Memory Consumption Analysis of Java Smart Cards

GERARDO SCHNEIDER

University of Oslo - Norway

Joint work with PABLO GIAMBIAGI (SICS, Sweden)

Partially supported by the RNTL French project CASTLES (*Conception d'Analyses Statiques et de Tests pour le Logiciel Embarqué Sécurisé*). IRISA/INRIA - Rennes, INRIA Sophia-Antipolis, Oberthur Card Systems and AQL

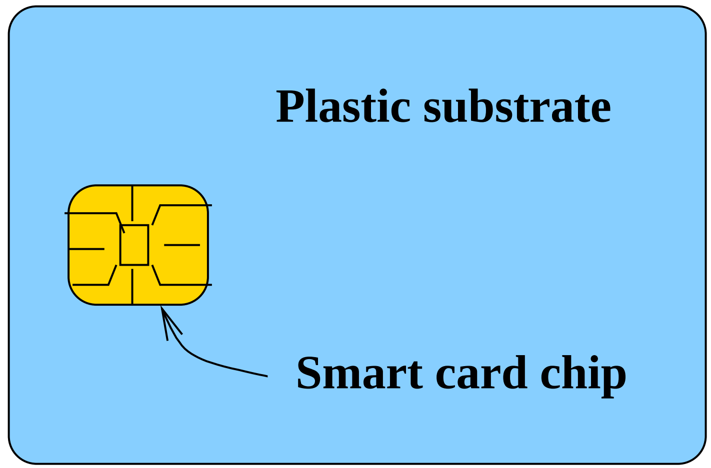
CLEI'05 - Cali, Colombia - October 2005

Overview

- Introduction and motivation
- Objective - Our approach
- Final discussion

Introduction and Motivation

Smart cards



- Small communicating devices with restricted resources
- Execute stand-alone applications specifically written for the hardware it runs on

New generation of Java smart cards

- High-level language for programming applets (JavaCard Language)
- Multi-application: various applets may be downloaded and interact in the same card
- Post-issuance: applets may be loaded on the card after issued by the manufacturer

Size (banking - high-tech cards): EEPROM (16K - 64K), ROM (16K - 200K), RAM (1K - 4K)

Applications: mobile phones, e-purse, e-identity, medical file management, etc

Security Issues

Downloaded applets may attack by leaking or modifying confidential information, causing malfunctioning, etc

Security Issues

Downloaded applets may attack by leaking or modifying confidential information, causing malfunctioning, etc

The “Sandbox” model relies on that applets are:

- Compiled to bytecode for a virtual machine
- Not given direct access to hardware resources
- Subject to a static analysis: bytecode verification (checks applets are well-typed)

Security Issues (cont.)

Extensions of the bytecode verifier are needed to guarantee (among others)

- Information flow (i.e. an applet does not “leak” confidential information)
- Reactiveness (bounding the running time of the applet between two interactions with the environment)
- Availability of services

Security Issues (cont.)

Extensions of the bytecode verifier are needed to guarantee (among others)

- Information flow (i.e. an applet does not “leak” confidential information)
- Reactiveness (bounding the running time of the applet between two interactions with the environment)
- Availability of services (resource-awareness analysis - Memory)

How to program in small devices?

Quoted from “Java Card Technology for Smart Cards - Sun Series” [Chen,2000; Chapter 13]

- “...neither persistent nor transient objects should be created willy-nilly.”
- “You should also limit nested method invocations...”
- “..applets should not use recursive calls.”
- “An applet should always check that an object is created only once.”

