Portable Resource Reification in Java-Based Mobile Agent Systems

Alex Villazón[1] and Walter Binder[2]

[1] University of Geneva, Switzerland
villazon@cui.unige.ch
[2] CoCo Software Engineering GmbH, Austria
w.binder@coco.co.at

Abstract. Resource awareness is an important step towards the realization of adaptable software, something which is particularly desirable in the context of mobile code and mobile agent environments. Since resources (CPU, memory, network bandwidth, etc.) are not available and manipulable as first-class entities in standard programming models, such as in the Java language, some kind of reification seems indispensable. This is however difficult to achieve, especially if portability is a requirement. In this paper we describe a mobile agent execution environment that reifies several aspects of both the execution environment itself and of the mobile agents it hosts. We explain how resources consumed by an agent are reified directly from the agent code. Performance measurements show that our approach incurs only moderate overhead.

1 Introduction

Resource awareness in the context of mobile agents has been identified as an important concept for agent adaptability. If a mobile agent is aware of its resource consumption, it may use this information e.g. to optimize its migration decisions. Furthermore, a mobile agent platform that executes unknown foreign code has to control resource allocation, i.e., the system has to account the resources consumed by an agent and to prohibit allocations exceeding the agent's resource limits, in order to prevent denial-of-service attacks caused by malicious (or buggy) agents, which may even crash the agent execution platform [21, 4]. Information about resource consumption may be used to implement different control algorithms (e.g., market-based [20,6], energy-based [2], applying different scheduling policies [4], etc.). Moreover, resource accounting and control may be targeted towards provision of quality of service or of usage-based billing, in order to amortize investments in hardware and software set at customers' disposal.

These considerations raise questions concerning the manipulation of resource-related information and the programmability of agents, since resource-related aspects are clearly non-functional, i.e., frequently these aspects are not directly related to the basic task of the agent, and therefore it is important to separate them from the base-level code of the agent. Another important issue is how resource consumption can be *reified*, i.e., how it can be made accessible for

G.P. Picco (Ed.): MA 2001, LNCS 2240, pp. 213–228, 2001.

manipulation. Unfortunately, most mobile agent execution environments do not provide any means for obtaining information regarding the resource consumption of different agents. As a remedy, we suggest to integrate reflective capabilities into the execution environment (EE). *Reflection* and *reification* are closely related concepts, since a reflective system requires the reification of some of its internals. That is, reflection is the capability of a system to reason about and act upon itself [15]. A reflective system is composed of a *base-level*, which is the part of the system reasoned about, and a *meta-level*, which has access to the reified information about the base-level.

Even though Java [12] is the predominant implementation language for mobile agent systems[1], it does not support resource accounting. Proposed solutions for resource control in Java are either incomplete, or rely on native code, on low-level resource control mechanisms offered by the underlying operating system, or on a modified Java Virtual Machine (JVM) [14]. Consequently, these systems are not well suited to be deployed in heterogeneous environments, such as the Internet, where a wide variety of different hardware platforms and operating systems has to be supported. Because portability is of paramount importance for the success of a mobile agent system, resource control facilities have to be provided on top of standard Java runtime systems.

Resource control has to cover physical resources, such as CPU, memory, and network bandwidth, as well as logical resources, such as threads, the number of agents, etc. Moreover, communication with agents or services is also subject to resource control policies, which may e.g. limit the communication bandwidth and the size of exchanged messages. The reification of physical resources poses some serious difficulties, as it should be based only on the information that can be obtained from the agent code itself, without resorting to any external functionalities, such as those provided by the operating system. Thus, one of our goals is to provide an abstract and portable representation of the physical resources mentioned above, as well as mechanisms allowing to manipulate them without relying on functionalities specific to a particular operating system.

In addition to this, our approach allows to fully exploit all advantages of mobile code, since the reification itself may be performed by mobile code. That is, special code is injected into the mobile agent platform in order to customize the reification process. This is achieved by allowing agents to interact with reflective components inside the EE, rather than only with an external interface of the reflective system.

This paper is organized as follows: In section 2 we discuss related work on reflection and resource control in mobile agent environments. In section 3 we describe the generic architecture of a mobile agent platform, which enables the reification of physical and logical resources, as well as of communication structures. In section 4 we explain some basic ideas for resource reification in Java. We focus on physical resources and give an overview of our techniques to transpar-

[1] For an (incomplete) list of different mobile agent platforms see *The Mobile Agent List* at `http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html`. Most of the systems presented there are based on Java.

ently reify memory and CPU resources by directly inspecting the code of mobile agents. In section 5 we present benchmarking results of our fully portable techniques for resource reification in Java. The last section concludes this paper.

