Abstract Interpretation-based Verification/Certification in the CiaoPP System*

Germán Puebla[1], Elvira Albert[2], and Manuel Hermenegildo[1,3]

[1] Facultad de Informática, Technical University of Madrid, german@fi.upm.es
[2] DSIP, Complutense University of Madrid, elvira@sip.ucm.es
[3] Depts. of Comp. Sci. and El. and Comp. Eng., U. of New Mexico, herme@unm.edu

Abstract. 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 programs. In CiaoPP, the semantic approximations thus produced have been applied to perform high- and low-level optimizations during program compilation, including transformations such as multiple abstract specialization, parallelization, partial evaluation, resource usage control, and program verification. More recently, novel and promising applications of such semantic approximations are being applied in the more general context of program development such as program verification. In this work, we describe our extension of the system to incorporate Abstraction-Carrying Code (ACC), a novel approach to mobile code safety. ACC follows the standard strategy of associating safety certificates to programs, originally proposed in Proof Carrying-Code. A distinguishing feature of ACC is that we use an abstraction (or abstract model) of the program computed by standard static analyzers as a certificate. The validity of the abstraction on the consumer side is checked in a single-pass by a very efficient and specialized abstract-interpreter. We have implemented and benchmarked ACC within CiaoPP. The experimental results show that the checking phase is indeed faster than the proof generation phase, and that the sizes of certificates are reasonable. Moreover, the preprocessor is based on compile-time (and run-time) tools for the certification of CLP programs with resource consumption assurances.

1 Abstract Interpretation-based Verification

We start by briefly describing an abstract interpretation-based approach to program verification [23, 5, 21] which constitutes the basis for the certification process carried out in Abstraction Carrying Code (ACC).

Abstract interpretation [7] is now a well established technique which has allowed the development of very sophisticated global static program analyses that are at the same time automatic, provably correct, and practical. The basic idea of abstract interpretation is to infer information on programs by interpreting

* Invited talk at MoveLog'05.

("running") them using abstract values rather than concrete ones, thus obtaining safe approximations of the behavior of the program. An abstract value of an *abstract domain* (D_α) is a finite representation of a, possibly infinite, set of actual values in the *concrete domain* (D). Our approach relies on the abstract interpretation theory [7], where the set of all possible abstract semantic values which represents D_α is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets, both for the concrete $\langle 2^D, \subseteq\rangle$ and abstract $\langle D_\alpha, \sqsubseteq\rangle$ domains. Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma\rangle$: *abstraction* $\alpha : 2^D \rightarrow D_\alpha$, and *concretization* $\gamma : D_\alpha \rightarrow 2^D$, such that $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. In general $\sqsubseteq$ is induced by $\subseteq$ and α. Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in a precise sense.

We consider the important class of semantics referred to as *fixed-point semantics*. In this setting, a (monotonic) semantic operator (which we refer to as S_P) is associated to each program P. This S_P function operates on a semantic domain which is generally assumed to be a complete lattice or, more generally, a chain-complete partial order. The meaning of the program (which we refer to as $[\![P]\!]$) is defined as the least fixed point of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$. A well-known result is that if S_P is continuous, the least fixed point is the limit of an iterative process involving at most ω applications of S_P and starting from the bottom element of the lattice.

Both program verification and debugging compare the *actual semantics* of the program, i.e., $[\![P]\!]$, with an *intended semantics* for the same program, which we denote by $\mathcal{I}$. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. The classical verification problem of verifying that P is *partially correct* w.r.t. $\mathcal{I}$ can be formulated as follows:

$$\boxed{P \text{ is } \textit{partially correct} \text{ w.r.t. } \mathcal{I} \text{ if } [\![P]\!] \subseteq \mathcal{I}}$$