The problem

- Nothing in the standards prevents a(n) (intentionally) badly written applet to allocate all persistent memory on a card!
- State-of-the-art tools do not detect whether a given applet will make the card run out of memory

Example:

```
public class Example
    ...
    while(arg > 0)
      new Example();
    ...
```

Objectives - Our Approach

Objective

An analyser for estimating memory usage on Java smart cards, which

- Statically analyses the bytecode
- Does not assume any structure on the bytecode
- Comprises intra- and inter-procedural analysis
- Is as precise as possible
- Is compositional/extensible
- Has low complexity (on-card analyser)

The JavaCard bytecode language

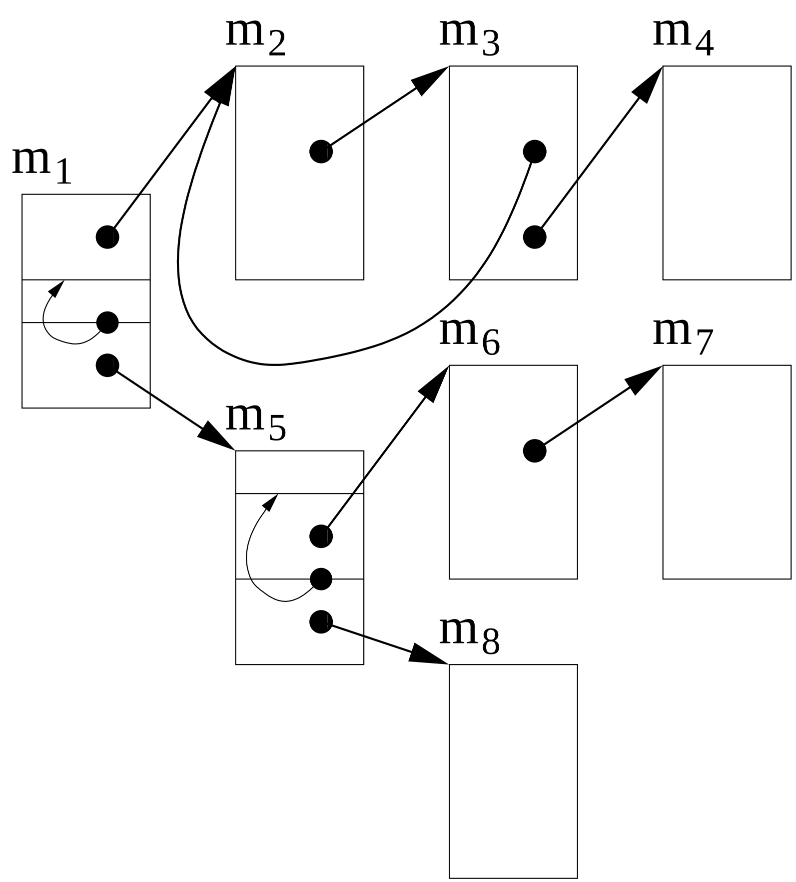
- Stack manipulation: `push`, `pop`, `dup`, `dup2`, `swap`, `numop`;
- Local variables manipulation: `load`, `store`;
- Jump instructions: `if`, `goto`;
- Heap manipulation: `new`, `putfield`, `getfield`;
- Array instructions: `arraystore`, `arrayload`;
- Method calls and return: `invokevirtual`, `invokedefinite`, `return`
- Exceptions and subroutines

