Resource Management for Safe Languages

Grzegorz Czajkowski[1] and Jan Vitek[2]

[1] Sun Microsystems Laboratories,
grzegorz.czajkowski@sun.com
[2] Purdue University,
jv@cs.purdue.edu

Abstract. Safe programming languages offer safety and security features making them attractive for developing extensible environments on a wide variety of platforms, ranging from large servers all the way down to hand-held devices. Extensible environments facilitate dynamic hosting of a variety of potentially untrusted codes. This requires mechanisms to guarantee isolation among hosted applications and to control their usage of resources. While most safe languages provide certain isolation properties, typically resource management is difficult with the current standard APIs and existing virtual machines.
This one-day workshop brought together practitioners and researchers working on various approaches to these problems to share ideas and experience.

1 Workshop Overview

The workshop consisted of four 90-minute sessions. In the first one Doug Lea from State University of New York in Oswego delivered an invited talk on the Application Isolation API proposed as an extension to the Java^{TM} programming language [1]. Presentations of accepted position papers were given in the next two sessions. Each author had 7 minutes to present the main idea of his/her work. After all of the authors in a given session finished, the presentations were discussed - this include the time for questions about specific presentations as well as general remarks and brain-storming.

The last session was a panel discussion, during which the workshop attendants listed a list of open or "really difficult" issues in the discussed domain.

The total of nine presentations were accepted for the workshop, and each of them was presented. About 25 people attended the workshop, the majority from Europe, with a few attendees from the US. Most of the participants were from the academia.

2 Position Paper Summaries

All the papers accepted and presented are available from `http://www.ovmj.org/workshops/resman`. The section below contains summaries of the papers, provided by the authors.

J. Hernández and A. Moreira (Eds.): ECOOP 2002 Workshops, LNCS 2548, pp. 1–14, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2.1 Creating a Resource-aware JDK

Authors Walter Binder (CoCo Software Engineering GmbH, Vienna, Austria), Vladimir Calderon (University of Geneva, Switzerland)
Contact e-mail w.binder@cocosoftware.com

Accounting and limiting the resource consumption of applications is a prerequisite to prevent denial-of-service attacks in mobile code environments. Moreover, it enables the monitoring and profiling of applications, which may be charged for their utilization of computing resources. Java has become the de facto standard for the implementation of mobile code environments. However, current Java runtime systems do not offer any mechanisms for resource accounting and resource control.

Prevailing approaches to provide resource control in Java-based platforms rely on a modified Java Virtual Machine (JVM), on native code libraries, or on program transformations. The Java resource accounting facility J-RAF is based completely on program transformations. In this approach the bytecode of applications is rewritten in order to expose its CPU and memory consumption (CPU and memory reification). Programs rewritten by J-RAF keep track of the number of executed bytecode instructions (CPU accounting) and update a memory account when objects are allocated or reclaimed by the garbage collector.

Resource control with the aid of program transformations offers an important advantage over the other approaches, because it is independent of a particular JVM and underlying operating system. It works with standard Java runtime systems and may be integrated into existing mobile code environments. Furthermore, this approach enables resource control within embedded systems based on modern Java processors, which provide a JVM implemented in hardware that cannot be easily modified.

Typically, rewriting the bytecode of an application is not sufficient to account and control its resource consumption, because Java applications use the comprehensive APIs of the Java Development Kit (JDK). Therefore, resource-aware versions of the JDK classes are necessary in order to monitor the total resource consumption of an application. Ideally, the same bytecode rewriting algorithm should be used to rewrite application classes as well as JDK classes. However, the JDK classes are tightly interwoven with native code of the Java runtime system, which causes subtle complications for the rewriting of JDK classes.

The authors outline the difficulties of modifying the JDK classes: The native code of the Java runtime system relies on several low-level assumptions regarding the dependencies of Java methods in certain JDK classes. Thus, program transformations that are correct for pure Java may break native code in the runtime system. Unfortunately, these dependencies are not well documented, which complicates the task of defining transformation rules that work with the Java class library. Moreover, the transformed JDK classes may seem to work as desired even with large-scale benchmarks, while the transformation may have compromised the security model of Java. Such security malfunctions are hard to detect, as they cannot be perceived when running well behaving applications.

