SCALABLE COMPUTING: PRACTICE AND EXPERIENCE ISSN 1895-1767
Volume 7, Number 4, pp. 89–99. http://www.scpe.org © 2006 SWPS

GENERATIVE MOBILE AGENT MIGRATION IN HETEROGENEOUS ENVIRONMENTS

B. J. OVEREINDER, D. R. A. DE GROOT, N. J. E. WIJNGAARDS, AND F. M. T. BRAZIER*

Abstract. Agent migration, in theory, makes it possible to bring computations to the resources required. In practice, however, homogeneity in programming language and/or agent platform is required. This paper presents an approach that supports heterogeneous agents and platforms: agents written in different languages can migrate between non-identical platforms. Instead of migrating the "code" (including data and state) of an agent, a blueprint of an agent's functionality is transferred (together with its state). An agent factory on the receiving platform generates new code on the basis of this blueprint. This approach of *generative mobility* not only has implications for interoperability but also for security, as discussed in this paper.

Key words. mobile agents, process migration, compositional design

1. Introduction.

Agents are used to solve complex problems in distributed environments, in which heterogeneity of protocols, runtime environment, operating system and hardware resources are a fact of life. Agents migrate from one machine to another: moving computations to resources and/or services, overcoming the high latency or limited bandwidth problem of traditional client-server interactions [19], and providing resource owners a means to control access to their resources. The current evolution of intelligent networked systems and Grid management, for example, is based on this technology [7]. A similar tendency is observed in the search and filtering of global information in the electronic marketplaces, e-commerce, and information retrieval on the World Wide Web [20].

To support agent mobility a distributed system needs provisions to physically migrate units of computation at runtime. This migration includes relocation of an agent's code base and state to another platform. Code and state migration is a complex task with technical complications such as incompatibility of platforms. The current solution to migration of active units of computation is to provide homogeneous platforms, either physically such that binary checkpoints can be restarted at another location, or virtually by using virtual machines, e.g., the Java Virtual Machine, providing a machine independent platform. The homogeneity requirement, physical or virtual, is a strong requirement: mobility is otherwise impossible. Another solution is for an agent to carry different versions of its binaries (or URLs) depending on the platforms it may be expected to encounter. The same objections, however, hold: the platforms need to be pre-specified.

Another important issue in mobile agents technology is security. In most current systems trust in the owners and in the machines on which an agent has previously run, are the only basis for a security model. Code signing and certificates are the techniques used to this purpose.

This paper presents an alternative approach to agent mobility and security. Not the code migrates, but an agent's blueprint (implementation independent description) and state. A receiving platform *regenerates* a mobile agent as it migrates to its new location. Homogeneity is no longer required: an agent programmed in Java can be transformed to, for example, a Python implementation of the agent with the same functionality. Trust in an agent coming from another machine increases considerably if the receiving platform uses its own trusted components to reconfigure an agent.

Section 2 discusses mobility of processes and agents in more detail. Section 3 describes the concept of an agent factory. Heterogeneous migration based on this concept is the topic of Section 4. Implications of this approach for heterogeneous migration and security are discussed at more length Section 6. Section 8 sums up the results and proposes future research.

2. Agent Migration.

2.1. Mobility.

An agent in a mobile agent system is typically associated with a unit of computation which resides in the lower layers of a virtual machine. A unit of computation is composed of the code describing its behaviour, the associated data, and its execution state. Mobile agent systems allow migration of the whole unit or a part thereof, i. e., one or more of the three constituents mentioned above. The most relevant differences among existing systems lie exactly in what is moved and how [31].

A distinction can be made between systems that migrate execution state along with the unit of computation and those that do not. Systems that do, are said to support *strong mobility*, as opposed to systems that discard the execution state across migration, and are hence said to provide *weak mobility*. In systems supporting strong mobility, migration is completely transparent. In systems supporting weak mobility, if part of an execution state is needed at a later state in an agent's lifetime, it needs to be explicitly saved.

*IIDS Group, Department of Computer Science, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Strong mobility, as found in NOMADS [37], Ara [29], D'Agents [15], requires that the entire state of an agent, including its execution stack and program counter, is saved before an agent is migrated to a new location. This process of saving the entire state of an executing process is called *checkpointing*. An important quality of strong mobility is transparent migration of the running process. That is, the agent is unaware of the migration, bindings to other agents and objects are transparently resolved, i. e., references to agents and objects are location independent. The checkpoint/migration facility is either implemented at the operating system level [16, 17, 28] or incorporated within the virtual machine of an interpreted language (e.g., within the Java Virtual Machine [37]).

Despite the advantages of strong mobility, many agent systems support weak mobility (like Ajanta [38], Aglets [18], Grasshopper [1], AgentScape [27] and JADE [2]). Most current agent systems are implemented on top of the Java Virtual Machine (JVM), which provides object serialization and basic mechanisms to implement weak mobility. The JVM does not provide mechanisms to deal with execution state transfer.

Agent mobility is relatively straightforward in homogeneous environments. For strong mobility with checkpoint/migration incorporated at the operating system level, agent mobility is limited to identical computer architectures running the same operating system. Agent mobility facilities implemented at the virtual machine level makes the migration of agents machine independent, but is still homogeneous in language, i. e., only migration of agents from Java to Java platforms is possible.

2.2. Heterogeneity & Interoperability. Migration of mobile agents does not need to be constrained by homogeneity of code bases and platforms. *Generative migration* is an option. Blueprints of the functionality of an agent, together with its state, are the basis of this type of migration. A blueprint defines the functional components of which an agent is composed: implementations of these components differ between platforms. When an agent decides to migrate its blueprint and state are transferred to the designated platform. The agent is regenerated on this platform according to its blueprint. The same functional components are used. The components, however, may be coded in a different code base, the code base supported by the agent platform to which the agent migrated. The agent platform may also differ: its the blueprint that matters.

