Abstraction Carrying Code and Resource-Awareness

Manuel V. Hermenegildo[1,3] Elvira Albert[2] Pedro López-García[1] Germán Puebla[1]

[1] School of Computer Science
T. U. of Madrid (UPM)
Madrid, Spain
{herme,german,pedro.lopez}@fi.upm.es

[2] DSIP
Complutense U. of Madrid
Madrid, Spain
elvira@sip.ucm.es

[3] Depts. of CS and ECE
U. of New Mexico
Albuquerque, NM, USA
herme@unm.edu

ABSTRACT

Proof-Carrying Code (PCC) is a general approach to mobile code safety in which the code supplier augments the program with a certificate (or proof). The intended benefit is that the program consumer can locally validate the certificate w.r.t. the "untrusted" program by means of a certificate checker—a process which should be much simpler, efficient, and automatic than generating the original proof. *Abstraction Carrying Code* (ACC) is an enabling technology for PCC in which an *abstract model* of the program plays the role of certificate. The generation of the certificate, i.e., the abstraction, is automatically carried out by an abstract interpretation-based analysis engine, which is parametric w.r.t. different abstract domains. While the analyzer on the producer side typically has to compute a semantic fixpoint in a complex, iterative process, on the receiver it is only necessary to check that the certificate is indeed a fixpoint of the abstract semantics equations representing the program. This is done in a single pass in a much more efficient process. ACC addresses the fundamental issues in PCC and opens the door to the applicability of the large body of frameworks and domains based on abstract interpretation as enabling technology for PCC. We present an overview of ACC and we describe in a tutorial fashion an application to the problem of resource-aware security in mobile code. Essentially the information computed by a cost analyzer is used to generate *cost certificates* which attest a safe and efficient use of a mobile code. A receiving side can then reject code which brings cost certificates (which it cannot validate or) which have too large cost requirements in terms of computing resources (in time and/or space) and accept mobile code which meets the established requirements.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Diagnostics, Symbolic execution*; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages, Multiparadigm languages*.

General Terms

Reliability, Security, Languages, Theory, Verification.

Keywords

Program verification, mobile code certification, resource awareness, program debugging, cost analysis, granularity control, distributed programming, abstract interpretation, programming languages.

1. INTRODUCTION

In traditional distributed execution receivers are assumed to be either dedicated and/or to trust and simply accept (or take, in the case of work-stealing schedulers) available tasks. Typically, tasks are run at the receiving end under some administrative domain that is previously agreed on by producer and consumer of the task. However, many recently proposed applications (such as peer-to-peer systems, the GRID, and other similar overlay computing systems) represent more open settings where the administrative domain of the receiver can be completely different from that of the producer. Also, in these applications the receiver is typically being used for other purposes (e.g., as a general-purpose workstation) in addition to being a party to the distributed computation.

In such an environment, interesting security- and resource-related issues arise. In particular, in order to accept some code and a particular task to be performed, the receiver must clearly have some assurance of the *correctness and characteristics of the code received* and also of *the kind of load the particular task is going to pose.* A receiver should be free to reject code that does not adhere to a particular *safety policy* involving both more traditional safety issues (such as, e.g., that it will not write on specific areas of the disk) and *resource*-related issues (such as, e.g., that it will not compute for more than a given amount of time, or that it will not take

up an amount of memory or other resources above a certain threshold). Regarding the latter, although it is obviously possible to interrupt a task after a certain time or if it starts taking too much memory, this will be wasteful of resources and require recovery measures. It is clearly more desirable to be able to detect these situations a priori. In other words, the need appears to develop security techniques for verifying that the execution of a program (possibly) supplied by an untrusted source meets certain properties according to a predefined *safety policy*, which will typically include *resource*-related requirements.

1.1 Proof Carrying Code

Proof-Carrying Code (PCC) [22] is an enabling technology for mobile code safety which proposes to associate safety information in the form of a *certificate* to programs. The certificate (or proof) is created at compile time, and packaged along with the untrusted code. The consumer who receives or downloads 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 "certificate-based" approach to mobile code safety is that the consumer's task is reduced from the level of proving to the level of checking. Indeed the (proof) checker performs a task that should be much simpler, efficient, and automatic than generating the original certificate. This is important since the implementation on the receiving end is part of the safety-critical infrastructure and it should be minimized. Also, the receiving host could be a small, embedded system that lacks the computing resources required to run large and complex programs. Finally, such checking will be performed by every consumer, while the certification generation is done only once by the supplier.