Algorithm - Outline

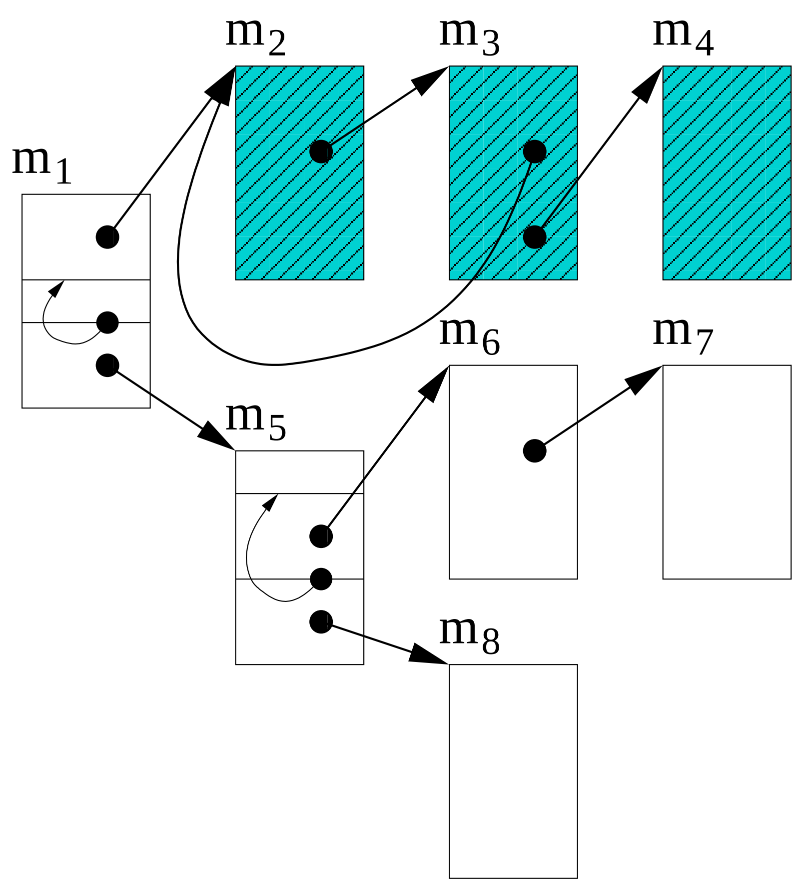
- Detection of (mutually) recursive methods and methods reachable from those (Rec)
- Detection of potential intra-method loops ($Loop$)
- Propagation of $Loop$ inter-procedurally ($Loop$')
- Identification of dynamic instantiation of classes (Γ)

Rec, $Loop$ and $Loop$' are functions associating a set to pairs (m, pc)

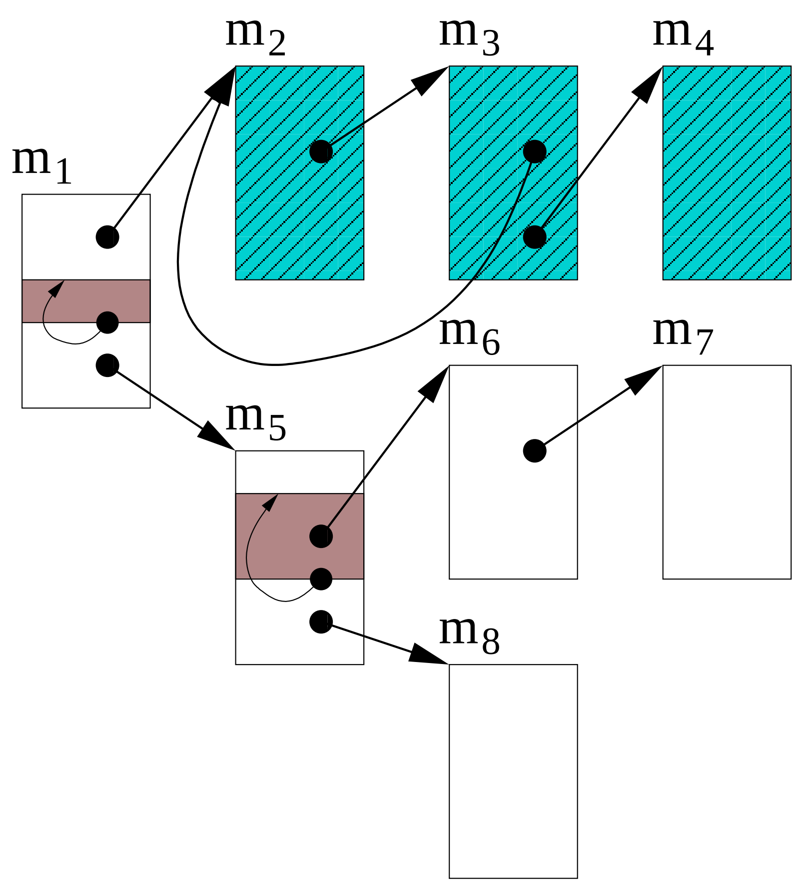
Example: *Rec*, *Loop* and *Loop*'