2 Related Work

We distinguish two broad categories of related work: Proposals which apply reflective techniques to mobile agents, and systems which support resource control and may be used to implement mobile agent platforms. Even though resource control is beneficial for all kinds of programming environments, we focus on Java technology, because it is the common implementation language for mobile agent systems.

2.1 Reflection in Mobile Agent Environments

Some ideas about applying reflection in the context of mobile agents have been sketched by Ledoux et al. [13] and by Watanabe et al. [23].

Ledoux et al. suggest to use reflection in order to reason about and act upon the agent's *transfer mechanism*. They point out that reification of resources, as well as of relationships between agents and resources, is crucial for the realization of an open architecture. In other words, mobile agents should be aware of the underlying infrastructure. In their approach reflection is exploited to allow different granularities in the migration process, and therefore to elaborate a fine-tuned model for code mobility. Reflection is thus placed inside the execution environment and not at the level of mobile agents.

Watanabe et al. take a different approach, placing reflection at the mobile agent level. This approach focuses on a fault-handling mechanism by defining meta-agents that can customize the base-agents' *fault-handling strategies* (e.g., by introducing user-defined before- and after-fault handling methods at the meta-level of each agent).

Applying reflection to mobile code technology is therefore recognized as an interesting approach to improve openness. Regarding the combination of reflection and mobile code, there are two complementary issues:

- Tuning of internal aspects of the EE, such as mobility and communication, through reflection and reification of internals of the EE.
- Introducing new strategies to handle specific aspects of agent execution, through reflection and reification of agents.

The approach presented in this paper takes advantage of these two aspects: Reflection is used at both the *execution environment* and *mobile agent* levels. This requires a particular infrastructure, which we call a *reflective EE* (REE), supporting reification of internals of the EE, as well as reflection of the agents. The REE enables the execution of mobile agents as reflective entities that interact with the reified internals of the environment. The REE allows mobile agents

to influence their own execution or the execution of other agents, and to interact with internals of the REE.

Kava [24] is a 'reflective Java', which is based on load-time bytecode rewriting and supports the adaptation of applications. Kava has been used to specify and implement security policies for mobile code [25]. Reflection is used to insert security checks into the compiled application, avoiding the need to re-compile the application when different security checks are required. The security mechanism is implemented by *meta-objects*, i.e., special objects containing reflective information [15] that act as reference monitors and enforce security policies. Meta-objects are part of the trusted computing base and can be securely loaded from a remote source. Kava also supports some limited forms of resource accounting, it enforces a limit on the number of threads an application may create. A meta-object acts as a resource monitor and throws an exception, if the thread limit is exceeded.

2.2 Resource Control in Java Environments

JRes [10] is a resource control library for Java, which takes CPU, memory, and network resource consumption into account. Accounting for CPU relies on native code and on the underlying operating system[2]. Memory accounting in JRes is closely related to the reification of memory resources presented in this paper, although JRes still needs the support of a native method (to account for memory occupied by array objects). To achieve accounting of network bandwidth, the authors of JRes also resort to native code, since they swapped the standard `java.net` package with their own version of it. Consequently, JRes does not meet our requirements regarding portability.

KaffeOS [1] is a Java runtime system based on a modified JVM. It supports the operating system abstraction of *process* to isolate applications or mobile agents from each other, as if they were run on their own JVM. Thanks to KaffeOS, it is possible to achieve resource control with a higher precision than what is possible with our portable techniques for resource reification. The KaffeOS approach should result in better performance by design, but is however inherently non-portable.

NOMADS [17] is a mobile agent system which has the ability to control resources used by agents, including protection against denial-of-service attacks. The NOMADS execution environment is based on a modified JVM, the Aroma VM, a copy of which is instantiated for each agent. There is no resource control model or API in NOMADS; resources are managed manually (on a per-agent basis) and the resource related information is not accessible to agent. Since NOMADS is based on a modified JVM, its portability is limited.

[2] More precisely, CPU accounting in JRes is based on native threads, a feature not supported by every JVM.

3 Reflective Execution Environment (REE)

Reflection is the capability of a computation system to reason about and act upon itself and to adjust itself to changing conditions [15]. Reflective systems provide more openness than traditional systems, since they allow inspection and modification of internal functionalities. Reflective systems require the *reification* of some aspects of the base-level computational system. In other words, reification makes something accessible, which normally is not available in the programming model.