Regeneration relies on an automated facility (*agent factory*) on the receiving platform to reconfigure an agent on the basis of its blueprint and state. The interface with which an agent communicates to the platform on which it runs can differ significantly.

Taking this interface into account, and the specific machines on which an agent runs, i. e., the host, the following migration scenarios can be distinguished.

Homogeneous migration An agent migrates to another host without any changes to the format of its executable code or the interfaces to the agent platform. This form of migration requires that source and destination platform offer the same interfaces, but also that the (virtual) machine that executes the agent is the same on both sides. In practice, this form of migration is most common.

Cross-platform migration An agent migrates to another host with a different agent platform, one that offers the same (virtual) machine architecture. This generally entails changes to the interface to the agent platform, but not necessarily changes to the format of its executable code. This form of migration may occur when, e.g., a Java-agent migrates from a Ajanta platform to a Zeus platform. One commonly applied solution is to offer wrapper interfaces that hide the differences between source and target platform. Another approach, followed in MAF [26] or FIPA, is to enforce platforms to implement a standard interface for interoperability.

Agent-regeneration migration An agent migrates to a host running a different (virtual) machine. This requires an agent to be regenerated, resulting in different executable code. Note that the target agent platform may be the same as that of the source, simplifying regeneration. To regenerate an agent, it is necessary that the target has a blueprint of the agent.

Heterogeneous migration An agent migrates to another host with a different agent platform with a different (virtual) machine. In this case, regeneration of the agent is necessary. As the underlying agent platform is also different the agent's blueprint must be platform independent.

This paper advocates the support of heterogeneous migration as it offers the most flexibility. As distributed systems are gradually required to scale worldwide across different administrative organizations, and to support a myriad of platforms, solutions are needed that anticipate heterogeneity and adaptability. Regeneration of agents for different underlying platforms is a step towards meeting such requirements.

The approach described in this paper combines heterogeneous migration with weak migration. The term proposed for our approach introduced above *generative migration*. Generative migration for agents may open the world of distributed

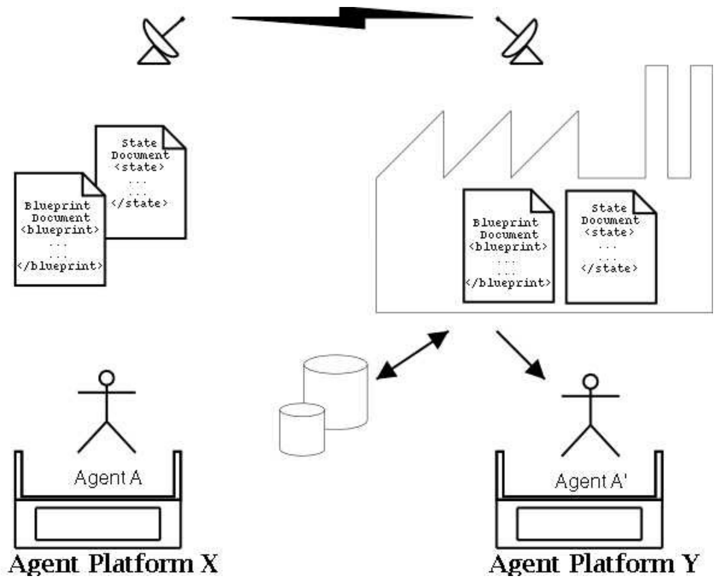
State Document <state> ... </state>
Blueprint Document <blueprint> ... </blueprint>
Agent A
Agent A'
Agent Platform X
Agent Platform Y

FIG. 3.1. *Principles of generative migration.*

systems to agent-developers. The adage "write once, run everywhere" is achieved while retaining heterogeneity and tackling the problem of interoperability.

Generative migration requires that a target host has access to an agent factory capable of generating an agent for that target. Ideally, this factory is placed on the target host or another machine on the same local-area network. An important issue is that the factory is trusted to generate an agent that the target host/platform can trust. Security and trust are briefly discussed in Section 6.

Our approach has the additional benefit that various optimizations become possible. For example, an agent generated by a factory may be optimized with respect to the target's machine architecture, or the way that local resources such as databases are accessed. In addition, transmission of an agent's blueprint and information on its current state will generally require fewer network resources than migrating an agent using more traditional approaches. On the downside, the agent-generation process may affect overall performance if agents migrate frequently.

3. Generative Agent Migration. Assuming agents have a compositional structure described by their *blueprints*, building an agent is, in fact, a configuration task: a task that can be automated. Automated (re-)design of agents is the task of an agent factory [5]. An agent factory generates from this blueprint an agent for a specific platform (in many ways this functionality is comparable to a compiler). This section describes reconfiguration by an agent factory.

The process of generative migration is illustrated in Figure 3.1. An agent's blueprint describes its structure and functionality. Its state (including private data) is needed for its execution. Both the blueprint and state are described in a format independent of the operating system and agent system, e.g., in XML. The blueprint and state are transferred to the destination platform requested by the agent. The destination's agent factory re-builds the agent on the basis of the blueprint and state it's received. Regeneration of the agent relies on availability of the appropriate building blocks. These building blocks are retrieved from a local repository. Finally, the regenerated agent is initiated with its state, on the platform for which it has been generated.

Section 3.1 discusses characteristics of an agent factory, and Section 3.2 illustrates this concept for a specific type of agent, namely an information retrieval agent. Section 3.3 describes a current prototype of this agent factory.

