# Resource Aware Programming

LUC MOREAU
University of Southampton
and
CHRISTIAN QUEINNEC
LIP6, Université de Paris 6

We introduce the *Resource Aware Programming* framework, which allows users to monitor the resources used by their programs and to programmatically express policies for the management of such resources. The framework is based on a notion of hierarchical groups, which act as resource containers for the computations they sponsor. Asynchronous notifications for resource exhaustion and for computation termination can be handled by arbitrary user code, which is also executed under the control of this hierarchical group structure. Resources are manipulated by the programmer using resource descriptors, whose operations are specified by a resource algebra. In this article, we overview the Resource Aware Programming framework and describe its semantics in the form of a language-independent abstract machine able to model both shared and distributed memory environments. Finally, we discuss a prototype implementation of the Resource Aware Programming framework in Java.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*

General Terms: Languages, Design

Additional Key Words and Phrases: Resource management, resource algebra, semantics, abstract machine

## 1. INTRODUCTION

Dynamic code loading has popularised 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 [Hartel and Moreau 2001], and in particular if

This research is funded in part by QinetiQ and the EPSRC Magnitude project (reference GR/N35816), and the EPSRC myGrid project (reference GR/R67743/01).
Authors' addresses: L. Moreau, School of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, U.K.; email: L.Moreau@ecs.soton.ac.uk; C. Queinnec, LIP6, 4, place Jussieu, 75252, Paris Cedex 05, France; email: Christian.Queinnec@lip6.fr.

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.

Another important trend is illustrated by multiagent systems [Roure et al. 2001] and services-based architectures [Booth et al. 2003] (in particular in the Grid context [Foster et al. 2002]): here, complex applications are the result of dynamic composition, opportunistic reuse, and on-the-fly creation of multiple distributed computations. Resource management is not only crucial, as illustrated by proposals for computational economies, but has now become a distributed problem.

Over the last few years, the authors of this article have designed a framework providing a programming interface for managing resources [Moreau and Queinnec 1997a, 1997b, 1998, 2002a]. Its initial design generalizes Kornfeld and Hewitt's [1981] group hierarchy, in which groups organized along a hierarchical structure sponsor computations. The framework introduces a new abstract notion of measurable resource: in this context, groups are seen as containers for such resources, which are consumed by their sponsored computations. Asynchronous notifications are introduced to mark the *exhaustion* of the resources contained in a group, and the *termination* of the computations sponsored by a group. In order to provide flexibility, it is desirable that notification handlers be defined by the programmer; therefore, these handlers can trigger arbitrary computations, whose resource usage must also be managed: the framework specifies how such notifications can be handled in an integrated manner. Additionally, explicit primitives are provided by the framework to *pause* computations, that is, to remove resources from their sponsoring group, or to *resume* computations, that is, to transfer resources back to their sponsoring group. Our previous work focused on the design of a language and its formalization [Moreau and Queinnec 1997a, 1997b]: we regarded this language as a "domain-specific language," with a functional core extended with primitives for resource management. We implemented a shared memory version of the framework, by modifying a POSIX thread library [Moreau and Queinnec 1998].

While our model of resource management had always been intended to support distribution and multiple resources of different types, these two aspects were rather poorly supported in our earlier work. Advances in mobile agent systems, and "Internet programming" languages, have shown that distribution has to be represented explicitly in the formalism: a shared memory model of resource management cannot and should not be extended transparently to the distributed setting; instead, specific primitives handling distribution have to be introduced. In addition, our motivation is to handle resources that have multiple forms, for example, processor, memory, and this requires introducing the means to refer to resources of a specific type [Moreau and Queinnec 2002a]. Besides these extensions, we also wanted to make this model of programming available in Java.

Extending the framework to multiple resources and to the distributed setting, and supporting the programming language Java, resulted in a radically different perception of the framework, which is the contribution of this article. As far as resources are concerned, a set of precise operations needs to be defined in order to support the framework, which resulted in a notion of *resource algebra*. The framework is no longer seen as a domain-specific language, but as a *library* that can be integrated with any programming language. Such a generic nature was also reflected in the semantics, which became programming language agnostic: we modeled the framework as a *message-passing system*, which can accommodate both shared and distributed memories.

The rest of the article is organized as follows. First, we clarify what we mean by resource and how the framework allows the programmer to refer to resources (Section 2). Then, we informally introduce the primitives of our Resource Aware Programming framework (Section 3). Given these primitives, we illustrate their usage with some examples that show how providers and consumers can program policies on the usage of resources (Section 4). We then model the framework by an asynchronous message-passing system (Section 5), and we sketch its implementation in Java (Section 6). Finally, in Section 7 we compare our system with related work and in Section 8 conclude the article. The Appendix gives an example of the abstract machine by showing the transitions it would perform on a sample program.

## 2. A NOTION OF RESOURCE

The purpose of this section is to introduce some terminology related to resources and associated concepts. Throughout this article, we use the notion of *resource* in the broadest sense. We obviously include hardware resources (or their software abstractions) managed by the operating system, such as CPU time, disk space, memory, files, threads, or sockets. In our Resource Aware Programming system, our goal is to be able to set limits on resource *consumption* (i.e., their usage) by applications. Examples of such limits include the maximum number of files to be opened by an application, the maximum size of data to be stored on disk, or the maximum number of threads to be created. Additionally, we also want to be able to refer to "rates," which are units of resources that can be consumed over a period of time: examples of these include the write rate of a program which should be limited to 50 kB/s, or the maximum number of messages allowed to be sent over the network during a period of time. All these resources are referred to as *rate*, *quantity*, or *space* resources in the Aroma VM [Groth and Suri 2002].

While physical resources are core resources that are required for applications to execute, the operating system, the environment, or the application context may set further constraints, which we also regard as resources. For instance, we consider the permissions attached to files or the rights to perform an action awarded by an authorization policy, such as a role certificate in role-based access control [Yao et al. 2001], as resources required by a computation to proceed. Similarly to physical resources, their availability changes over time, and we want application programmers to be able to reason over and manipulate them in a uniform manner.

Therefore, we define a *resource* as a hardware or software entity that is required by a computation to proceed and upon which the user wishes to impose a resource management policy. As computations proceed, they *consume* the resources that they were awarded. When all resources have been consumed by a computation, we expect an *exhaustion notification* to be raised. In addition, one of the system's requirements is that we must be able to grant resources to computations in a dynamic manner; as resources cannot be created, resources have to be *transferred* across computations.

In this article, we will present two views of resources. From a programmer's viewpoint, it is necessary to refer to quantities of resources, for instance, to indicate the amount of resources to allocate or to transfer to a computation. For this purpose, we introduce the notion of *resource descriptor*, which is a programmatic entity available to the programmer to refer to resources. On the other hand, the implementation of the Resource Aware Programming system, in collaboration with its hosting environment, for example, operation system or virtual machine, has to refer to concrete resource values: such resource values must remain under the system's control, and they must not be forgeable by the user. Therefore, we introduce *resource values* as a system-only accessible measure of resources.

Under precise circumstances, which we will elaborate on when describing the semantics of the framework, the system may give an indication of currently existing resource values: it has therefore to convert resource values into resource descriptors, so that they become examinable by the programmer. Symmetrically, the programmer may specify an amount of resources through a descriptor, which will be converted into concrete resource values, provided that they can be taken from a given supply of resources. Such a principle guarantees that resources cannot be created ex nihilo.

While resource values are necessarily absolute and they extensively enumerate the amount of resources available, descriptors do not have to be absolute. For instance, relative descriptors should allow programmers to specify that "*half*" of their resources should be reserved for a specific task, or that "*all*" their permissions must be transferred to another entity. To this end, a conversion function will convert relative descriptors into concrete resource values according to the amount of resources currently available.

Finally, the idea of resource transfer described above requires us to introduce two abstract functions. First, when transferred to a computation, the new resource values should be "*merged*" with the existing resource values. Second, as resource values cannot be created ex nihilo, they must be "*subtracted*" from an existing set of resource values. Therefore, we will respectively refer to these operations as *addition* and *subtraction* of resources. These are naturally specified by an algebra of resource operations, which we will characterize in the rest of the article, and for which we will discuss a possible concrete implementation.

## 3. INTUITIVE MODEL

In this section, we overview the essence of our Resource Aware Programming model—RAP for short—as described in previous publications [Moreau and Queinnec 1997a, 1997b, 1998, 2002a]. The RAP model is independent of the primitives for parallelism or distribution. In the sequel, we shall assume a *multithreaded* model of execution, and we will use the term *thread* to denote an execution thread created by some primitive for parallelism.

Our goal is to be able to allocate resources to computations, and to monitor and control their use as evaluations proceed. Thus, two events have a special importance in the lifetime of a computation, and they may trigger customisable actions. First, the *termination* of a computation marks the end of its life, and we would expect unconsumed resources to be transferred to a more suitable computation. Second, the *exhaustion* of the resources allocated to a computation triggers a notification that can be used to decide whether more resources need to be supplied to the computation.

Against this background, in order to be notified of the termination or resource exhaustion of a computation, we introduce an entity that represents a computation: a *group* is an object that can be used to refer to a computation. Specifically, a group is associated with a computation composed of several threads proceeding in parallel; in turn, these threads can initiate subcomputations by creating subgroups. As a result, our computation model is hierarchical. Following Kornfeld and Hewitt's terminology, a group is said to *sponsor* [Kornfeld and Hewitt 1981; Osborne 1990; Halstead 1990] the computation it is associated with. Symmetrically, every computation has a sponsoring group, and so does every thread.

At creation time, a group must be provided with an initial set of resources. More specifically, under the sponsorship of a group $G_1$, a computation can invoke a primitive newGroup, with arguments $f, \vec{R}, h_e, h_t$ (which we explain below). This creates a new first-class group $G_2$ that is allocated an initial set of resources $\vec{R}$ and whose parent is $G_1$. Furthermore, this initiates a computation under the sponsorship of $G_2$ by executing the user code $f$ ($f$ would be a function in a functional language or a `Runnable` object in Java.). As the framework keeps track of resource consumption, the resources $\vec{R}$ allocated to $G_2$ are deducted from the resources of $G_1$. Figure 1 displays the behavior of the primitive newGroup. Before execution, we see a configuration where a group $G_1$ is sponsoring two threads $T_1$ and $T_2$ and a subgroup $G_3$ itself sponsoring a thread $T_3$. After evaluating the primitive newGroup, the new subgroup $G_2$ is sponsoring the application of $f$, with $\vec{R}$ resources transferred from $G_1$ to $G_2$. The two remaining arguments of the primitive newGroup, $h_e, h_t$, are notification handlers, which we shall discuss later.

In a framework for managing resources, every action should be accounted for, and therefore should be costed. Figure 1 shows that the resources of $G_1$ become $\vec{R}_1 - \vec{R} - K_g$ after transition: the value $-\vec{R}$ is the amount of resources transferred to the new group $G_2$, whereas $-K_g$ represents the cost of the group creation operation. In order to ensure that resources do not get created ex nihilo, this transition assumes that enough resources are available in $G_1$, that is, $\vec{R}_1 > \vec{R} + K_g$.

The RAP model enforces the following principle: any computation consumes resources from its sponsoring group. Therefore, not only is a group perceived as a way of naming computations, but also it must be regarded as a *tank of resources* for the computation. Such a principle is illustrated by Figure 2, which displays a thread $T$ evolving to state $T'$ by performing an action, whose cost $\vec{R}_1$ is charged to the sponsoring group $G_1$.

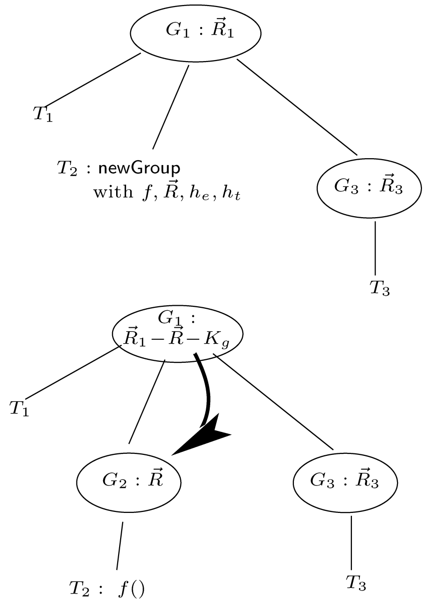

Fig. 1. Group creation.

$G_1 : \vec{R}$
$T$
$G_1 : \vec{R} - \vec{R}_1$
$T'$

Fig. 2. Resource consumption when $T$ evolves to $T'$ using $\vec{R}_1$ resources.

Once a group is created, two moments have a special importance in its lifetime. A group is said to be *terminated* if it has no subgroup and it does not sponsor any thread, that is, no more activity is sponsored by the group. Likewise, a group is said to be *exhausted* if is has no resource left, that is, the threads it sponsors can no longer progress by lack of resources. In RAP, group termination and resource exhaustion are asynchronously notified by applying the user specified handler functions $h_t$ and $h_e$, respectively. (Subscript $t$ denotes termination, whereas subscript $e$ denotes exhaustion.) In Figure 3, when the only thread $T_4$ of group $G_3$ is terminating, the function $h_t$ is asynchronously called, with $G_3$ as argument, to notify its termination, and the resource surplus of $G_3$ is transferred back to $G_1$, minus the cost of notification $K_n$. Note that the execution of the handler $h_t$ is sponsored by $G_1$, that is, the parent of $G_3$; indeed, it would not make any sense to have it sponsored by $G_3$, since it has just terminated its lifetime. The termination handler is also given an indication of the amount of resources that were available when $G_3$'s termination occurred in the form of a descriptor $\vec{D}_3$ denoting the amount of resources $\vec{R}_3$ that remained in $G_3$ at termination time.