Mobile agents offer high flexibility for application deployment, dynamic application extensibility, and configurability. However, frequently the corresponding EE provides insufficient means for manipulating internal functionalities, and hence limits the resulting software adaptability. The application of reflection to mobile agents aims at providing enhanced adaptability and flexibility for the implementation of agent applications.

Fig. 1 shows how it is possible to enhance adaptability using reflection and mobile code. The application of mobile code usually improves adaptability w.r.t. a classical application, since parts of the service can be implemented as mobile components, which are pushed into the system dynamically. However, the adaptation of mobile components is normally performed based on external information requiring to stop the application, to modify the code, and to deploy it again (see (a) and (b)). The reflective approach enables mobile components to perform the adaptation exploiting internal information about the EE and their own execution (see (c)), without any external configuration.

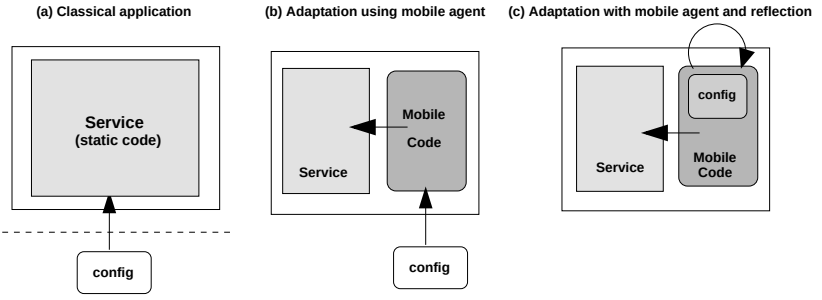

Fig. 1. Adaptation through mobile code and reflection.

Thus, the design of a REE for mobile agents requires the following considerations: **(1)** which aspects of the EE shall be reified, and **(2)** which parts of incoming mobile agents have to be reified. These considerations help to define the architecture of the different elements inside the EE and their dependencies. In the case of reification of mobile agents, the role of the reflective part of the agents is also established. Thus, a reflective mobile agent EE has to:

- Reify internal aspects of the EE and of agents;
- Allow the manipulation of the reified information;
- Allow separation of concerns of orthogonal aspect in mobile agents.

The architecture of such a REE consists of two levels: the **base-level** and the **meta-level**. The first level is composed of components that handle the execution of *base-level agents*, whereas the second level supports the execution of *meta-agents* and the manipulation of the reified information. The base-level can be seen as a conventional mobile agent EE without any support for reflection, which acts as a *black-box* (i.e., a closed environment that gives minimal and ad-hoc access to internal details). The meta-level includes components that are related to the reified aspects of both agents and the EE itself.

Fig. 2 illustrates the conceptual architecture of the REE. The different components in both base- and meta-level are described in the following subsections.

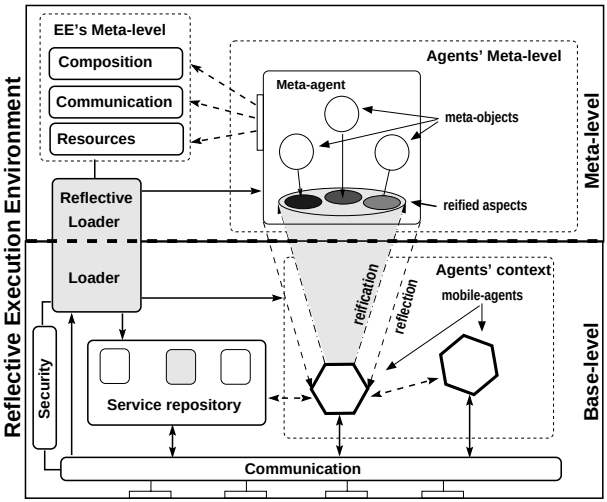

Fig. 2. The architecture of the Reflective Execution Environment.

3.1 Base-Level of the REE

The base-level provides the basic functionalities of the EE. It consists of the following elements:

Communication: The communication element handles communications between base-level agents and local services. It also supports communication with remote EEs or applications, i.e., it provides external connectivity.

Loader: This component is responsible for retrieving the code of mobile agents. The loader is connected to the external communication channel that receives agent code from remote EEs. The loader is logically divided into two parts: the **loader** at the base-level and the **reflective loader** at the meta-level. Both components are complementary, since the base-level loader is responsible for loading mobile agents and new services, while the reflective one performs the reification of the agent and associates the corresponding meta-agent (the reified information is made accessible through *meta-objects*). The combination of both loading components allows the creation of *reflective mobile agents*.