3.1. Concepts of an Agent Factory. Whether the need for adaptation is identified by an agent itself, or by another agent, is irrelevant in the context of this paper. An agent factory simply constructs new agents and/or modifies existing agents [5]. The (re-)design of agents is fully automated, with very limited interaction with outside parties. The concept of an agent factory requires (i) agents to have a compositional structure, (ii) one or more libraries of re-usable agent components, and (iii) one or more ways to describe the functionality of these agent components.

In the agent factory discussed in this section two additional assumptions hold: (i) two levels of description are distinguished: conceptual and detailed, and (ii) no commitments are made to specific programming languages and/or ontologies.

A conceptual description of (parts of) an agent is an architectural description: a blueprint of the components, interfaces and interactions between components. A detailed description includes code, together with definitions of, e.g., interfaces. A mapping between these descriptions defines the relationship between the elements at one level with elements at the other. This mapping may be structure preserving. (Note that this is not always ideal.) A number of detailed components may exist for each conceptual component (e.g., one in C, another in Java), and vice versa.

The concept of a building block is used to describe the components within an agent factory at both levels. Some building blocks contain open slots, others are fully specified and operational. Both define their functionality on the basis of their interfaces. Open slots define the interfaces of the building blocks to be inserted.

Depending on availability and domain of application libraries of building blocks may include: partial agent designs (cf., generic models/design patterns [13, 30, 33]), knowledge-based models (e.g., problem-solving models [35] or generic task models [4]), agent-wrappers (providing cross platform interfaces) (e.g., AgentScape, Zeus [25], message parsing Ajanta [38]), et cetera. Building blocks may be written in, e.g., UML, Python, C++, CommonKads, etc.

3.2. Agents and Building Blocks. A blueprint of an agent contains descriptions of the interfaces of building blocks and their open slots, and additional information on the relation between the building block configurations at the two levels of detail.

Consider, for example, the architecture of a simple information retrieval agent. This simple agent is only capable of communication with one other agent, e.g., its owner, and interaction with external resources using one protocol. The template for this simple information retrieval agent contains the agent architecture shown in Fig 3.2. A number of the components and data structures contain open slots in this model specifying the interfaces of the required building blocks. These are not depicted in this figure.

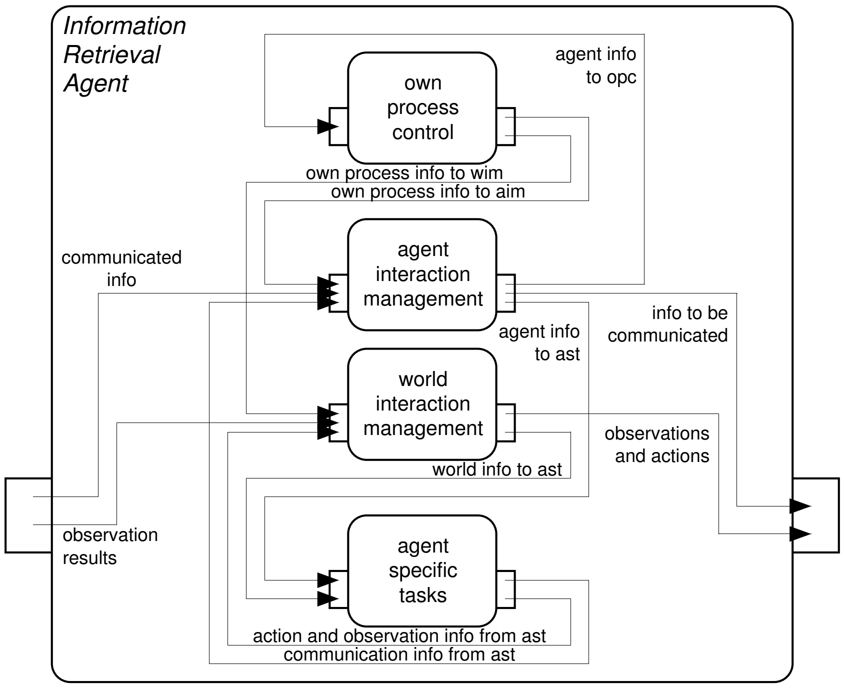

FIG. 3.2. *Architecture of a simple information retrieval agent.*

The component Own Process Control (opc) has an open slot for a building block with knowledge of an agent's identity and the agent's preferences for interactions with resources. The component World Interaction Management (wim) has an open slot for a building block capable of managing interaction with external resources. The component Agent Specific Task (ast) has an open slot for a building block capable of transforming user requests into queries, and query results into a format/language users may understand. The fourth component of the agent, Agent Interaction Management (aim), does not have an open slot.

The control inside this building block is pre-defined, no control slot is available for extension. A number of datastructures used by the agent need to be extended, see Section 4. The library contains a detailed building block for this information retrieval agent template in Java. The structure of the code mirrors the architecture of the agent.

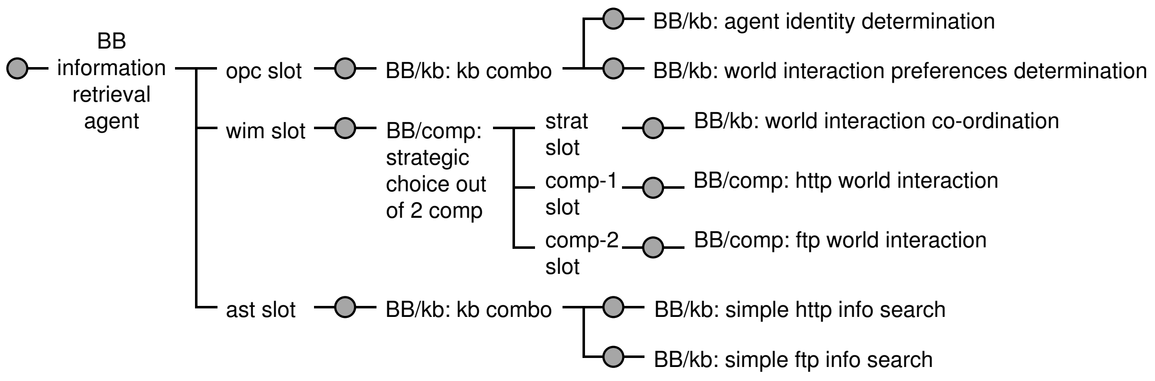
BB information retrieval agent
opc slot
BB/kb: kb combo
BB/kb: agent identity determination
BB/kb: world interaction preferences determination
wim slot
BB/comp: strategic choice out of 2 comp
strat slot
BB/kb: world interaction co-ordination
comp-1 slot
BB/comp: http world interaction
comp-2 slot
BB/comp: ftp world interaction
ast slot
BB/kb: kb combo
BB/kb: simple http info search
BB/kb: simple ftp info search