In Figure 4, a computation $T_2$ sponsored by $G_2$ requires an amount $\vec{D}$ of resources that is not available in $G_2$; the function $h_e$ is asynchronously called, with

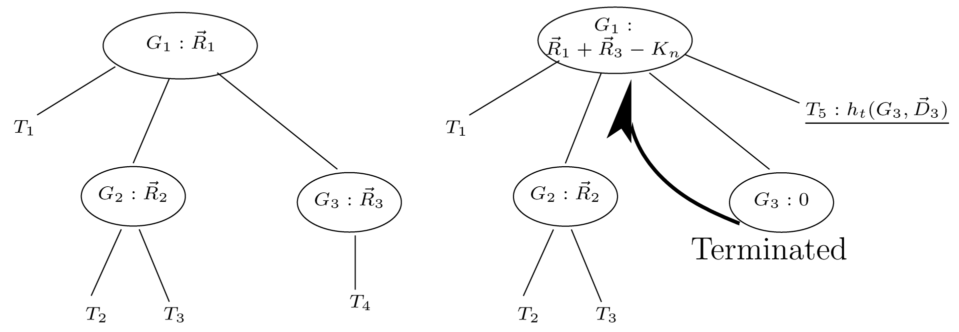

Fig. 3. Termination of group $G_3$ when thread $T_4$ finishes its execution.

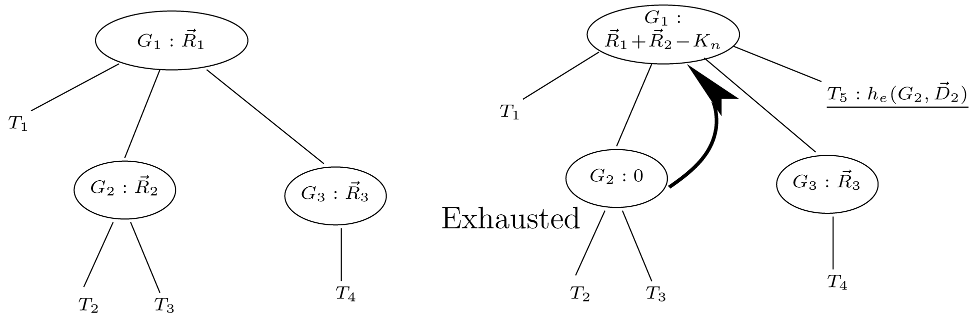

Fig. 4. Exhaustion of group $G_2$ when $T_2$ requests $\vec{D}$ resources ($\vec{D} > \vec{R}_2$).

argument $G_2$, to notify its resource exhaustion, also under the sponsorship of $G_1$, for similar reasons as the termination handler. Furthermore, the remaining resources of $G_2$ are transferred to $G_1$, minus the cost of notification $K_n$, to ensure that no resource is leaked by the system. Similarly, the exhaustion handler is provided with the resource descriptor $\vec{D}_2$ denoting the amount of resources that existed at the time of $G_2$'s exhaustion, and the requested amount of resources that resulted in the exhaustion.

Figure 5 displays the state transition diagram for groups. At creation time, a group is in the running state; this means that the threads it sponsors can proceed as long as they do not require more resources than available. Asynchronous notifications are represented by dotted lines. Once a computation requires more resources than available in its sponsoring group, the state of its group changes to exhausted, and at the same time an asynchronous notification $h_e$ is run. When all the subgroups and all the threads sponsored by a group terminate, its state becomes terminated, and the asynchronous notifier $h_t$ is called. Let us observe that the terminated state is a dead end in the state diagram; this guarantees the *stability* of the termination property: once a computation terminates, it is not allowed to restart (as the resource that it did not consume may have been reallocated).

In addition, resources may be transferred between groups, independently of the group hierarchy, under the control of the user's program. Thus, we define two primitives that operate on groups: pause and awaken. The primitive pause

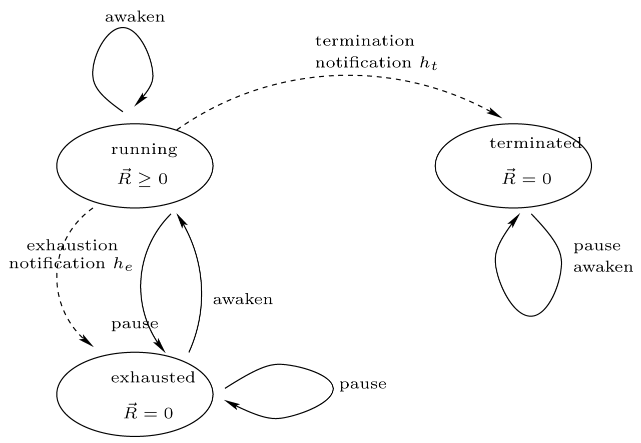

Fig. 5. State transitions.

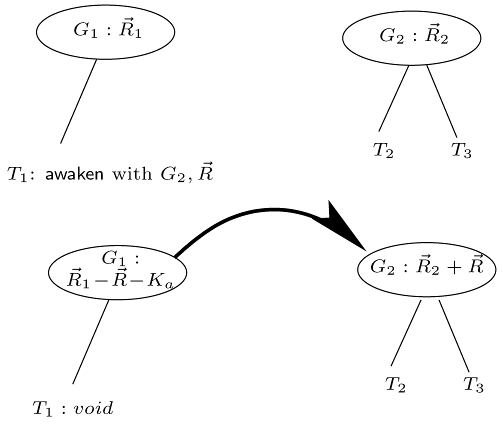
$G_1 : \vec{R}_1$
$G_2 : \vec{R}_2$
$T_1$: awaken with $G_2, \vec{R}$
$T_2$
$T_3$
$G_1 : \vec{R}_1 - \vec{R} - K_a$
$G_2 : \vec{R}_2 + \vec{R}$
$T_1 : void$

Fig. 6. Awakening a group.

forces a running group *and its subgroups* into the exhausted state; all the resources that were available in this hierarchy are transferred to the group that sponsored the pause action. The primitive awaken transfers some resources to a target group $g$, after deducting them from the group sponsoring the awaken action. If the target group $g$ is in the exhausted state, its state is changed to running. On the other hand, if the group is in the terminated state, the awaken operation is void, and does not result in any transfer of resources[1] to the target group. Figure 6 displays the behavior of awaken, assuming that $\vec{R}_1 > \vec{R} + K_a$ and $G_2$ is not terminated.

Let us observe the asymmetric behavior of pause and awaken: the former operates recursively on a group hierarchy, while the latter acts on a group and

[1]Technically, resources are transferred, but once the group is observed to be terminated, resources are transferred back to the group sponsoring the awaken primitive. It is the responsibility of the initiator of the awaken primitive to ensure that it does not terminate before resources are transferred back; otherwise they may be lost.

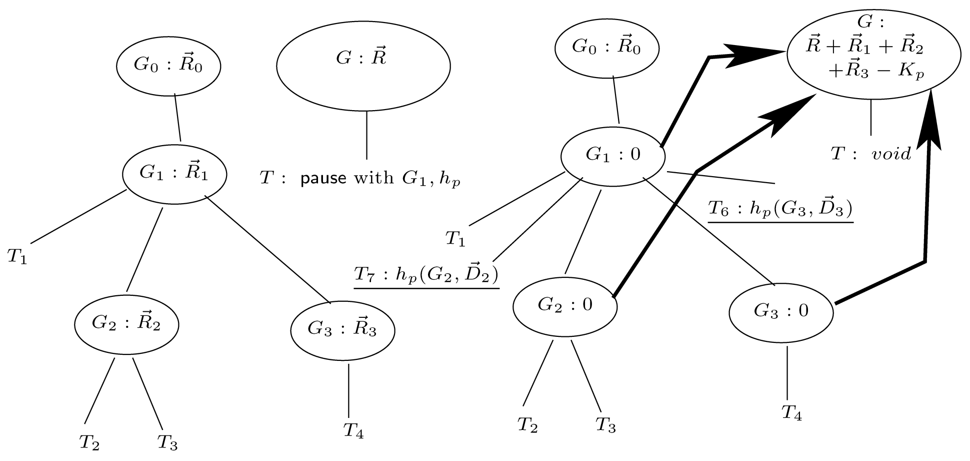

Fig. 7. Pausing group $G_1$.

not its descendants. We however might like to awaken a hierarchy recursively, for instance, in order to resume a paused parallel search. In particular, we might wish to resume the search with the resource distribution that existed when the hierarchy was paused. Unfortunately, such information is no longer available because groups are *memoryless*. By this, we mean that a group does not remember the amount of resources it contained before being paused. It is therefore the programmer's responsibility to leave some information at pausing-time about the way a hierarchy could be awakened. Thus, we designed pause such that, not only does it transfer resources, but also it posts a notification for each group in the hierarchy. Figure 7 displays the precise behavior of pause: executing pause with arguments $G_1$ and $h_p$ forces into the exhausted state each group $G'$ in the hierarchy rooted by $G_1$; moreover, for each $G'$, an activation of $h_p$ with $G'$ as argument is created under the sponsorship of the parent of $G'$ (except for the root $G_1$ to avoid intruding its parent). Notifications are prevented from running since all groups in the hierarchy have been dried out. Once the group $G_1$ is awakened, any notification sponsored by the group will be activated and may decide to awaken the group it is applied to, and step by step, resources may be redistributed among the hierarchy.

A group is a stateful object in the sense that it is a resource container, whose contents are affected by computations as they proceed and consume resources. Once exhausted, groups become stateless and memoryless: by this, we mean that their state no longer changes (until they become running again), and they do not even remember the resources they contained at the time of pausing. It is no doubt useful to know what amount of resources a group owned, but we did not feel the group itself was the best data-structure to make this information available through. Typically, this information is used to decide if and how many resources need to be supplied to the group again. Such a decision making takes place outside the group, and, in particular, in a distributed setting, in all likelihood, at another location. It is therefore more flexible to make this information available to a handler that the user can program in order to store the information in the most suitable location, for the current application.

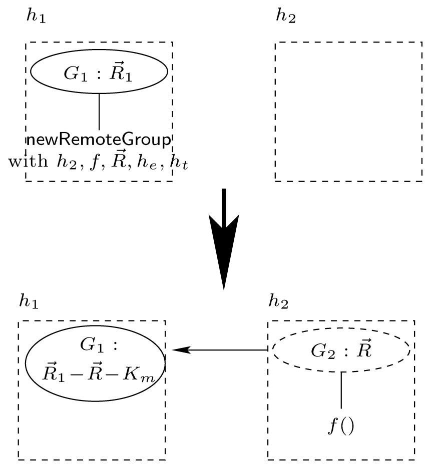

Fig. 8. Remote group creation at location $h_2$.

So far, we have discussed the RAP model in multithreaded shared-memory environments. In order to extend the model to the distributed setting, we now introduce a notion of "location," that is, a hosting environment where groups and their associated computations are situated. First, we revisit the newGroup primitive to understand its effect in terms of locations: newGroup creates a new group at the location where it is executed, that is, at the same location as its sponsoring group. As far as the other primitives are concerned, they are all independent of groups' locations, and therefore can operate on both local and remote groups.

In order to control the location at which groups and associated computations should be created, we introduce a new primitive newRemoteGroup that requests a location in addition to the arguments required by newGroup. Its effect is the creation of a group at the remote location, as depicted by Figure 8.

Before transition, at location $h_1$, newRemoteGroup is called with arguments $h_2$, $f$, $\vec{R}, h_e, h_t$ with a sponsoring group $G_1$ containing $\vec{R}_1$ resources. After transition, a new group $G_2$ has been created at the remote location $h_2$, with resources $\vec{R}$, which have been deducted from $G_1$ as well as the administrative cost of remote group creation $K_m$. Let us note that, after transition, the execution thread at $h_1$ has terminated. In essence, this primitive is similar to a weak form of migration [Fuggetta et al. 1998].

Following the creation of group $G_2$, the parent of $G_2$ is defined as $G_1$. In this situation, a difficulty is to ensure that the handlers for exhaustion and termination of $G_2$ be executed under the sponsorship of $G_1$, which is located at $h_1$. In Figure 8, we represent $G_2$ as a dashed ellipse, because this group was created by the newRemoteGroup primitive. For such a group, which we refer to as a *remote group*, the handlers for termination and exhaustion are executed at the location of the remote group's parent, as illustrated by Figures 9(a) and 9(b), similarly to Figures 3 and 4.