Agent context: This element is used to create the context in which the agent is executing and to trigger the agent's initialization. The context provides access to the different services available in the EE.

Service repository: Local services, which may be accessed by mobile agents running in the EE, are stored in the service repository. Agents may trigger the installation of new services, which are loaded dynamically. The loader handles the loading and linking of services.

Security: The security component has to mediate the external communication with remote EEs or applications, as well as to enforce security policies during the loading of agent code. For example, this component may ensure that agents are loaded from a trusted remote EE or application, and verify that agents do not refer to forbidden objects or services (such as internals of the EE).

3.2 Meta-level of the REE

At the meta-level internal aspects of the EE are reified, i.e., the EE exhibits internal components to be manipulated at the meta-level. The meta-level is populated by meta-agents, which form the reflective part of mobile agents executing in the base-level. From a logical point of view, the combination of a base-level agent and its associated meta-agent may be considered a **reflective mobile agent**, which is able to think and act upon itself, to access its internal representation (code and state), and to change its behavior. Meta-agents are located at the meta-level and manipulate the reified information (about the EE internals and about the base-level agent) in order to adapt the base-agent. Meta-agents are located in the **Agents' meta-level**, and the reified internals of the EE reside in the **EE's meta-level** (see Fig. 2).

Reification of EE Internals. The REE is a *white-box*, which allows its internals to be reified. This architecture is based on *multi-model reflection*, which allows the separation of concerns at the meta-level itself [16]. Considering the specificity of the mobile agent paradigm, three aspects have been identified that are necessary to provide openness in the REE: **composition**, **communication**, and **resources**:

Composition: The composition of different elements in the system is reified in order to expose the interconnections of base-level elements. For instance, this allows to inspect the binding of communication facilities to network interfaces, or to discover all services that employ the communication component.

Communication: The communications between different elements of the EE are reified in order to provide an entry point for the introduction of filters, to account for messages exchanged, or to redirect communication messages in the EE. Communication with local services are of particular interest and are closely related to the reification of agents.

Resources: Resources are reified in the REE. Physical resources (CPU, memory, network bandwidth, etc.) have abstract representations, since it is difficult to map low-level resources that are typically dependent on the underlying operating system. Logical resources (such as threads, agents, etc.) are also reified. The representation of reified logical resources allows to manipulate and to modify the way the EE allocates those resources (e.g., enforcing resource limits).

Similar architectures have been proposed in the context of reflective middleware [5,9], which provide hooks to add new behavior to the environment. In the case of mobile agents, this aspect is not necessary, since the agents themselves provide such functionality.

Reification of Mobile Agents. The REE allows mobile agents to be reified when they arrive. For each base-level agent, the REE associates a meta-agent, which is able to manipulate the reified information of the base-level agent. The reification of the agent is a process that takes an agent as input, reifies internal aspects of the agent, and binds the agent to a meta-agent. All these adaptations transform the agent into a new *reflective mobile agent* as shown in Fig. 3.

The meta-agents are compositions of several meta-objects that are related to the different reified aspects of the base-level agent. Similarly as for the EE, we have identified three aspects of the base-level agent that are reified: **structure**, **bindings**, and **resources**. The associated meta-objects play the role of micromanagers of the base-level agent. The separation of the meta-agent into domain-specific meta-objects allows to simplify the modification and composition of the different aspects handled at the meta-level.

Structural representation of the agent: The structural representation allows the meta-agent to adapt the base-level agent using the information about composition, communication, and resources, which is exposed by the REE. The reified agent structure allows to write special code to modify this structure. Structural reification involves a high-level representation of the agent code, which can be easily manipulated.

Bindings to services and other agents: On arrival, the base-level agent has unresolved references to other agents or services. By reifying these bindings, the meta-agent can adapt the interactions of the agent. The meta-agent may collect information about agent bindings and use it for optimizations, for accounting, to apply particular communication policies, and for debugging.

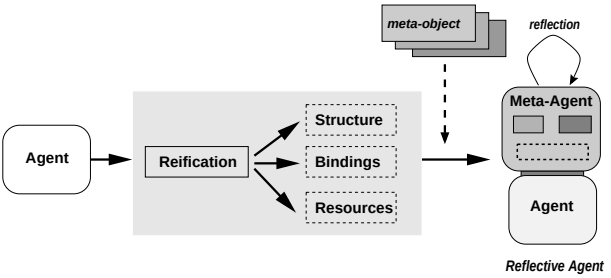

Fig. 3. Reification of mobile agents.

