Controlling and Optimizing the Usage of One Resource

Antoine Galland[1,3] and Mathieu Baudet[2]

[1] Gemplus Research Labs,
La Vigie, avenue du Jujubier,
ZI Athelia IV, 13705 La Ciotat Cedex, France
antoine.galland@research.gemplus.com
[2] LSV/CNRS UMR 8643 & INRIA Futurs Projet SECSI & ENS Cachan,
61, avenue du Président Wilson, 94235 Cachan Cedex, France
mathieu.baudet@lsv.ens-cachan.fr
[3] Pierre & Marie Curie University, LIP6 Laboratory,
8, rue du Capitaine Scott, 75015 Paris, France

Abstract. This paper studies the problem of resource availability in the context of mobile code for embedded systems such as smart cards. It presents an architecture dedicated to controlling the usage of a single resource in a multi-process operating system. Its specificity lies in its ability to improve the task scheduling in order to spare resources. Our architecture comprises two parts. The first statically computes the resource needs using a dedicated lattice. The second guarantees at runtime that there will always be enough resources for every application to terminate, thanks to an efficient deadlock-avoidance algorithm. The example studied here is an implementation on a JVM (Java Virtual Machine) for smart cards, dealing with a realistic subset of the Java bytecode.

1 Introduction

A smart card is a device with stringent hardware constraints: low-power CPU, low throughput serial I/O, little memory (typically 1−4 kb RAM, 32−128 kb ROM and 16−64 kb Flash RAM). But its tamper resistance [19] makes it one of the mobile computing devices of choice. From the emergence of smart cards [26] to the present, integrating a high level of safety with so few resources has always been the main challenge of smart card manufacturers. Hence, one has to keep in mind that optimizing physical and logic resources usage is of prime importance in such a constrained system.

Modern smart card platforms offer the opportunity to download code into the card while it is in the user's possession—this is called *post issuance*. This new functionality raises new problems as far as the security of mobile code for smart cards is concerned, since hostile applets can be developed and downloaded into the card. In Java Card [4], various solutions have been studied to integrate a Java bytecode verifier into a smart card in order to make sure that programs are well-typed [2, 11, 23]. After type-safe verification, resource control is the logical next step to ensure reliability [10]. Indeed application providers would like guarantees that their applets will have all the required resources for safe execution throughout their lifespan.

The aim of this paper is to propose an architecture that solves these two problems, i.e., optimizing resource usage and guaranteeing availability in a multi-application environment. Section 2 studies different models of resource control and introduces the problem of deadlock avoidance. Section 3 presents our framework of code analysis and deadlock avoidance on a single resource. Then Section 4 details its implementation on an architecture for smart cards. Finally, Section 5 provides some benchmarks and Section 6 is the conclusion.

2 Related Literature

2.1 Resource Control

When mobile code is uploaded to a new host—in our study a smart card—there is no guarantee that there will be enough resources to complete its execution. The most commonly adopted solution is to use a contract-based approach of resource management [22]. In this approach, each applet must declare its resource requirements in a contract. Once the smart card accepts a contract, it must meet its requirements during the applet's lifespan. Since the uploaded applet is not considered as trustworthy, safeguards must be established. Runtime techniques are generally used to control allocations, which implies costly monitoring. Moreover, when the contract is canceled, it is often too late to recover applet execution even if a call-back mechanism can be used [8].

To reduce runtime extra-costs, it may be preferable to check once and for all whether an applet respects its own contract. This generally implies bounding its resource consumptions by means of static control-flow analysis or type systems [7, 16]. These approaches are complex, as a consequence, it is difficult to incorporate them directly into a smart card. In this case, some techniques inspired of "Proof-Carrying Code" from Lee and Necula [25] can be used to check on line the resource bounds computed off line [24].

Once a smart card commits itself to respecting specific resource requirements, the simplest solution to ensure availability is to reserve and lock all the required resources at start-up. This is the solution used in Java Card 2.1 [4] for heap memory allocation. Smart card developers gather all necessary memory initializations[4] needed at start-up to avoid running out of memory later. This implies that additional memory allocations cannot be ensured.

The drawback of this solution is the waste of resources when multiple applets are used. Indeed applets will seldom use all their reserved quotas simultaneously. On the contrary it is more likely that peaks of resource usage will occur at different times. Moreover, if this is not the case, we might consider delaying one or more tasks to avoid this situation. In short, resource usage is currently far from being optimized.

[4] in the `javacard.framework.Applet.install()` method.

Our objective is to guarantee resource availability to every applet, while minimizing the global needs of the system. Most approaches described above do not solve these two problems simultaneously. Thus we are looking for a framework that is more economical than one that blocks all resources at start-up, while offering the same level of dependability.

2.2 Relation to Deadlock Avoidance

In most systems, when a program requests more resources than the amount available, an error is reported and the task terminated. For our purpose, however, a thriftier solution would be to suspend the requesting task temporarily, in the hope that some resource might be released later. In this case, allocation and deallocation methods are equivalent to the P (locking) and V (unlocking) synchronizing primitives [12].

This approach leads to two well-known problems of concurrency theory: starvation and deadlocks. Starvation only occurs in the presence of non-terminating programs, and therefore is usually not a problem on smart cards. Concerning the problem of deadlock prevention, it was first studied by Dijkstra [12]. A convenient geometrical representation exists and it is called *progress graphs* (see Figure 1). The axes represent the progress rate of each task through the time.