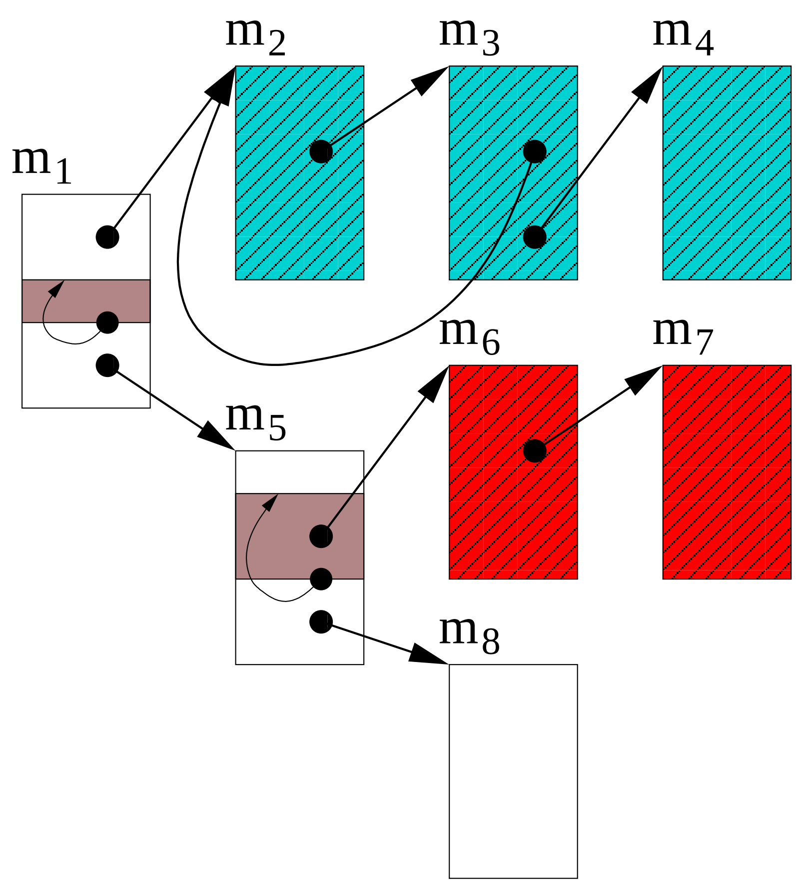
Example: *Rec*, *Loop* and *Loop*’



Example: *Rec*, *Loop* and *Loop*'



Example: *Rec*, *Loop* **and** *Loop'*



Example - Detecting loops ($Loop$)

method m

1 goto 4

2 ...

3 goto 2

4 return

Example - Detecting loops ($Loop$)

method m

1 goto 4	$Loop$(m,1) = {1}
2 ...	$Loop$(m,2) = {}
3 goto 2	$Loop$(m,3) = {}
4 return	$Loop$(m,4) = {}

Example - Detecting loops ($Loop$)

method m

1 goto 4	$Loop$(m,1) = {1}
2 ...	$Loop$(m,2) = {}
3 goto 2	$Loop$(m,3) = {}
4 return	$Loop$(m,4) = {1,4}

Example - Detecting loops ($Loop$)

method m

1 goto 4	$Loop$(m,1) = {1}
2 ...	$Loop$(m,2) = {2}
3 goto 2	$Loop$(m,3) = {}
4 return	$Loop$(m,4) = {1,4}

Example - Detecting loops ($Loop$)

method m

1 goto 4	$Loop$(m,1) = {1}
2 ...	$Loop$(m,2) = {2}
3 goto 2	$Loop$(m,3) = {2}
4 return	$Loop$(m,4) = {1,4}

Example - Detecting loops ($Loop$)

method m

1 goto 4	$Loop$(m,1) = {1}
2 ...	$Loop$(m,2) = {2,●}
3 goto 2	$Loop$(m,3) = {2}
4 return	$Loop$(m,4) = {1,4}

Example - Detecting loops ($Loop$)

method m

1 goto 4	$Loop$(m,1) = {1}
2 ...	$Loop$(m,2) = {2,•}
3 goto 2	$Loop$(m,3) = {2,•}
4 return	$Loop$(m,4) = {1,4}

Example - Detecting loops ($Loop$)

method m

1 goto 4	$Loop$(m,1) = {1}
2 ...	$Loop$(m,2) = {2,•}
3 goto 2	$Loop$(m,3) = {2,•}
4 return	$Loop$(m,4) = {1,4}

A reasonable complex applet may have hundreds of LoC and around 50 jumps!