In Figure 9(a), when a remote group $G_2$ detects the termination of the computation it sponsors, its resources are transferred back to its parent $G_1$, minus the

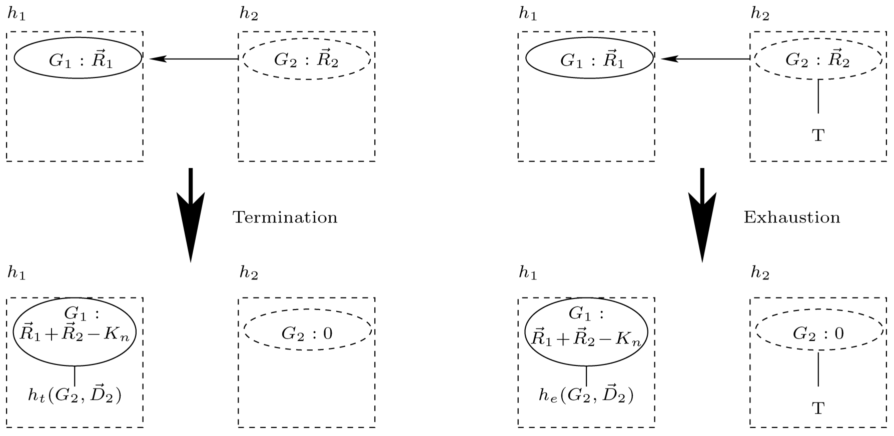

Fig. 9. Remote group: (a) termination; (b) exhaustion.

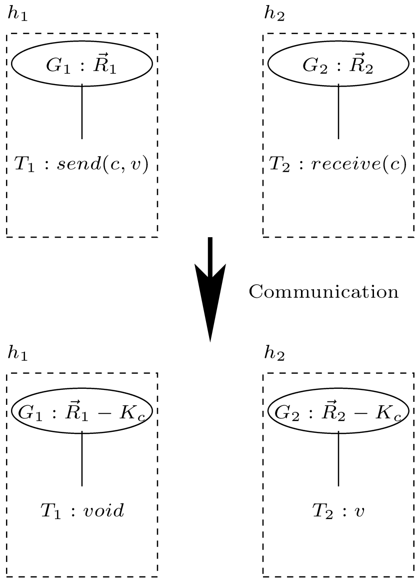

Fig. 10. Synchronous communications.

amount $K_n$ necessary for the notification. Then, the notification is run under the sponsorship of $G_1$ *at $G_1$'s location*. Likewise, in Figure 9(b), exhaustion of $G_2$ triggers an exhaustion notification at location $h_1$, under the sponsorship of $G_1$.

In addition, we need a communication mechanism to allow distributed computations to exchange messages. Figure 10 displays the semantics of synchronous communications using primitives *send* and *receive*. No group is created here; instead, both sponsoring groups are charged with the cost of communication $K_c$.

This section concludes our intuitive presentation of the Resource Aware Programming model. In the next section, we will illustrate how the model can be used to program some applications.

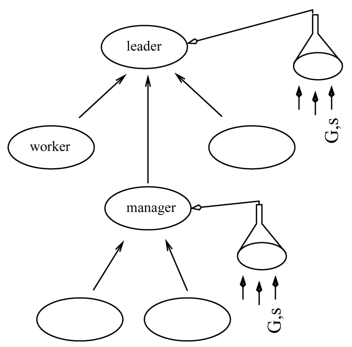

Fig. 11. Return on investments.

## 4. EXAMPLES

As our framework is programming language independent, we will present examples in an agnostic discursive way. We examine two examples of resource aware programming: in the first one, resources are used by the program to coordinate the exploration of a complex search, whereas the second example demonstrates a policy of resource usage.

### 4.1 Return on Investments

Let us consider a problem with a large space to explore. This may be the quest for a cheap round-the-world trip, some natural numbers with unusual properties, or some statistical analysis of genome sequences. A single best solution for such problems might not exist; however, assuming that we are able to compare solutions, our aim is to retain the most interesting ones. Several strategies to find a solution might exist, although the best ones are unknown. Here, we propose a scheme that dynamically favors the strategies that produce improving solutions.

A *leader* group starts the exploration and spawns several groups with various strategies (see Figure 11). These groups are referred to as *workers* when they try to find solutions, or as *managers* when they coordinate submanagers or subworkers testing alternate strategies, dividing or refining the initial problem, etc. All groups send the solutions they independently find (paired with their own identity) to a *comparator* set up by their manager.

The comparator compares the received solutions and notifies its associated manager of the results of the comparison: if a group sends a solution that is better than the solutions sent by its siblings (or competitors), then its relative priority with respect to its manager is increased, and transitively so the priorities of its managers with respect to their respective managers. Comparators eventually provide a reward to workers finding good solutions, whereas workers providing non-best solutions will gradually receive less resources to continue their search.

As far as implementation is concerned, groups (whether workers or managers) are created with some minimal resources so that they are forced to request resources from their parent group (their manager) in order to proceed.

The manager gives some new resources according to the performance of the group among all the groups it manages. If a group does not produce enough solutions or its solutions are not good enough, then its future resource requests will be delayed and therefore its share of resources will decrease. For this to happen, a manager must keep up to date information about the groups it manages: for instance, it may maintain the total amount of given resources, the best solution found so far, the number of best found solutions, etc. Hence, managers may rank, at any time, the performance of their managed subgroups.

When a group notifies its manager that its resources are exhausted, this manager will check whether the group is among the most promising (or profitable) ones, in which case it will give some resources to pursue its work or will delay its request until the group becomes promising again (this may occur if its competitors become less profitable).

This scheme analyzes the return on investments and favors those workers and their management that are successful. The solution benefits from the use of RAP because it offers a framework to control the parallel execution of the multiple search activities, which are prioritized according to their ability to find good solutions. With RAP, we have successfully separated the different concerns in this application, namely, the search function, the measure of result quality, and the result-driven scheduling of the search activities. As a result, domain-specific code searching for solutions does not need to be modified with RAP primitives to run in this framework.

## 4.2 Dynamic Authorization

In their current form, authorization policies in JDK are static, that is, they are constant during the lifetime of a running program. Encoding authorization as resource allows more dynamic patterns.

Let us consider that some private keys are held in several files within some directory, and that we want to define a policy according to which a program may be allowed to use at most one such key. Static policies may be used to restrict the access to that directory or to those files but cannot prevent the use of more than one such key.

Alternatively, one may create a new type of resource, which we call a *key-resource*, corresponding to the use of one key. A new group $G_K$ can then be defined with one and only one such resource. Any new computation that must obey this policy will have to be created as a subgroup of $G_K$. Whenever a new subgroup is created, it is initially not given such a key-resource: as a result, any attempt to use such key will result in an exhaustion notification. When a notification handler for a key-resource exhaustion is executed in $G_K$, one may decide to transfer one key-resource to the group, which may consume it: the only key-resource having been consumed, no further request for accessing keys can be satisfied.

In this example, once the resource for the use of at most one key is consumed, it becomes nonavailable for future use. Alternatively, one may create a nonconsumable resource acting as a token that confers some rights to the computation that has access to it. This would limit some rights to at most one group at any time. That token may pass from group to group cooperatively via the awaken primitive or preemptively via the pause operation.

## 5. SEMANTICS

The previous sections have introduced the notion of resource and associated basic operations. In this section, we present a resource algebra describing two fundamental operations on resource values and descriptors. We then formulate the semantics of the Resource Aware Programming framework as a message-passing abstract machine. The combination of the resource algebra and the abstract machine results in a precise and operational definition of RAP, which we have used to build our reference implementation.

### 5.1 Resource Algebra

The RAP model was designed so that users' programs never manipulate resource values, but only refer to resources through the use of descriptors. Thus, the conversion between descriptors and resources must be handled by the system, in order to guarantee that resources cannot be forged by users. In addition, we introduce a conversion function that converts resource values back into descriptors, with resource values elements of $\mathbb{P}(\mathcal{R})$ and descriptors as elements of the set $\mathbb{P}(\mathcal{D})$:

$$toDesc() : \mathbb{P}(\mathcal{R}) \rightarrow \mathbb{P}(\mathcal{D}).$$

Precise definitions of the sets $\mathcal{R}$ and $\mathcal{D}$ will be introduced later in the article. We now discuss the resource values and resource descriptors.

5.1.1 *Resource Values.* In our prototype implementation, we support a range of resources, for which we now present a more formal characterization. There are three broad categories of resource values, which can be expressed in terms of (i) a numeric value and a type, (ii) a set of values over a discrete domain, or (iii) an infinite resource.

A numerical resource value can be used for instance to count the maximum number of files or the maximum number of sockets that can be opened; an enumerated resource value can be used to list all the Java permissions a computation is allowed to use; finally, the infinite resource value is used for the root of the group hierarchy. We represent such resources symbolically as follows:

| | |
|---|---|
| $\langle \mathsf{NumV}\ n, t\rangle$ | (Numerical resource value) |
| $\langle \mathsf{EnumV}\ s, t\rangle$ | (Enumerated resource value) |
| $\langle \mathsf{AllV}\rangle$ | (Infinite resource set) |
| $t$ | (Resource type) |
| $n$ | (Integer) |
| $s$ | (Set) |

As illustrated by the intuitive description of Section 3, we need to be able to merge resource values. Therefore, we define an algebra of resource values with an addition operation. Our algebra distinguishes different kinds of numeric resource values. There are resource values that are additive: for instance, the number of files allowed to be opened. On the other hand, there are also resource values that denote a maximal limit, such as the maximum number of hops a computation is allowed to travel from a given node. Such differences are encapsulated in the operator $+_t$ describing how to add two numeric resources of a given type $t$:

$$\begin{array}{rclcll}
\langle \mathsf{NumV}\ n_1, t_1\rangle & \oplus & \langle \mathsf{NumV}\ n_2, t_2\rangle & = & \langle \mathsf{NumV}\ n_1 +_t n_2, t_1\rangle & \text{if } t_1 = t_2,\\
\langle \mathsf{EnumV}\ s_1, t_1\rangle & \oplus & \langle \mathsf{EnumV}\ s_2, t_2\rangle & = & \langle \mathsf{EnumV}\ s_1 \cup s_2, t_1\rangle & \text{if } t_1 = t_2,\\
\langle \mathsf{AllV}\rangle & \oplus & \langle \mathit{Resource\ Value}\rangle & = & \langle \mathsf{AllV}\rangle, & \\
n_1 & +_t & n_2 & = & n_1 + n_2 & \text{if } \mathit{Additive}(t),\\
n_1 & +_t & n_2 & = & \max(n_1, n_2) & \text{if } \mathit{MaxLimit}(t).
\end{array}$$

We can see that the Infinite resource set is an absorbent. Such a resource value is used for the root of the group hierarchy. We do not expect any of the groups created by the user to contain the infinite resource value; such a property is enforced by the subtraction operator, which we discuss in the following section.

5.1.2 *Resource Descriptors.* Programmers do not have direct access to resource values, but instead they can use resource descriptors to denote resource values. Resource descriptors can be absolute: they then denote the corresponding amount of resource values. They can be relative (or symbolic): their meaning is specified according to an algebra, which we describe below. In the table below, the first two descriptors are absolute, whereas the next two are relative.

| | |
|---|---|
| $\langle \mathsf{NumD}\ n, t\rangle$ | (Numerical resource descriptor) |
| $\langle \mathsf{EnumD}\ s, t\rangle$ | (Enumerated resource descriptor) |
| $\langle \mathsf{PercentD}\ p\rangle$ | (Percentage resource descriptor) |
| $\langle \mathsf{AllD}\rangle$ | (All Resources Descriptor set) |
| $t$ | (Resource type) |
| $n$ | (Integer) |
| $p$ | (Percentage $0 \leq p \leq 1$) |
| $s$ | (Set) |

The operation $\ominus$ is opposite to $\oplus$: it specifies how to split an amount of resource values in two parts according to a resource descriptor. The operator $\ominus$ has the following signature:

$$\begin{aligned}\ominus\ :\ &\langle \mathit{Resource\ Value}\rangle \times \langle \mathit{Resource\ Descriptor}\rangle\\ &\quad \to \langle \mathit{Resource\ Value}\rangle \times \langle \mathit{Resource\ Value}\rangle \mid \bot.\end{aligned}$$

Given a value and a descriptor, $\ominus$ either produces two resource values or fails to split the resource value in two parts. The second part represents the amount subtracted from the initial value, while the first part represents what remains of the initial value after subtraction.