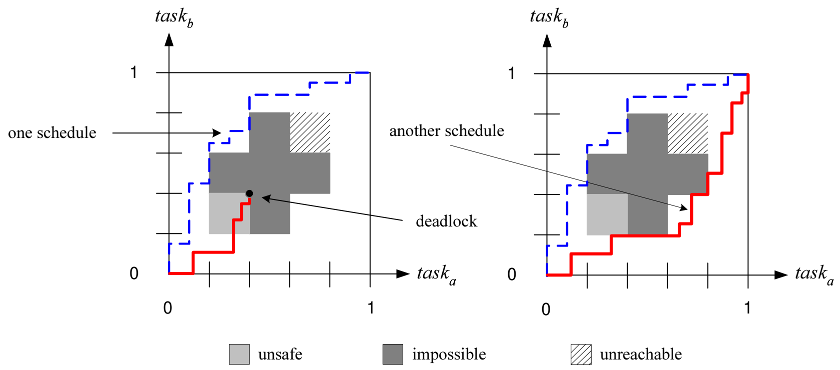

Fig. 1. Progress graph and deadlock avoidance

Impossible areas are those for which the sum of the instantaneous consumptions is greater than the amount of resource available. A *valid schedule* is a path which avoids impossible areas. A state is considered *safe* if there exists a schedule starting from this thread that ends every task. On the contrary, *unsafe states* lead to deadlock. Deadlock-avoidance algorithms generally consist in detecting unsafe states (deadlock *prediction*) and avoiding them at runtime.

The first deadlock-avoidance algorithm, *the banker's algorithm*, was introduced by Dijkstra [12]. In the case of reusable resources without partial requests, Habermann [15] solved the deadlock-prediction problem in polynomial time. Later, Gold [14] proved that deadlock prediction is NP-complete in general and is polynomial for several sub-problems including the case of one resource. Deadlock avoidance has since been studied and used in many resource allocation systems, such as the flexible manufacturing systems (FMS) [21, 13].

In the rest of this paper we first present a generalization of Gold's safety criterion for a single resource [14]. Gold reasons directly on lists of partial requests, not processes. Accordingly it is unclear from Gold's paper how to compile processes into partial requests (or "blocks" in the present study). Furthermore, Gold's method only applies to processes where parallel composition is not nested. Our algorithm can handle a more general class of systems, modeled by a simple process algebra. It relies on a dedicated data-structure called "normalized lists" which allows for very fast computations and thus is perfectly suitable for dynamic scheduling. Thanks to the lattice structure of normalized lists, the framework of abstract interpretation applies; this allows us to design a static analysis of the program requirements in most programming languages, provided allocations and deallocations are explicit. Once the programs have been annotated with the results of the analysis, they can be safely executed under the control of a deadlock-avoidance unit. This unit works by reading the static information at runtime and by applying the safety criterion to the term which models the overall state of the system. Finally, as an application of our deadlock-avoidance framework, an implementation for the Java bytecode is presented and discussed. A noticeable property of this implementation is that it is an ordinary Java package. Thanks to the synchronization primitives of Java, no modification of any specific JVM is necessary.

3 Deadlock Avoidance on a Single Resource

3.1 Process Algebra

We first consider a simple language of processes, intended to model a pool of threads accessing a single resource. This will prove useful later on. Terms of our language, called *processes* or *programs*, are defined by the following grammar:

$$p \ ::= \ \epsilon \mid x \mid (p\,p) \mid (p \,\|\, p)$$

where x denotes an integer, positive or negative. Intuitively, the meaning of this syntax can be described as follows:

- ϵ represents the empty program (thus already terminated),
- x stands for an instruction that allocates an amount x of the resource (which means deallocating some of it when x is negative),
- $(p_1\,p_2)$ denotes a sequence of processes: p_1 then p_2,
- $(p_1 \,\|\, p_2)$ is the program that executes p_1 and p_2 concurrently.

$\dfrac{}{(\epsilon\ p) \xrightarrow{\epsilon} p}$	$\dfrac{}{(\epsilon \Vert p) \xrightarrow{\epsilon} p}$	$\dfrac{}{(p \Vert \epsilon) \xrightarrow{\epsilon} p}$	(structural rules)
$\dfrac{}{x \xrightarrow{x} \epsilon}$	$\dfrac{p \xrightarrow{\epsilon *} \xrightarrow{x} \xrightarrow{\epsilon *} p'}{p \xrightarrow{x} p'}$		(evaluation rules)
$\dfrac{p_1 \xrightarrow{x} p_1'}{(p_1\ p_2) \xrightarrow{x} (p_1'\ p_2)}$	$\dfrac{p_1 \xrightarrow{x} p_1'}{(p_1 \parallel p_2) \xrightarrow{x} (p_1' \parallel p_2)}$	$\dfrac{p_2 \xrightarrow{x} p_2'}{(p_1 \parallel p_2) \xrightarrow{x} (p_1 \parallel p_2')}$	(context rules)

Table 1. Small-step semantics

To formalize our language's semantics, we choose a system of labelled transitions described in Table 1. ϵ-transitions do nothing but simplify terms, whereas normal transitions *emit* an allocation x.

