A resource-control model based on deadlock avoidance

Antoine Galland

antoine.galland@gemplus.com

Gemplus Research Labs – LIP6

Mathieu Baudet

mathieu.baudet@lsv.ens-cachan.fr

ENS Cachan – INRIA Futurs

Industrial Context

- Gemplus, world's leader in smart card manufacturing

- Smart card applications
  - Banking (Debit and credit cards, Electronic purse)
  - Security & access control (Identity, Biometrics, Pay TV)
  - Health care cards
  - SIM cards (GSM/GPRS/UMTS networks)
  - Multi-applications cards (Multos, Java Card)

State-of-the-art smart card

- Embedded system with major hardware constraints
- *post-issuance* principle $\Rightarrow$ mobile code security
- Next generation: multi-threading, garbage-collection, IP-networking...

$\Rightarrow$ always more and more reliability:

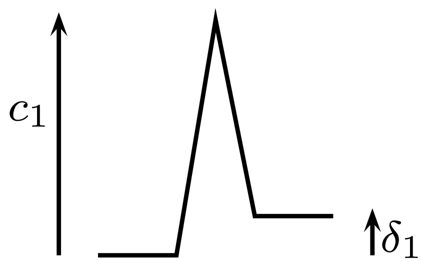
- Information protection
  - hardware: *tamper resistance*, software: cryptography
- Safety of application
  - Mobile code verification: Leroy (2002), Casset *et al.* (2002)
- Guarantee of execution
  - Resource control

Problem

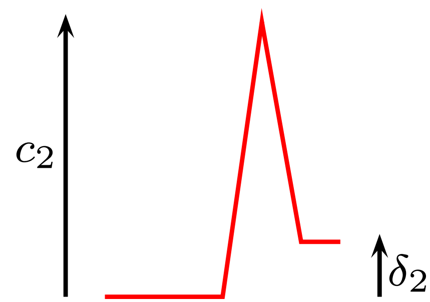
- "Contract-based approach"
  Problem of trust: verify that the contract is valid (safe)
  - *runtime* : monitoring
  - *loading* : code analysis, proof
- Resource management:
  - reserve and lock all the required resource at start-up (Java Card)
    $\Rightarrow$ waste of resource when multiple applets are used
- *Goals:*
  (1) Guarantee resource availability for a safe execution
  (2) Optimizing resource usage

Problem

- One limited resource, several applications
- Usual contracts:

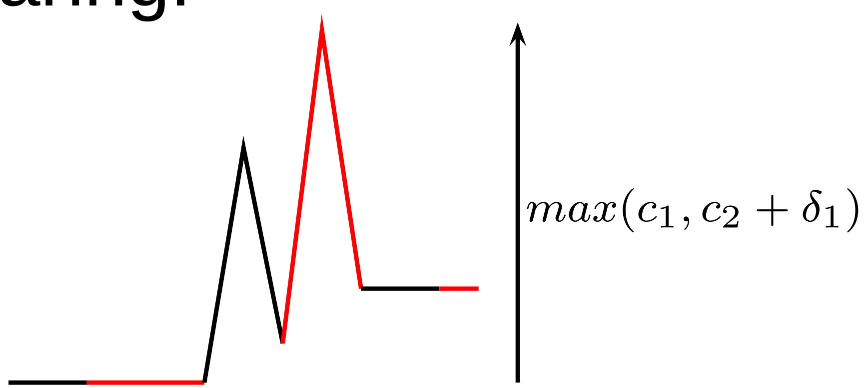


and



require $c_1 + c_2$.

- Could be more sparing:



Our approach

- Improve contracts and task-scheduling
- Three ingredients:
  - Tasks suspended on impossible allocations
  - *Deadlock-avoidance* algorithm
  - Static analysis to annotate the code and compute precise contracts
- Hypothesis:
  - Possible to bound (de)allocations statically
  - Finite execution times (so no starvation)
  - No other interaction