$$
\begin{array}{rcll}
\langle \mathsf{NumV}\ n_1, t_1\rangle & \ominus & \langle \mathsf{NumD}\ n_2, t_2\rangle & = (\langle \mathsf{NumV}\ n_1 -_{t_1} n_2, t_1\rangle, \langle \mathsf{NumV}\ n_2, t_1\rangle) \\
 & & & \quad \text{if } t_1 = t_2 \text{ and } n_1 \geq n_2, \\
\langle \mathsf{NumV}\ n_1, t_1\rangle & \ominus & \langle \mathsf{NumD}\ n_2, t_2\rangle & = \bot \text{ if } t_1 \neq t_2 \text{ or } n_1 < n_2, \\
\langle \mathsf{NumV}\ n_1, t_1\rangle & \ominus & \langle \mathsf{PercentD}\ p\rangle & = (\langle \mathsf{NumV}\ n_1 -_{t_1} n_1 * p, t_1\rangle, \langle \mathsf{NumV}\ n_1 * p, t_1\rangle), \\
\langle \mathsf{NumV}\ n_1, t_1\rangle & \ominus & \langle \mathsf{EnumD}\ s, t\rangle & = \bot, \\
\langle \mathsf{NumV}\ n_1, t_1\rangle & \ominus & \langle \mathsf{AllD}\rangle & = (\langle \mathsf{NumV}\ 0, t_1\rangle, \langle \mathsf{NumV}\ n_2, t_1\rangle), \\
 & & & \\
\langle \mathsf{EnumV}\ s_1, t_1\rangle & \ominus & \langle \mathsf{NumD}\ n, t\rangle & = \bot, \\
\langle \mathsf{EnumV}\ s_1, t_1\rangle & \ominus & \langle \mathsf{EnumD}\ s_2, t_2\rangle & = (\langle \mathsf{EnumV}\ s_1, t_1\rangle, \langle \mathsf{EnumV}\ s_2, t_2\rangle) \\
 & & & \quad \text{if } t_1 = t_2 \text{ and } s_1 \supseteq s_2, \\
\langle \mathsf{EnumV}\ s_1, t_1\rangle & \ominus & \langle \mathsf{EnumD}\ s_2, t_2\rangle & = \bot \text{ if } s_1 \not\supseteq s_2 \text{ or } t_1 \neq t_2, \\
\langle \mathsf{EnumV}\ s, t\rangle & \ominus & \langle \mathsf{PercentD}\ p\rangle & = \bot, \\
\langle \mathsf{EnumV}\ s, t\rangle & \ominus & \langle \mathsf{AllD}\rangle & = (\langle \mathsf{EnumV}\ s_1\rangle, \langle \mathsf{EnumV}\ s_1\rangle), \\
 & & & \\
\langle \mathsf{AllV}\rangle & \ominus & \langle \mathsf{NumD}\ n, t\rangle & = (\langle \mathsf{AllV}\rangle, \langle \mathsf{NumV}\ n, t\rangle), \\
\langle \mathsf{AllV}\rangle & \ominus & \langle \mathsf{EnumD}\ s\rangle & = (\langle \mathsf{AllV}\rangle, \langle \mathsf{EnumV}\ s\rangle), \\
\langle \mathsf{AllV}\rangle & \ominus & \langle \mathsf{PercentD}\ p\rangle & = \bot, \\
\langle \mathsf{AllV}\rangle & \ominus & \langle \mathsf{AllD}\rangle & = \bot, \\
n_1 & -_t & n_2 & = n_1 - n_2 \quad \text{if } \mathit{Additive}(t), \\
n_1 & -_t & n_2 & = n_1 \quad \text{if } \mathit{MaxLimit}(t).
\end{array}
$$

The above algebra has defined the operations $\oplus$ and $\ominus$ on resource entities (values or descriptors). In the rest of the article, we refer to multiple resources managed by a group. Therefore, we extend this notation to vectors of resources. Additionally, we introduce the following abbreviations:

$$
\left.\begin{array}{rcl}
\vec{R} \ominus_1 \vec{D} & = & \vec{R}_1 \\
\vec{R} \ominus_2 \vec{D} & = & \vec{R}_2
\end{array}\right\} \text{ if } \vec{R} \ominus \vec{D} = (\vec{R}_1, \vec{R}_2).
$$

No resource is intended to be created or lost by the operations $\oplus$ and $\ominus$, which therefore must satisfy the following relation:

$$\text{If } \vec{R} \ominus \vec{D} \text{ results in } (\vec{R}_1, \vec{R}_2), \text{ then } \vec{R}_1 \oplus \vec{R}_2 = \vec{R}.$$

We should note that the $\ominus$ operation is successful for a set of infinite resource values, only if the resource descriptor is absolute. This constraint guarantees that no user group will ever be given infinite resources, even though the root of the group hierarchy is allocated infinite resource values.

## 5.2 Operational Semantics

In this section, we formalize our model of resource programming using an abstract machine that is programming-language independent so as to be as generic as possible. The abstract machine comprises a notion of group and makes interactions between groups explicit through the use of communication channels. As a result, our model is suitable for both shared and distributed memory systems.

$$
\begin{array}{rcll}
\mathcal{G} & & & \text{(Groups)}\\
\mathcal{R} & & & \text{(Resource values)}\\
\mathcal{D} & & & \text{(Resource descriptors)}\\
\mathcal{F} & = & Void \rightarrow Void & \text{(User function)}\\
\mathcal{T} & & & \text{(Threads)}\\
\mathcal{S} & = & \{\mathsf{running}, \mathsf{exhausted}, \mathsf{terminated}\} & \text{(Group states)}\\
\mathcal{H}_t & = & \mathcal{G} \times \mathbb{P}(\mathcal{D}) \rightarrow Void & \text{(Termination handlers)}\\
\mathcal{H}_e & = & \mathcal{G} \times \mathbb{P}(\mathcal{D}) \times \mathbb{P}(\mathcal{D}) \rightarrow Void & \text{(Exhaustion handlers)}\\
\mathcal{H}_p & = & \mathcal{G} \times \mathbb{P}(\mathcal{D}) \rightarrow Void & \text{(Pause handlers)}\\
\mathcal{M}_s & = & \mathsf{exhaustion} : \mathcal{G} \times \mathbb{P}(\mathcal{D}) \times \mathbb{P}(\mathcal{R}) \rightarrow \mathcal{M} & \text{(User independent messages)}\\
& & \mid\ \mathsf{termination} : \mathcal{G} \times \mathbb{P}(\mathcal{R}) \rightarrow \mathcal{M} & \\
& & \mid\ \mathsf{thread} : \mathcal{F} \rightarrow \mathcal{M} & \\
& & \mid\ \mathsf{sendPause} : \mathcal{G} \times \mathcal{H}_p \rightarrow \mathcal{M} & \\
& & \mid\ \mathsf{awaken} : \mathcal{G} \times \mathbb{P}(\mathcal{R}) \rightarrow \mathcal{M} & \\
& & \mid\ \mathsf{pause} : \mathcal{G} \times \mathcal{G} \times \mathcal{G} \times \mathcal{H}_p \rightarrow \mathcal{M} & \\
& & \mid\ \mathsf{paused} : \mathcal{G} \times \mathbb{P}(\mathcal{R}) \times \mathcal{H}_p \rightarrow \mathcal{M} & \\
& & \mid\ \mathsf{awakenFailure} : \mathcal{G} \times \mathcal{G} \times \mathbb{P}(\mathcal{R}) \rightarrow \mathcal{M} & \\
\mathcal{M}_{\mathcal{R}} & = & \mathsf{group} : \mathbb{P}(\mathcal{R}) \times \mathcal{F} \times \mathcal{H}_t \times \mathcal{H}_e \rightarrow \mathcal{M} & \text{(Instantiated messages)}\\
& & \mid\ \mathsf{sendAwaken} : \mathcal{G} \times \mathbb{P}(\mathcal{R}) \rightarrow \mathcal{M} & \\
\mathcal{M}_{\mathcal{D}} & = & \mathsf{group} : \mathbb{P}(\mathcal{D}) \times \mathcal{F} \times \mathcal{H}_t \times \mathcal{H}_e \rightarrow \mathcal{M}_n & \text{(Uninstantiated messages)}\\
& & \mid\ \mathsf{sendAwaken} : \mathcal{G} \times \mathbb{P}(\mathcal{D}) \rightarrow \mathcal{M}_n & \\
\mathcal{M}_u & = & \mathcal{M}_s \cup \mathcal{M}_{\mathcal{D}} & \text{(User messages)}\\
\mathcal{M} & = & \mathcal{M}_s \cup \mathcal{M}_{\mathcal{R}} & \text{(System messages)}\\
\mathcal{K} & = & \mathcal{G} \times \mathcal{G} \rightarrow Bag(\mathcal{M}) & \text{(Channels)}\\
\mathcal{GC} & = & \mathcal{S} \times \mathbb{P}(\mathcal{R}) \times \mathbb{P}(\mathcal{T}) \times \mathbb{P}(\mathcal{G}) \times \mathcal{G} & \\
& & \times\ Queue(\mathcal{M}) \times \mathcal{H}_t \times \mathcal{H}_e & \text{(Group configurations)}\\
\mathcal{C} & = & (\mathcal{G} \rightarrow \mathcal{GC}) \times \mathcal{K} & \text{(Configurations)}
\end{array}
$$

Characteristic variables:

$$G \in \mathcal{G}, R \in \mathcal{R}, D \in \mathcal{D}, \vec{R} \in \mathbb{P}(\mathcal{R}), \vec{D} \in \mathbb{P}(\mathcal{D}), f \in \mathcal{F},$$
$$M \in \mathcal{M}_u, M \in \mathcal{M}, M^* \in Queue(\mathcal{M}), s \in \mathcal{S}, h_t \in \mathcal{H}_t, h_e \in \mathcal{H}_e, \langle \Gamma, k \rangle \in \mathcal{C}$$

The initial state is defined in terms of $f_{init}, \vec{D}_{init}, h_{t\,init}, h_{e\,init}$, provided by the user, with $G_\perp$ the system predefined *root group*, with resources and handlers $\vec{R}_\perp, h_{t\perp}, h_{e\perp}$, and no parent.

$$
\begin{array}{rcl}
\vec{R}_{init} & = & \vec{R}_\perp \ominus_1 \vec{D}_{init}\\
M_{init} & = & \mathsf{group}(\vec{R}_{init}, f_{init}, h_{t\,init}, h_{e\,init})\\
\Gamma_{init}(G_\perp) & = & \langle \mathsf{running}, \vec{R}_\perp, \emptyset, \emptyset, \perp, [M_{init}], h_{t\perp}, h_{e\perp} \rangle\\
\Gamma_{init}(G) & = & \perp, \text{ for any } G \neq G_\perp\\
k_{init} & = & G_1, G_2 \rightarrow \emptyset, \quad \forall G_1, G_2 \in \mathcal{G}\\
c_{init} & = & \langle \Gamma_{init}, k_{init} \rangle
\end{array}
$$

Fig. 12. State space.

5.2.1 *State Space.* The abstract machine state space is displayed in Figure 12. In this abstract machine, we model only the computations that pertain to resource management, and we do not model any other form of computation. Groups are a key component of the abstract machine; they are containers for *resources values*, which the programmer refers to by using *resource descriptors*. Messages are exchanged between groups, and are defined by an inductive type, whose constructors are exhaustion, termination, thread, group, sendAwaken, sendPause, awaken, pause, paused, and awakenFailure. We do not make any assumption of the order of message delivery in communication channels, and therefore we represent such channels as bags of messages between pairs of groups. Each group is associated with some information, referred to as *group configuration*: its status, its set of resources, its threads, its subgroups, its parent group, a queue of incoming messages waiting to be processed, and its two handlers for termination and exhaustion. A complete system configuration is defined by all the groups and their associated group configuration and communication channels.

We note here a peculiarity of the set of messages: we distinguish *user messages* in $\mathcal{M}_u$ from *system messages* in $\mathcal{M}_s$. The former are messages that result from a library call by the programmer, and therefore refer to resources by descriptors, whereas the latter are messages where the resource descriptors have been replaced by resource values. In order to convert user messages into system messages, we use a conversion function, which we note $M\{\vec{D}/\vec{R}\}$, which substitutes a resource descriptor for a resource value in a user message; it is defined as the identity function for all messages in $\mathcal{M}_s$ and as follows for messages of $\mathcal{M}_{\mathcal{D}}$:

$$\begin{aligned} \_\,\{\vec{D}/\vec{R}\} &:: \mathcal{M}_{\mathcal{D}} \rightarrow \mathcal{M}_{\mathcal{R}}, \\ \mathsf{group}(\vec{D}, f, h_t, h_e)\ \{\vec{D}/\vec{R}\} &= \mathsf{group}(\vec{R}, f, h_t, h_e), \\ \mathsf{sendAwaken}(G, \vec{D})\ \{\vec{D}/\vec{R}\} &= \mathsf{sendAwaken}(G, \vec{R}). \end{aligned}$$

For convenience, we shall use the extension of the conversion function to sequences of messages.

5.2.2 *Initial State.* The abstract machine is characterized by an initial state and a set of transitions. In the initial state $c_{init}$, defined in Figure 12, we find empty communication channels and a single group $G_\perp$. The *root group* $G_\perp$ is a system-defined group that is the root of the hierarchy, with resources $\vec{R}_\perp$ and handlers $h_{t\perp}, h_{e\perp}$. In Section 6, we will see how the system group $G_\perp$ is defined in a concrete implementation. In group $G_\perp$, there is a message requesting the creation of a new group, with user code $f_{init}$, resources $\vec{R}_{init}$ (computed from $\vec{D}_{init}$), and notification handlers $h_{t\,init}, h_{e\,init}$, all specified by the user.

5.2.3 *Thread Execution.* Our aim is to present a language-independent semantics of our hierarchical model of resource programming. This model is however meant to be integrated into a programming language that contains constructs, primitives, or library functions able to request resources managed by our model. Against this background, we identify here a set of primitives, but we do *not* provide their exact syntax, nor their operational definition. Instead, we express their effect on RAP, by the descriptors of the resources they are meant to request, the sequence of user messages they generate, and their administrative cost. Their definitions appear in Figure 13.