Resources consumed by the agent: The reification of the resources consumed by an agent exposes the agent's resource consumption to the meta-agent, which may use this information for monitoring, to enforce resource limits, or for billing purpose.

The reified aspects of the EE and of agents are closely related. However, the manipulations supported at the agents' meta-level are more flexible than those in the EE itself. The information associated with the reification in the EE's meta-level is useful for the adaptation in the agents' meta-level. Only in a restricted number of cases, modifications are performed in the EE's meta-level, because the components of the EE are rather static when compared to the dynamic nature of mobile agents.

The reification and adaptations applied in a REE are transparent to the agent programmer, who does not need to consider the non-functional aspects, which are incorporated just before the agent starts executing, i.e. at load-time.

One disadvantage of load-time reification is the overhead caused by the reification process. For the reification of structure and bindings, the overhead is rather small, because the necessary information can be obtained without complex processing. However, resource reification may cause considerable overhead. We have studied these aspects in a concrete implementation and evaluated the overhead of resource reification, focusing on CPU and memory. Our results, which are presented in section 5, show that sophisticated implementation techniques keep the overhead due to resource reification reasonably small.

4 Reification of Resources

For agent resource reification, we analyze and modify the agent code in order to extract information related to resource consumption and to insert the meta-objects that collect and maintain this information. Modification of source code is a common practice in some reflective systems, since it allows to manipulate and adapt applications [18]. In a mobile agent context *load-time transformation* based on the (compiled) agent code is better adapted, because it does not depend

on the source code of agents (which usually is not available) and enables the necessary modifications to be applied before an agent starts executing. The REE allows reification and adaptation directly on agent arrival, avoiding the need to implement different versions of the agent for distinct EEs. The reification process is performed using special adaptation code (*agent modifiers*) that can be dynamically inserted into the REE. Using mobile code for the reification process itself further increases the flexibility of the model. As illustrated in Fig. 4, it is possible that an agent visits EEs that do not support any adaptations.

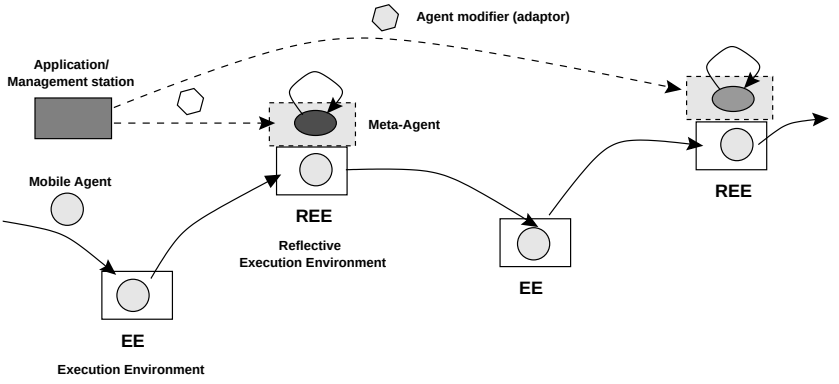

Fig. 4. Overview of adaptations in REEs.

To illustrate the reification of resources in the REE, let us consider Java as the target language, because it is the common implementation language for mobile agent systems. Moreover, Java offers many features that ease the implementation of our REE, such as e.g. object-orientation, language safety, multi-threading, a portable code format (bytecode), as well as the support for dynamic and customized class-loading, which enables sophisticated adaptation of agent classes before they are loaded and linked by the JVM.

Recently, we have observed an increasing interest in applying reflection to Java-based environments. Frameworks such as Javassist [8] allow structural reflection of Java programs. Javassist reifies the whole structural representation of an application directly from the bytecode and allows the modification of the application structure at load-time, hiding the low-level aspects of bytecode manipulation.

Our approach is based on rewriting of Java bytecode, because it enables a fully portable solution that does not rely on any low-level operating system functionalities. In the following subsections we briefly explain some basic ideas for resource reification in Java environments. More details can be found in [4].

Reification of Network Bandwidth. For the reification of network bandwidth, a straightforward solution consists in redirecting the calls to the component that provides the network service and associating a meta-object to this call (see Fig. 5). This is done by before/after processing of the method call and by trapping of well-known methods. The network resource is reified by an object that maintains information on consumption, thresholds, etc. This solution exposes e.g. the network traffic caused by an agent, and it allows to set limits or to add some filtering of network messages at the meta-level. The necessary modifications at the bytecode level are rather simple, because method invocations are explicit in the agent code.