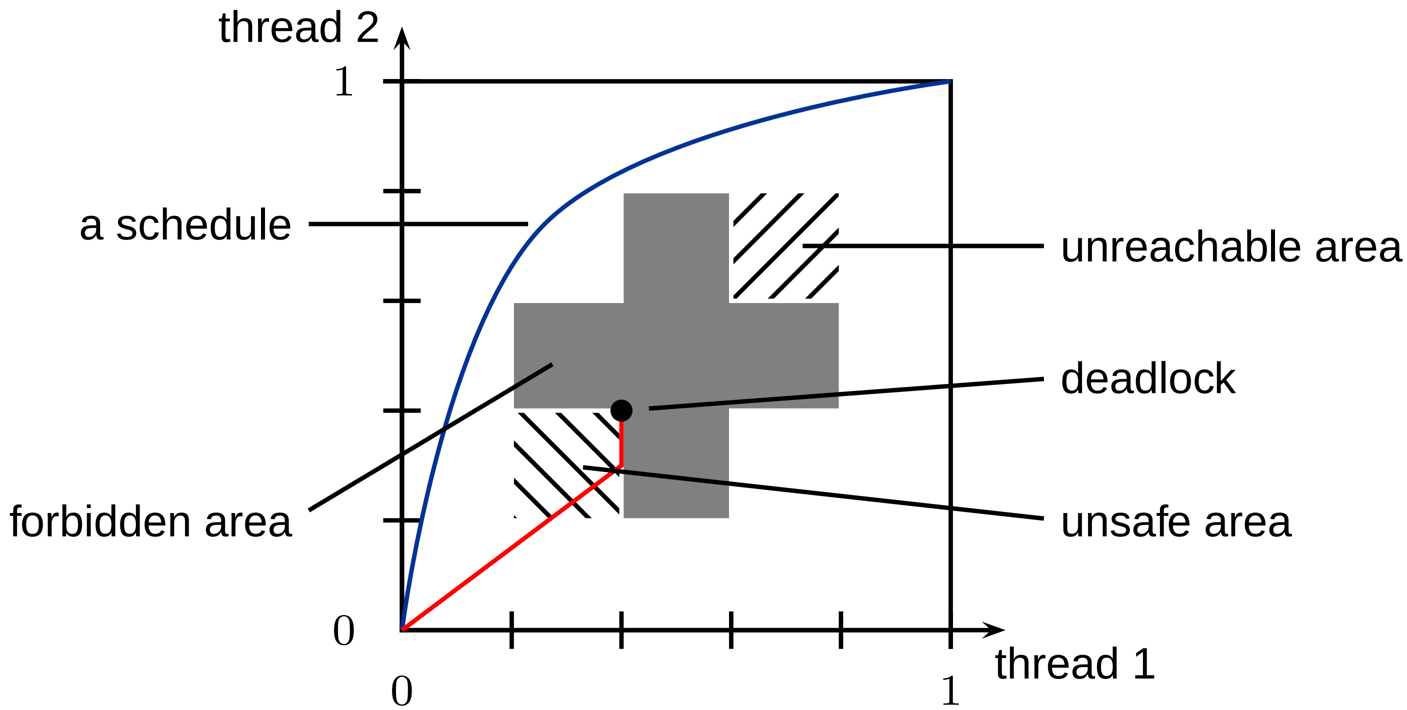
Outline

- Deadlock avoidance
- Theoretical materials
  - Process algebra
  - Efficient safety criterion
  - Abstract domain
- Practical results
  - Java bytecode analyzer
  - Deadlock-avoidance library for Java