FIG. 3.3. *Building block configuration of a simple information retrieval agent.*

Figure 3.3 illustrates a building block-configuration in which two levels of building blocks were required: each open slot required a building block that itself contained other open slots. Note that the lower level building blocks make a distinction between open slots for data, and open slots for processes.

3.3. Agent Factory Prototype. A prototype of the agent factory has been implemented. This prototype automatically (re-)configures an information retrieval agent depending on the site to which it migrates.

Conceptual components are specified in the DESIRE framework [3, 4]. The compositional nature of DESIRE models, and the separation between processes and knowledge makes it possible to specify the functionality of knowledge intensive systems as the composition of (reusable) components. A structure-preserving mapping exists between the configuration of building blocks at the conceptual level of abstraction (i. e., DESIRE specifications) and the configuration of building blocks at the detailed level of abstraction. The detailed components are in Java.[1]

The prototype agent factory itself is written in Java, and contains the knowledge needed to (re-)design simple information retrieval agents.

4. Migration Using the Agent Factory Service. One of the strengths of the agent factory concept is that it provides a means to support migration of agents in heterogeneous environments that require a high level of security. Section 4.1 discusses pre-conditions for successful migration of agents. Section 4.2 describes the approach in agent-factory-enhanced migration.

4.1. Migration Pre-Conditions. To facilitate the description of agent migration, an agent is assumed to consist of executable code and state. Executable code may contain "code and data," if these can be distinguished, or may be inseparable (as in Prolog). When an agent migrates, it needs to retain sufficient information from its state to resume execution at its destination. Note that this description leans towards weak-mobility: it may not be necessary to transport the entire state of an agent. This, however, depends on the application.

Although it is not necessary for the source and destination host to both have access to an agent factory, it greatly simplifies descriptions of the migration process. An agent needs to be able to store and restore information on its state; this is a requirement for interoperability. An implementation-independent format such as XML, RDF, or OIL can be used to this purpose.

The agent factories on the hosts need to share the same functional components. That is, both have the same libraries of building blocks at the conceptual level of abstraction, but may have different libraries of building blocks at the detailed level, depending on the nature of the destination location. For example, an agent factory may have a mapping from a conceptual agent architecture building block (its functional components) to a detailed building block written in Java, while another agent factory may have a detailed building block written in C++.

4.2. Approach to Migration. In essence, migration entails moving an agent from one machine to another. This usually involves pre-packaging an agent before its move, such that its executable code and state may be restored at the destination host. Migration using an agent factory diverges from standard mobility of agents in that *not* an agent's executable code with state is migrated, but its blueprint together with (parts of) its state. This might seem to be similar

[1]Automated prototype generation within the DESIRE framework on the basis of detailed formal specifications facilitates verification and validation of knowledge intensive systems; this feature is not used within the current prototype of the agent factory.

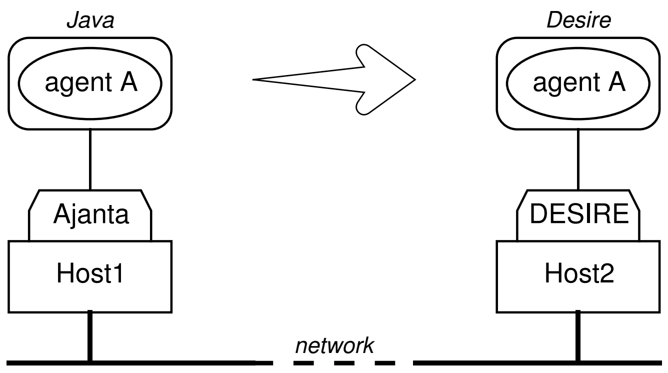

FIG. 4.1. *Example migration scenario in which agent A on host 1 (written in Java, running on Ajanta) migrates to host 2 (where it will be specified in* DESIRE *and running on* DESIRE*).*

to Java agents and their interaction with class loader objects that allows specific implementations of Java classes to be loaded; however, in our approach a blueprint of an agent can be sent to any arbitrary platform like Java, Python, or Prolog.

Consider the scenario for heterogeneous mobility, depicted in Fig. 4.1. An information retrieval agent A resides on a host machine H1. This host runs the Ajanta [38] agent platform, and, as such, supports Java agents. The agent wishes to move to another host: host H2. The host H2 runs the DESIRE platform, and its agents run code generated by the DESIRE platform.

In the process of migrating the agent A from host H1 to host H2, the agent first needs to offload information on its state. Then the agent factory on host H1 sends the blueprint of the agent, together with the state information of the agent to host H2.

Host H2's local agent factory receives the blueprint of the agent and state information. This agent factory designs a DESIRE agent A on the basis of the blueprint of agent A. This DESIRE agent A (i. e., a functionally equivalent incarnation of the Java agent A) runs on DESIRE's virtual machine (the DESIRE interpreter), and is able to incorporate information on its state.