Form of the constraint rules

For each function Δ (Rec, $Loop$ and $Loop$'), the specification is given by a set of constraint rules of the form:

$$\frac{(m, pc) : \texttt{Instr} \qquad \texttt{Cond}}{f(\Delta(m, pc)) \sqsubseteq \Delta(m', pc')}$$

- $\texttt{Instr}$ is the current instruction
- $\texttt{Cond}$ is a set of conditions (predicate)
- f is a monotonic function
- (m', pc') is the *next* instruction

Detecting loops ($Loop$)

$$\frac{}{\{1\} \sqsubseteq Loop(m, 1)}$$

$$\frac{(m, pc) : \texttt{invokevirtual}\ m'}{Loop(m, pc) \sqsubseteq Loop(m, pc+1)}$$

$$\frac{(m, pc) : \texttt{goto}\ pc'}{F(Loop(m, pc), pc') \sqsubseteq Loop(m, pc')}$$

$$\frac{(m, pc) : \texttt{return}}{\bot \sqsubseteq Loop(m, \mathrm{END}_m)}$$

$$\frac{(m, pc) : \texttt{if}\ t\ op\ \texttt{goto}\ pc'}{\begin{array}{l} F(Loop(m, pc), pc') \sqsubseteq Loop(m, pc') \\ F(Loop(m, pc), pc+1) \sqsubseteq Loop(m, pc+1) \end{array}}$$

$$\frac{(m, pc) : \texttt{Instr}}{Loop(m, pc) \sqsubseteq Loop(m, pc+1)}$$

`Instr` is any instruction different from the ones appearing in the rules and also from `throw` and `jsr`

Spec. of the main algorithm - Γ

- Similar rules to $Loop$ are defined for $Loop$' and Rec

Spec. of the main algorithm - Γ

- Similar rules to $Loop$ are defined for $Loop$' and Rec

Let $Cycle_{m,pc} \equiv Loop_{m,pc} \vee Loop'_{m,pc} \vee Rec_{m,pc}$

$$\Gamma(m, pc) = \begin{cases} \infty & \text{if } (m, pc) : \texttt{new}(cl) \wedge Cycle_{m,pc} \\ 1 & \text{if } (m, pc) : \texttt{new}(cl) \wedge \neg Cycle_{m,pc} \\ 0 & \text{otherwise} \end{cases}$$

Spec. of the main algorithm - Γ

- Similar rules to $Loop$ are defined for $Loop$' and Rec

Let $Cycle_{m,pc} \equiv Loop_{m,pc} \vee Loop'_{m,pc} \vee Rec_{m,pc}$

$$\Gamma(m, pc) = \begin{cases} \infty & \text{if } (m, pc) : \texttt{new}(cl) \wedge Cycle_{m,pc} \\ 1 & \text{if } (m, pc) : \texttt{new}(cl) \wedge \neg Cycle_{m,pc} \\ 0 & \text{otherwise} \end{cases}$$

Fix-point computations: Rec, $Loop$ and $Loop$'!

Algorithm - How does it work?

- The domains (lattices) used and the “form” of the constraints guarantee the existence of a *least fix-point*
- The well-foundedness of the lattices guarantees termination
- A constraint solver computes the least fix-point

Exceptions and Subroutines

- The `finally` block of a `try...finally` Java construct is compiled into a subroutine, a fragment of code called with the `jsr` bytecode instruction
- In Java, exceptions are thrown using the `throw` instruction, compiled into `throw`
- Other forms of exceptions (`try...catch`) are compiled into `invokevirtual` method calls (accessing the Exception Table)

Exceptions and Subroutines (cont.)