Following the usual conventions of automata, sequences of transitions emit a list of integers: $p \xrightarrow{x_1 x_2 \ldots x_n} p'$ if $p \xrightarrow{x_1} \xrightarrow{x_2} \cdots \xrightarrow{x_n} p'$. It is easy to prove, using these semantics, that every program eventually terminates—after emitting all its allocations. Thus we can define the execution traces of program p as the set of every list $l = x_1 x_2 \ldots x_n$ such that $p \xrightarrow{l} \epsilon$. For instance, the traces of $((1\,2) \Vert {-3})$ are: $1\,2\,(-3)$, $1\,(-3)\,2$ and $(-3)\,1\,2$.

Given an execution trace, we can easily define the amount of resource that is needed, which we call *cost* of the trace:

$$\mathcal{C}(x_1 x_2 \ldots x_n) \stackrel{def}{=} \max_{0 \le i \le n} \left(\sum_{1 \le j \le i} x_j \right) \quad \text{thus } \mathcal{C}(l) \ge 0 \text{ and } \mathcal{C}(\epsilon) \stackrel{def}{=} 0.$$

Using this, we are now able to formulate a simple criterion of safety. Assuming that a state is described by term p of the process algebra, and given a certain amount M of available resource, the state p will be safe if and only if there exists a trace l such that: $p \xrightarrow{l} \epsilon$ and $\mathcal{C}(l) \le M$. Thus a criterion for the safety of state p is simply: $\min \{\mathcal{C}(l), p \xrightarrow{l} \epsilon\} \ \le\ M$.

What we need now is an efficient way to compute this minimum, which we call the *cost of process* p:

$$\mathcal{C}(p) \stackrel{def}{=} \min \{\mathcal{C}(l), p \xrightarrow{l} \epsilon\}$$

3.2 Computing Process Costs with Normalized Lists.

A first class of algorithms which can be thought of to compute $\mathcal{C}(p)$ consists in exploring the transition system between p and ϵ. With dynamic-programming techniques, such algorithms would require linear time dependent on the size of the transition system. Unfortunately this size grows exponentially with the number of threads (state-explosion phenomenon), so these algorithms are not suited to our purpose.

Instead, we shall rely upon the fact that only one resource is considered and investigate a more semantical approach, where $\mathcal{C}(p)$ is computed recursively over p.

As a first attempt in this direction, the algorithm presented in Table 2 recursively computes a pair of integer $B(p)$. It can be shown that, for each p, the

$B(\epsilon)$	$= (0,\ 0)$
$B(x)$	$= (\max(x, 0),\ x)$
$B(p_1\, p_2)$	$= B(p_1) \cdot B(p_2)$
$B(p_1 \parallel p_2)$	$= B(p_1) \times B(p_2)$
where:	
$(c_1, \delta_1) \cdot (c_2, \delta_2)$	$= (\max(c_1, \delta_1 + c_2),\ \delta_1 + \delta_2)$
$(c_1, \delta_1) \times (c_2, \delta_2)$	$= (\min(\max(c_1, \delta_1 + c_2), \max(c_2, \delta_2 + c_1)),\ \delta_1 + \delta_2)$

Table 2. A first algorithm with blocks

first coordinate of $B(p)$ is an upper-approximation of $\mathcal{C}(p)$, whereas the second one computes the sum of all the allocations. For instance, a run on the previous example: $B((1\,2)\|{-3}) = B(1\,2) \times (0, -3) = (3, 3) \times (0, -3) = (0, 0)$ gives here the exact answer $\mathcal{C}((1\,2)\|{-3}) = 0$.

Pairs of integers computed by the function $B(\_)$ are of the form (c, δ) where $c \geq \max(0, \delta)$. As they take a natural place in our problem, we shall henceforth call such pairs *blocks*. These are equivalent to Gold's *partial requests* [14] which are actually the pairs $(c, c - \delta)$. Figure 2 illustrates the intuition behind blocks and the $\cdot$-operator.

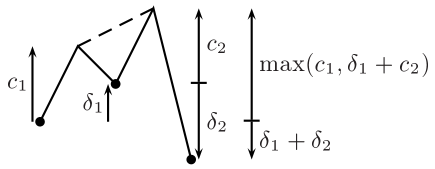

Fig. 2. Merging blocks with $\cdot$

Although fast and simple, the algorithm in Table 2, that uses blocks, unfortunately gives very poor results on more complicated entries. Two reasons may be put advanced to account for this fact:

- First, the $\times$-operator does not reflect the associativity of parallel composition. For instance, $((2, 0) \times (1, 1)) \times (0, -1) = (2, 1) \times (0, -1) = (1, 0)$ whereas $(2, 0) \times ((1, 1) \times (0, -1)) = (2, 0) \times (0, 0) = (2, 0)$.

– Second, the $\cdot$-operator causes a significant loss of information, by merging the internal steps of a sequence together. For example, although the cost of $(-1\,1)\|(2\,-1)$ is $\mathbf{1}$, the algorithm fails to find the optimal trace $(-1\,2\,-1\,1)$: $B((-1\,1)\|(2\,-1)) = B((-1\,1)) \times B((2\,-1)) = (0,0) \times (2,1) = (\mathbf{2},1)$.