However, using the exact either actual or intended semantics for automatic verification and debugging is in general not realistic, since the exact semantics can be infinite, too expensive to compute, only partially known, etc. An alternative approach is to work with approximations of the semantics. This is interesting, among other reasons, because the technique of abstract interpretation can provide *safe* approximations of the program semantics. For now, we assume that the program specification is given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program is most easily done in the same domain, since then the operators on the abstract lattice, that are typically already defined in the analyzer, can be used to perform this comparison. Thus, for comparison we need in principle $\alpha([\![P]\!])$ and we proceed as follows:

$$\boxed{P \text{ is } \textit{partially correct} \text{ w.r.t. } \mathcal{I}_\alpha \text{ if } \alpha([\![P]\!]) \sqsubseteq \mathcal{I}_\alpha}$$

However, using abstract interpretation, we can usually only compute $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$ and it is computed by the analyzer as

$[\![P]\!]_\alpha = \mathrm{lfp}(S_P^\alpha)$. The operator S_P^α is the abstract counterpart of S_P. A key idea in abstract interpretation-based verification is to use $[\![P]\!]_\alpha$ directly in debugging and verification tasks. The possible loss of accuracy due to approximation prevents full verification in general. However, and interestingly, it turns out that in many cases useful verification and debugging conclusions can still be derived by comparing the approximations of the actual semantics of a program to the (also possibly approximated) abstract intended semantics. Thus, we are interested in studying the implications of comparing $\mathcal{I}_\alpha$ and $[\![P]\!]_\alpha$. Analyses which over-approximate the actual semantics (which we denote $[\![P]\!]_{\alpha^+}$), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification $\mathcal{I}_\alpha$. In particular, a sufficient condition for demonstrating that P is partially correct is as follows:

$$\boxed{P \text{ is } \textit{partially correct} \text{ w.r.t. } \mathcal{I}_\alpha \text{ if } [\![P]\!]_{\alpha^+} \sqsubseteq \mathcal{I}_\alpha}$$

In our approach, we compare $[\![P]\!]_\alpha$ directly to the (also approximate) intention which is given in terms of *assertions* [22]. Such assertions are linguistic constructions which allow expressing properties of programs, as we will explain in the next section.

2 The Abstraction-Carrying Code Framework

Current approaches to mobile code safety, inspired by the technique of *Proof-Carrying Code* (PCC) [20], associate safety information in the form of a *certificate* to programs. The certificate (or proof) is created by the code supplier at compile time, and packaged along with the untrusted code. The consumer who receives the code+certificate package can then run a *checker* which by a straightforward inspection of the code and the certificate, can verify the validity of the certificate and thus compliance with the safety policy. The key benefit of this approach is that the burden of ensuring compliance with the desired safety policy is shifted from the consumer to the supplier. Indeed the (proof) checker performs a task that should be much simpler, efficient, and automatic than generating the original certificate. For instance, in the first PCC system [20], the certificate is originally a proof in first-order logic of certain *verification conditions* and the checking process involves ensuring that the certificate is indeed a valid first-order proof.

The main practical difficulty of PCC techniques is in generating safety certificates which at the same time:

- allow expressing interesting safety *properties*,
- can be generated *automatically* and,
- are easy and *efficient* to check.

The idea of Abstraction Carrying Code (ACC) [1, 2] is a natural extension of our approach to abstract interpretation-based program verification which offers a number of advantages for dealing with the aforementioned issues in the context