We have extended the above algorithm to handle subroutines and `throw` exceptions by adding rules to $Loop$ and Rec

- Added rules for handling subroutines

$$\frac{(m, pc) : \texttt{jsr}\ pc'}{\begin{array}{l} F(Loop(m,pc)) \sqsubseteq Loop(m, pc') \\ F(Loop(m,pc)) \sqsubseteq Loop(m, pc+1) \end{array}} \qquad \frac{(m, pc) : \texttt{ret}\ i}{\bot \sqsubseteq Loop(m, \mathrm{END}_{ret})}$$

- Similar rules for treating exceptions

Exceptions and Subroutines (cont.)

We have extended the above algorithm to handle subroutines and `throw` exceptions by adding rules to $Loop$ and Rec

- Added rules for handling subroutines

$$\frac{(m, pc) : \texttt{jsr}\ pc'}{\begin{array}{l} F(Loop(m, pc)) \sqsubseteq Loop(m, pc') \\ F(Loop(m, pc)) \sqsubseteq Loop(m, pc + 1) \end{array}} \qquad \frac{(m, pc) : \texttt{ret}\ i}{\bot \sqsubseteq Loop(m, \mathrm{END}_{ret})}$$

- Similar rules for treating exceptions

We don't need to change the previous defined rules!

Final Discussion

Achievements

- We have written a constraint-based algorithm for detecting possible memory overflow due to dynamic instantiation of classes inside cycles
- Handwritten proof of
  - Termination
  - Soundness and completeness w.r.t. to an abstraction of the operational semantics

Features of our algorithm

- \+ Written in a “good” way to be fed into Coq (certification)
- \+ Rec, $Loop$ and $Loop$’ reusable/extensible
- \+ Static analysis
- +/- Low space and time complexity
- +/- Compositional
- – Over-approximation:
  - It detects (all the) syntactic cycles
  - An instruction in a method (not in a cycle) called more than once is counted once

Related Work

- In [CJPS05]: a certified analyser for Java card bytecode
  - Constraint-based
  - Formalisation based on abstract interpretation
  - A proof of the algorithm soundness in Coq
  - Extraction of OCAML code from its Coq's proof

[CJPS05] D. Cachera, T. Jensen, D. Pichardie and G. Schneider. Certified Memory Usage Analysis. In: Formal Methods. LNCS 3582, p.91-106. July 2005

Contributions (comparison)

- Improved the algorithm presented in [CJPS05]
  - Our algorithm performs better in terms of space-complexity (for a method with 200 lines and 50 basic blocks $Loop$ uses 10 KB vs 40 KB)
  - We treat exceptions (partially)
  - We treat subroutines
- Time complexity is similar (computation of fix-points converges at most in 4 iterations)
- No Coq proof in our work (paper-proof of its correctness and completeness)

Improvements to be done

- Implementation would improve efficiency
- Treat all the cases of exceptions (not difficult!)
- Propagate the pc-numbers of basic blocks only to relevant points (not difficult!)
  - For analysing an applet with methods containing 50 basic blocks (independently of the Nr of LoC) $Loop$ would need only 2.5 KB!
- Extend the analysis for “open” composite applets (a bit more difficult!)

Thank you very much!
Questions?

Research on this topic?

- Fortunately, there are many interesting M.Sc. (Ph.D.) research possibilities related to the topic of this talk

Research on this topic?