We have experienced that a minor restructuring of the method call sequence completely breaks several security checks, which are based on stack introspection and assume a fixed call sequence. Consequently, modifications and updates of the JDK are highly error-prone.

Having explained the difficulties of rewriting the JDK classes, the authors present and compare different bytecode transformation schemes that allow to create resource-aware JDK classes. The distinct rewriting strategies are evaluated based on standard benchmark suites for Java.

2.2 Scoped Memory

Authors Greg Bollella (Sun Mircosystems, Inc.), Kirk Reinholtz (NASA Jet Propulsion Laboratories)
Contact e-mail greg.bollella@sun.com

The full text of this short position paper is available on the workshop's Web site.

2.3 Resource Accounting in a J2ME Environment

Authors Walter Binder, Balazs Lichtl (both from CoCo Software Engineering GmbH, Vienna, Austria)
Contact e-mail w.binder@cocosoftware.com

Nowadays Java has become a common implementation language for applications which have to be executed in different hardware environments without changes. With the specification of the Java 2 Micro Edition (J2ME), Sun has created a standard for embedded applications developed in Java.

Resource accounting and control is getting increasingly important in various settings. One particularly important case for resource control is the protection of platforms executing dynamically uploaded applications (mobile code) from faulty or malicious code, which may overuse the computing resources of the system. Another interesting setting is the charging of applications for their resource consumption.

Unfortunately, current standard Java runtime systems do not support resource control. Hence, until resource control may become part of future releases of the Java Development Kit, resource control in Java environments has to be based either on modified JVMs or on program transformation techniques. So far, modified JVMs have not been deployed widely, because typically they suffer from limited portability and from low performance, since usually they do not support a state-of-the-art JIT compiler as provided by standard Java runtime systems. Moreover, there are embedded systems based on recent Java processors, which provide JVMs implemented in hardware that cannot be easily modified to enable resource control. Thus, program transformation techniques are an attractive alternative to modifying the JVM.

Ideally, program transformations for resource control shall be compatible with existing Java runtime systems, shall cause only moderate overhead, and

allow accurate accounting. While the accounting accuracy on a Java 2 Standard Edition JVM is limited, because of the significant part of native code that is not accounted for and due to optimizations performed by the JIT compiler, the accuracy on a Java processor can be much better, as the execution time of individual bytecode instructions can be measured and only very simple and well documented optimizations are performed, such as the combination of certain JVM instructions. However, regarding the accounting overhead, sophisticated optimizations can be beneficial, and consequently the relative overhead on an embedded Java processor may be significantly higher than on a JVM with a modern JIT compiler.

The authors report their initial results from applying program transformations for resource accounting, which are described in an accompanying work, to embedded applications running on a Java processor. The authors created a benchmark suite for the J2ME environment and measured the overhead of CPU accounting with different rewriting schemes and optimizations.

2.4 JRAF - The Java Resource Accounting Facility

Authors Vladimir Calderon (Computer Science Department, University of Geneva, Switzerland), and Walter Binder (CoCo Software Engineering GmbH, Vienna, Austria)
Contact e-mail caldero6@cuimail.unige.ch

Program transformations are a portable approach to extend existing Java environments with resource management functionality [4, 3]. Using such techniques, the bytecode of Java applications is modified to expose its resource consumption (resource reification through bytecode rewriting).

In this paper we outline the concept and structure of JRAF[1], the Java Resource Accounting Facility, which comprises a series of tools for resource reification. These tools offer CPU and memory accounting with pluggable accounting strategies, bytecode optimizations, calibration mechanisms to fine-tune the accounting schemes and the accuracy of accounting to particular execution environments, as well as higher level components that make use of the collected accounting information. To manage such a huge set of tools, JRAF provides a powerful and flexible configuration mechanism and controls the application and proper composition of the separate resource management tools. It has a layered architecture offering an accounting and a resource control interface.

JRAF Components

Accounting Interface The accounting interface of JRAF manages all resource reification tools, as well as the structure of the corresponding accounts. Currently, JRAF supports interfaces for CPU and memory accounting, but it can

[1] `http://abone.unige.ch/`

be easily extended for accounting of additional resources. The accounting interface of JRAF consists of two parts:

- Interfaces for the resource reification tools.
- Interfaces defining the accounts.