To solve the first problem, one possibility would be to generalize the $\times$-operator so as to deal with a pool of blocks. A natural way to do this is to define:

$$\begin{aligned}(c_1,\delta_1) \times \ldots \times (c_n,\delta_n) &\stackrel{def}{=} \prod_{\sigma \in \Sigma_n} (c_{\sigma 1},\delta_{\sigma 1}) \cdot \ldots \cdot (c_{\sigma n},\delta_{\sigma n}) \\ &= \left(\min_{\sigma \in \Sigma_n} \max_{1 \le i \le n} \left(c_{\sigma i} + \sum_{1 \le j < i} \delta_{\sigma j} \right), \sum_{1 \le i \le n} \delta_i \right)\end{aligned}$$

where Σ_n stands for the set of all permutations on $\{1 \ldots n\}$ and $\sqcap$ is the greatest lower bound on pairs of integers. After doing this, we face a non-trivial minimization problem—which can be seen as a scheduling problem. Fortunately it can be solved in polynomial time, one of the optimal permutations being obtained by sorting the set of blocks according to the following total quasi-ordering (total, reflexive, transitive relation):

$$(c_1,\delta_1) \preceq (c_2,\delta_2) \text{ iff: } \begin{array}{r|ll} & \delta_1 \le 0 \text{ and } \delta_2 \ge 0 & (1) \\ \text{or} & \delta_1 < 0, \delta_2 < 0 \text{ and } c_1 \le c_2 & (2) \\ \text{or} & \delta_1 > 0, \delta_2 > 0 \text{ and } \delta_1 - c_1 \le \delta_2 - c_2 & (3) \end{array}$$

Thus the previous quantity can be computed in $O(n \log(n))$ (i.e., the cost of sorting), which gives us the generalization of $\times$ we were looking for. The definition of $\preceq$ relies basically on the same ideas as Gold's generalized banker's algorithm [14]: for instance rule (1) means "producers before consumers," rule (2) means "better producers first" and rule (3) can be seen as the time-reversed image of (2).

We now address the second issue: to reduce the loss of information caused by the $\cdot$-operator. What the previous example $(-1\,1)\|(2\,-1)$ suggests, is that merging blocks implies losing many valid schedules. Actually the traces that are lost are those that interleave the processes together.

For this reason we work with *lists of blocks* instead of blocks so as to keep more information about code behavior. In this way, the $\cdot$-operator will not always *merge* blocks as it did previously. For instance, we decide that $(0,-2) \cdot (2,2) = (0,-2)(2,2)$, which is more precise than $(0,0)$. Indeed $(0,-2)(2,2)$ in parallel with $(2,0)$ can be scheduled $(0,-2)(2,0)(2,2)$ which costs 0, whereas $(0,0)$ in parallel with $(2,0)$ costs 2. By cost of a list of blocks, we mean the quantity defined by: $\mathcal{C}(\,(c_1,\delta_1)\ldots(c_n,\delta_n)\,) \stackrel{def}{=} \max_{1 \le i \le n} \left(c_i + \sum_{1 \le j < i} \delta_j \right)$ and $\mathcal{C}(\epsilon) \stackrel{def}{=} 0$ where ϵ stands for the empty list.

Defining $\cdot$ as the usual concatenation of lists (working in the free monoid on blocks) would lead to structures with nearly the same size as the program, so we need a way to simplify the data. Thus, we introduce a procedure of *normalization* of lists. To this end we define a relation $\mathcal{S}$ between blocks called the *simplification relation*, which determines which blocks can be soundly merged. Intuitively two blocks can be merged if every schedule that separates them can be rewritten in an equivalent or possibly less expensive schedule that leaves them adjacent. This idea is expressed by the following definition:

$$(c_1,\delta_1)\mathcal{S}(c_2,\delta_2) \text{ iff: } \forall l, \quad \begin{array}{r|l} & \mathcal{C}((c_1,\delta_1)l(c_2,\delta_2)) \geq \mathcal{C}((c_1,\delta_1)(c_2,\delta_2)l) \\ \text{or} & \mathcal{C}((c_1,\delta_1)l(c_2,\delta_2)) \geq \mathcal{C}(l(c_1,\delta_1)(c_2,\delta_2)) \end{array}$$

which can be written in a more convenient way:

$$(c_1,\delta_1)\mathcal{S}(c_2,\delta_2) \text{ iff: } \quad \begin{array}{r|l} & \delta_2 \leq 0 \text{ and } c_1 \geq c_2+\delta_1 \\ \text{or} & \delta_1 \geq 0 \text{ and } c_1 \leq c_2+\delta_1 \end{array}$$

We then consider the following rewriting rules on lists of blocks:

$$\begin{aligned} (c_1,\delta_1)(c_2,\delta_2) &\to (\max(c_1,\delta_1+c_2),\delta_1+\delta_2) \text{ whenever } (c_1,\delta_1)\mathcal{S}(c_2,\delta_2) \\ (0,0) &\to \epsilon \end{aligned}$$

It is an easy check that these rules are both confluent and strongly normalizing; therefore $\to$ terminates for every list, on a unique *normal form*. From now on, lists in normal form will be called *normalized lists*. Figure 3 illustrates the strong constraints imposed by the normalization.