The agent factory on the receiving side regenerates the agent, possible in a different implementation language and in a different environment.

5. Experimental Validation. The experimental setting consists of two agent platforms: JADE (version 3.01b) and FIPA-OS (version 2.10). The goal of our experiment is to connect the platforms to enable the exchange of messages and the migration of agents following the principles on generative migration (i. e., a blueprint and state are transferred).

5.1. Agent Platforms. Agent platforms provide an environment for the execution of agents. JADE [2] the Java Agent Development Environment, is a FIPA-compliant agent platform. The main objective of JADE is to simplify the development of agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. As a secondary objective the JADE agent platform tries to optimize the performance of a distributed agent system implemented in the Java language. JADE consists of two parts. The first part is the runtime environment for FIPA compliant multi-agent systems. The second is a framework for FIPA compliant agent system development. JADE supports weak agent migration using Java serialization.

FIPA-OS [32] is an open source agent platform implemented in the Java programming language. The main purpose of the FIPA-OS agent platform is to provide a reference implementation for the standard specifications of the FIPA organization. A key focus of the platform is that it supports openness through a loose coupling between the platform components and compliance to the FIPA Agent Management reference model. Official releases of FIPA-OS do not support agent migration.

5.2. Assumptions. Given the assumptions for Generative Migration, extra assumptions are necessary for the realization of Cross-Platform Generative Migration implementation:

- Appropriate communication facilities in the agent platforms exist: for example, facilities for sending ACL messages via HTTP are needed. A bidirectional channel for communication is needed for sending messages. The communication channel is used for requesting migration and sending/receiving information about the agents (e.g., blueprint and state documents).

TABLE 5.1
Test results.

	Message Structure	Communication Protocols	Creation Process
JADE	ACL	HTTP, CORBA ORB (IIOP)	simple, can be isolated
FIPA-OS	ACL	HTTP, CORBA ORB (IIOP)	currently part of the GUI, difficult to isolate

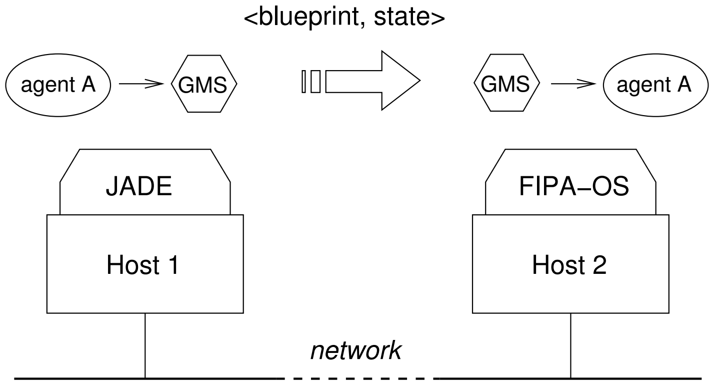

FIG. 5.1. *Generative migration experiment with JADE and FIPA-OS.*

- There is a service for agent creation or regulated control over the agent creation process. By means of an agent creation process a new agent can be created on a target platform. Control over the creation process allows for the implementation of a service that starts and initializes a new agent.

Agent creation, as such, was a challenge in FIPA-OS. The agent creation process is difficult to isolate in FIPA-OS: it is integrated in the User Interface classes and other classes. Implementation of a separate service for automatic agent creation and initialization is not currently supported. JADE does support separate services for this purpose.

5.3. Scenario. The scenario focuses on the technical aspects of generative agent migration. The environmental setting consists of two Agent Platforms: FIPA-OS and JADE. A number of services is assumed to be present on both platforms: a standard Directory Facilitator (DF) and a Generative Migration Service (GMS). The GMS is our own implementation and its primary task is to take care of migration, agent creation and initialization. The GMSs are registered at the local DF and can be found by agents and other services on both platforms if the DFs are cross-registered.

An agent sends a migration request to the local GMS for migration to a remote platform. The GMS contacts the remote GMS and forwards the request. The agent receives an acknowledgement of the migration request, or denial with an indication of the reason for denial (e.g., no GMS on target location, building-blocks not present on target or an agent with (requested) identical name exists). On acknowledgement the agent hands over its blueprint and state to the local GMS. The local GMS contacts the remote GMS for creation of a new agent with a given blueprint, state and agent name (see experiment scenario in Fig 5.1). The remote GMS needs to be able to assemble a new agent based on the blueprint of the requesting agent, it must have capabilities and permission to create a new agent on the platform and the means to initialize the new agent with the state of the original agent. Once a new agent is running the agent in the source platform can be stopped and removed.

5.4. The Implementation. As explained in Section 3, a compositional agent can be described in a blueprint document, written in an independent format that can be exchanged between locations. Additionally, state information can be used to initialize a newly generated agent on the target location.

To implement generative agent migration in agent platforms, two frameworks had been developed: a conceptual framework and an operational framework. The conceptual framework is based on the DESIRE modelling framework [16] and has four kinds of structure elements: components, links, information types, and control. Each component has an input and an output buffer and components define their input and output information types. Links can be placed between components to connect input and output buffers. A link transfers information elements (instances of information types) from an output buffer to an input buffer. Which information types a link transports can be predefined. In this framework only component structure elements can contain other structure elements.

On the operational level, an implementation-language specific framework is needed. For building-block components implemented in the Java programming language a framework had been developed which has a one-to-one mapping with the conceptual framework. The operational framework is used to dynamically load Java components via the provided mechanisms for class loading.

A compositional agent is embedded in a wrapper, the wrapper is an extension of the base-agent of an agent platform, e.g., in JADE it is `jade.core.Agent` and in FIPA-OS `fipaos.agent.FIPAOSAgent`.