- Fortunately, there are many interesting M.Sc. (Ph.D.) research possibilities related to the topic of this talk

- Unfortunately, I don't have money for scholarships

Detecting recursive methods (Rec)

$$\frac{(m, pc) : \texttt{invokevirtual}\ m' \qquad m = m'}{\begin{array}{c} Rec(m, pc) \cup \{m, \bullet\} \sqsubseteq Rec(m', 1) \\ Rec(m, pc) \sqsubseteq Rec(m, pc+1) \end{array}}$$

$$\frac{(m, pc) : \texttt{return}}{Rec(m, pc) \sqsubseteq Rec(m, \mathrm{END}_m)}$$

$$\frac{(m, pc) : \texttt{invokevirtual}\ m' \qquad m \neq m'}{\begin{array}{c} G(Rec(m, pc), m') \sqsubseteq Rec(m', 1) \\ Rec(m, pc) \sqsubseteq Rec(m, pc+1) \end{array}}$$

$$\frac{(m, pc) : \texttt{Instr}}{Rec(m, pc) \sqsubseteq Rec(m, pc+1)}$$

Rules for *Loop*'

$$\frac{(m, pc) : \texttt{invokevirtual}\ m' \quad Loop_{m,pc}}{\begin{array}{l}\bullet \sqsubseteq Loop'(m', 1) \\ Loop'(m, pc) \sqsubseteq Loop'(m, pc+1)\end{array}}$$

$$\frac{(m, pc) : \texttt{Instr}}{Loop'(m, pc) \sqsubseteq Loop'(m, pc+1)}$$

$$\frac{(m, pc) : \texttt{invokevirtual}\ m' \quad \neg Loop_{m,pc}}{\begin{array}{l}Loop'(m, pc) \sqsubseteq Loop'(m', 1) \\ Loop'(m, pc) \sqsubseteq Loop'(m, pc+1)\end{array}}$$

$$\frac{(m, pc) : \texttt{return}}{\bot \sqsubseteq Loop'(m, \mathrm{END}_m)}$$

Definition of the functions F and G

$$F(L_{m,pc}, pc') = \begin{cases} L_{m,pc} \cup \{\bullet\} & \text{if } pc' \in L_{m,pc} \\ L_{m,pc} \setminus \{\bullet\} \cup \{pc'\} & \text{otherwise} \end{cases}$$

$$G(R_{m,pc}, m') = \begin{cases} R_{m,pc} \cup \{m, \bullet\} & \text{if } m' \in R_{m,pc} \\ R_{m,pc} \cup \{m\} & \text{if } m' \notin R_{m,pc} \end{cases}$$

Rules for Handling Exceptions

$$\frac{(m, pc) : \texttt{throw}\ e \qquad (m, pc') \in \mathit{findHandler}(m, pc, e)}{F(\mathit{Loop}(m, pc)) \sqsubseteq \mathit{Loop}(m, pc')}$$

$$\frac{(m, pc) : \texttt{throw}\ e \qquad (m', pc') \in \mathit{findHandler}(m, pc, e) \qquad m' \neq m}{G(\mathit{Rec}(m, pc), m') \sqsubseteq \mathit{Rec}(m', pc')}$$

Some M.Sc. (Ph.D.) subjects

- Implement the O.S. of the JCVM, and the (optimised) analysis in Maude
- Prove correctness of the algorithm in Coq (using a prefix semantics) and extract the program
- Specify an implement a modular analysis in order to minimise global fix-point computations

Objective (Cont.)

The technique used should allow us to:

- Develop a **certified analyser**
- Extract a correct analyser

Moreover, we want the formalism to be compatible with previous work (certified Data Flow Analyser developed at IRISA)

How to obtain a certified analyser?

- Formalise the operational semantics of the language in a Proof Assistant (Coq)
- Define the abstract domains (lattices)
- Prove well-foundedness of the lattices
- Code the algorithm into Coq (as a constraint-based algorithm)
- Prove the correctness of the algorithm w.r.t. (an abstraction of) the operational semantics
- Extract a program (proof-as-program paradigm) using Coq's extraction mechanism

How to obtain a certified analyser?

- Formalise the operational semantics of the language in a Proof Assistant (Coq)
- Define the abstract domains (lattices)
- Prove well-foundedness of the lattices
- Code the algorithm into Coq (as a constraint-based algorithm)
- Prove the correctness of the algorithm w.r.t. (an abstraction of) the operational semantics
- Extract a program (proof-as-program paradigm) using Coq's extraction mechanism