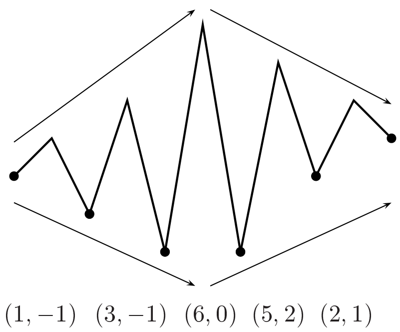

Fig. 3. General aspect of normalized lists

A key property of normalized lists is that they are sorted for $\preceq$, the previously mentioned quasi-ordering. Thanks to this property and to the normalization rules, we are able to generalize the $\times$-operator to any set of lists, so that it still corresponds to optimal schedules of blocks. Instead of sorting the blocks, we shall simply *merge* the normalized lists—in the sense of the merge sort for $\preceq$ (see e.g., [18]).

Finally Table 3 describes an algorithm that computes recursively a list of blocks $L(p)$ that is "equivalent" to p. It can be shown that for each p, $\mathcal{C}(p) = \mathcal{C}(L(p))$. The fact that the algorithm is sound *and* complete results from the careful choice of the simplification relation $\mathcal{S}$ and from the existence of an exact scheduling algorithm given by the total quasi-ordering $\preceq$.

$L(\epsilon)$	$= \epsilon$
$L(x)$	$= (\max(x, 0),\ x)$ if $x \neq 0$, ϵ otherwise
$L(p_1\, p_2)$	$= L(p_1) \cdot L(p_2)$
$L(p_1 \parallel p_2)$	$= L(p_1) \times L(p_2)$
where:	
$l_1 \cdot l_2$	$= normalize(l_1 l_2)$
$l_1 \times l_2$	$= normalize(merge(l_1, l_2))$

Table 3. Final algorithm with lists of blocks

Given the linear complexity of the *normalize*- and *merge*-operations, our algorithm finally computes $\mathcal{C}(p)$ in time $O(m\, n)$, where n is the size of p and m its depth (as a binary tree). In practice, moreover, we noticed that the size of normalized lists seldom exceeds ten blocks (see our experimental study in section 5.1). Hence process costs are computed in practically linear time.

3.3 Static Analysis

So far we have explained how to quickly compute the cost of a system—and thus whether it is safe or not—provided that it is modeled by a term of our process algebra. We now need a way to represent systems that execute real programs. This is done by analyzing the programs statically, and by annotating them with the extracted information. In this way, the deadlock-avoidance unit will be able to track the system state at runtime, and to detect unsafe areas before they occur.

Given the previous algorithm for computing process costs, normalized lists appear as a natural data-structure to represent information on resource demands during the analysis. However, branching instructions (if, while...) are not handled directly by the previous algorithm. A general framework for analyzing programs is abstract interpretation [6]. This usually requires a lattice structure.

Fortunately, it turns out that normalized lists can be equipped with a lattice structure, where the ordering $\sqsubseteq$ (reflexive, transitive, antisymmetric) represents loss of information in a natural way. For instance $l_1 \sqsubseteq l_2$ iff $\forall l, \mathcal{C}(l_1 \times l) \leq \mathcal{C}(l_2 \times l)$. $\sqsubseteq$ is also stable by any context: $l_1 \sqsubseteq l_2$ implies $\forall l$, $(l_1 \times l) \sqsubseteq (l_2 \times l)$, $(l_1\, l) \sqsubseteq (l_2\, l)$ and $(l\, l_1) \sqsubseteq (l\, l_2)$.

Thanks to this structure we are able to apply the theory of abstract interpretation to analyze general programming languages, provided allocations and deallocations are explicit. Otherwise another preliminary static analysis may be

Ordering	: $l_1 \sqsubseteq l_2$ iff $\mathcal{C}(l_1 \times -l_2) = 0$
Least upper bound	: $l_1 \sqcup l_2 = normalize(join(0, l_1, l_2))$
Greatest lower bound	: $l_1 \sqcap l_2 = -((-l_1) \sqcup (-l_2))$
Least element	: $\perp = (0, -\infty)$
Greatest element	: $\top = (+\infty, +\infty)$
Opposite	: $-(\,(c_1, \delta_1) \dots (c_n, \delta_n)\,) =$
$normalize(\,(0, -c_1)(\max(-\delta_1, 0), -\delta_1) \dots (0, -c_n)(\max(-\delta_n, 0), -\delta_n)\,)$	

Table 4. The lattice of normalized lists

$join(x, \epsilon, \epsilon)$	$= \epsilon$
$join(x, (c_1, \delta_1)q_1, \epsilon)$	$= shift(x, c_1, \delta_1)\, join(x + \delta_1, q_1, \epsilon)$
$join(x, \epsilon, (c_2, \delta_2)q_2)$	$= shift(-x, c_2, \delta_2)\, join(x - \delta_2, \epsilon, q_2)$
$join(x, (c_1, \delta_1)q_1, (c_2, \delta_2)q_2))$	$= shift(x, c_1, \delta_1)\, join(x + \delta_1, q_1, l_2)$ when $(c_1, \delta_1) \preceq (c_2, \delta_2)$
$join(x, (c_1, \delta_1)q_1, (c_2, \delta_2)q_2))$	$= shift(-x, c_2, \delta_2)\, join(x - \delta_2, l_1, q_2)$ otherwise
$shift(x, c, \delta)$	$= (\max(c, x) - \max(0, x),\ \max(\delta, x) - \max(0, x))$

Table 5. Computing the least upper bound

required so as to infer upper bounds to the allocations (and lower bounds to the deallocations); this falls outside the scope of this paper.