We explain how Figure 13 is structured by discussing the primitive *awaken*, which expects two arguments: a target group and some resource descriptors. The resources requested by this primitive are exactly those passed to the

| Operation | Descriptor | User messages | Administrative cost |
|---|---|---|---|
| $awaken(G, \vec{D})$ | $\vec{D}$ | [sendAwaken$(G, \vec{D})$] | $\vec{D}_{awaken}$ |
| $pause(G, h_p)$ | | [sendPause$(G, h_p)$] | $\vec{D}_{pause}$ |
| $newGroup(\vec{D}, f, h_t, h_e)$ | $\vec{D}$ | [group$(\vec{D}, f, h_t, h_e)$] | $\vec{D}_{newGroup}$ |
| $new\ runnable(f).start()$ | $\{thread : 1\}$ | [thread$(f)$] | $\vec{D}_{thread}$ |
| $malloc(n)$ | $\{memory : n\}$ | [] | $\vec{D}_{malloc}$ |
| $fileOpen(name)$ | $\{file : 1\}$ | [] | $\vec{D}_{fileOpen}$ |
| $thread.stop()$ | $\{thread : -1\}$ | [] | $\vec{D}_{threadstop}$ |
| $free(n)$ | $\{memory : -n\}$ | [] | $\vec{D}_{free}$ |
| $fileClose(file)$ | $\{file : -1\}$ | [] | $\vec{D}_{fileClose}$ |

Fig. 13. Operations.

primitive in the form of descriptor $\vec{D}$. The primitive generates a user message sequence [sendAwaken$(G, \vec{D})$], which will be handled by a transition rule described in this section. Finally, each specific implementation of the system must define the cost of executing such a primitive: $\vec{D}_{awaken}$. We note that the messages sendAwaken and group in Figure 13 are uninstantiated messages, since they refer to a resource descriptor $\vec{D}$ waiting to be substituted for a concrete resource value. Other primitives in the figure entail the consumption of resources, such as thread creation, memory allocation, and file opening, which respectively require thread, memory, and file resources to be available to proceed. The figure also contains their symmetric primitives, which release resources. For instance, closing a file makes one unit of the file resource available again, which we represent by a negative number.

We rely on this definition of primitives to identify the effect of thread execution on the system resources in a language-independent manner, which we capture with the relationship $\Rightarrow$:

$$T_1 \Rightarrow T_2 \text{ requesting } \vec{D} \text{ generating } M_1^* \text{ with administrative cost } \vec{D}_a.$$

It states that a thread with "state" $T_1$ evolves to a new "state" $T_2$, by executing some constructs, whose cumulative effect is to request a set of resources $\vec{D}$ and generate a sequence of user messages $M_1^*$ with a given administrative cost $\vec{D}_a$. When resources are released, following the convention of Figure 13, resources in $\vec{D}$ are negative. By convention, a thread that executes a *thread.stop*() instruction will result in a final state, which we note void, which marks the end of its lifetime.

5.2.4 *Configuration Transformers.* We use some *pseudostatements* such as *send*, *receive*, or table updates, which give an imperative look to the specification. Such a notation helps the reader understand how the specification could be implemented. Formally, such pseudostatements have a precise meaning: they act as configuration transformers and are defined as follows:

—Let $\Gamma$ be the function mapping groups to group configurations in a configuration $\langle \Gamma, k \rangle$. Then the expression $\Gamma(g) := V$ denotes the configuration $\langle \Gamma', k \rangle$, where $\Gamma(g_1) = \Gamma'(g_1)$ if $g_1 \neq g$, and $\Gamma'(g) = V$.

—Let $k$ be the set of message channels of a configuration $\langle \Gamma, k \rangle$. Then the expression *send*$(G_1, G_2, m)$ denotes the configuration $\langle \Gamma, k' \rangle$, with $k'(G_1, G_2) =$

$k(G_1, G_2) \uplus \{M\}$, and $k'(G_i, G_j) = k(G_i, G_j)$, $\forall (G_i, G_j) \neq (G_1, G_2)$, where the operator $\uplus$ denotes the union operator on bags.

—Let $k$ be the set of message channels of a configuration $\langle \Gamma, k \rangle$. Then the expression *receive*$(G_1, G_2, m)$ denotes the configuration $\langle \Gamma, k' \rangle$, with $k'(G_1, G_2) = k(G_1, G_2) \setminus \{M\}$, and $k'(G_i, G_j) = k(G_i, G_j)$, $\forall (G_i, G_j) \neq (G_1, G_2)$, where the operator $\setminus$ denotes the difference operator on bags.

5.2.5 *Transition Rules.* The rules which we are now going to define adopt the following syntax:

```
    $rule\_name(v_1, v_2, \ldots)$ :
    $condition_1(v_1, v_2, \ldots) \wedge condition_2(v_1, v_2, \ldots) \wedge \ldots$
$\rightarrow$ {
      $pseudo\ statement_1$;

      $pseudo\ statement_n$;
    }
```

A rule is identified by its name, and is parameterized by a number of variables. Some conditions can appear to the left-hand side of the arrow: these are guards that must be satisfied in order for the transition to be fireable. The right-hand side of the arrow denotes the configuration that is reached after transition: its value results from applying the configuration transformer obtained by composing all the pseudostatements to the configuration that satisfied the guard. From a concurrency viewpoint, we assume that the execution of a transition, that is, verification of guards and pseudostatements execution, is performed atomically.

Figures 14, 15, 16, and 17 contain the transition rules of our abstract machine, which we discuss in the rest of the section, by making explicit references to some of the rule features. Appendix A presents an example of a RAP program and its evaluation step by step using the machine transition rules. First, we consider Figure 14 concerned with a group's life cycle.

—*Note* 1. In rule *consumption*, we consider a thread able to evolve from state $T$ to $T'$, requesting resources specified by a descriptor $\vec{D}$, generating a set of user messages $M_1^*$, with an administrative cost $\vec{D}_a$. If the current resources are greater than the described resources $\vec{D}$, the administrative cost $\vec{D}_a$, and the cost of notification $\vec{D}_n$, then transition *consumption* is fireable. Such a condition, expressed as

$$\vec{R} \ominus \vec{D} = (\vec{R}_1, \vec{R}_D) \wedge \vec{R}_1 \ominus \vec{D}_a = (\vec{R}_2, \vec{R}_a) \wedge \vec{R}_2 \ominus \vec{D}_n \neq \perp,$$

means that we can subtract $\vec{D}$, $\vec{D}_a$, and $\vec{D}_n$ from $\vec{R}$. After transition, the sponsoring group will see its resources decremented by the request resources $\vec{R}_D$ and the administrative cost; the thread will evolve to its state $T'$, with the insurance that enough resources remain to emit a notification of cost $\vec{D}_n$; in other words, we have set aside some resources to be sure that we can raise a notification in the future. All the messages $M^*$ generated by the execution

```
$consumption(G, G^*, G_p, T, T', T^*, M^*, M_1^*, \vec{R}, \vec{D}_a, \vec{D}_n, \vec{D}, h_t, h_e)$ :
  $\Gamma(G) = \langle \mathsf{running}, \vec{R}, \{T\} \cup T^*, G^*, G_p, M^*, h_t, h_e \rangle$
  $\wedge\ T \Rrightarrow T'$ requesting $\vec{D}$ generating $M_1^*$ with admin cost $\vec{D}_a$
  $\wedge\ \vec{R} \ominus \vec{D} = (\vec{R}_1, \vec{R}_D)\ \wedge\ \vec{R}_1 \ominus \vec{D}_a = (\vec{R}_2, \vec{R}_a)\ \wedge\ \vec{R}_2 \ominus \vec{D}_n \neq \perp$        (Note 1)
  $\rightarrow$ {
      $if\ T' = \mathsf{void}$
      $then\ \Gamma(G) := \langle \mathsf{running}, \vec{R}_2, \{T'\} \cup T^*, G^*, G_p, M^* : M_1^*\{\vec{D}/\vec{R}_D\}, h_t, h_e \rangle$;
      $else\ \Gamma(G) := \langle \mathsf{running}, \vec{R}_2, T^*, G^*, G_p, M^* : M_1^*\{\vec{D}/\vec{R}_D\}, h_t, h_e \rangle$;
    }

$exhaustion(G, G^*, G_p, T, T', T^*, M^*, M_1^*, \vec{R}, \vec{R}_D, \vec{D}_a, \vec{D}, h_t, h_e)$ :
  $\Gamma(G) = \langle \mathsf{running}, \vec{R}, \{T\} \cup T^*, G^*, G_p, M^*, h_t, h_e \rangle$
  $\wedge\ T \Rrightarrow T'$ requesting $\vec{D}$ generating $M_1^*$ with admin cost $\vec{D}_a$
  $\wedge\ \vec{R} \ominus_1 \vec{D} \ominus_1 \vec{D}_a \ominus \vec{D}_n = \perp$        (Note 2)
  $\rightarrow$ {
      $\Gamma(G) := \langle \mathsf{exhausted}, \vec{0}, \{T\} \cup T^*, G^*, G_p, M^*, h_t, h_e \rangle$;
      $send(G, G_p, \mathsf{exhaustion}(G, \vec{R} \ominus_1 \vec{D}_n, \vec{D}))$;
    }

$termination(G, G_p, \vec{R}, \vec{R}_t, h_t, h_e)$ :
  $\Gamma(G) = \langle \mathsf{running}, \vec{R}, \emptyset, \emptyset, G_p, \emptyset, h_t, h_e \rangle$        (Note 3)
  $\rightarrow$ {
      $\Gamma(G) := \langle \mathsf{terminated}, \vec{0}, \emptyset, \emptyset, G_p, \emptyset, h_t, h_e \rangle$;
      $send(G, G_p, \mathsf{termination}(G, \vec{R} \ominus_1 \vec{D}_n))$;
  }

$message(G, G^*, G_1, G_p, T^*, M^*, M, \vec{R}, s, h_t, h_e)$ :
  $\Gamma(G) = \langle s, \vec{R}, T^*, G^*, G_p, M^*, h_t, h_e \rangle\ \wedge\ \langle M \rangle \in k(G_1, G)$        (Note 4)
  $\rightarrow$ {
      $receive(G_1, G, M)$;
      $\Gamma(G) := \langle s, \vec{R}, T^*, G^*, G_p, [M^* : M], h_t, h_e \rangle$;
    }
```

Fig. 14. Group life cycle.

of the thread are concatenated to the group's message list. Additionally, references to the descriptors $\vec{D}$ in the set of messages $M_1^*$ are replaced by the exact resources $\vec{R}_D$ they denote, using the conversion function to system messages $M_1^*\{\vec{D}/\vec{R}_D\}$. We see here that such a conversion of descriptors into resource values is under control of the system, which has checked that enough resources existed with the guard of rule *makeGroup* before allowing the transition to be fired.

—*Note* 2. The preconditions of rule *exhaustion* are similar to those of rule *consumption*, except that the amount of requested resources (plus the administrative cost) is smaller than the amount of resources available in the

```
$makeGroup(G, G^*, G_p, G_1, T^*, M^*, \vec{R}, \vec{R}_1, f, h_t, h_e, h'_t, h'_e)$ :
  $\Gamma(G) = \langle \mathsf{running}, \vec{R}, T^*, G^*, G_p, [\mathsf{group}(\vec{R}_1, f, h'_t, h'_e) : M^*], h_t, h_e \rangle$        (Note 5)
  $\wedge\ \Gamma(G_1) = \bot \quad \wedge \quad \vec{R}_1 \ominus \vec{D}_n \neq \bot$
  $\rightarrow$ {
      $\Gamma(G) := \langle \mathsf{running}, \vec{R}, T^*, \{G_1\} \cup G^*, G_p, M^*, h_t, h_e \rangle;$
      $\Gamma(G_1) := \langle \mathsf{running}, \vec{R}_1, \emptyset, \emptyset, G, [\mathsf{thread}(f)], h'_t, h'_e \rangle;$
  }

$makeThread(G, G^*, G_p, T^*, M^*, \vec{R}, f, h_t, h_e)$ :
  $\Gamma(G) = \langle \mathsf{running}, \vec{R}, T^*, G^*, G_p, [\mathsf{thread}(f) : M^*], h_t, h_e \rangle$        (Note 6)
  $\rightarrow$ {
      $\Gamma(G) := \langle \mathsf{running}, \vec{R}, \{f()\} \cup T^*, G^*, G_p, M^*, h_t, h_e \rangle;$
  }

$notifyTermination(G, G^*, G_p, G_t, T^*, M^*, \vec{R}, \vec{R}_t, s, h_t, h_e)$ :
  $\Gamma(G) = \langle s, \vec{R}, T^*, \{G_t\} \cup G^*, G_p, [\mathsf{termination}(G_t, \vec{R}_t) : M^*], h_t, h_e \rangle$        (Note 7)
  $\rightarrow$ {
      $\Gamma(G) := \langle \mathsf{running}, \vec{R} \oplus \vec{R}_t, T^*, G^*, G_p, [M^* : M_n], h_t, h_e \rangle;$
       where $M_n = \mathsf{thread}(\Gamma(G_t).h_t(G_t, toDesc(\vec{R}_t)))$
  }

$notifyExhaustion(G, G^*, G_p, G_e, T^*, M^*, \vec{R}, \vec{R}_e, \vec{D}_e, s, h_t, h_e)$ :
  $\Gamma(G) = \langle s, \vec{R}, T^*, G^*, G_p, [\mathsf{exhaustion}(G_e, \vec{R}_e, \vec{D}_e) : M^*], h_t, h_e \rangle$        (Note 8)
  $\wedge\ s \neq \mathsf{terminated}$
  $\rightarrow$ {
      $\Gamma(G) := \langle \mathsf{running}, \vec{R} \oplus \vec{R}_e, T^*, G^*, G_p, [M^* : M_n], h_t, h_e \rangle;$
       where $M_n = \mathsf{thread}(\Gamma(G_e).h_e(G_e, \vec{D}_e, toDesc(\vec{R}_e)))$
  }
```

Fig. 15. Message processing (1).

sponsoring group $G$. Such an event triggers the exhaustion of $G$, marked by the change of its status to exhausted and the resetting of its resources; any other data of $G$ remains unchanged. Additionally, we send an exhaustion message to the parent of $G$, which contains a reference to $G$ (the exhausted group), the resources it had left, and a descriptor indicating the amount of resources that we failed to grant. The latter descriptor can be useful to the handler to determine the amount of resources that should be transferred back to the group. When the thread being executed reaches its final state void, it is removed from the group list of threads.