6. Security Issues. Migration of an agent involves security from a number of perspectives. Security issues related to authenticating an agent, and deciding whether an agent is allowed to migrate to its destination, are not discussed in this paper. What remains is how to protect an agent against attacks during and after its migration, and how to protect a target against attacks from a malicious agent. Considerable research has already been conducted with respect to both issues which can be applied to our approach. In the following, the role of security is briefly considered. It should be noted, however, that security in our approach is subject of current research.

6.1. Protecting an Agent. A mobile agent may be preyed upon while it is in transit, or while running on a malicious host. It is impossible to protect an agent against modifications during its transfer or execution in an untrusted environment [12]. At best, it can check whether an agent has been maliciously modified and take appropriate measures after the fact. Our approach to migration can help here.

It is important to realize that, in principle, an agent's blueprint does not change during its lifetime. (Except for reconfiguration at an agent factory.) Consequently, by adding an integrity check to a blueprint using standard techniques for digital signatures [34], it is easy to detect whether a blueprint has been changed. When a factory notices that a blueprint has been changed, it can either discard the agent or generate it from the original blueprint. The latter is possible only if that blueprint is locally available, or if it can be retrieved in a secure way. Securely retrieving a blueprint requires that a factory can set up a secure channel to a blueprint repository, that is, a channel that provides authentication and transmission integrity.

Of course, it should be possible to support *evolutionary agents* for which new blueprints are generated. However, blueprint generation should be done only by trusted factories and never as a solution to migration. As such, it falls outside the scope of this paper.

6.2. Protecting a Host. A host that admits foreign mobile agents to its resources takes a risk: some of the agents may be malicious, and may try to subvert (parts of) the host. The problem with traditional approaches to agent migration is that it is impossible to check in advance whether or not imported code does only what it promises. The solution is to construct what are known as sandboxes [39]: a restricted environment in which, effectively, each instruction is monitored and checked before being executed. If access to resources is violated, execution halts. The sandbox model is quite restrictive, and has been extended since its initial introduction (see, for example, [14, 23]).

Regenerating agents from blueprints may considerably help to protect a host against malicious code. Normally, blueprints do not contain code descriptions, but refer only to interfaces and components that should be locally available to an agent factory. The code contained in these components may have been verified by the owner of the factory, or have been obtained from trusted sources. Of course, protection will fail if verification has not been done properly. In effect, a requirement is that trusted code is available before an agent migrates to a target, or that can be retrieved from a trusted repository through a secure channel.

In those cases that blueprints require execution of untrusted code, traditional approaches based on sandboxing techniques or protection domains need to be implemented as part of the target platform.

A mobile agent arriving at the host is regenerated on the basis of its blueprint, using only detailed building blocks of which the host approves. Although the specific configuration of building blocks may be new to the host, a number of security risks can be removed. The mobile agent may still be untrustworthy, but is prevented from executing certain calls on the host.

As an example, consider a bank that wishes to use mobile agents which may transact money from one account to another. The bank offers libraries of building blocks written in Java to its clients. These clients may build mobile agents that can perform transactions at the bank. The bank admits only those mobile agents that can be regenerated on the basis of their blueprint using building blocks written in their programming language of choice, running on their machines. Note that cheating, using other people's passwords and certificates is not necessarily stopped by this approach.

7. Related Research. Related research with respect to the agent factory and other heterogeneous agent migration initiatives is shortly discussed.

In this paper an Agent Factory is a service to design, adapt and (re-) assemble agents from building blocks. The Agent Factory to which this paper refers is the Agent Factory described in [6, 36]: the IIDS Agent Factory. There are other agent factories. The Factory of the Agents [8, 9] currently being developed to provide "a cohesive framework that supports a structured approach to the development and deployment of agent oriented-applications", providing extensive support for the creation of Belief-Desire-Intention (BDI) agents.

The Agent Factory described in [10, 11] is designed to provide designers with a tool for building agents, and a repository of patterns that can be used to add new capabilities. Agents in this project are developed according to the PASSI design methodology, with which models of multi-agent systems and agent interactions are specified and expressed in UML.

The IIDS Agent Factory has been designed for automated design and redesign of single agents, whereas the two other agent factories are being designed to assist human designers designing multi-agent systems. However, some similarities can be found in the approaches to agent design and the assembly process.

Also related to this research are other heterogeneous agent migration approaches. Known implementations are based on wrappers, intermediate interface layers, and agent servers. Especially worth mentioning are the Guest [21, 22] and Monads [24] research efforts.

The developers of Guest [21, 22] propose a middleware-based model that introduces an intermediate layer for the support of their agents on top of an existing agent platform. Regardless of the underlying platform, if it supports the Guest Layer, Guest Agents can be executed. The goals of the project are (i) to allow interoperability between platforms, i. e., allowing agents to run, communicate and move between them; (ii) To provide a uniform view of those platforms, i. e. agents can be written and manipulated without considering the kind of servers they will run on; (iii) To add new adaptive features to the platforms, i. e., plug-in mechanisms for dynamic extensions. Though still an experimental prototype, Guest enables interoperability between Voyager, Aglets, Grasshopper and JADE.

The Monads project [24] concentrates on the needs of nomadic users and adaptability. By adaptability they primarily mean the ways in which services adapt themselves to properties of terminal equipment and to characteristics of communications. This involves both mobile and intelligent agents as well as learning and predicting temporary changes in the available Quality-of-Service along the communications paths. The goal of Monads is to design an efficient and reliable software architecture based on adaptive services and agents, and to develop prototypes based on that architecture. The agents themselves are not adaptive, only used to steer changes in, e.g., quality of service.