We describe the lattice operations on normalized lists in Table 4. The least upper bound $\sqcup$ requires an auxiliary function *join*, described in Table 5, whose first argument x is seen intuitively as an offset between the beginning of the two lists. The opposite operator $-$ is meant to reverse the signs of the variations of a list. Arithmetic operations on $\mathbb{Z}$ are extended to $\mathbb{Z} \cup \{\pm\infty\}$ in the usual way with the convention $(+\infty) + (-\infty) = +\infty$. Proving these algorithms would require more involved tools than we intend discuss here (see [1]).

In the following section, we shall see more closely how to apply these ideas in practice, through the implementation of a framework to analyze and execute Java class files for smart cards.

4 Application to Java on Smart Cards

We now describe an architecture dedicated to smart cards, which can analyze Java class files and control their access to one resource. Due to the limited computing power of smart cards (memory and CPU), not all operations can be done on card. Therefore, we use a split architecture (off card/on card) to delegate complex computations off the card. The off-card part is in charge of resource prediction on Java applications while the on-card part prevents resource deadlock through a resource server. The following sections explain in more detail the role of each part.

4.1 Static Analysis of Java Programs (off card)

The analysis algorithm, programmed in Java, comprises three parts. First, we summarize the control-flow graph of each application. Then we extract and summarize information about resource requirements using normalized lists. Finally, this information is used to annotate the Java class files.

Control-Flow Graph. The aim of this part is to build the most precise representation of the application's control-flow graph. This stage constructs the method call graph (inter-method analysis) and a control-flow graph for each method (intra-method analysis).

Runtime method dispatch, exception handling and dynamic thread creation make Java control-flow hard to compute statically. Control flow estimation for object-oriented languages is usually called "class analysis" [9]. In our case, virtual methods can be handled in a conservative way by computing for each object variable a super-set of the parent classes. As for the exceptions, one possibility is to over-approximate the set of exception handlers associated with each program point [3]. Both solutions require there be fixed a pool of Java classes to analyze (Java API plus smart card applications). It means that our approach would not work in an environment with dynamic class loading. Concerning Java thread creation and destruction (`thread.start()` and `thread.join()`), at this point our analysis requires that when a thread is *joined*, its creation point can be determined statically.

To speed up the next stages of the analysis, the control-flow graph of each method is stored as a directed graph where only control instructions (branch, fork, join, call) and allocation/deallocation operations are kept.

Abstract Interpretation Using Normalized-List Lattice. This stage analyzes the resource allocations of each method and determines the annotations that will be added to the bytecode. Annotations consist basically in normalized lists that represent the future allocations of the thread. These are computed by a backward abstract interpretation [6] using lattice operations from Table 4. The least upper bound operation $\sqcup$ is used for branches and loops. For loops, the convergence of the computation is ensured by the fact that: for each normalized list $l = (x_1, \delta_1) \ldots (x_n, \delta_n)$, if $\sum_i \delta_i \leq 0$ (no leak of resource) then $l \cdot l \sqsubseteq l$; in other words, only one iteration over the loop is needed. Otherwise, when a resource leak is detected, we put directly $\top$ as a result of the analysis for this portion of code. Due to the fact that each method is analyzed independently, i.e., ignoring call contexts, the deadlock-avoidance algorithm has to track and restore the real call contexts at runtime. Hence, annotations are added when we enter and return from a method (the same applies for threads). Figure 4 gives an illustration—translated into Java for convenience—of the annotations added to the Java bytecode. The "global contract" for instance, corresponds to:

$$(5,2)(2,1) = \underbrace{L(1)}_{\texttt{alloc(1)}} \cdot (\underbrace{(L(4) \cdot L(-3))}_{\texttt{thread.start()}} \times (\underbrace{(L(-2) \sqcup L(2))}_{\texttt{foo(args)}} \cdot \underbrace{L(-1)}_{\texttt{alloc(-1)}}))$$

Before

```
class SimpleExample implements Executable {

  int [] getGlobalAnnotation() {
    return null;
  }

  void run(String[] args){
    Server.alloc(1)
    SimpleThread thread = new SimpleThread();

    thread.start();

    foo(args);

    Server.alloc(-1);

  }

  void foo(Object obj) {
    if (obj == null) {
      Server.alloc(-2);
    } else {
      Server.alloc(2);
    }

  }

  static class SimpleThread extends Thread {
    public void run() {
      Server.alloc(4);
      Server.alloc(-3);

  }
}
```

After

```
class SimpleExample implements Executable {

  int [] getGlobalAnnotation() {
    return [(5,2),(2,1)]; // global contract
  }

  void run(String[] args){
    Server.alloc(1,[(4,1)(2,1)]);
    SimpleThread thread = new SimpleThread();
    Server.fork([(2,1)], thread, [(4,1)]);
    thread.start();
    Server.call([(2,2)], [0,-1]);
    foo(args);
    Server.discard();
    Server.alloc(-1,[]);
    Server.end();
  }

  void foo(Object obj) {
    if (obj == null) {
      Server.alloc(-2,[]);
    } else {
      Server.alloc(2,[]);
    }
    Server.end();
  }

  static class SimpleThread extends Thread {
    public void run() {
      Server.alloc(4, [(0,-3)]);
      Server.alloc(-3, []);
      Server.end();
  }
}
```

Fig. 4. Example of annotations using the resource server API

Optimizations can be carried out to reduce unnecessary calls to the server. For example, when a method does not handle a resource, it not necessary to enclose calls to it in a `Server.call(..)`–`Server.discard()` pair. During the final stage, the parameters for the server calls (normalized lists) are stored in the corresponding Java classes. Once the Java classes have been annotated, they can be used by the on-card resource server.