—*Note* 3. Termination of a group is defined as the simultaneous absence of threads in the group, of messages in the queue of messages, and of subgroups. A termination message is sent to the parent of $G$, which contains a reference to the terminated group $G$ and its remaining resources. Such a termination property can be observed locally, that is, at a given location, and therefore is also tractable for a distributed environment.

```
$sendAwaken(G, G^*, G_p, G_1, T^*, M^*, \vec{R}, \vec{R}_1, h_t, h_e)$ :
  $\Gamma(G) = \langle \mathsf{running}, \vec{R}, T^*, G^*, G_p, [\mathsf{sendAwaken}(G_1, \vec{R}_1) : M^*], h_t, h_e \rangle$        (Note 9)
  $\rightarrow$ {
      $\Gamma(G) := \langle \mathsf{running}, \vec{R}, T^*, G^*, G_p, M^*, h_t, h_e \rangle$;
      $send(G, G_1, \mathsf{awaken}(G, \vec{R}_1))$;
    }

$awaken(G, G^*, G_p, G_1, T^*, M^*, \vec{R}, \vec{R}_1, s, h_t, h_e)$ :
  $\Gamma(G) = \langle s, \vec{R}, T^*, G^*, G_p, [\mathsf{awaken}(G_1, \vec{R}_1) : M^*], h_t, h_e \rangle$        (Note 10)
  $\rightarrow$ {
      $if\ (s \neq \mathsf{terminated})\ then$
        $\Gamma(G) := \langle \mathsf{running}, \vec{R} \oplus \vec{R}_1, T^*, G^*, G_p, M^*, h_t, h_e \rangle$;
      $else$
        $send(\mathsf{awakenFailure}(G, G_1, \vec{R}_1))$;
    }

$awakenFailure(G, G^*, G_p, G_1, T^*, M^*, \vec{R}, \vec{R}_1, s, h_t, h_e)$ :
  $\Gamma(G) = \langle s, \vec{R}, T^*, G^*, G_p, [\mathsf{awakenFailure}(G_1, \vec{R}_1) : M^*], h_t, h_e \rangle$        (Note 11)
  $\rightarrow$ {
      $if\ (s \neq \mathsf{terminated})\ then$
        $\Gamma(G) := \langle \mathsf{running}, \vec{R} \oplus \vec{R}_1, T^*, G^*, G_p, M^*, h_t, h_e \rangle$;
      $else$
        $//return\ resource\ \vec{R}_1\ to\ resource\ sink$
    }
```

Fig. 16. Message processing (2): group awakening.

—*Note* 4. Rule *message* is concerned with the asynchronous handling of messages in transit between two groups. Any message in a communication channel from group $G_1$ to group $G$ is added at the end of $G$'s queue of messages. While order of messages is not necessarily preserved by communication channels between groups, messages are handled in a strictly ordered manner by groups.

We now examine the transitions of Figure 15.

—*Note* 5. User's programs can request new groups to be created by providing a resource descriptor, a user function, and notification handlers. The resource descriptor, specified by the group creation primitive in Figure 13, is converted into an actual resource value (cf. Note 1). A new group $G_1$ is created, with the current group $G$ defined as its parent. A thread message is also sent, referring to the user code to be executed. We require the amount of resources allocated to a new group to be greater than $\vec{D}_n$, the cost of notification, so as to ensure that enough resource exist to be able to support the cost of raising a notification.

—*Note* 6. The processing of a message requesting the creation of a thread results in a new thread appearing in the group, which begins the execution of the

```
$sendPause(G, G^*, G_p, G_1, T^*, M^*, \vec{R}, h_t, h_e, h_p)$ :
  $\Gamma(G) = \langle \mathsf{running}, \vec{R}, T^*, G^*, G_p, [\mathsf{sendPause}(G_1, h_p) : M^*], h_t, h_e \rangle$        (Note 12)
  $\rightarrow$ {
      $\Gamma(G) := \langle \mathsf{running}, \vec{R}, T^*, G^*, G_p, M^*, h_t, h_e \rangle$;
      $send(G, G_1, \mathsf{pause}(G, G_1, G_1, h_p))$;
    }

$pause(G, G^*, G_p, G_1, G_t, T^*, M^*, \vec{R}, \vec{R}_p, s, h_t, h_e, h_p)$ :
  $\Gamma(G) = \langle s, \vec{R}, T^*, G^*, G_p, [\mathsf{pause}(G_1, G, G_t, h_p) : M^*], h_t, h_e \rangle$        (Note 13)
  $\rightarrow$ {
      $if\ (s \neq \mathsf{terminated})\ then$
        $\Gamma(G) := \langle \mathsf{exhausted}, \vec{0}, T^*, G^*, G_p, M^*, h_t, h_e \rangle$;
        $if\ (\vec{R} \ominus \vec{R}_p \neq \perp)\ then\ send(G, G_1, \mathsf{awaken}(G, G_1, \vec{R} \ominus \vec{R}_p))$;
        $if\ (G \neq G_t)\ then\ send(G, G_p, \mathsf{paused}(G, \vec{R}, h_p))$;
        $for\ all\ g\ \in\ G^*\ send(G, g, \mathsf{pause}(G_1, g, G_t, h_p))$;
    }

$notifyPause(G, G^*, G_p, G_t, T^*, M^*, \vec{R}, \vec{R}_t, s, h_t, h_e)$ :
  $\Gamma(G) = \langle s, \vec{R}, T^*, G^*, G_p, [\mathsf{paused}(G_t, \vec{R}_t, h_p) : M^*], h_t, h_e \rangle$        (Note 14)
  $\rightarrow$ {
      $\Gamma(G) := \langle s, \vec{R}, T^*, G^*, G_p, [M^* : \mathsf{thread}(h_p(G_t, toDesc(\vec{R}_t)))], h_t, h_e \rangle$;
    }
```

Fig. 17. Message processing (3): group pausing.

user function. The creation of a thread is conditional to the group having the necessary resources to create one thread, as specified by Figure 13, which imposes that one thread-resource is available for consumption,

—*Note* 7. A termination message issued by rule *termination* executed by group $G_t$ results in the creation of a new thread that activates the termination handler of group $G_t$. The termination handler expects the terminated group $G_t$ and resource descriptors denoting the amount of resources that remained in $G_t$ when termination was observed; such resource descriptors are obtained by converting the resource values $\vec{R}_t$.

—*Note* 8. Handling an exhaustion message is similar to handling a termination message. Instead, the exhaustion handler is called; it requires a description of the remaining resources and a description of the resource requested.

In the following, we comment on Figures 16 and 17.

—*Note* 9. The operation *awaken* of Figure 13 requests a message awaken to be sent to a group $G_1$ with an amount of resources $\vec{R}_1$. Such a message is simply added to the appropriate communication channels.

—*Note* 10. Rule *awaken* handles an incoming awaken message by adding the specified amount of resources $\vec{R}_1$ to the resources of the receiving group, provided it is not terminated; as the group is given the running status after transition, the awaken message is able to awaken exhausted groups. If the receiving group is terminated, we avoid losing the received resources, by sending them back to the emitter using an awakenFailure message.

—*Note* 11. Rule *awakenFailure* handles awakenFailure messages marking the failure to awaken a group. The resources are simply returned to the receiving group if it has not terminated. If it has terminated, then both the group that initiated the awaken message and the target group for the message have terminated: the resources are therefore "recycled" by the system.

—*Note* 12. The operation *pause* of Figure 13 requests a message pause to be sent to a group $G_1$. Rule *sendPause* of Figure 17 handles this request by sending such a message to the relevant communication channel. A pause message is composed of four arguments: the group that initiates the pause action, the current group to be paused, the targeted group to be paused and a pause handler. Initially, the current group and targeted group are identical: the former changes as the pause message is forwarded in the hierarchy of groups rooted at the targeted group.

—*Note* 13. Rule *pause* is triggered by the arrival of a pause message. The receiving group changes to a status exhausted and all its resources are sent to the emitter of the pause message using an awaken message. If the group $G$ is not the group initially targeted by the pause message (but is one of its direct or indirect child), a paused notification message is sent to its parent. Additionally, for each of its children, a pause message is sent, with the same targeted group $G_t$. No action is performed if the receiving group has already terminated. In a first approximation, Figure 7 defined the cost of pausing as $K_p$; rule *pause* refines this cost by taking into account that pausing a hierarchy is an operation that is propertional to the number of groups contained in the hierarchy; therefore, an administrative cost $\vec{R}_p$ is charged for each group encountered by rule *pause*.

—*Note* 14. Rule *notifyPause* handles a paused message and creates a thread executing the pause handler. It is implicit that the current group $G$ is not terminated because the queue of messages for $G$ is not empty at the beginning of the transition.

5.2.6 *Discussion.* The rules presented in this section do not make any assumption about the memory model: all communications between groups take place over communication channels, and therefore can be hosted in shared and distributed memories. The newRemoteGroup primitive, like newGroup, creates a group message, which has to be sent to a remote host, using the existing communication channel mechanism.

In this section, we have sought to present a simple semantics highlighting the key principles of resource transfers. In particular, when the awaken primitive is called on a terminated group, an awakenFailure message transfers resources back to the group that sponsored the invocation of awaken. Likewise, a pause primitive results into resources being transferred back to the group that sponsored its execution. In both cases, resources could be leaked if the group initiating the operation terminates before the resources are returned. It is therefore the programmer's responsibility to ensure these groups do not terminate prematurely. In order to achieve this in a reliable manner, it may be desirable to extend the existing semantics with acknowledgment messages marking a successful awaken or pause operation. However, we did not introduce such messages to avoid cluttering the semantics with non-resource-specific messages. Additionally, handlers for such messages would have to interface with the language the system is embedded in, with a view to unblock the corresponding primitives: such interfacing is difficult to express in a language independent manner.

Our RAP model also supports time in two different ways. First, time can be defined as a deadline by which a computation must have completed its execution; otherwise its group becomes suspended. This model of time assumes that all locations involved in a computation are synchronised, say using a protocol such as NTP (`www.ntp.org`); deadlines will be shared by all locations, up to the quality of synchronization between the different clocks involved. Alternatively, RAP can also support a notion of "tick" [Haynes and Friedman 1987], which is an abstract notion of cost associated with each instruction of a programming language; it is independent of the hardware the machine is operating on, and really represents a semantic "unit of computing."

## 6. RAP MODEL IMPLEMENTATION

In this section, we discuss a prototype implementation of our Resource Aware Programming model in Java. The prototype implementation is a proof of concept aiming to demonstrate the feasibility of a RAP implementation; in particular, it does *not* require a modified JVM, but we discuss in the latter part of this section how our implementation could benefit from monitoring primitives as in the Aroma VM [Groth and Suri 2002] or JVM extensions [JSR-121 2003].

Our implementation supports both a shared memory and a distributed memory. While a single API has been defined, we have provided implementations of communication channels that operate in both memory models; the distributed implementation of communication channels relies of Java RMI [Sun MicroSystems 1996].

### 6.1 Overview

Figure 18 presents the key elements of our implementation of Resource Aware Programming in Java. The RAP package introduces new notions of group (`RAPGroup`) and thread (`RAPThread`). Each `RAPGroup` contains explicit references of the `RAPThreads` it sponsors, to all of its subgroups, to its parent group, and to the set of resources it contains. For each group, we find a synchronous communication channel and a thread, which we call a *communication thread*, processing messages coming over the communication channel. While the semantics of Section 5 does not require communication channels to be synchronous, we found this was a useful property in our implementation as the communication thread (combined with the communication channel) was then able to execute its code in a group-critical section. The communication thread processes incoming messages according to the semantics of Figures 15, 16, and 17.

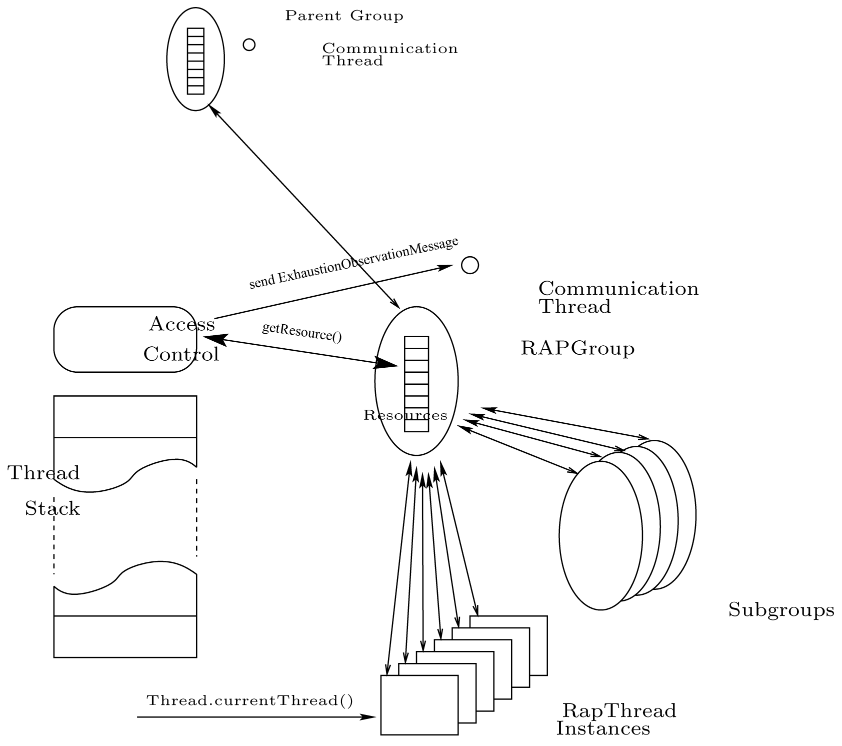

Fig. 18. Implementation.

