A resource-control model based on deadlock avoidance

Antoine Galland[*,†] Mathieu Baudet[‡]

[*]Gemplus Applied Research and Security Centre,
antoine.galland@gemplus.com
[‡]LSV/CNRS UMR 8643 & INRIA Futurs Projet SECSI & ENS Cachan,
mathieu.baudet@lsv.ens-cachan.fr
[†]Pierre & Marie Curie University, LIP6 Laboratory,

Abstract

This talk 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. Details of this work have been presented in [5].

1 Introduction

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, 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. After type-safe verification, resource control is the logical next step to ensure reliability. 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 talk is to propose an architecture that solves these two problems, i.e., optimizing resource usage and guaranteeing availability in a multi-application environment.

2 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 [9]. 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 cancelled, it is often too late to recover applet execution even if a call-back mechanism can be used [3].

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 [2, 8]. 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 [11] can be used to check on line the resource bounds computed off line [10].

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 [1] for heap memory allocation. Smart card developers gather all necessary memory initializations[1] 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.

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.

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

2.1 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 [4].

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 [4] by the name of *the banker's algorithm*. In the case of reusable resources without partial requests, Habermann [7] solved the deadlock-prediction problem in polynomial time. Later, Gold [6] proved that deadlock prediction is NP-complete in general and is polynomial for several sub-problems including the case of one resource.

This talk presents a generalization of Gold's safety criterion for a single resource [6]. 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. Furthermore, Gold's method only applies to processes where parallel composition is not nested. Our algorithm can handle a more general class of systems, modelled 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.

References

[1] Zhiqun Chen. *Java Card^{TM} Technology for Smart Cards : Architecture and Programmer's Guide*. The Java$^{\text{TM}}$ Series. Addison Wesley, 2000.

[2] Karl Crary and Stephanie 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.

[3] Grzegorz Czajkowski and Thorsten 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.

[4] Edsger Wybe Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[5] Antoine Galland and Mathieu Baudet. Controlling and Optimizing the Usage of One Resource. In Atsushi Ohori, editor, *1st Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*, pages 195–211, Beijing, China, November 27-29 2003. Springer-Verlag.

[6] E. Mark Gold. Deadlock prediction: Easy and difficult cases. *SIAM Journal on Computing*, 7(3):320–336, August 1978.

[7] Arie Nicolaas Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–377, 1969.

[8] Martin Hofmann and Steffen 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.

[9] Nicolas le Sommer and Frédéric 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.

[10] Mobile Resource Guarantees (MRG). European Project IST-2001-33149, 2002. `http://www.dcs.ed.ac.uk/home/mrg/`.

[11] Georges C. Necula. Proof–Carrying Code. In *the 24th ACM SIGPLAN-SIGACT symposium on principles of programming Languages*, Paris, France, January 1997.