

# Abstraction-Carrying Code

**Elvira Albert\***, **Germán Puebla\*\*** and **Manuel Hermenegildo\*\*,\*\***

*(\*) Complutense University of Madrid (Spain)*

*(\*\*) Technical University of Madrid (Spain)*

*(\*\*\* ) University of New Mexico (USA)*

11TH INTERNATIONAL CONFERENCE ON LOGIC FOR  
PROGRAMMING ARTIFICIAL INTELLIGENCE AND REASONING  
(LPAR'04)

Uruguay, March 14-18, 2005



# Motivation

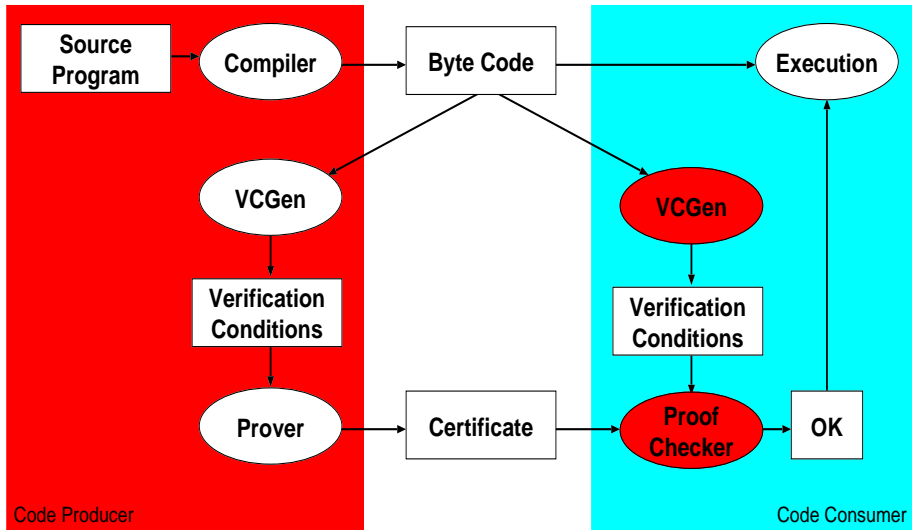
- **Mobile computing** is characterized by having a relatively large number of *untrusted* computing devices which interact.
  - ▶ Assurance of the safety and characteristics of the code received and also the kind of load it is going to pose.
  - ▶ At the same time, it is essential to simplify such verification process and reduce its resource usage.
- **Proof-Carrying Code** (PCC) and related approaches to mobile code safety involve associating safety information in the form of a *certificate* to programs.
- Code suppliers compute tamper-proof certificates and pass them along with the code which simplify code verification.
- **Abstraction-carrying code** follows the standard approach of associating certificates to programs but it is based throughout on the technique of abstract interpretation.



# Introduction: Proof-Carrying Code

- Security techniques verify that the execution of a program is *safe*, i.e., meets certain properties of predefined *safety policy*.
- PCC and related approaches to mobile code safety involve associating safety information in the form of a *certificate* to programs.
- Certificate (or proof) created at compile time by the code supplier, packaged with untrusted code.
- Code consumer can then run a *verifier* which, by a straightforward inspection of code and certificate, verifies validity of certificate and thus compliance with safety policy.
- The key benefit of this “certificate-based” approach to mobile code safety is that the burden of ensuring compliance with the desired safety policy is shifted from the consumer to the supplier.





# Advantages of PCC

- The verifier, or proof checker, performs a task that should be much simpler, efficient, and automatic than generating the certificate
- The implementation of the checking algorithm is part of the safety-critical infrastructure and we want to minimize it
- The local host could be a small embedded system that lacks computing resources to run large and complex programs
- The checking will be performed by every consumer (whilst the certification generation is done only once by the supplier).



# Certificate-based Approaches

- Well-known methods following this approach are:
  - ▶ Proof-Carrying Code (PCC) [Necula'97]
  - ▶ Typed Assembly Languages (TAL) [Morrisett et al.'99].
- The certificate may take different forms:
  - ▶ In PCC the certificate is originally a proof in first-order logic of certain verification conditions.
  - ▶ A recent proposal [Bernard and P. Lee'02] uses temporal logic to specify security policies in PCC.
  - ▶ In TAL, the certificate is a type annotation of the assembly language program.



# Fundamental Challenges

- The design of mobile code safety systems based on certificates shares the same, fundamental challenges:
  - 1 defining *expressive safety policies* covering a wide range of properties,
  - 2 solving the problem of how to *automatically generate the certificates* and,
  - 3 designing *simple, reliable, and efficient checkers* for the certificates.
- The various approaches differ in expressiveness, flexibility, and efficiency, but share the goal of using safety information to make untrusted mobile code safe and efficient.



# Abstraction-Carrying Code

- The design of our abstract interpretation-based system for mobile code safety follows the “certificate-based” approach but based on abstract interpretation:
  - 1 An assertion language for specifying complex program properties including safety and resource-related properties.
  - 2 A fixpoint static analyzer is used to automatically infer information about the mobile code which can then be used to prove that the code is safe.
  - 3 A simple, easy-to-trust analysis checker used at receiving end to verify the validity of the information on the mobile code. It is indeed a specialized abstract interpreter.
- The resulting scheme has been incorporated in CiaoPP (the preprocessor of the Ciao system).