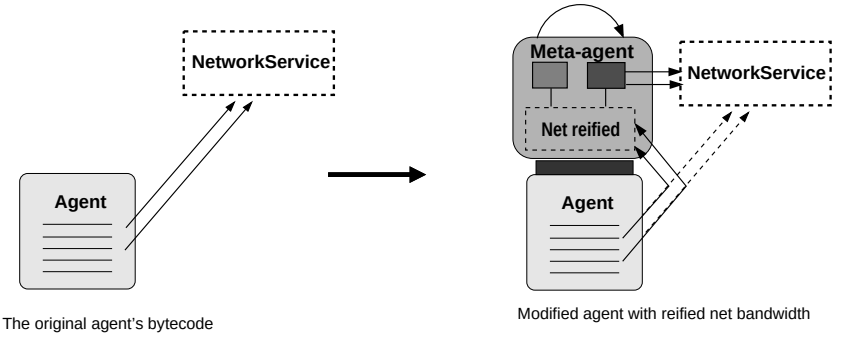

Fig. 5. Reification of network bandwidth.

Reification of Memory. Memory reification is more complex, since it requires accounting of all object allocations. The difficulty comes from the fact that it is no longer sufficient to redirect a call to a given service. We have to modify the way objects are allocated and deallocated in the EE. Another difficulty is to accurately estimate the size of allocated objects and the size of agent code. We have to calculate an estimation of the size of objects that the agent creates and to provide wrappers for the construction of these objects. This also requires forcing the agent to use the wrappers.

Our approach consist in dynamically adding our own version of object allocator to the agent code and replacing all object allocation instructions and invocations of constructors with our reified version (see Fig. 6). The memory reification process is supported by *agent modifier* code, which is retrieved from a remote node and performs the adaptation of agent. This allows to implement different allocation strategies without having to hard-code the actual reification in the EE.

Object deallocation is also difficult to account for when memory is garbage collected, as in Java, because there are no explicit application-level operations that could be easily tracked to this end. Details can be found in [4].

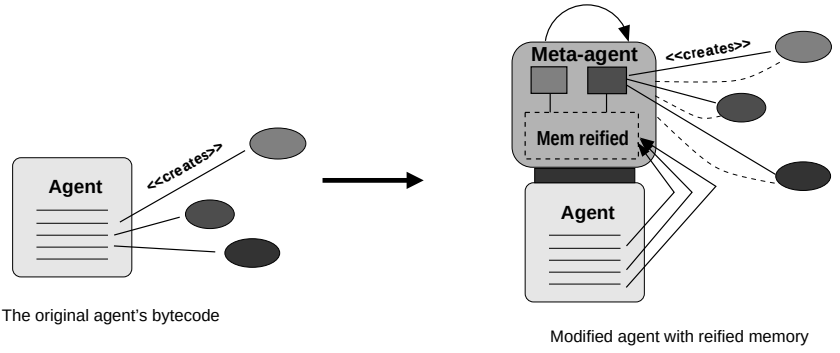

Fig. 6. Reification of memory.

Reification of CPU. CPU is probably the most difficult resource to be reified, because it is not explicitly 'visible' as a method invocation or an object allocation. Most operating systems provide some means for CPU control that may be exploited by an agent EE. However, such a solution limits portability and therefore should not be applied as a general-purpose solution in the context of a mobile agent system. Consequently, the REE provides some means in order to reify the CPU resource directly from the agents' code. We introduce an abstract unit of measurement based on the *number of bytecode instructions* executed by an agent. The reified CPU resource is associated with a meta-object that monitors the CPU consumption of all threads of an agent.

On a JVM using Just-in-Time compilation, this approach only gives an estimation of the actual CPU consumption. Nevertheless, an approximation is sufficient for many purposes, such as for the prevention of denial-of-service attacks. On a JVM implemented in hardware, like recently emerging Java processors that offer competitive performance and low power consumption, accounting the number of executed bytecode instructions gives a precise information on the actual CPU usage. Furthermore, as such processors will be integrated in mobile devices, where preservation of battery power is of paramount importance, the information on CPU consumption may be used to estimate and limit the power consumption of mobile agents.

In our approach the bytecode of an agent is analyzed in order to build control-flow graphs of the agent's methods (see Fig. 7). The resulting graphs are used to insert accounting instructions at strategic places into the bytecode. Thus, the reified accounting information is updated while the agent is executing. At the meta-level this information may be used to implement dedicated scheduling algorithms, which may e.g. reduce the priority of threads of an agent, if it exceeds its CPU limit.