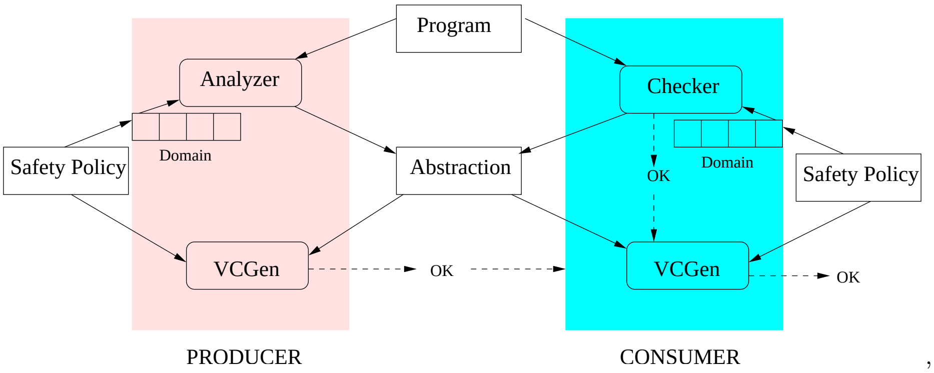

Fig. 1. Abstraction-Carrying Code

of mobile code. In particular, the expressiveness of existing abstract domains will be implicitly available in abstract interpretation-based code certification to define a wide range of safety properties. Figure 1 presents an overview of ACC. The certification process carried out by the code producer is depicted to the left of the figure while the checking process performed by the code consumer appears to the right. In particular, ACC has the following fundamental elements which can handle the challenges of PCC.

The first element, which is common to both producer and consumers, is the Safety Policy. We rely on an expressive class of safety policies based on "abstract"—i.e. symbolic—properties over different abstract domains. Thus, our framework is parametric w.r.t. the abstract domain(s) of interest, which gives us generality and expressiveness. As in the case of simple verification, an expressive assertion language is used to define the safety policy. Given an initial program P, we first define its Safety Policy by means of a set of assertions AS in the context of an abstract domain D_α. The domain is appropriately chosen among a repertoire of Domains available in the system. The assertions are obtained from the assertions for system predicates and those provided by the user.

Once the safety policy is specified, the next element at the producer's side is a fixpoint-based static Analyzer which automatically and efficiently infers an abstract model (or simply *abstraction*) of the mobile code, $[\![P]\!]_\alpha$, in terms of the abstract domain D_α. This abstraction can then be used to prove that the code is safe w.r.t. the given policy. Thus, our certification method is based on the following key idea:

> *An abstraction of the program computed by abstract interpretation-based analyzers can play the role of certificate for attesting program safety.*

The process of returning this *abstraction* of P's execution, $[\![P]\!]_\alpha$, in terms of the abstract domain D_α is well understood for several general types of analyses for Prolog and its (constraint or multi-paradigm) extensions [8, 3, 18, 16, 6, 11]. In particular, our implementation is based on PLAI [18, 17], a generic engine which has the description domain and functions on this domain as parameters.

The verification condition generator, VCGen in the figure, generates, from the initial safety policy and the abstraction, a *Verification Condition* (VC) which can be proved only if the execution of the code does not violate the safety policy. The formal definition of VCGen is outside the scope of this paper (it can be found in [2]). Intuitively, the verification condition is a conjunction of boolean expressions whose validity ensures the consistency of a set of assertions w.r.t. the answer table computed by the analyzer. The condition is sent to an automatic validator which attempts to check its validity w.r.t. the answer table. This validation may yield three different possible status: i) the verification condition is indeed checked (marked as OK in Fig. 1), then the certificate (i.e., the abstraction) is considered valid, ii) it is disproved, and thus the certificate is not valid and the code is definitely not safe to run (we should obviously correct the program before continuing the process); iii) it cannot be proved nor disproved, which may be due to several circumstances. For instance, it can happen that the analysis is not able to infer precise enough information to verify the conditions. The user can then provide a more refined description of initial calling patterns or choose a different, finer-grained, domain. Although, it is not showed in the picture, in both the ii) and iii) cases, the certification process needs to be restarted until achieving a verification condition which meets i). If it succeeds, the answer table constitutes a valid certificate and can be sent to the consumer together with the program.

The *checking* process performed by the consumer is illustrated on the right hand side of Fig. 1. Initially, the supplier sends the program P together with the certificate to the consumer. To retain the safety guarantees, the consumer can provide a new set of assertions which specify the Safety Policy required by this particular consumer. It should be noted that ACC is very flexible in that it allows different implementations of the way the safety policy is provided. Clearly, the same assertions AS used by the producer can be sent to the consumer. But, more interestingly, the consumer can decide to impose a weaker safety condition which can still be proved with the submitted abstraction. Also, the imposed safety condition can be stronger and it may not be proved if it is not implied by the current abstraction (which means that the code would be rejected). From the provided assertions, the consumer must generate again a trustworthy VC and use the incoming certificate to efficiently check that the VC holds. The re-generation of the VC (and its corresponding validation) is identical to the process done in the producer.