# The Ciao System

- **Ciao:** next-generation multi-paradigm language with:
  - ▶ declarative (LP-based) kernel designed to be:
    - ★ *Very extensible*  $\Rightarrow$  multi-paradigm:  
ISO-Prolog, functions, constraints, higher-order, objects.
    - ★ *Analysis “aware”*  $\Rightarrow$   
assertion language, automatic static inference and checking, autodoc,  
...
  - ▶ “Industry standard” performance.
  - ▶ Robust module/object system, separate/incremental compilation.
  - ▶ Concurrency, parallelism, distributed execution, ...
  - ▶ (Semi-automatic) interfaces to other languages, databases, etc.
  - ▶ Complete program development environment.

(Free Software – GNU LGPL license.)



# Supporting Framework: the CiaoPP System

- CiaoPP is the abstract interpretation-based preprocessor of the Ciao multi-paradigm constraint logic programming system.
- It uses modular, incremental abstract interpretation as a fundamental tool to obtain information about the program.
- In CiaoPP, the semantic approximations produced by the analysis have been applied to:
  - ▶ high- and low-level optimizations during program compilation
  - ▶ and the more general context of program development.
- This work extends the validation/verification framework available in CiaoPP to mobile code safety



# The Certification Process

- The certification process starts from an initial program and a set of assertions which encode the safety policy the program should meet

```
:- entry reverse : list * var.  
reverse( [] ) := [].  
reverse( [H|L] ) := append( reverse(L), [H] ).
```

- The consumer will only accept *pure* tasks, i.e., tasks that have no side effects, and only those of polynomial complexity.

```
:- check comp reverse  
  + sideff(free).  
:- check comp reverse(A,_B)  
  : list * var  
  + steps_ub( o(exp(length(A),2)) ).
```



# The Safety Policy

- The code will be accepted at the receiving end, provided all assertions can be checked, i.e., the intended semantics expressed in the above assertions determines the safety condition.
  - ▶ This can be a policy agreed a priori or exchanged dynamically.
- Unlike traditional safety properties such as, e.g., type correctness, resource-related properties should take into account issues such as load and available computing resources in each particular system.
  - ▶ Thus, for resource-related properties different devices may impose different policies for the acceptance of tasks (mobile code).



# Generation of the Certificate

- Given the previous assertions defining the safety policy, the certificate is automatically generated by a *goal dependent analysis engine*.
- This analyzer receives as input a set of entries which define the base, boundary assumptions on the input data.
- The computation of the analysis process terminates when a fixpoint of a set of equations is reached (*analysis fixpoint*).
- The certification process is based on the idea that the role of certificate can be played by a *particular and small subset of the analysis results* computed by abstract interpretation-based analyses.



## An Example: The Certificate

- The analyzers available in CiaoPP infer the following information:

```
:- true pred reverse(A,_B)
   : ( list * var )
   => ( list * list)
   + ( not_fails, is_det, sideff(free),
       steps_ub( 0.5*exp(length(A),2)+1.5*length(A)+1 ) )
```

- The two check assertions become checked:

```
:- checked comp reverse(A,B)
   + sideff(free).
:- checked comp reverse(A,B): list * var
   + steps_ub( o(exp(length(A),2)) ).
```

- The analysis results above can themselves be used as the *cost and safety certificate* to attest a safe and efficient use of `reverse`.



# The Verification Condition

- The verification process requires generating a *verification condition* (VC) encoding the information in check assertions to be verified.
- This condition is sent to an automatic *validator* which attempts to check its validity w.r.t. the analysis fixpoint.
  - i) The VC is checked (:-checked);
  - ii) It is disproved (:-false);
  - iii) It cannot be proved nor disproved (:-check)
- What does the user do?:
  - ▶ fix program;
  - ▶ more assertions;
  - ▶ another domain.
- The certification process needs to be restarted until reaching i):



## Validation in the Consumer:

- The *validation* process performed by the consumer device is similar to the above certification process except that the analysis engine is replaced by an *analysis checker*.
- The definition of the analysis checker is centered around the observation that the checking algorithm can be defined as a very simplified “one-pass” analyzer.
- Intuitively since the certification process already provides the fixpoint result as certificate, an additional analysis pass over it cannot change the result.
- Thus, as long as the fixpoint is valid, one single execution of the abstract interpreter validates the certificate.



## An Example: Validation in the Consumer

- The code consumer rejects code which does not adhere to some specification, including usage of computing resources.

```
:- check comp reverse(A,_B) : ( list * var )  
  + steps_ub( length(A) + 1 ).
```

- If certificate is valid, the code will be accepted only if the upper bound in the certificate is lower or equal than that in the assertion.
- The certificate contains the (valid) information that `reverse` will take at most  $0,5 (length(A))^2 + 1,5 length(A) + 1$  resolution steps. The assertion requires the cost to be at most  $length(A) + 1$  steps.
- The code received by the consumer cannot be proved to satisfy the efficiency requirements imposed.



## Conclusions / Future Work

- Extended the validation/verification framework available in CiaoPP to mobile code safety by following the standard strategy of associating safety certificates to programs
  - ▶ The certificate takes the form of a particular slice of the analysis results generated by the abstract interpreter
  - ▶ The burden on the consumer side is reduced by replacing an analysis phase by a simple one-traversal analysis checking
  - ▶ The certificate checker on the consumer side is in fact a very simplified and efficient specialized abstract interpreter
- We believe the proposed approach is of interest for bringing the automation and expressiveness which is inherent in the abstract interpretation techniques to the area of mobile code safety.
- The size of certificates needs to be minimized as much as possible. We believe that they can be further reduced by omitting the information which has to be necessarily re-computed by the checker.