Analyzer Tool This is a very important JRAF component. In fact, the analyzer is an essential tool used by other accounting tools. Its main purpose is to ease the comprehension of the deep characteristics of the input classes to be reified, crucial for the rewriting process.

CPU Reification Tool In order to show a more concrete implementation of the accounting interface we explain the interface to the CPU tool in more detail, which supports specific features: an interface for optimization algorithms that help to reduce the overhead of CPU accounting (see [2] for examples of such optimizations); an interface allowing to associate different accounting weights with JVM instructions and common initializations for CPU tools.

Memory Reification Tool Another very important resource is the memory. A memory reification tools was already implemented [5]), we needed then to make this tool compliant to JRAF. This tool is now fully integrated to JRAF, similarly to the CPU tool, and can be used within a single JRAF reification process along with the other tools.

Resource Control Interface The resource control interface comprises all tools using the accounts, such as resource monitors, schedulers, logging components, etc. JRAF aims at joining the accounting with these components. It allows to plug resource control tools to an application through the accounting interface.

JRAF in Action We have successfully applied JRAF on the Java 2D Demo[2] to demonstrate the application of JRAF on a general multithreaded application. In this example, JRAF reifies the CPU consumption of the individual threads and displays this information in an extra window.

The reader may want to apply JRAF to his own Java applications using the JRAF demo available at URL `http://abone.unige.ch/`, which currently is configured to reify arbitrary applications using one of our CPU tools and to attach a simple thread monitor.

Conclusion JRAF has a layered architecture, it is extensible and provides an XML-based configuration mechanism. JRAF manages, coordinates, and combines the application of various bytecode rewriting tools for resource accounting. It has become a general tool to add different resource management strategies to arbitrary Java applications. Currently, JRAF includes three different CPU reification tools with various optimizations, one memory reification tool, as well as a graphical monitor and scheduler.

[2] The Java2D Demo is available at `http://java.sun.com/products/java-media/2D/samples/index.html`.

2.5 Resource Consumption Interfaces for Java Application Programming - A Proposal

Authors Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper (all from Sun Microsystems, Inc.)
Contact e-mail grzegorz.czajkowski@sun.com

Software systems in many circumstances need an awareness of their resource impact on the underlying executing platform, so they can satisfy externally imposed performance requirements. Programs constituting such systems need the ability to control their consumption of resources provided by the platform. This document summarizes the current status of a proposal being prepared to define an extensible, flexible, and widely applicable resource consumption API for the Java platform. Control over resource consumption is developed using a set of abstractions that allow the statement of reservations and constraints for individual resources utilized by the executing application.

The Java programming language and its associated runtime environment have grown beyond their initial goal of writing portable applications. The advent of the Web, applications servers, and the enterprise and micro editions of the Java platform has created pressure to make more system programming features available to programmers, as they develop progressively more sophisticated applications in an increasingly wide range of environments. This document addresses one such need: the ability to monitor or control the consumption of resources by an application. The proposed resource consumption interface (RC API) controls resources available to collections of isolates and, as such, depends on the availability of the isolate abstraction. (The isolate abstraction is provided by Java Specification Request (JSR) 121 [1]. However, the RC API is designed so that new methods need not be added to the Isolate API.

The general goals for the resource consumption API are as follows:

1. Extensibility: The resource consumption interface should support the addition of new controlled resources.
2. Cross-platform applicability: The interface, as well as its underlying abstractions, should be applicable to all kinds of Java platforms.
3. Cross-scenario applicability. The interface should support the different forms of application management supported by the Isolate API, as well as being meaningful in a single application context.
4. Flexibility. The interface should be able to describe and control a broad range of resource types.
5. Completeness of abstraction. The interface should hide from applications whether a given resource is managed by the Java Virtual Machine (JVM^{TM}), by a core library, or by an underlying operating system facility.
6. Lack of specialization. The interface should not require an implementation to depend on specialized support from an operating system or from hardware, although implementations may take advantage of it if available.

The proposal examines the resource consumption interfaces from the perspectives of three classes of developers, each of whom is a participant in resource

management: the application developer, the middleware developer, and the runtime environment developer.