Regarding the definition of the Checker, although global analysis is efficient enough to be now used routinely as a practical tool, it is still unacceptable to run the whole Analyzer to validate the certificate since it involves considerable cost. One of the main reasons is that the analysis algorithm is an iterative process which often computes answers (repeatedly) for the same call due to possible updates introduced by further computations. At each iteration, the algorithm has to manipulate rather complex data structures—which involve performing updates, lookups, etc.—until the fixpoint is reached. The whole validation process is centered around the following observation: *the checking algorithm can be de-*

fined as a very simplified "one-pass" analyzer. The computation of the Analyzer can be understood as:

$$[\![P]\!]_\alpha = \mathsf{Analyzer} = \mathrm{lfp}(\mathit{analysis\_step})$$

I.e., a process which repeatedly performs a traversal of the analysis graph (denoted by *analysis_step*) until the computed information does not change, i.e., it reaches a fixpoint. The idea is that the simple, non-iterative *analysis_step* process can play the role of abstract interpretation-based checker (or simply analysis checker). In other words,

$$\mathsf{Checker} \equiv \mathit{analysis\_step}$$

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 analysis results are a valid fixpoint one single execution of *analysis_step* validates the certificate.

Another efficiency issue that the ACC model addresses is which particular *subset* of $[\![P]\!]_\alpha$ is sufficient for verification purposes. It turns out, not surprisingly, that there is a tradeoff between the amount of information sent and the cost of the checking phase. However, we have also shown that only a very small portion of $[\![P]\!]_\alpha$ (the "guesses" in the recursive cliques) is sufficient to ensure that the checker does not need to iterate. In any case, the analysis checker for efficiently validating the certificate can be designed in a simple way which does not require the use of many of the complex data structures which are needed in the implementation of a practical analyzer. More details are presented in [2].

3 Some Examples in `CiaoPP`

The above abstract interpretation-based code certification framework has been implemented in `CiaoPP` [13]: the preprocessor of the `Ciao` program development system [4]. `Ciao` is a multi-paradigm programming system, allowing programming in logic, constraint, and functional styles. At the heart of `Ciao` is an efficient logic programming-based kernel language. This allows the use of the very large body of approximation domains, inference techniques and tools for abstract interpretation-based semantic analysis which have been developed to a powerful and mature level in this area (see, e.g., [19, 6, 11, 14] and their references). These techniques and systems can approximate at compile-time, always safely, and with a significance degree of precision, a wide range of properties which is much richer than, for example, traditional types. This includes data structure shape (including pointer sharing), independence, bounds on data structure sizes, and other operational variable instantiation properties as well as procedure-level properties such as determinacy, termination, non-failure and bounds on resource consumption (time or space cost). The latter tasks are performed in an integrated fashion in `CiaoPP`.

In the context of `CiaoPP`, the abstract interpretation-based certification system is implemented in Ciao 1.11#200 [4] with compilation to bytecode. In essence, we have used the efficient, highly optimized, state-of-the-art analysis system of CiaoPP (which is part of a working compiler) as fixpoint analyzer for generating safety certificates. The checker has been implemented also as a simplification of such generic abstract interpreter. Our aim here is to present not the techniques used by CiaoPP for code certification (which are described in [2]) but its main functionalities by means of some examples.

Example 1 (sharing+freeness). The next program mmultiply multiplies two matrices by using two auxiliary predicates: multiply which performs the multiplication of a matrix and an array and vmul which computes the vectorial product of two arrays (by multiplying all their elements):

```
mmultiply([],_,[]).
mmultiply([V0|Rest], V1, [Result|Others]):-
        mmultiply(Rest, V1, Others),
        multiply(V1,V0,Result).

multiply([],_,[]).
multiply([V0|Rest], V1, [Result|Others]):-
        multiply(Rest, V1, Others),
        vmul(V0,V1,Result).

vmul([],[],0).
vmul([H1|T1], [H2|T2], Result):-
        vmul(T1,T2, Newresult),
        Product is H1*H2,
        Result is Product+Newresult.
```