The basic approach to mobile agents in Monads is a separation of an agent into a head and a body. The body handles the agent-programming interface of each agent platform, and a head can be placed on top of it. The agent-head can migrate via a service called the Monads Agent Gateway (MAG). The Monads approach has been implemented on JADE, Voyager and Grasshopper agent platforms.

There are some differences and similarities with our approach. The Guest Interface Layer defines a generic agent interface that can be supported on other platforms. This differs with the approach explained in this paper because our agents are assembled (i. e. adapted) before execution on any platform takes place and agents in Guest are not adapted. A similarity with Monads and our approach concerns one of the goals of Guest: to provide a uniform view of the underlying platform. In Monads an agent consists of two parts: a head and a body. In our approach a similar division into agent-head and agent-body can be made. In contrast with the Monads agent-head our agent consists of multiple components and can be migrated using generative migration whereas Monads does not make this distinction and uses Java object serialization for migration. Additionally the functionality of their MAG-service is comparable with our Generative Migration Service.

8. Discussion and Future Work. Agents, and in particular mobile agents, offer a means for application developers to build distributed applications. Mobility of agents is often required for various reasons, notably performance. Current agent platforms offer a wide range of services to agent developers, including mobility. However, mobility of agents is usually limited to hosts running the same agent platform and that have the same (virtual) machine architecture. In other words, it is often restrained to a homogeneous environment.

The approach described in this paper transcends this homogeneity and proposes *generative mobility*. In generative mobility, a blueprint of an agent's functionality is transported, together with information on the agent's state. At its destination, an agent factory regenerates the executable code of the agent on the basis of its blueprint. An agent may then restore its state and resume execution.

With generative mobility an agent may travel to locations that offer a different platform and that require it to adopt a different (virtual) machine architecture. In other words, generative mobility supports true heterogeneous mobility, offering an agent maximum flexibility with respect to where it wants to go. In addition, an agent's executable code can be optimized for its destination, while retaining its required agent-level functionality. In their own way, agent factories and blueprints offer a language and agent platform independent virtual machine that allows for heterogeneous migration.

Agent factories play an important role in generative mobility as they offer the services needed to generate executable code on the basis of blueprints. Agent factories rely on libraries of building blocks from which agents can be configured. As a consequence, agent factories need to share these (conceptual) building blocks to understand an agent's blueprint and be able to generate its associated executable code. Homogeneity in agent architectures is a likely consequence of this approach.

Research on generative mobility is clearly not finished. In particular, the use of blueprints needs to be investigated to determine to what extent blueprints are flexible enough to describe agents, and how security can be adequately dealt with. Agent factories form an important component within our worldwide distributed AgentScape system that allows agents to be automatically (re-)designed. Currently a new prototype of the agent factory (namely the libraries of components) in AgentScape is being built that supports generative migration, based on the factory described in this article.

The use of generative mobility for relatively closed environments, such as hospitals, is currently being studied. Generative mobility with trusted code libraries on the hospital side may possibly provide a solution to controlled access to medical dossiers. Insurance companies, for example, are allowed limited access to specific types of information and processing. Control over the executable code of an insurance company's agent provides a means for a hospital to control the calls and data an agent may execute inside the hospital.

REFERENCES

[1] C. Bäumer, M. Breugst, S. Choy, and T. Magedanz, *Grasshopper—A universal agent platform based on OMG MASIF and FIPA standards*, in First International Workshop on Mobile Agents for Telecommunication Applications (MATA'99), Ottawa, Canada, Oct. 1999, pp. 1–18.

[2] F. Bellifemine, A. Poggi, and G. Rimassa, *Developing multi-agent systems with a FIPA-compliant agent framework*, Software: Practice and Experience, 31 (2001), pp. 103–128.

[3] F. Brazier, C. Jonker, and J. Treur, *Compositional design and reuse of a generic agent model*, Applied Artificial Intelligence Journal, 14 (2000), pp. 491–538.

[4] ———, *Principles of component-based design of intelligent agents*, Data and Knowledge Engineering, 41 (2002), pp. 1–28.

[5] F. Brazier and N. Wijngaards, *Automated servicing of agents*, in Proceedings of the AISB-01 Symposium on Adaptive Agents and Multi-Agent Systems, Mar. 2001, pp. 54–64.

[6] ———, *Designing self-modifying agents*, in Proceedings of Computational and Cognitive Models of Creative Design, the Fifth International Roundtable Conference, Dec. 2001, pp. 93–112.

[7] A. Chakravarti, G. Baumgartner, and M. Lauria, *The organic grid: Self-organizing computation on a peer-to-peer network*, in Proceedings of the First International Conference on Autonomic Computing (ICAC 2004), New York, NY, May 2004, pp. 96–103.

[8] R. Collier and G. O'Hare, *Agent factory: A revised agent prototyping environment*, in Proceedings of the 10th Irish Artificial Intelligence and Cognative Science Conference (AICS'99), Cork, Ireland, Sept. 1999.

[9] R. Collier, G. O'Hare, T. Lowen, and C. Rooney, *Beyond prototyping in the factory of the agents*, in Proceeding of the 3rd Central and Easern European Conference on Multi-Agent Systems (CEEMAS'03), Prague, Czech Republic, June 2003.

[10] M. Cossentino, *Different perspectives in designing multi-agent systems*, in Proceedings of the AGES'02 Workshop at NODe02, Erfurt, Germany, Oct. 2002.

[11] M. Cossentino, P. Burrafato, S. Lombardo, and L. Sabatucci, *Introducing pattern reuse in the design of multi-agent systems*, in Proceedings of the AITA'02 workshop at NODe02, Erfurt, Germany, Oct. 2002.