Deadlock avoidance

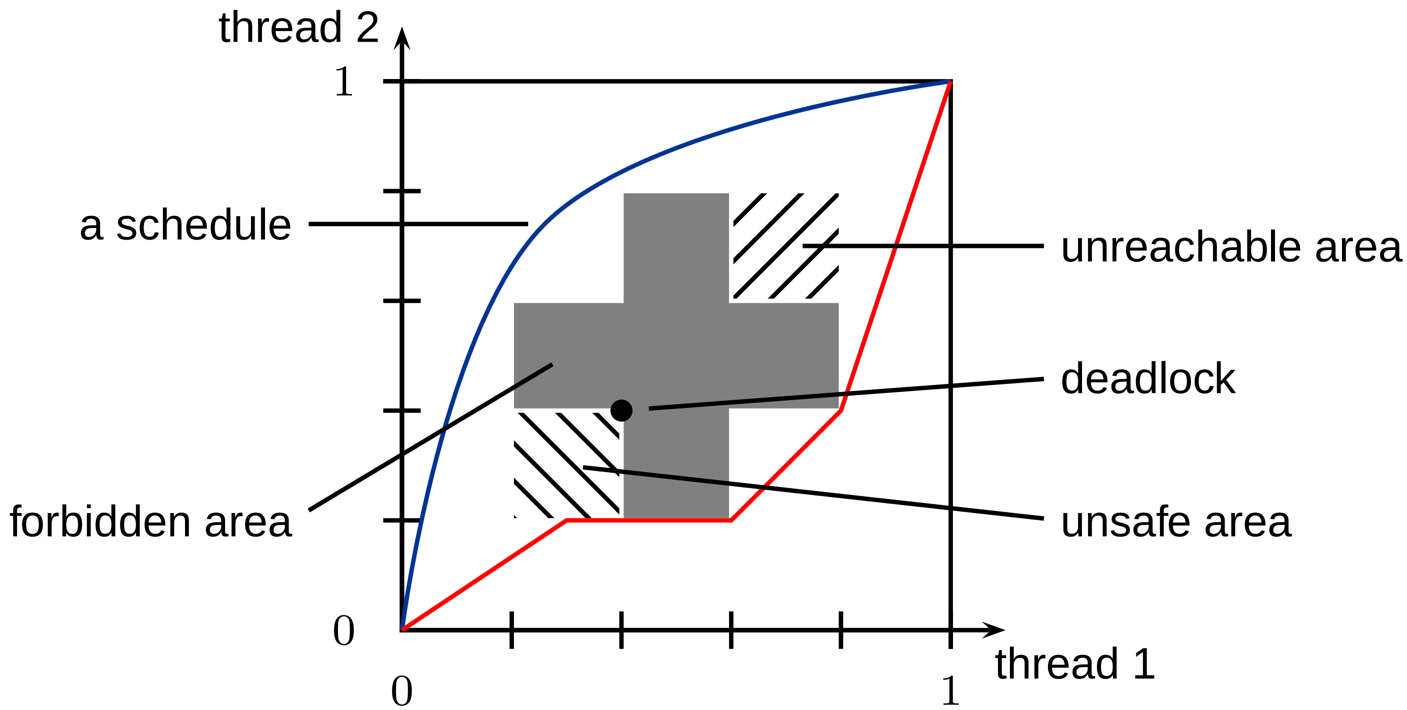
Principle of deadlock avoidance

- Progress graphs (Dijkstra):



Principle of deadlock avoidance

- Progress graphs (Dijkstra):



→ Detect and avoid unsafe areas to avoid deadlocks

- Conservative approximations possible, but beware of liveness

Why new algorithms ?

- Existing works: Dijkstra (1965), Habermann (1969), Holt (1972), Gold (1978)

- Allocations inside real programs: nested forks, branches, loops, function calls. . .
  $\longrightarrow$ *semantic* objects.

- Need to compute contracts from applications, and to add code annotations
  $\longrightarrow$ static code analysis

$\longrightarrow$ “Semantic approach” to deadlock avoidance

Theoretical materials

Process algebra

- Abstract model for the system state:

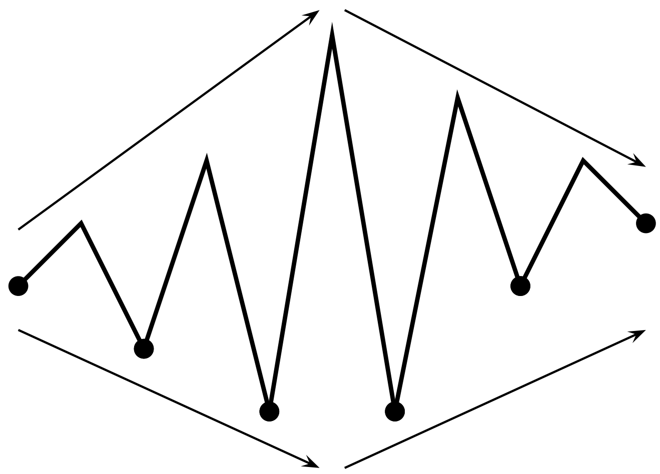
$$\begin{array}{rcll} p & ::= & \epsilon & \text{empty process} \\ & | & x & \text{variation } x \in \mathbb{Z} \text{ of resource} \\ & | & (p_1\, p_2) & \text{sequence} \\ & | & (p_1 \parallel p_2) & \text{concurrent execution} \end{array}$$

- Small-step semantics $\stackrel{x}{\longrightarrow}$, execution traces l
- Safery criterion: enough resource to end

$$\mathcal{C}(p) \stackrel{def}{=} \min_{p \stackrel{l}{\longrightarrow} \epsilon} \mathcal{C}(l) \leq M$$

Efficient computation of $\mathcal{C}(p)$

- Recursive translation $L(p)$ to *normalized lists*



$(1,-1)(3,-1)(6,0)\ (5,2)\ (2,1)$

- Exact computation: $\mathcal{C}(L(p)) = \mathcal{C}(p)$
- Worst-case complexity: $O(depth \times size)$
- Linear in practice