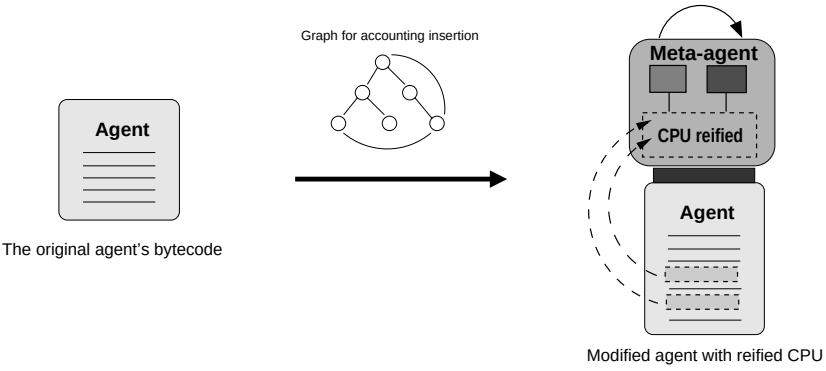

Fig. 7. Reification of CPU.

5 Evaluation

We have implemented a tool based on bytecode rewriting techniques, which transforms Java classes in order to reify resource consumption. While our transformations for memory accounting are related to techniques used by JRes [10], a similar approach for CPU accounting in Java has not been used before. Our current implementation supports off-line transformations of arbitrary Java classes (including JDK classes).

We are also integrating resource reification and appropriate control mechanisms into the J-SEAL2 mobile agent kernel [3], which requires load-time rewriting of mobile objects. J-SEAL2 is a secure mobile agent system implemented in pure Java, which supports the hierarchical process model of the Seal Calculus [22] that was first implemented by the JavaSeal mobile agent system [7]. Resource reification in J-SEAL2 concerns only memory and CPU resources, since the J-SEAL2 design already supports network accounting and the integrating of application-specific security policies.

In this section we present performance measurements showing that the overhead due to our completely portable implementation of CPU and memory accounting is acceptable on modern JVM implementations. We are not measuring the overhead incurred by the utilization of the reified information[3]. Our goal is to show that resource reification causes only moderate overhead and opens up interesting possibilities to improve flexibility and adaptability by allowing the application to use the reified information.

Our bytecode rewriting tool is based on BCEL (Byte Code Engineering Library) [11], which allows bytecode manipulations of Java classes and is also entirely written in Java. We chose BCEL since it is one of the most mature bytecode

[3] E.g. for CPU control, the overhead caused by a dedicated scheduler that uses the reified information can be kept small by choosing an appropriate time-slice.

instrumentation frameworks and provides a powerful and intuitive API that is well adapted for our requirements.

To show that our approach may be applied to complex Java applications (and also because there is a lack of standard benchmarks for mobile agent applications), we measured the standard SPEC JVM98 benchmarks [19] on a Linux platform (Intel Pentium III, 733MHz clock rate, 128MB RAM, Linux kernel 2.2.16) with IBM's JDK 1.3 implementation, which includes one of the best Just-in-Time compilers available today. We measured the overhead due to CPU and memory accounting in three different configurations[4]:

- Unmodified benchmarks.
- Rewritten benchmarks for CPU reification.
- Rewritten benchmarks for memory reification.

For each measurement, table 1 shows the execution times of the benchmarks in seconds (rounded to 3 decimal places), as well as the speedup of the original code compared to the rewritten version (rounded to 2 decimal places). In order to minimize the impact of compilation and garbage collection, all results represent the median of 101 different measurements. Furthermore, we also computed the geometric mean for each configuration. We rewrote about 520 Java class-files for the CPU and memory-aware versions of the SPEC JVM98 benchmarks.

Table 1. Benchmarks measuring the overhead of CPU and memory accounting (time in seconds).

Benchmark	Unmodified	CPU reified	Mem reified
_227_mtrt	5,823 (1,00)	7,336 (1,26)	6,898 (1,18)
_202_jess	7,779 (1,00)	9,145 (1,18)	8,608 (1,11)
_201_compress	19,130 (1,00)	23,156 (1,21)	19,500 (1,02)
_209_db	26,740 (1,00)	27,777 (1,04)	27,031 (1,01)
_222_mpegaudio	8,694 (1,00)	12,425 (1,43)	10,358 (1,19)
_228_jack	8,184 (1,00)	8,771 (1,07)	9,226 (1,13)
_213_javac	14,150 (1,00)	15,618 (1,10)	16,016 (1,13)
Geometric Mean	11,286 (1,00)	13,296 (1,18)	12,508 (1,11)

The results in table 1 show that the overhead due to CPU accounting is about 20%, while in the case of memory accounting the observed overhead is only about 10%. Note that we did not apply any optimizations to reduce the accounting overhead. Simple optimization rules, as discussed in [4], can help to reduce the overhead significantly. The implementation of the optimization algorithm is still in progress.