2.6 Towards Resource Consumption Accounting and Control in Java: A Practical Experience

Authors Frederic Guidec, Nicolas Le Sommer (both from VALORIA Laboratory, University of South Brittany, France)
Contact e-mail Frederic.Guidec@univ-ubs.fr

All software components are not equivalent as far as resource access and consumption are concerned. Some components can do very well with sparse or even missing resources, while others require guaranteed access to the resources they need. In order to deal with non-functional requirements pertaining to resource utilization we propose a contractual approach of resource management and access control. This idea is being investigated in the context of project RASC[3] *(Resource-Aware Software Components)*. In this project our objective is to provide software components with means to specify their requirements regarding hardware and/or software resources, and to design methods and models for utilizing this kind of information at any stage of a component's life-cycle.

The remaining of this paper gives an overview of two software products whose development is in progress in the context of project RASC.

Raje: A Resource-aware Java Environment RAJE can be perceived as an extension of the traditional JRE (Java Runtime Environment). This extension provides means to monitor resource access and consumption at middleware level. It makes it possible to monitor the usage of "global" resources (CPU, system memory and swap, network interfaces, etc.) as well as that of the "conceptual" resources used by Java programs (TCP and UDP sockets, CPU time and memory consumed by each Java thread, etc.). In RAJE all resources are modeled as first-class Java objects. Information about any kind of resource can thus be gathered by calling appropriate methods on the Java object modeling this resource.

Since resource objects can be created and destroyed dynamically by Java programs, RAJE implements a resource register, whose role is to identify and to keep track of resource objects at runtime. By consulting the resource register a program can locate all the objects that model resources in its name space.

Resource monitoring can be performed in either a synchronous or asynchronous way. Resource monitoring is said to be achieved synchronously when any attempt to access a given resource can be intercepted and checked immediately. Monitoring a resource asynchronously consists in consulting the state of this resource explicitly every now and then, in such a way that the time of the observation does not necessarily coincide with the time of an attempt to use the resource. Both monitoring models are indeed complementary models. In any case RAJE provides facilities for performing both kinds of monitoring. For

[3] `http://www.univ-ubs.fr/valoria/Orcade/RASC`

the programmer, deciding which model should be applied mostly comes down to making a tradeoff between the precision and the cost of monitoring.

Most of the code included in RAJE is pure Java and, as such, is readily portable. However, part of this code consists of C functions that permit the extraction of information from the underlying OS, and the interaction with inner parts of the JVM (Java Virtual Machine). To date RAJE is implemented under Linux, and the JVM it relies on is a variant of TransVirtual Technology's Kaffe 1.0.6.

Jamus: Java Accommodation of Mobile Untrusted Software JAMUS is an experimental platform we develop on top of RAJE in order to experiment with the idea of resource contracting. JAMUS supports the deployment of "untrusted" software components, provided that these components can specify their requirements regarding resource utilization in both qualitative (e.g., access rights to parts of the file system) and quantitative terms (eg read and write quotas). Emphasis is put on providing a safe and guaranteed runtime environment for such components.

Resource control in JAMUS is based on a contractual approach. Whenever a software component applies for being deployed on the platform, it must specify explicitly what resources it will need at runtime, and in what conditions. The main originality of this approach lies in the fact that a specific contract must be subscribed between the JAMUS platform and each software component it accommodates. By specifying its requirements regarding resource access and consumption, the candidate component requests a specific service from the JAMUS platform. At the same time it promises to use no other resource than those mentioned explicitly. In return, whenever the platform accepts a candidate component, it promises to provide this component with all the resources it requires. At the same time it reserves the right to sanction any component that would try to access other resources than those it required.

Based on this notion of resource contracting, JAMUS can provide some level of quality of service regarding resource availability. It also provides components with a relatively safe runtime environment, since no component can access or monopolize resources to the detriment of other components.

Further Readings The development of both RAJE and JAMUS is still in progress. Interested readers can refer to [11, 12] for a more detailed description of this work. Up-to-date information about this development (and about other topics addressed in project RASC) can also be found at http://www.univ-ubs.fr/valoria/-Orcade/RASC.

2.7 Safe Mobile Code - CPPCC: Certified Proved-Property-Carrying Code

Author Zoltan Horvath and Tamas Kozsik (both from Department of General Computer Science, University Eotvos, Lorand, Budapest, Hungary)
Contact e-mail hz@inf.elte.hu

Some systems (e.g. Java virtual machines) make it possible for an application to download a component over the network, link it dynamically to the application and execute it. In such cases a safety-critical application may want to get guarantees that the downloaded component does no harm: does not use too much resources, does not read or modify data unauthorised, etc.