Remaining issues

- Wish to use *normalized lists* for: static analysis, code annotations, contracts.
- Semantic quasi-ordering: $L(p_1) \sqsubseteq L(p_2)$ iff $\mathcal{C}(C[p_1]) \leq \mathcal{C}(C[p_2])$ for every context C
- Minimal data-structure ? (antisymmetry)
- How to decide $\sqsubseteq$?
- Existence of a l.u.b. operator $\sqcup$?
  $\longrightarrow$ Useful for abstract interpretation (branches, loops)

Properties of normalized lists

- A rich data-structure:
  - allocations
  - concatenation
  - parallel product
  - ordering $\sqsubseteq$
  - least upper bound $\sqcup$
  - greatest lower bound $\sqcap$
  - least element $\bot$
  - greatest element $\top$
- Linear complexities w.r.t. length.

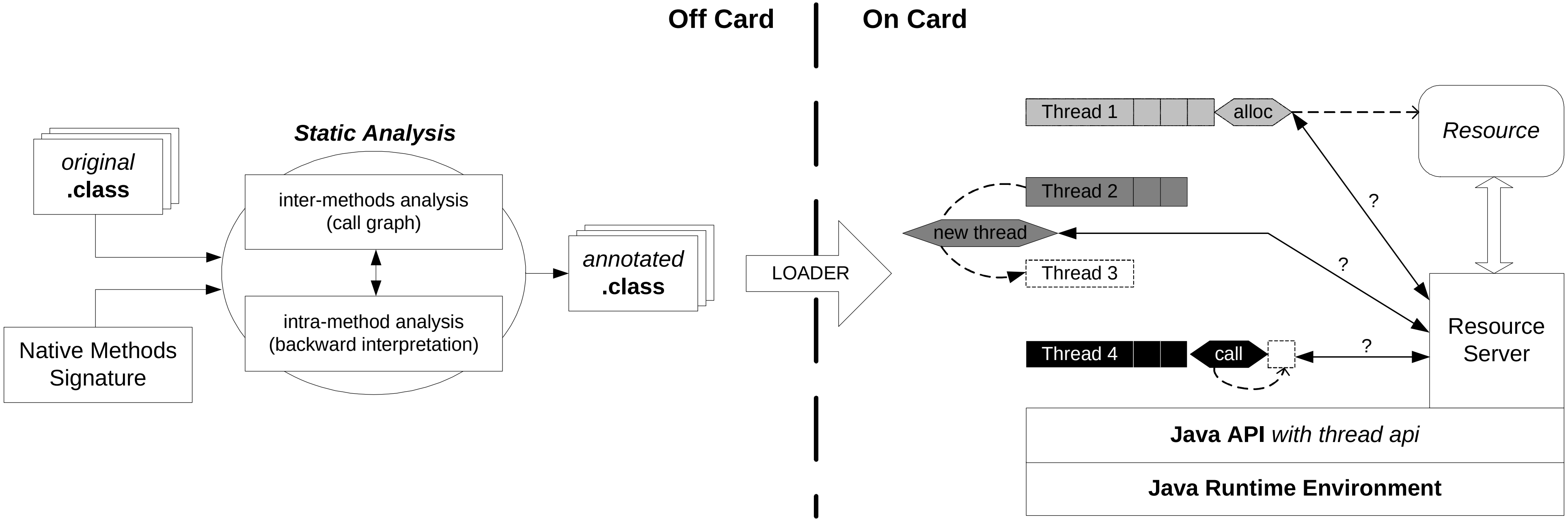
$\longrightarrow$ Domain for abstract interpretation

see Galland and Baudet (APLAS 2003)

Practical results

Overview

- Prototype in Java for Java bytecode,
- Abstract scalar resource,
- Global architecture:



Annotations and runtime library

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);