4.2 Dynamic Resource Avoidance (on card)

We have designed our resource server completely in Java over a Java Virtual Machine (JVM) dedicated to low-end embedded systems and next generation smart cards [20]. This experimental JVM is more similar to J2ME [17] than to a standard Java Card [4]. It supports, for example, Java threads which are necessary for our resource server.

The resource server is informed of any context change through annotations (resource allocation, deallocation, thread creation, application start...) which we call *requests*. For each request, our deadlock-avoidance algorithm simulates the transaction, that is: the state of the server is updated and our safety algorithm is applied to compute an estimated cost C, to be compared with the amount of resource still available M. If it leads to a safe state ($C \leq M$), the transaction is made. If not (unsafe state detected), the thread of the requesting application is suspended and the server state restored. The thread will be woken up when the server state has changed. The Java thread API is used (`wait()` and `notifyAll()`) to suspend and wake up threads. The chosen implementation for the server state is a tree similar to the syntax trees of our previous process algebra, and having each thread mapped to one of its leaves.

The scheduling algorithm is distributed between the internal JVM scheduler and the resource server. The former is in charge of equity access to the CPU among threads (using a round-robin algorithm for example), and the latter ensures deadlock avoidance over the JVM scheduler. The advantage of this design is that the JVM and its internal scheduler do not need to be modified. Thanks to the Java Thread API, the resource server is implemented completely in Java. The next section presents experimental results of this architecture.

5 Experimental Results

5.1 Experimental study of list normalization

To have an idea of the statistical behavior of normalization, we studied the length after normalization of random lists of length n. One drawing of lots consisted in generating a vector $(a_0, \ldots, a_n)$ whose elements were uniformly selected between $-m$ and $+m$. This vector was then translated into a list of n elementary blocks $(\max(x_1, 0), x_1) \ldots (\max(x_n, 0), x_n)$ where $x_k = a_k - a_{k-1}$ represents the variation of resource between states a_{k-1} and a_k. The next table shows the length after normalization of samples of 1000 random lists for $m = 100$:

initial length	minimum	average	maximum
10	1	2.47	5
100	1	4.21	9
1000	1	4.43	9
10000	1	4.41	9

As n tends to infinity with m fixed, the average size of normalized lists tends to an extremely low limit. This observation was done for other models of randomization as well. In practice the size of normalized lists seldom exceeds ten blocks.

5.2 Experimental Validation

We have tested our resource server on a generic resource (a simple resource counter). The goal was to evaluate the cost and benefit of this architecture before adapting it to a real resource. Results are given for three test programs

(`Test`$_i$). Each test is made of several threads that model a group of concurrent applications. It performs no operations but allocations and deallocations on the resource. Table 6 compares the resource necessary according to three models of resource control and discusses equity (i.e., CPU time-sharing) between applications:

- The first model reserves all resources needed for applications at start-up. The reserved resource represents the *worst case* consumption for the entire test. There is no resource optimization, but each application can run at anytime so the equity between applications is full.
- The second model runs an approximated deadlock-avoidance algorithm based on blocks. Although interleaved schedules are possible, these are not considered while detecting unsafe states. Hence the deadlock prediction is over-conservative and the equity between applications is weak.
- The third model is the one presented in this paper, where deadlock avoidance is based on normalized lists. Threads are suspended only when a deadlock may occur—although one cannot be sure it will, because of the branching instructions. In practice equity is strong because applications are seldom suspended for a long time.

	Resource needs per model		
	No optimization	Blocks	Norm. Lists
`Test1` (2 threads)	18	16	16
`Test2` (4 threads)	12	6	2
`Test3` (2 threads)	7	4	4
Equity (in practice)	full	weak	strong

Table 6. Resource needs and equity for each model.

These tests demonstrate that large gains can be obtained (e.g., Table 6, `Test2` and `Test3`). Sometimes, there is less incentive for using our deadlock-avoidance scheme (e.g., `Test1`). In these cases, it is important to notice that gains (or not) can be computed in advance, and a card loader may therefore decide whether to apply our techniques prior to launching an application, depending on the estimated gains and the available resources.

As for the cost of code annotation, the Java binary file (`.class`) of an annotated application is 47% larger than the original code. That represents worst cases possible increase, as the test programs *only* perform allocations and deallocations and no other computation. Moreover the increase is due more to the inserted code (server calls) than to the data (normalized lists); hence, appropriate coding of the server calls would allow large savings.

Concerning the dynamic on-card behavior of the resource server, CPU time-sharing measurements confirm that mixed scheduling between the resource server and the internal Java scheduler is suitable for CPU equity and deadlock avoidance. Moreover, the dynamic deadlock avoidance caused no noticeable slowing down in practice thanks mainly to the small size of normalized lists (see section 5.1).