Resource aware primitives trigger the invocation of the Java security manager—in this case, a RAP-aware security manager. In addition to the Java stack inspection, the RAP manager ensures that the group sponsoring the execution of the current primitive owns a sufficient amount of resources. The explicit representation of resources can be obtained by accessing the current thread, then its sponsoring group, and finally its associated resources. Should resources be insufficient, a message is sent to the group's communication channel, indicating that resource exhaustion has been observed; in a same atomic operation (due to the synchronous nature of the communication channel), the current thread then suspends itself, by a call to `wait()` on the group object, and all the threads in the group are suspended by explicitly calling `suspend()` on each of them. When some resources are transferred to a group (e.g., with an awaken message), all threads waiting for the group monitor and all suspended threads in the group are resumed.

## 6.2 Key Classes and Interfaces

The RAP package [Moreau and Queinnec 2002b] presents a set of public interfaces and a very small set of public classes, which we summarize in this section. We provide interfaces for communication channels, communication channel factories, exhaustion and termination handlers, messages, and resource descriptors. We have minimized the number of classes visible by the programmer using the RAP package; only four important classes are required.

—*Group*. This class contains static factory methods for threads, local and remote groups, and communication channel factories. Additionally, it contains instance methods `pause` and `awaken` following the semantics of Section 5.

—*AccessManager*. An implementation of the resource-aware security manager described in the previous section.

—*Descriptors*. This class contains several factory methods for the descriptors we discussed in Section 5.1.2.

—*Server*. A class able to start up and register a RAP-aware platform where groups can be migrated to.

### 6.3 Extensions to the Semantics

In our implementation, we have considered some variants of the semantics. From an implementation viewpoint, it may be inefficient to create a thread in the parent group every time an exhaustion or termination exhaustion has to be run. As an alternative, one can create a "listener thread," sponsored by the parent group, listening on a specified communication channel. Instead of specifying handlers when creating groups, we can now specify communication channels to which notifications much be sent. The listener thread will then perform a suitable action on exhaustion and termination messages. Such a listener thread and notification channels were used in our implementation of remote groups to ensure that the notification handler was executed on the same host as the parent group.

The semantics of Section 5 forces all threads of a group to be suspended when a resource exhaustion is observed. A common usage pattern of groups is to program them to monitor the usage of resources: a group is created with an initial amount of $u$ units of the resources to be monitored, with an exhaustion handler that counts the number of times it has been invoked and awakens the exhausted group with $u$ more units of resources. Suspending all threads when resources are exhausted and resuming them all when more resources are provided is very inefficient. Instead, one prefers an asynchronous notification to be propagated, with only the thread that caused the notification to be suspended.

### 6.4 Support for JVM Extensions

Our current implementation uses a customized Java security manager to intercept calls to resource-oriented primitives. The key advantage of this approach is its compatibility with the JDK1.4, which therefore guarantees that it can run on existing JVMs. The disadvantage is that *not all* resource-oriented primitives result in a call to the security manager. Furthermore, it should not be regarded as an efficient implementation of RAP since it relies on the onerous stack inspection mechanism.

For instance, some Java API calls such as closing files do not invoke the security manager, and therefore are not "trapped" by our implementation. It essentially means that we are able to control the total maximum number of files an application can open, but not the maximum number of files opened at any one time. Offering a new API with methods such as `close`, but invoking RAP when called, would allow us to support the latter facility. However, this is not a satisfactory solution, since RAP is meant to be able to control *any* code it sponsors, without the code being rewritten for this purpose. Another solution would be to provide a new runtime library that would be RAP aware.

The usage of some resources is currently not trapped by RAP, such as rates (number of bytes stored per second, or number of packets sent per second). In order to integrate these, we need some support at the level of the JVM, for instance, as provided by the Aroma VM [Groth and Suri 2002]. As long as the extended API provides a callback mechanism, and it allows us to set the level at which notifications should be triggered, our implementation can be extended to support such basic monitoring extensions; we would also require the API to identify the thread that triggered the notification, so that, using the implementation of Figure 18, we can find the group the exhaustion was observed in.

Another resource whose usage is difficult to monitor is memory. The JSR 121 [JSR-121 2003] offers a beginning of solution to this end, since it introduces the explicit notion of memory space in the form of "isolate." Still, monitoring usage of memory in the presence of a garbage collector is not a trivial matter: while allocation can be seen as resource consumption, its counterpart, memory reclaiming, will have to return resources to a group, but without a straightforward answer to the problem of which group it will have to be returned to. Indeed, the group that allocated a recycled object may no longer exist. On the other hand, returning recycled memory space to the system would penalize programs that keep a bounded reachable memory, and rely on garbage collection to recycle garbage data. In fact, these options describe policies for managing memory, which our framework could implement if a virtual machine makes available the sensors necessary to observe memory allocation and deallocation.

## 7. RELATED WORK

A number of existing systems support resources accounting. Telescript [White 1996] featured "clicks" that are a unit of charge deducted from an agent's account. JRes and JKernel [Czajkowski and von Eicken 1998] support accounting of memory, CPU, and network usage. Nomads [Suri et al. 2000], through a modified JVM, supports strong migration of agents and resource accounting; in particular, a limit file is able to specify both quantity limits (such as disk space or memory) but also rate limits (such as disk usage rate and transfer rate). Java Seal2 [Villazón and Binder 2001] is an extension of Java Seal [Bryce and Vitek 2001] which provides portable resource accounting. A notion of process is introduced in KaffeOS [Back et al. 2000], a modified JVM, which allows resource control in a fine manner.

All these systems are complementary to the Resource Aware Programming framework: indeed, they implement the accounting of resource usage and they raise a notification when a resource quota is reached, while RAP provides the mechanism to transfer (i.e., add or remove) resource dynamically between distributed computations. RAP also provides a programming model to support the execution of asynchronous notifications, under the control of the same resource management system. Notifications are therefore becoming a key programming technique that can be made available to the programmer. Additionally, our model also supports resources that do not necessarily have a physical reality, but can be defined in an application-specific setting.

Java Seal [Bryce and Vitek 2001] is able to migrate nested seals. In our proposed model, we have not considered migration of groups. The primitive newRemoteGroup is able to create a group remotely. This primitive in essence offers a *weak* form of migration (where threads of control are not migrated, but only data is). Thus, if a newly created remote group itself called a remote group, a series of nested groups would be created, none of them directly sponsoring a computation, but acting as a parent for its nested group. There is here an opportunity for short-cut pointers [Moreau 2001a], and a tree rerooting techique [Moreau 2001b] could be used in addition to flatten the group hierarchy.

Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] is not directly related to RAP although AOP may be used to insert RAP-related instructions within a program. AOP may be viewed as a kind of macroprocessor able to transform the abstract syntax tree of a program into another instrumented program. The transformations are usually expressed as syntactic rules. Many interesting features may be woven in a program with these rules, including traces, replacement and bug fixing.

RAP, on the other hand, only deals with resource consumption. However, one should distinguish between the RAP user and the RAP designer. The RAP user programs within a framework that takes care automatically of resources and does not need to be aware of them. Sometimes the code is suspended and resumed according to its resource greediness. The RAP user programs implicitly with resources and only needs to cope with resource exhaustion.

The RAP designer builds the framework, sets resources up, and manages them explicitly. The specific RAP instructions that consume or release resources should be set on every followed path; this is where AOP may be used to insert consistently these instructions. For instance, the method to close a file, discussed above, could be extended by AOP to invoke the necessary RAP related operation.

The Grid is a large-scale computer system that is capable of coordinating resources that are not subject to centralized control, while using standard, open, general-purpose protocols and interfaces, and delivering nontrivial qualities of service [Foster 2002]. As part of the endeavor to define the Grid, a service-oriented approach has been adopted by which computational and storage units, networks, programs, and databases are all represented by services [Foster et al. 2002]. In this context, WS-Agreement [Czajkowski et al. 2004], a standards proposal from the Global Grid Forum, introduces the notion of agreement negotiation, which captures the idea of dynamically adjusting policies that affect service behavior without necessarily exposing the details required to enact or enforce the policies. An agreement captures a mutual understanding of (future) service provider behavior. The scope of WS-agreement negotiation includes the resources we discussed here in the context of RAP, and in particular includes access policies, resource consumption, or performance goals (i.e., provisioning of services). Agreements in WS-Agreement must be defined as policy elements of the Web services policy framework WS-Policy [Box et al. 2003]. WS-Policy is a language allowing a set of assertions to be associated to a subject; a *policy assertion* conveys a requirement, preference, or capability. Examples of security assertions for Web services are defined in WS-SecurityPolicy [Della-Libera et al. 2002] and include requirements for security tokens, integrity, confidentiality, visibility, header information, and message age. An alternative specification language for Web services is the Web service Level Agreement language (WSLA) [Ludwig et al. 2003], which is capable of defining assertions regarding agreed to guarantees that a service provider has to meet, regarding resources such as response time and throughput. The language can also specify measures to be taken when a service fails to meet the asserted guarantees, such as notification of the customer.

Policy languages have been the focus of much attention lately: a number of authors have proposed services and tools for the specification, management, conflict resolution, and enforcement of policies within an organizational policy domain. As an illustration, the KAoS [Johnson et al. 2003] and Ponder [Damianou et al. 2001] policy languages distinguish between *authorizations*, that is, constraints that permit or forbid some actions, and *obligations*, that is, constraints that require some action to be performed. Related to this work, KAoS policies have been used to specify rates and resolution of image streaming [Suri et al. 2003]. A future area of research is to understand how a policy language for resources could be enforced by an interpreter extended with RAP capabilities for managing resources. In particular, soundness and completeness of policy-based resource specifications with respect to RAP-based resource management are interesting properties to investigate: soundness would prove that resource-related behavior enforceable at runtime is specifiable by a policy, whereas completeness would establish that all properties specifiable by such a policy language are enforceable by the RAP-based runtime.

## 8. CONCLUSION

In this article, we have presented a language-independent model of programming for the monitoring and management of resources. It relies on a resource algebra composed of two operations on resources, addition and subtraction, and a message-passing operational semantics that supports a shared memory and a distributed memory view of the model.

This framework is able to create "resource-aware sandboxes" which allow untrusted code to be loaded dynamically and executed, while the monitoring and management of resources can be programmed in a uniform manner. The framework allows the policies to change over time, hence supporting better customization of the sandbox to the prevailing execution circumstances.

As far as future work is concerned, a number of issues remain to be investigated. Our prototype implementation relies on the Java Security Manager

```
public class Example {
    static Runnable r = new Runnable () {
            public void run () {
                try {
                    new Socket("www.ecs.soton.ac.uk", 80);
                } catch (Exception e) {
                    // handler
                }
            }};

    public static void main (String [] arg) {
        AccessManager am = new AccessManager(true);
        System.setSecurityManager(am);

        ResourceDescriptors rv =
            Descriptors.newResourceDescriptors(new ResourceDescriptor [] {});
        Group.initialGroup(r,Group.newSharedMemoryChannelFactory(),rv);

        // wait
    }
}
```

Fig. 19. An example.

to intercept resource-related calls; we have discussed the limitations of this approach both in terms of efficiency and the kind of calls that could be intercepted. Our framework would benefit fully of the monitoring and notification mechanisms of some extended virtual machines, such as Aroma VM [Groth and Suri 2002], to better control resources of the system. Ultimately, end-users do not want to program their resource management policies, but they want to specify them, in an abstract resource-management policy language. Work is required in order to define and enforce such a policy language, using RAP primitives. A more fundamental study of the expressiveness of the policy language and the enforcer is required in order to ensure that the policies enforced by the enforcer are the ones specified by the language.

## APPENDIX: EXECUTION TRACE

In this section, we illustrate the abstract machine by showing the transitions it would perform on a sample program. The program is written in Java and makes uses of our implementation of RAP. An excerpt of the source code appears in Figure 19.