The widely-used way to avoid malicious code is that the application accepts code only from trusted partners who guarantee the expected behaviour. The identity of the partner can be proved by e.g. public-key cryptography systems or certificates. But still, there is a danger that the trusted partner makes a mistake, sends a wrong or an outdated component, or one with a programming error in it.

The class file verifier mechanism of Java, for example, can prove static and structural constraints and type correctness when linking the downloaded component to the application. These checks ensure that the code is well-formed, and the applied data-flow analysis guarantees that there will be no operand stack overflow or underflow, that local variable uses and stores are valid, and that the arguments of the instructions are of valid types.

Proof-carrying code is a technique for providing further assurances. With PCC, the code consumer specifies safety requirements, which tell under what conditions parts of memory can be read or written or how much of a resource is allowed to use. The code producer must provide a set of program properties and encoded proofs packed together with the code transmitted, to certify that the code really satisfies the requirements. Complex methods may be used to construct the proofs, but their verification should be a simple computation.

It would be very fruitful to include further semantical information to the code compared to the above approaches. Based on such information the code consumer could decide more precisely to which extent the obtained code is appropriate for solving a certain specified problem. For example the consumer could expect a piece of code that can sort a list of integer numbers. Type requirements for such a code would be: take a list of integers and produce a list of integers. These requirements can be type-checked. Structural constraints can prescribe that the sorting operation does not perform illegal memory access – the aforementioned Java class file verifier can make such a guarantee. On the other hand, the following additional semantic requirement would also be necessary: the result of the operation is such that every element of the list is either the first element or an element that is greater than or equal to the previous element. A safety-critical application must be sure at least that the downloaded code, if it terminates, satisfies this requirement. (A time-critical application would also set up a requirement on termination.)

As the requirements made for the downloaded components are getting more and more complex, the proofs of correctness get longer and harder to verify. Hence efficiency comes to the front. The goal is to reduce the memory and time consumption of the technology for communicating safe code. In the solution we suggest for this problem there are three main parties involved. Here is a summary of their tasks.

1. The code producer packages the machine code with the original source code, type information, some important properties of the code, and the proof of these properties designed by the help of a theorem prover.
2. The certifier checks the proofs in respect to the source code and the properties, then it checks whether the machine code is really the result of a proper compilation of the source code, and finally it checks whether the type information is correct. If every check succeeds, then the machine code, the type information, the properties and a certificate will be packaged and sent back to the code producer, which can place it in a library for later use.
3. If an application needs a component available at a mobile code producer, it can request and receive a certified proved-properties-carrying code, which can be dynamically linked into the application – of course, after the necessary safety checks. First the certificate is verified, then the type information and the carried properties are compared to the type and semantic requirements specified for the dynamically loaded component.

A prototype system to illustrate the ideas above is being assembled. Its already developed components are based on the "dynamic" construct of the Concurrent Clean language and a proof tool specifically designed for Concurrent Clean.

2.8 Resource Control in Component-Based Systems

Author Jarle Hulaas (Computer Science Department, University of Geneva, Switzerland), Walter Binder (CoCo Software Engineering GmbH, Vienna, Austria)
Contact e-mail Jarle.Hulaas@cui.unige.ch

In the approach followed in this position paper, we address various applications of resource control (RC), like security, Quality-of-Service, and billing, with an emphasis on the prevention of Denial-of-Service (DoS) attacks. We assume a multi-threaded component model with resource limits enforced by the (Java-based) kernel at the level of individual components, in order to confine abnormal behaviour. Using a general communication facility (e.g., method invocation, message passing, tuple space, etc.), a component C (client, caller) may request a service from another component S (service, callee). The problem addressed here is: *Which component shall be charged for the resources consumed by S while executing a request on behalf of C, even when S and C do not trust each other, and how can the communication model be kept simple while still allowing resource allocation to be managed efficiently at the application level ?*

As already noted by other researchers (e.g. [9]), the range of possible interaction patterns between C and S is wide: anonymous, asynchronous, callbacks or one-to-many service invocations should be supported. We show that the most secure and comprehensive solution is to resort to an abstraction called *resource container* for explicitly transmitting resources between donators and consumers.

Components may then freely decide when to switch from one available resource container to another.