One of the distinguishing features of logic programming is that arguments to procedures can be uninstantiated variables. This, together with the search execution mechanism available (generally backtracking) makes it possible to have multi-directional procedures. I.e., rather than having fixed input and output arguments, execution can be "reversed". Thus, we may compute the "input" arguments from known "output" arguments. However, predicate is/2 (used as an infix binary operator) is mono-directional. It computes the arithmetic value of its second (right) argument and unifies it with its first (left) argument. The execution of is with an uninstantiation rightmost argument results in a runtime error. Therefore, a safety issue in this example is to ensure that calls to the built-in predicate is are performed with ground data in the right argument.

We can infer this safety information by analyzing the above program in CiaoPP using a mode and independence analysis ("sharing+freeness"). In the "sharing+freeness" domain, var denotes variables that do not point yet to any data structure, mshare denotes pointer sharing patterns between variables and ground variables which point to data structures which contain no pointers. The analysis is performed with the following entry assertion which allows specifying a restricted class of calls to the predicate.

```
:- entry mmultiply(X,Y,Z):( var(Z), ground(X), ground(Y) ).
```

It denotes that calls to mmultiply will be performed with ground terms in the first two arguments and a free variable in the last one.

For the above entry, the output of CiaoPP yields, among others, the following set of assertions which constitute our safety certificate:

```
:- true pred A is B+C
       : ( mshare([[A]]),var(A),ground([B,C]) )
      => ( ground([A,B,C]) ).
:- true pred A is B*C
       : ( mshare([[A]]),var(A),ground([B,C]) )
      => ( ground([A,B,C]) ).
```

The "true pred" assertions above specify in a combined way properties of both: ":" the entry (i.e., upon calling) and "=>" the exit (i.e., upon success) points of all calls to the predicate. These assertions for predicate is express that the leftmost argument is a free unaliased variable while the rightmost arguments are input values (i.e., ground on call) when is is called (:). Upon success, all three arguments will get instantiated. Given this information, we can verify that the safety condition is accomplished and thus the code is safe to run. Thus, the above analysis output can be used as a certificate to attest a safe use of predicate is.

The above experiment has been performed using a sharing+freeness domain. However, the whole method is domain–independent. This allows plugging in different abstract domains, provided suitable interfacing functions are defined. From the user point of view, it is sufficient to specify the particular abstract domain desired. For instance, CiaoPP can also infer (parametric) types for programs both at the predicate level and at the literal level [11, 12, 25]. Clearly, type information is very useful for program certification, verification, optimization, debugging (see, e.g., [13]).

Example 2 (eterms). Our next experiment uses the *regular type* domain *eterms* [25] to analyze the same program of Ex. 1. We use in our examples term as the most general type (i.e., it corresponds to all possible terms), list to represent lists and num for numbers. We also allow parametric types such as list(T) which denotes lists whose elements are all of type T. Type list is clearly equivalent to list(term).

The program is analyzed w.r.t. the following entry assertion which specifies that calls to mmultiply are performed with matrices in the first two arguments:

```
:- entry mmultiply(X,Y,Z): (var(Z),
         list(X,list(num)),list(Y,list(num))).
```

CiaoPP output yields, among other, the following assertions for the built-in predicate is:

```
:- true pred A is B+C
         : ( term(A),num(B),num(C) )
        => ( num(A),num(B),num(C) ).
```

```
:- true pred A is B*C
         : ( term(A),num(B),num(C) )
        => ( num(A),num(B),num(C) ).
```

They indicate that calls to is will be performed with numbers in the rightmost argument (thus, ground terms) and will return, upon success, a number in the first argument. Therefore, they also constitute a valid (and more precise) certificate for the safety issue described in Ex. 1.