  }
}
```

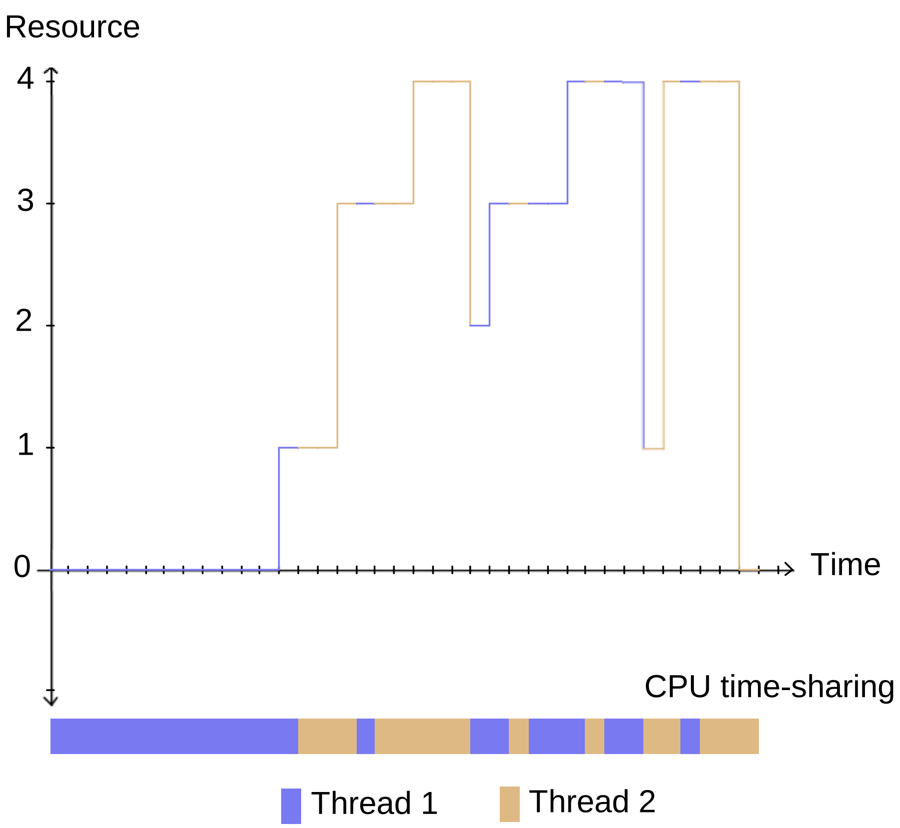
After

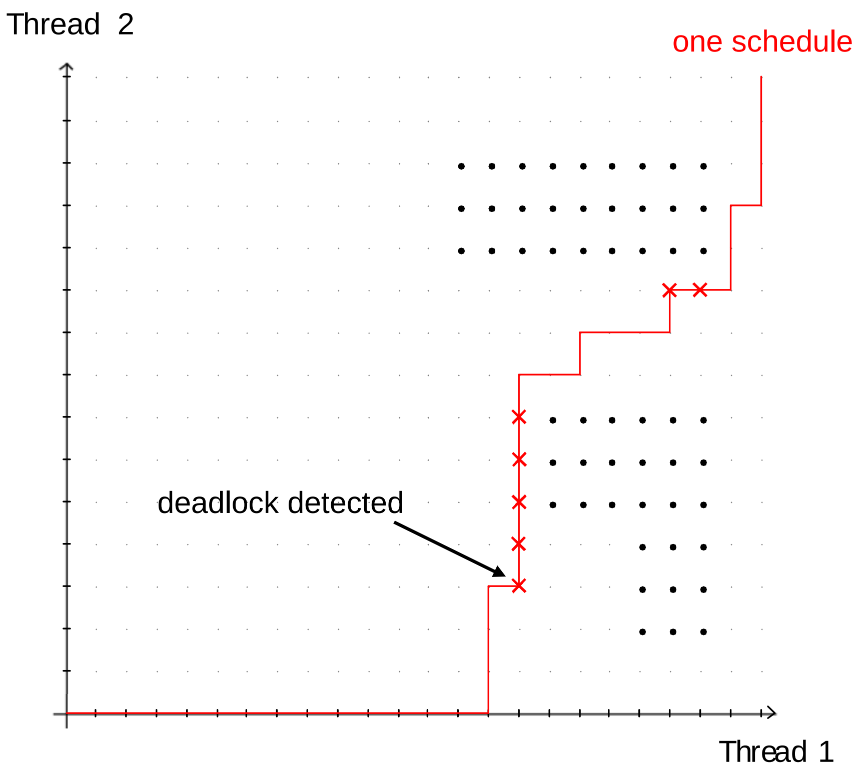
```
class SimpleExample implements Executable {

  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();
  }
}
```

$$(5,2)(2,1) = \underbrace{L(1)}_{} \cdot (\underbrace{(L(4) \cdot L(-3))}_{} \times (\underbrace{(L(-2) \sqcup L(2))}_{} \cdot \underbrace{L(-1)}_{}))$$

An example

- Simple Java program with two threads





- Allocated resource

- Progress graph

Conclusion

- A more sparing approach to resource control:
  - fast deadlock-avoidance algorithm
  - new abstract domain for static analysis
- Applied to Java
- Future works:
  - Non-terminating idioms
  - Contract verification
  - Many resources
  - Apply these results to a realistic resource. Why not memory ? (escape analysis)

Thank you !

Q&A