Resource exhaustion may either stem from malicious or accidental resource over-use or from intentional resource revocation. Such an event, when occuring in S, must be signalled to C even when executing asynchronous invocations. S must also be able to ensure its own consistency and to terminate properly the request being serviced. To this end, a callback-based notification mechanism is needed. The API described in Section 2.5 is a good fit. We propose additionally a means for identifying the specific invocation where the problem occurred to facilitate the work of the callback; it is indeed a very delicate task for a notification routine to execute properly in presence of violated resource constraints.

Finally, we notice that non-intrusive monitoring of resource consumption is a valuable facility that is not supported by other approaches we are aware of.

2.9 Distributed and Multi-type Resource Management

Authors Luc Moreau (Department of Electronics and Computer Science, University of Southampton, UK) and Christian Queinnec (LIP6, Paris, France)
Contact e-mail L.Moreau@ecs.soton.ac.uk

Dynamic code loading has popularized the idea of Internet servers able to reconfigure themselves and to extend their capabilities by uploading code dynamically — examples of such systems can be found in the mobile agent literature. The full power of this paradigm shift can be achieved if untrusted code can be run in a safe manner, and in particular if malicious code can be prevented from using too many resources. This raises the problem of resource management, both for the provider and the consumer of resources.

In previous work [7, 8], the authors introduced *Quantum*, a framework generalizing Kornfeld and Hewitt's group hierarchy [6] and providing a programmatic interface for managing resources in a distributed setting. Quantum is based on the notion of *energy*, an abstract notion denoting a quantity of resources, and on *groups* acting as tanks of energy. Groups are organized along a hierarchical structure. Groups *sponsor* computations that consume energy from the group they are directly sponsored by. Two forms of notification are supported: *exhaustion* of the energy contained in a group and *termination* of the computation sponsored by a group. Additionally, Quantum provides a mechanism for pausing and resuming hierarchies of computations. Notifications are made available to the programmer and therefore can be arbitrary computations, whose resources must also be managed: Quantum specifies how such notifications can be integrated in a single framework. Our previous work focused on its formalization [7] and its implementation in a shared memory [8].

Our present contribution is twofold:

1. the introduction of two different primitives related to distribution — migration and communications — and their semantics in terms of groups.
2. the support for multiple types of resources.

Distributed Resource Management As far as distribution is concerned, we distinguish the transfer of data between hosts from the transfer of groups between hosts. The former can easily be expressed by send and receive primitives 'à la' π-calculus. The latter is reminiscent of remote procedure calls and migration of mobile agents.

We introduce the primitive $migrate(h, f)$, which requires two arguments: h a host name and f a procedure without argument (a thunk). The effect of the *migrate* primitive is displayed in Figure 1. The energy (less the cost of migration) contained in the group that sponsors the *migrate* primitive is transferred to a newly created *remote group*. We require *migrate* to be executed in a group that sponsors *only one* thread.

When a remote group detects the termination of the computation it sponsors, its energy (less the cost of the return) is transferred back to its parent group. Exhaustion in the remote group triggers an exhaustion notification in the parent group; in turn, the latter notification triggers an exhaustion notification in its parent group, which may be able to transfer energy through the use of the *awake* primitive, according to the handler programmed by the user.

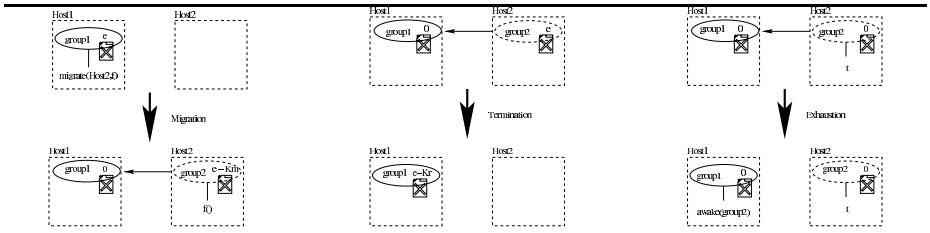

Fig. 1. Migration, Return from Migration: (a) Termination — (b) Exhaustion

Management of Multiple Resources While our previous work mainly focused on processor time, our set of primitives is able to address other kinds of energies. From an energy system implementor's viewpoint, there are only three primitive operations that deal with energy tanks. These operations specify how energy is

1. merged when a subgroup terminates and gives its energy back to its parent,
2. consumed while a group performs some work (a descriptor tells how much is consumed) and
3. split between the creating and created groups (a descriptor details the split).