[4] The JDK was not rewritten for the measurements presented in this paper. See [4] for an evaluation of the performance impact of JDK rewriting.

6 Conclusion

In this paper we have presented the architecture of a reflective mobile agent EE supporting the reification of agents and of internal aspects of the EE itself. The reflective EE allows to manipulate information on resource consumption. We suggest an implementation scheme for Java, which is entirely portable and entails only moderate overhead. Moreover, our approach is not restricted to mobile agent applications, but opens up the perspective of building portable resource management policies as dynamic add-on modules for commercial off-the-shelf components.

Acknowledgments. Many thanks to Jarle Hulaas and Klaus Rapf for their valuable comments, and to Rory Vidal for his work on CPU reification. This work was financed by the Swiss National Foundation, grants 5003-057753 and 20-54014.98, and by CoCo Software Engineering GmbH.

References

[1] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego, CA, USA, Oct. 2000.

[2] J. Baumann. *Control Algorithms for Mobile Agents.* PhD thesis, University of Stuttgart, Germany, 1999.

[3] W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, Jan. 2001.

[4] W. Binder, J. Hulaas, A. Villazón, and R. Vidal. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, Tampa Bay, Florida, USA, Oct. 2001.

[5] G. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In N. Davies, K. Raymond, and J. Seitz, editors, *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lake District, UK, 1998. Springer-Verlag.

[6] J. Bredin, D. Kotz, and D. Rus. Market-based resource control for mobile agents. In *Second International Conference on Autonomous Agents*, Minneapolis,USA, May 1998.

[7] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, Oct. 1999.

[8] S. Chiba. Load-time structural reflection in Java. In *ECOOP*, pages 313–336, 2000.

[9] M. F. Costa, H. Duran, N. Parlavantzas, K. Saikoski, G. Blair, and G. Coulson. The Role of Reflective Middleware in Supporing the Engineering of Complex Applications. In W. Cazzola, R. Stroud, and F. Tisato, editors, *OOPSLA '99, Workshop on Object-Oriented Reflection and Software Engineering*, Denver, Colorado, USA, Nov. 1999.

[10] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *OOPSLA-98*, pages 21–35, New York, USA, Oct. 18–22 1998. ACM Press.

[11] M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. `http://bcel.sourceforge.net/`.

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

[13] T. Ledoux and N. Bouraqadi-Saâdani. Adaptability in mobile agent systems using reflection. In *ECOOP 2000, Workshop on Reflection and Metalevel Architectures*, Cannes, France, 2000.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[15] P. Maes. Concepts and experiments in computation reflection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, Orlando, Florida, USA, Oct. 1987.

[16] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A distributed programming sysyem with multi-model reflection framework. In *Workshop on New Models for Software Architecture*, Nov. 1992.

[17] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith. NOMADS: toward a strong and safe mobile agent system. In C. Sierra, G. Maria, and J. S. Rosenschein, editors, *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00)*, pages 163–164, NY, June 3–7 2000. ACM Press.

[18] M. Tatsubori. An Extension Mechanism for the Java Language. Master's thesis, Graduate School of Engineering, University of Tsukuba, Ibaraki, Japan, Feb. 1999.

[19] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`, 1998.

[20] C. F. Tschudin. Funny money arbitrage for mobile code. In *Proceedings of the Second Dartmouth Workshop on Transportable Agents*, Sept. 1997.

[21] C. F. Tschudin. Open resource allocation for mobile code. In *Proceedings of The First Workshop on Mobile Agents*, Berlin, Germany, Apr. 1997.

[22] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.

[23] T. Watanabe, A. Noriki, and K. Shinbori. A Reflective Framework for Reliable Mobile Agent. In *ECOOP 2000, Workshop on Reflection and Metalevel Architectures*, Cannes, France, 2000.

[24] I. Welch and R. Stroud. Kava - A Reflective Java Based on Bytecode Rewriting. In W. Cazzola, R. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, LNCS, pages 155–167. Springer Verlag, 2000.

[25] I. Welch and R. J. Stroud. Using reflection as a mechanism for enforcing security policies in mobile code. In *ESORICS*, pages 309–323, 2000.