[12] W. Farmer, J. Guttman, and V. Swarup, *Security for mobile agents: Issues and requirements*, in Proceedings of the 19th National Information Systems Security Conference, Baltimore, MD, Oct. 1996, pp. 591–597.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1994.

[14] L. Gong and R. Schemers, *Implementing protection domains in the Java Development Kit 1.2*, in Proceedings of the Symposium on Network and Distributed System Security, San Diego, CA, Mar. 1998, pp. 125–134.

[15] R. Gray, G. Cybenko, D. Kotz, R. Peterson, and D. Rus, *D'Agents: Applications and performance of a mobile-agent system*, Software: Practice and Experience, (2001). In press.

[16] K. Iskra, F. van der Linden, Z. Hendrikse, B. Overeinder, G. van Albada, and P. Sloot, *The implementation of Dynamite: An environment for migrating PVM tasks*, Operating Systems Review, 34 (2000), pp. 40–55.

[17] D. Johansen, R. van Renesse, and F. Schneider, *Operating system support for mobile agents*, in Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), Orcas Island, WA, May 1995, pp. 42–45.

[18] D. Lange, M. Oshima, G. Karjoth, and K. Kosaka, *Aglets: Programming mobile agents in Java*, in Worldwide Computing and Its Applications, vol. 1274 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 1997, pp. 253–266.

[19] D. B. Lange and M. Oshima, *Seven good reasons for mobile agents*, Communications of the ACM, 42 (1999), pp. 88–89.

[20] M. Luck, P. McBurney, O. Shehory, and S. Willmott, *Agent technology: Computing as interaction—A roadmap for agent-based computing*, tech. rep., AgentLink III, Sept. 2005. `http://www.agentlink.org/roadmap/`

[21] L. Magnin, T. V. Pham, A. Dury, N. Besson, and A. Thiefaine, *Our guest agents are welcome to your agent platforms*, in Proceedings of the ACM Symposium on Applied Computing (SAC 2002), Madrid, Spain, Mar. 2002, pp. 107–114.

[22] L. Magnin, H. Snoussi, V. T. Pham, A. Dury, and J.-Y. Nie, *Agents need to become welcome*, in Proceedings of the 3rd International Symposium on Multi-Agent Systems, Large Complex Systems, and E-Businesses (MALCEB'2002), Erfurt, Germany, Oct. 2002.

[23] D. Malkhi and M. Reiter, *Secure execution of Java applets using a remote playground*, IEEE Transactions on Software Engineering, 26 (2000), pp. 1197–1209.

[24] P. Misikangas and K. Raatikainen, *Agent migration between incompatible platform*, in Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan, Republic of China, Apr. 2000.

[25] H. Nwana, D. Ndumu, L. Lee, and J. Collis, *ZEUS: A tool-kit for building distributed multi-agent systems*, Applied Artifical Intelligence Journal, 13 (1999), pp. 129–186.

[26] OMG, *Mobile agent facility specification*, OMG Document formal/00-01-02, Object Management Group, Framingham, MA, Jan. 2000.

[27] B. Overeinder and F. Brazier, *Scalable middleware environment for agent-based Internet applications*, in Proceedings of the Workshop on State-of-the-Art in Scientific Computing (PARA'04), Copenhagen, Denmark, June 2004, pp. 675–679. Published in Applied Parallel Computing, LNCS 3732, Springer, Berlin, 2006.

[28] B. Overeinder, P. Sloot, R. Heederik, and L. Hertzberger, *A dynamic load balancing system for parallel cluster computing*, Future Generation Computer Systems, 12 (1996), pp. 101–115.

[29] H. Peine and T. Stolpmann, *The architecture of the Ara platform for mobile agents*, in Proceedings of the First International Workshop on Mobile Agents (MA'97), vol. 1219 of Lecture Notes in Computer Science, Berlin, Germany, Apr. 1997, Springer-Verlag, pp. 50–61.

[30] F. Peña-Mora and S. Vadhavkar, *Design rationale and design patterns in reusable software design*, in Artificial Intelligence in Design (AID'96), Dordrecht, 1996, Kluwer Academic Publishers, pp. 251–268.

[31] G. P. Picco, *Mobile agents: An introduction*, Microprocessors and Microsystems, 25 (2001), pp. 65–74.

[32] S. Poslad, P. Buckle, and R. Hadingham, *The FIPA-OS agent platform: Open source for open standards*, in Fifth International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, Manchester, UK, Apr. 2000, pp. 355–368.

[33] A. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, Reading, MA, 1996.

[34] B. Schneier, *Applied Cryptography*, John Wiley, New York, NY, 2nd ed., 1996.

[35] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. V. de Velde, and B. Wielinga, *Knowledge Engineering and Management, the CommonKADS Methodology*, MIT Press, 1999.

[36] S. v. Splunter, M. Sabou, F. Brazier, and D. Richards, *Configuring web service, using structurings and techniques from agent configuration*, in Proceedings of the 2003 IEEE/WIC International Conference on Web Intelligence (WI 2003), Halifax, Canada, Oct. 2003, pp. 153–160.

[37] N. Suri, J. Bradshaw, M. Breedy, P. Groth, G. Hill, and R. Jeffers, *Strong mobility and fine-grained resource control in NOMADS*, in Proceedings of the Joint Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA 2000), Zurich, Switzerland, Sept. 2000, pp. 2–15.

[38] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh, *Mobile agent programming in Ajanta*, in Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99), Austin, TX, May 1999, pp. 190–197.

[39] D. Wallach, D. Balfanz, D. Dean, and E. Felten, *Extensible security architectures for Java*, in Proceedings of the 16th Symposium on Operating System Principles, St. Malo, France, Oct. 1997, ACM, pp. 116–128.

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 16, 2006