This model allows the user to create his own types of energies and have the same machinery take care of these energies.

3 Final Session

In the final session a list of open or not yet satisfactorily solved problems related to resource management for safe languages was created. We have not found answers to them but the list itself is quite interesting. It is reproduced here in more or less verbatim form:

Languages vs operating systems?
The entity: thread, isolate, process?
Flexible consumer groupings/hierarchies
Economics/trading (trade disk space for network bandwidth, etc.)
Minimal resources, how to find granularity
Granularity/units
Observing vs controlling
Primitive/basic/minimal complete set of resources
Higher level abstractions vs low-level APIs
Different views of a resource
Synchrony vs Asynchrony
Shared OS resource overhead
Revocation/dynamic behavior
Sharing and termination
Security/compatibility with access control
Resource-aware applications/cost of being resource aware
Enforcing cooperation
Reservations/price prediction
Accuracy vs efficiency
Platform independence
Who's to blame/charge
Real time issues
Reclamation-when? correct? delay
Resource policies
Scalability
Transactional behavior
Extensibility vs Portability vs Efficiency
Simplicity
Useful and complete minimum set of resources
Taxonomy: renewable vs revocable

4 Workshop Conclusions

The participant concluded that it was a very useful meeting, although still there are more questions than answers. Perhaps a follow-up workshop will be organized in conjunction with ECOOP'03, and hopefully some of the issues will have been addressed by then. Certainly, this area does not suffer from the lack of interest!

References

[1] Java Community Process. JSR-121: Application Isolation API Specification. http://jcp.org/jsr/detail/121.jsp.

[2] Walter Binder, Jarle Hulaas, and Alex Villazón. Resource control in J-SEAL2. Technical Report Cahier du CUI No. 124, University of Geneva, October 2000. `ftp://cui.unige.ch/pub/tios/papers/TR-124-2000.pdf`.

[3] Walter Binder, Jarle Hulaas, Alex Villazón, and Rory Vidal. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2001)*, Tampa Bay, Florida, USA, October 2001.

[4] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 21–35, New York, USA, October 18–22 1998. ACM Press.

[5] Alex Villazón and Walter Binder. Portable resource reification in Java-based mobile agent systems. In *Fifth IEEE International Conference on Mobile Agents (MA-2001)*, Atlanta, Georgia, USA, December 2001.

[6] William A. Kornfeld and Carl E. Hewitt. The Scientific Community Metaphor. *IEEE Trans. on Systems, Man, and Cybernetics*, pages 24–33, January 1981.

[7] Luc Moreau and Christian Queinnec. Design and Semantics of Quantum: a Language to Control Resource Consumption in Distributed Computing. In *Usenix Conference on Domain-Specific Languages (DSL'97)*, pages 183–197, Santa-Barbara, California, October 1997.

[8] Luc Moreau and Christian Queinnec. Distributed Computations Driven by Resource Consumption. In *IEEE International Conference on Computer Languages (ICCL'98)*, pages 68–77, Chicago, Illinois, May 1998.

[9] G. Banga, P. Druschel, and J. Mogul. Resource containers: a new facility for resource management in server systems. In *In Proceedings of the 3rd USENIX Symposium on Operating system design and implementation*, Feb. 1999.

[10] Walter Binder. J-SEAL2 – A secure high-performance mobile agent system. In *IAT'99 Workshop on Agents in Electronic Commerce*, Hong Kong, December 1999.

[11] Nicolas Le Sommer and Frederic Guidec. A Contract-Based Approach of Resource-Constrained Software Deployment. In J. van Leeuwen G. Goos, J. Hartmanis, editor, *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment (CD'2002, Berlin, Germany)*, number 2370 in Lecture Notes in Computer Science, pages 15–30. Springer, June 2002.

[12] Nicolas Le Sommer and Frederic Guidec. JAMUS: Java Accommodation of Mobile Untrusted Software. In *4th EurOpen/USENIX Conference (NordU'2002, Helsinki, Finland)*, February 2002. http://www.univ-ubs.fr/valoria/Orcade/RASC/Publications/NordU2002.pdf.