In the class `Example`, a "runnable" object is defined to open a socket to port 80 of an http server. In the `main` method, after setting a RAP-aware security manager, a group is created to run the runnable object with an initially empty set of resources. The group that we create is provided with an exhaustion handler (not shown in the code), which always grants the resources requested. Formally, the handler is written as follows, it awakes the exhausted group with the resources that were available ($D_a$) and the requested resources ($D_r$):

$$h = \lambda G D_a D_r.awaken(G, D_a + D_r).$$

$$\begin{array}{lll}
c_{init} & = & \{\{G_\perp \rightarrow gc_0^\perp\}, k_{init}\} \\
& & \text{with } gc_0^\perp = \langle \mathsf{running}, \vec{R}_\perp, \emptyset, \emptyset, \perp, [M_{init}], h_{t\perp}, h_{e\perp}\rangle \\[1em]
\rightarrow^{makeGroup} & & \{\{G_\perp \rightarrow gc_1^\perp, G \rightarrow gc_1\}, k_{init}\} \\
& & \text{with } gc_1 = \langle \mathsf{running}, [], \{\mathtt{r}\}, \emptyset, G_\perp, [], h_{tinit}, h_{einit}\rangle \\
& & \text{and } gc_1^\perp = \langle \mathsf{running}, \vec{R}_\perp, \emptyset, \{G\}, \perp, [], h_{t\perp}, h_{e\perp}\rangle \\[1em]
\rightarrow^{consumption} & & \{\{G_\perp \rightarrow gc_1^\perp, G \rightarrow gc_2\}, k_{init}\} \\
& & \text{with } gc_2 = \langle \mathsf{running}, [], \{\mathtt{newSocket}(...)\}, \emptyset, G_\perp, [], h_{tinit}, h_{einit}\rangle \\[1em]
\rightarrow^{exhaustion} & & \{\{G_\perp \rightarrow gc_1^\perp, G \rightarrow gc_3\}, k_{init}[(G, G_\perp, \mathsf{exhaustion}(G, [], \vec{D}))]\} \\
& & \text{with } gc_3 = \langle \mathsf{exhausted}, [], \{\mathtt{newSocket}(...)\}, \emptyset, G_\perp, [], h_{tinit}, h_{einit}\rangle \\[1em]
\rightarrow^{message} & & \{\{G_\perp \rightarrow gc_2^\perp, G \rightarrow gc_3\}, k_{init}\} \\
& & \text{with } gc_2^\perp = \langle \mathsf{running}, \vec{R}_\perp, \emptyset, \{G\}, \perp, [\mathsf{exhaustion}(G, [], \vec{D})], h_{t\perp}, h_{e\perp}\rangle \\[1em]
\rightarrow^{notifyExhaustion} & & \{\{G_\perp \rightarrow gc_3^\perp, G \rightarrow gc_3\}, k_{init}\} \\
& & \text{with } gc_3^\perp = \langle \mathsf{running}, \vec{R}_\perp, \{h_{einit}(G, \vec{D}, [])\}, \{G\}, \perp, [], h_{t\perp}, h_{e\perp}\rangle \\[1em]
\rightarrow^{consumption} & & \{\{G_\perp \rightarrow gc_4^\perp, G \rightarrow gc_3\}, k_{init}\} \\
& & \text{with } gc_4^\perp = \langle \mathsf{running}, \vec{R}_\perp, \{void\}, \{G\}, \perp, [\mathsf{sendAwaken}(G, \vec{R})], h_{t\perp}, h_{e\perp}\rangle \\[1em]
\rightarrow^{sendAwaken} & & \{\{G_\perp \rightarrow gc_5^\perp, G \rightarrow gc_3\}, k_{init}[(G_\perp, G, \mathsf{awaken}(G, \vec{R}))]\} \\
& & \text{with } gc_5^\perp = \langle \mathsf{running}, \vec{R}_\perp, \{void\}, \{G\}, \perp, [], h_{t\perp}, h_{e\perp}\rangle \\[1em]
\rightarrow^{message} & & \{\{G_\perp \rightarrow gc_5^\perp, G \rightarrow gc_4\}, k_{init}\} \\
& & \text{with } gc_4 = \langle \mathsf{exhausted}, [], \{\mathtt{newSocket}(...)\}, \emptyset, G_\perp, [\mathsf{awaken}(G, \vec{R})], h_{tinit}, h_{einit}\rangle \\[1em]
\rightarrow^{awaken} & & \{\{G_\perp \rightarrow gc_5^\perp, G \rightarrow gc_5\}, k_{init}\} \\
& & \text{with } gc_5 = \langle \mathsf{running}, \vec{R}, \{\mathtt{newSocket}(...)\}, \emptyset, G_\perp, [], h_{tinit}, h_{einit}\rangle
\end{array}$$

Fig. 20. Transitions of the RAP abstract machine.

Referring to definition of the initial configuration in Figure 12, the Java code of Figure 19 is specifying the following values for the initial message $M_{init}$:

$$\begin{array}{rcl}
\vec{R}_{init} & = & [], \\
f_{init} & = & \textit{runnable object } \mathtt{r}, \\
h_{einit} & = & \lambda G D_a D_r.awaken(G, D_a + D_r), \\
h_{tinit} & : & \textit{not discussed in this example}.
\end{array}$$

In order to simplify the presentation, we assume administrative costs are zero, and that the root group $G_\perp$ contains an infinite set of resources $\langle \mathsf{AllV} \rangle$. Figure 20 shows the transitions of the RAP abstract machine for the example shown in Figure 19. We start the execution with the initial configuration. The initial message $M_{init}$ can be processed by rule *makeGroup*, which results in a new group $G$ being created, with one thread consisting of the invocation of the runnable object `r`. Its execution sets up an exception handler (rule *consumption*) to finally arrive at the socket creation primitive.

Experience with running this code shows that the JVM requires several resources to open a socket, including a file resource and multiple permissions. Therefore, we symbolically represent by $\vec{D}$ the resources requested by this operation. Since these resources are not available in $G$, an exhaustion is observed. An exhaustion message is propagated to group $G_{\perp}$, which can start a thread to execute $G$'s exhaustion handler (rule *notifyExhaustion*). The handler directly invokes the *awaken* operation. Since $G_{\perp}$ has an infinite set of resources, the descriptor $\vec{D}$ can be substituted for values $\vec{V}$ inside an sendAwaken message. Its processing by rule *sendAwaken* results into an awaken message being sent to $G$. When received, the group can be awakened, its status is set to running, and execution can continue.

## REFERENCES

Back, G., Hsieh, W. C., and Lepreau, J. 2000. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation* (San Diego, CA). Usenix Association, Berkeley, CA.

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D. 2003. Web services architecture. W3C Working Draft. World Wide Web Consortium (W. C.). Web site: `www.w3.org`.

Box, D., Curbera, F., Hondo, M., Kaler, C., Langworthy, D., Nadalin, A., Nagaratnam, N., Nottingham, M., von Riegen, C., and Shewchuk, J. 2003. Web services policy framework (ws-policy). Available online at `http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-policy.asp`.

Bryce, C. and Vitek, J. 2001. The JavaSeal mobile agent kernel. *Auton. Agents Multi-Agent Syst. 4*, 359–384.

Czajkowski, G. and von Eicken, T. 1998. JRes: A resource accounting interface for Java. In *Proceedings of ACM OOPSLA Conference* (Vancouver, BC, Canada). ACM Press, New York, NY.

Czajkowski, K., Dan, A., an S. Tuecke, J. R., and Xu, M. 2004. Agreement-based service management (ws-agreement). Tech. rep. Global Grid Forum, Lemont, IL. Web site: `www.gridforum.org`.

Damianou, N., Dulay, N., Lupu, E., and Sloman, M. 2001. The ponder policy specification language. In *Workshop on Poicies for Distributed Systems and Networks (POLICY 2001)*. Lecture Notes in Computer Science, vol. 1995. Springer-Verlag, Berlin, Germany.

Della-Libera, G., Hallam-Baker, P., Hondo, M., Janczuk, T., Kaler, C., Maruyama, H., Nagaratnam, N., Nash, A., Philpott, R., Prafullchandra, H., Shewchuk, J., Waingold, E., and Zolfonoon, R. 2002. Web services security policy (ws-securitypolicy). Web site: `http://www.ibm.com/developerworks/library/ws-secpol/index.html`.

Foster, I. 2002. What is the grid? a three point checklist. Web site: `http://www-fp.mcs.anl.gov/~foster/`.

Foster, I., Kesselman, C., Nick, J. M., and Tuecke, S. 2002. The physiology of the Grid—an open grid services architecture for distributed systems integration. Tech. rep. Argonne National Laboratory, Argonne, IL.

Fuggetta, A., Picco, G. P., and Vigna, G. 1998. Analyzing mobile code languages, mobile object systems. *IEEE Trans. Softw. Eng. 24*, 5 (May), 352–361.

Groth, P. T. and Suri, N. 2002. CPU resource control and accounting in the NOMADS mobile agent system. Tech. rep. Institute for Human & Machine Cognition, University of West Florida, Pensacola, FL.

Halstead, Jr., R. H. 1990. New ideas in Parallel Lisp: Language design, implementation. In *Parallel Lisp: Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, T. Ito and R. H. Halstead, Eds. Lecture Notes in Computer Science, vol. 441. Springer-Verlag, Berlin, Germany, 2–57.

Hartel, P. H. and Moreau, L. 2001. Formalizing the safety of Java, the Java Virtual Machine and Java Card. *ACM Comput. Surv. 33*, 4 (Dec.), 517–558.

Haynes, C. T. and Friedman, D. P. 1987. Abstracting timed preemption with engines. *Comput. Lang. 12*, 2, 109–121.

Johnson, M., Chang, P., Jeffers, R., Bradshaw, J., Breedy, M., Bunch, L., Kulkarni, S., Lott, J., Suri, N., Uszok, A., and Soo, V.-W. 2003. Kaos semantic policy and domain services: An application of DAML to Web services-based grid architectures. In *Proceedings of the AAMAS Workshop on Web-Services and Agent-Based Engineering* (Merlbourne, Australia).

JSR-121. 2003. Application isolation api specification. Web site: `http://www.jcp.org/en/jsr/detail?id=121`.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming (ECOOP'97)*. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, Berlin, Germany, 220–242.

Kornfeld, W. A. and Hewitt, C. E. 1981. The scientific community metaphor. *IEEE Trans. Syst., Man, Cybernet. 11*, 1 (Jan.), 24–33.

Ludwig, H., Keller, A., Dan, A., King, R. P., and Franck, R. 2003. Web service level agreement (WSLA), language specification. Tech. rep. IBM Corporation, York town Heights, NY.

Moreau, L. 2001a. Distributed directory service and message router for mobile agents. *Sci. Comput. Programm. 39*, 2–3, 249–272.

Moreau, L. 2001b. Tree rerooting in distributed garbage collection: Implementation and performance evaluation. *Higher-Order Symboli. Computat. 14*, 4 (Dec.), 357–386. (Colored figures can be found online at `http://www.ecs.soton.ac.uk/lavm/papers/hosc01-colour.tar.gz`).

Moreau, L. and Queinnec, C. 1997a. Design and semantics of Quantum: A language to control resource consumption in distributed computing. In *Proceedings of theUsenix Conference on Domain-Specific Languages* (DSL'97, Santa-Barbara, CA). Usenix Association, Berkely, CA, 183–197.

Moreau, L. and Queinnec, C. 1997b. On the finiteness of resources in distributed computing. Research rep. RR-3147. INRIA, Rocquen Count, France.

Moreau, L. and Queinnec, C. 1998. Distributed computations driven by resource consumption. In *IEEE International Conference on Computer Languages* (ICCL'98, Chicago, IL). IEEE Computer Press, Los Alamitos, CA, 68–77.

Moreau, L. and Queinnec, C. 2002a. Distributed and multi-type resource management. In *Proceedings of the ECOOP'02 Workshop on Resource Management for Safe Languages* (Malaga, Spain). 15. Short version appears in G. Czajkowski and J. Vitek, Resource management for safe languages, in *ECOOP'2002 Workshop Reader*, J. Hernandeg and A. Moreira, Eds. Lecture Notes in Computer Science. 1–14.

Moreau, L. and Queinnec, C. 2002b. Resource aware programming package. Available online at `www.ecs.soton.ac.uk/~lavm/rap`.

Osborne, R. B. 1990. Speculative computation in Multilisp. An overview. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (Nice, France). ACM Press, New York, NY, 198–208.

Roure, D. D., Jennings, N., and Shadbolt, N. 2001. The semantic grid. Tech. rep. University of Southampton, Southampton, U.K. Available online at `www.semanticgrid.org`.

Sun MicroSystems. 1996. *Java Remote Method Invocation Specification*. Sun MicroSystems, Santa Clara, CA.

Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A., Jeffers, R., Mitrovich, T. S., Pouliot, B. R., and Smith, D. S. 2000. NOMADS: Toward a strong and safe mobile agent system. In *Proceedings of the Fourth International Conference on Autonomous Agents* (Barcelona, Catalonia, Spain). ACM Press, New York, NY, 163–164.

Suri, N., University, L., Carvalho, M., Bradshaw, J. M., Breedy, M. R., Cowin, T. B., Groth, P. T., Saavedra, R., and Uszok, A. 2003. Enforcement of communications policies in software agent systems through mobile code. In *Proceedings of the IEEE 4th International Workshop on Policies* *for Distributed Systems and Networks* (Lake Como, Italy). IEEE Computer Society, Press, Los Alamitos, CA, 247–250.

VILLAZÓN, A. AND BINDER, W. 2001. Portable resource reification in java-based mobile agent systems. In *The Fifth IEEE International Conference on Mobile Agents* (MA'2001; Atlanta, GA). Springer-Verlag, Berlin, Germany.

WHITE, J. E. 1996. Telescript technology: Mobile agents. In *Software Agents*, J. Bradshaw, Ed. AAAI Press, Menlo Park, CA/MIT Press, Cambridge, MA.

YAO, W., MOODY, K., AND BACON, J. 2001. A model of oasis role-based access control and its support for active security. In *Proceedings of Sixth ACM Symposium on Access Control Models and Technologies* (SACMAT'2001, Chantilly, VA). ACM Press, New York, NY, 171–181.

Received March 2003; revised February 2004; accepted April 2004