Thus, the practical uptake of PCC greatly depends on the existence of a variety of enabling technologies which allow:

1. defining *expressive safety policies* covering a wide range of properties,
2. solving the problem of how to *automatically generate the certificates* (i.e., automatically proving the programs correct), and
3. replacing a costly verification process by an efficient checking procedure on the consumer side.

The main approaches applied up to now are based on theorem proving and type analysis. For instance, in PCC 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. λProlog is used in [3] to define a representation of lemmas and definitions which helps keep the proofs small. A recent proposal [4] uses temporal logic to specify the security policies. In Typed Assembly Languages [19], the certificate is a type annotation of the assembly language program and the checking process involves a form of type checking.

1.2 Abstraction Carrying Code

In [1, 2] we have proposed *Abstraction-Carrying Code* (ACC). This novel approach follows the certificate-based scheme but uses *abstract interpretation* [9] (rather than proofs or type analysis) as enabling technology to handle the three previously listed practical (and difficult) challenges of PCC. Abstract interpretation 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 ("running") them using abstract values rather than concrete ones, and obtaining safe over- or under-approximations of the behavior of the program. This technique allows inferring much richer information than, for example, traditional types [8]. And, while it is typically more limited than theorem proving, it is amenable to full automation and as a result it has been incorporated in practical compilers. In some cases such compilers use abstract interpretation to perform a large number of tasks including verification, debugging, partial evaluation, and low-level optimization (see, e.g., [16]). Examples of properties that are routinely inferred with a comparatively high degree of precision using abstract interpretation include data structure shape (with pointer sharing), 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).

Our proposal, ACC, opens the door to the applicability of the frameworks and domains used for inferring these properties as enabling technology for PCC. The fundamental idea of ACC is to use an *abstraction* (or abstract model) of the program computed by standard static analyzers as a certificate. The safety policy is specified by using an expressive assertion language defined over abstract domains. The certificate implies the compliance with that safety policy. The validity of the abstraction on the consumer side is checked in a single-pass by a very efficient and specialized abstract-interpreter. While the analyzer on the producer side typically has to compute a semantic fixpoint in a complex, iterative process, on the receiving side it is only necessary to check that the certificate received is indeed a fixpoint of the abstract semantic equations representing the program, which is a much more efficient process, and to check again the implication. Also, because the checker is a simpler program, the code base that must be trusted is reduced. The approach thus addresses the fundamental PCC issues of defining expressive safety policies (through the assertion language and the abstract domain), automatically generating the certificates (through abstract interpretation, by computing the fixpoint), and replacing a costly verification process by an efficient checking procedure on the consumer side (through the checking of the fixpoint).

1.3 ACC for Resource-related Properties

Because of its parametric nature, ACC is a very flexible framework. Our objective in this paper is to describe in a tutorial fashion an application of ACC to the problem of resource-aware security in mobile code. This will be done by enhancing such mobile code with safety certificates which guarantee that the execution of the (in principle untrusted) code received from another node in the network is *safe* but also, as mentioned above, *efficient*, according to a predefined safety policy *which includes properties related to resource consumption*. Information computed by a cost analysis will be used to generate *cost certificates* which are packaged along with the untrusted code. The receiving side can then reject code which brings cost certificates (which it

cannot validate or) which have too large cost requirements in terms of computing resources (in time and/or space) and accept mobile code which meets the established requirements.

1.4 Demonstration Platform: Ciao/CiaoPP

It is important to note that ACC is a general approach that applies directly to all programming paradigms, including logic, constraint, functional, and imperative programming, as long as a static analyzer/checker is available. Note that the fundamental components of the approach (fixpoint semantics and abstract interpretation) have both been widely applied also in all these paradigms. However, for concreteness we will work with an incarnation of it in the context of (Constraint) Logic Programming, (C)LP, because this paradigm offers a good number of advantages, an important one being the maturity and sophistication of the analysis tools available for it. The advanced state of program analysis technology and the expressiveness of existing abstract analysis domains used in the analysis of these paradigms has become very useful for defining, manipulating, and inferring a wide range of properties.

Also for concreteness, we build on the algorithms of (and report on an implementation on) `CiaoPP` [16], the abstract interpretation-based preprocessor of the `Ciao` system. `Ciao` is a modern multi-paradigm programming language and environment which in fact allows coding programs combining the styles of logic, constraint, functional, and a particular version of object-oriented programming. One of the advantages of this system in our context is that we have available a number of practical analysis tools within it. Indeed, `CiaoPP` uses modular, incremental abstract interpretation as a fundamental tool to obtain information about programs, including, as mentioned before, independence, determinacy, non-failure, termination, bounds on data structure sizes, computational cost, etc. In `CiaoPP`, the semantic approximations thus produced are applied to performing high- and low-level optimizations during program compilation, including transformations such as multiple abstract specialization, parallelization, and resource usage control, all in a provably correct way. In addition, and most importantly for our purposes, such semantic approximations are applied in the general context of program development to perform program static debugging, verification and, as we discuss in this paper, to perform ACC.