4 Computational Properties

Abstract interpretation-based techniques are able to reason about computational properties which can be useful for controlling efficiency issues, an interesting issue for instance in the context of pervasive computing systems. CiaoPP can infer lower and upper bounds on the sizes of terms and the computational cost of predicates [9, 10]. Cost bounds are expressed as functions on the sizes of the input arguments and yield the number of resolution steps. Various measures can be used for the "size" of the input, such as list-length, term-size, term-depth, integer-value, etc. The idea is that the system can disregard code which makes requirement that are too large in terms of computing resources (in time and/or space). Let us see an example.

Example 3. The following program inc_all increments all elements of a list by adding one to each of them.

```
inc_all([],[]).
inc_all([H|T],[NH|NT]) :-
        NH is H+1,
        inc_all(T,NT).
```

The following assertions have been added by the user of the pervasive computing system:

```
:- entry inc_all(A,B) : (list(A,num),var(B)).
:- check calls inc_all(A,B)
         :  list(A,num).
:- check success inc_all(A,B)
         =>  list(B,num).
:- check comp inc_all(A,B)
         : ( list(A,num), var(B) )
         + steps_ub(length(A)+1).
```

The entry assertion specifies that calls to inc_all must be performed with a list of numbers in the first argument while the second one must be a free variable. The next three check assertions express the intended semantics of the program. The third one intends to check that, upon success, the second argument of calls to inc_all will be a list of numbers. Finally, the last computational (comp) assertion tries to verify that the upper bound of the predicate is the sum of the length of the first list and one. The idea is that the code will be accepted provided all assertions can be checked.

The cost analysis available in CiaoPP infers, among others, the following assertions for the above program and entries:

```
:- checked calls inc_all(A,B)
        : list(A,num).
:- checked success inc_all(A,B)
       => list(B,num).
:- checked comp inc_all(A,B)
        : ( list(A,num), var(B) )
        + steps_ub(inc_all(A,B),length(A)+1).
:- true pred inc_all(A,B)
        : ( list(A,num), var(B) )
       => ( list(A,num), list(B,num) )
        + ( not_fails, is_det,
            steps_ub(length(A)+1) ).
```

Therefore, the status of the last three check assertions has become checked, which means that they have been validated and thus the program is safe to run (according to the intended meaning). The last procedure-level assertion merges them all and, additionally, indicates that calls to the predicate do not fail and their execution is deterministic by combining information available for other abstract domains.

PCC techniques—based on certificates which are computed outside the device—constitute a good scenario for the certification of software deployed in systems with limited computed resources, which may lack computing resources to perform static analysis. They compute tamper-proof certificates which simplify code verification and pass them along with the code. In our abstract interpretation-based context, although global analysis is now routinely used as a practical tool, it is still unacceptable to run the whole analyzer to validate the certificate as it involves considerable cost. One of the main reasons is that the fixpoint algorithm is an iterative process which often computes answers (repeatedly) for the same call due to possible updates introduced by further computations. At each iteration, the algorithm has to manipulate rather complex data structures—which involve performing updates, lookups, etc.—until the fixpoint is reached. Luckily, in abstract interpretation-based code certification, the burden on the consumer side is reduced by using a simple one-*traversal* checker, which is a very simplified and efficient abstract interpreter which does not need to compute a fixpoint. The benchmark results in [2] show that the speedup achieved by the checking is approximately 1.63 in just analysis time which, we believe, makes our approach practically applicable in pervasive contexts.

A similar proposal is presented in [24] to split the type-based bytecode verification of the KVM (an embedded variant of the JVM) in two phases, where the producer first computes the certificate by means of a type-based dataflow analyzer and then the consumer simply checks that the types provided in the code certificate are valid. This approach is extended in [15] to real world Java Software. As in our case, the validation can be done in a single, linear pass over the bytecode. However, these approaches are designed limited to types, whereas our approach supports a very rich set of domains especially well-suited for this purpose, including complex properties such as computational and memory cost, non-failure, determinacy, etc. (as we have seen in the examples in this section) and possibly even combining several of them.