6 Conclusion

Reserving all the required resources at start-up is a simple and efficient way to guarantee resource availability. The drawback of this solution is the waste of resources as it is scaled up. We have proposed an architecture that can both guarantee the availability of one resource and optimize its usage with a deadlock-avoidance algorithm. Our contribution is first a generalization of Gold's results for a single resource to a more expressive class of systems, modeled by a simple process algebra. Our algorithm, based on a dedicated data-structure called "normalized lists," takes linear time in practice and thus, is perfectly suitable for dynamic scheduling. Thanks to the lattice structure of normalized lists, abstract interpretation techniques can be applied and yield to an efficient deadlock-avoidance algorithm for real programming languages. We successfully applied our architecture to a realistic subset of Java bytecode. We should stress that no modification of the JVM is needed and that our annotated class files remain Java compliant.

The model of resource control discussed is this paper has been designed to optimize resource saving. Even though this resource gain has a price (our algorithms are non-trivial), we believe our model is appropriate for many heavily constrained systems.

In a theoretical way, several improvements can be considered in the future. First, we may want to handle possibly non-terminating idioms (I/O functions, endless loops). Second, it would be interesting to extend this framework to deal with several resources. However, solving this difficult problem (NP-complete in general) would certainly involve drastic over-approximations. Lastly, we have not yet considered checking the consistency of program annotations. This is actually a concern since static analysis is done off card and thus is untrusted. As with Java Card type verification, three approaches may be thought of: certification by a third party, on-card static verification and on-card runtime verification.

In a practical way, several studies are currently investigated. The main one is to apply our model to a real resource, among the choices : memory, files, buffers of communication. The main problem is to find where allocations/deallocations are implied in a program and to determine the amount of resource needed. Memory is the most critical resource for smart card manufacturers. But controlling memory usage becomes especially difficult in garbage-collected languages. We are currently adapting *escape analysis* [5] techniques in order to statically predict the releases of memory of a Java application. The objective is to determine the relevant conditions of using this model of resource.

References

1. M. Baudet. Contrôle de ressource et évitement des interblocages sur la mémoire. Master's thesis, DEA Programmation (Paris), Gemplus Research Labs, September 2002.
2. L. Casset, L. Burdy, and A. Requet. Formal Development of an embedded verifier for Java Card Byte Code. In *the IEEE International Conference on Dependable Systems & Networks (DSN)*, pages 51–58, Washington, D.C., USA, June 2002.
3. B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe. Interprocedural Exception Analysis for Java. In *16th ACM Symposium on Applied Computing (SAC)*, LasVegas, USA, March 2001.
4. Z. Chen. *Java Card^{TM} Technology for Smart Cards : Architecture and Programmer's Guide*. The Java™ Series. Addison Wesley, 2000.
5. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java™. In *ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 2–19, Denver USA, 1999.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *the 4th Annual ACM SIGPLAN-SIGACT*, pages 238–252, Los Angeles, California, 1977.
7. K. Crary and S. Weirich. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 184–198, Boston, MA, January 2000.
8. G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–35, Canada, 1998.
9. G. DeFouw, D. Grove, and C. Chambers. Fast Interprocedural Class Analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 222–236, San Diego, CA, USA, 1998.
10. D. Deville, A. Galland, G. Grimaud, and S. Jean. Assessing the future of Smart Card Operating Systems. In *4th e-Smart Conference*, Sophia Antipolis, France, September 17-19 2003.
11. D. Deville and G. Grimaud. Building an "impossible" verifier on a Java Card. In *2nd USENIX Workshop on Industrial Experiences with Systems Software (WIESS)*, Boston, USA, December 2002.
12. E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.
13. J. Ezpeleta, F. Tricas, F. Garcia-Valles, and J. Colom. A banker's solution for deadlock avoidance in FMS with flexible routing and multiresource states. *IEEE Transactions on Robotics & Automation*, 18(4):621–625, August 2002.
14. E. M. Gold. Deadlock prediction: Easy and difficult cases. *SIAM Journal on Computing*, 7(3):320–336, August 1978.
15. A. N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–377, 1969.
16. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, New Orleans, 2003.
17. Java™ 2 Platform Micro Edition (J2ME™). Connected Limited Device Configuration (CLDC) Specification Version 1.0. `http://java.sun.com/j2me/`.

18. D. E. Knuth. *The Art of Computer Programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., 1998.
19. O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *USENIX Workshop on Smartcard Technology (Smartcard'99)*, pages 9–20, Chicago, Illinois, USA, May 10–11 1999.
20. L. Lagosanto. Next-Generation Embedded Java Operating System for Smart Cards. In *Gemplus Developer Conference*, Singapore, November 2002.
21. M. Lawley and S. Reveliotis. Deadlock Avoidance for Sequential Resource Allocation Systems: Hard and Easy Cases. *The Journal of FMS*, 13(4):385–404, 2001.
22. N. le Sommer and F. Guidec. A Contract-Based Approach of Resource-Constrained Software Deployment. In *the 1st IFIP/ACM Working Conference on Component Deployment*, volume 2370 of *LNCS*, pages 15–30, Berlin, Germany, June 2002.
23. X. Leroy. Bytecode Verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.
24. Mobile Resource Guarantees (MRG). European Project IST-2001-33149, 2002. http://www.dcs.ed.ac.uk/home/mrg/.
25. G. C. Necula. Proof–Carrying Code. In *the 24th ACM SIGPLAN-SIGACT symposium on principles of programming Languages*, Paris, France, January 1997.
26. J.-J. Quisquater. The adolescence of smart cards. *Future Generation Computer Systems*, 13:3–7, 1997.