5 Conclusions

Abstract interpretation-based verification forms the corner stone of the safety model of `CiaoPP`: the preprocessor of the `Ciao` multi-paradigm programming system. It ensures the integrity of the runtime environment even in the presence of untrusted code. The framework uses modular, incremental, abstract interpretation as a fundamental tool to infer information about programs. This information is used to certify and validate programs, to detect bugs with respect to partial specifications written using program assertions, to generate and simplify run-time tests and to perform high-level optimizations such as multiple abstract specialization, parallelization, and resource usage control. Among these applications, we herein focus on the use of abstract interpretation-based verification for the purpose of mobile code safety by following the standard PCC methodology. We report on some experiments in `CiaoPP` at work which illustrate how the actual process of program certification is aided in an implementation of this framework. We point out that computational properties inferred by `CiaoPP` can be useful for controlling resource usage and filtering out mobile code which does not meet certain cost requirements. Also, the fact that our approach follows PCC techniques—in which the certificate is generated outside the device—makes it potentially applicable in this pervasive context. However, controlling it in a perfect way proves far from obvious, and a range of challenging open problems remain as topics for further research. For instance, we plan to study a more precise model of the memory requirements of small devices. The size of certificates needs to be minimized as much as possible to fit in such limited systems. We believe that they can be further reduced by omitting the information which has to be necessarily re-computed by the checker. This is the subject of ongoing research.

Acknowledgements.

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2001-38059 *ASAP* project and by the Spanish Ministry of Education under the TIC 2002-0055 *CUBICO* project. Part of this work was performed during a research stay of Elvira Albert and Germán Puebla at UNM supported by respective grants from the Secretaría de Estado de Educación y Universidades. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

The Ciao/CiaoPP system is a collaborative effort of members of several institutions, including UPM, UNM, U. Melbourne, Monash U., U.Arizona, Linköping U., NMSU, K.U. Leuven, Roskilde U., Ben-Gurion U, and INRIA. The system documentation and related publications contain more specific credits. The authors would also like to acknowledge the work all the other current Ciao and CiaoPP team members, in particular: Francisco Bueno, Manuel Carro, Daniel Cabeza, Jesús Correas, Pedro López-García, José Morales, Claudio Ochoa, David Trallero, Pawel Pietrzak and Claudio Vaucheret, without whose collaboration this work would not have been possible.

Finally, thanks are also due to the MoveLog program committee members for their generous invitation to present this work and demo the system at the conference.

References

1. E. Albert, G. Puebla, and M. Hermenegildo. An Abstract Interpretation-based Approach to Mobile Code Safety. In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, Electronic Notes in Theoretical Computer Science 132(1), pages 113–129. Elsevier - North Holland, April 2005.
2. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, number 3452 in LNAI, pages 380–397. Springer-Verlag, March 2005.
3. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
4. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual (v1.8). Technical Report CLIP4/2002.1, School of Computer Science, UPM, 2002. Available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
5. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.
6. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
7. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
8. S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.
9. S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Proc. of SAS'94*, number 864 in LNCS, pages 255–265. Springer-Verlag, 1994. Invited Talk.

10. S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *Proc. of ILPS'97*, pages 291–305. MIT Press, Cambridge, MA, 1997.
11. J. Gallagher and D. de Waal. Fast and Precise Regular Approximations of Logic Programs. In *Proc. of ICLP'94*, pages 599–613. MIT Press, 1994.
12. J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Proc. of PADL'02*, LNCS, pages 243–261, 2002.
13. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *Proc. of SAS'03*, pages 127–152. Springer LNCS 2694, 2003.
14. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
15. K. Klohs and U. Kastens. Memory Requirements of Java Bytecode Verification on Limited Devices. In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, 2004.
16. K. Marriott, H. Søndergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.
17. K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.
18. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
19. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.
20. G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.
21. G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.
22. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.
23. G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, 2000.
24. K. Rose, E. Rose. Lightweight bytecode verification. In *OOPSALA Workshop on Formal Underpinnings of Java*, 1998.
25. C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *Proc. of SAS'02*, pages 102–116. Springer LNCS 2477, 2002.
26. M. Weiser. The computer for the twenty-first century. *Scientific American*, 3(265):94–104, September 1991.