1.5 Outline

The rest of the paper proceeds as follows. In Section 2, we recall the abstract interpretation-based approach to program verification. In Section 3 we present an overview of the Abstraction Carrying Code framework. Section 4 recalls the basic techniques used for inferring resource-related properties in our approach, including upper and lower bounds on computational cost and data sizes. In Section 5 we then illustrate through an example the application of our framework which uses safety certificates with resource consumption assurances. Finally, Section 6 presents some conclusions.

2. ABSTRACT INTERPRETATION-BASED VERIFICATION

In this section, we briefly describe the abstract interpretation-based approach to program verification [6, 25] which constitutes the basis for the certification process carried out in ACC. This is also the method implemented in `CiaoPP`.

2.1 Program Verification

We consider the important class of semantics referred to as *fixpoint semantics*. In this setting, a (monotonic) semantic operator (which we refer to as S_P) is associated with 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 fixpoint of the S_P operator, i.e., $[\![P]\!] = \text{lfp}(S_P)$. A well-known result is that if S_P is continuous, the least fixpoint 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 proving 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 only partially known, infinite, too expensive to compute, 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.

2.2 Abstract Interpretation

In abstract interpretation [10] the program P is interpreted over a non-standard domain called the *abstract* domain (D_α) which is simpler than the selected *concrete domain* (D). The abstract domain D_α is usually constructed with the objective of computing precise, yet always safe approximations of the semantics of programs, and the semantics w.r.t. this abstract domain, i.e., the *abstract semantics* of the program, is computed (or approximated) by replacing the operators in the program by their abstract counterparts. An abstract value is a finite representation of a, possibly infinite, set of actual values (or states or even complete executions) in the concrete domain. The set of all possible abstract semantic values which D_α represents is usually also a complete lattice or cpo which is ascending chain finite. As in the previous section we assume for tutorial purposes complete lattices over *sets*, both for the concrete domain $\langle 2^D, \subseteq \rangle$ (we consider *sets* of values or states) and the abstract domain $\langle D_\alpha, \sqsubseteq \rangle$. 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 $\forall x \in 2^D : \gamma(\alpha(x)) \supseteq x$ and $\forall y \in D_\alpha : \alpha(\gamma(y)) = y$. 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.

One of the fundamental results of abstract interpretation is that an abstract semantic operator S_P^α for a program P can be defined which is correct w.r.t. S_P in the sense that

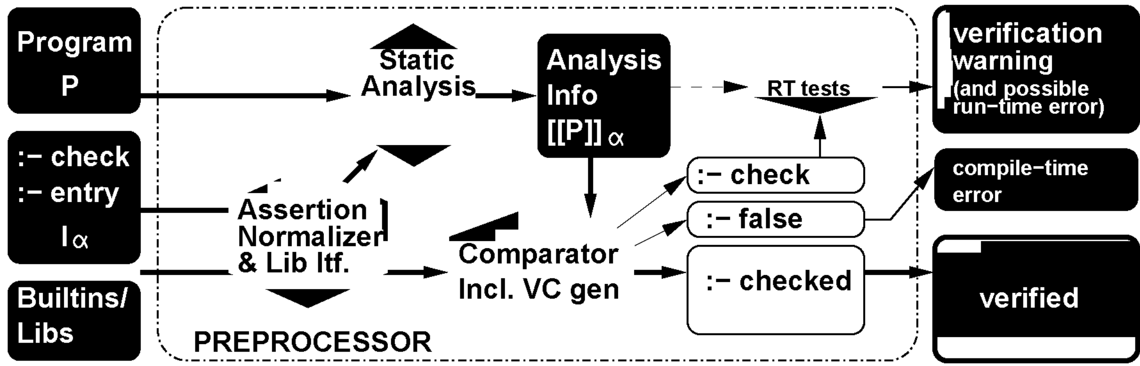

Figure 1: Program Verification Framework (CiaoPP)

$\gamma(\text{lfp}(S_P^\alpha))$ is an approximation of $[\![P]\!]$, and, if certain conditions hold (e.g., ascending chains are finite in the D_α lattice), then the computation of $\text{lfp}(S_P^\alpha)$ terminates in a finite number of steps. We will denote $\text{lfp}(S_P^\alpha)$, i.e., the result of abstract interpretation for a program P, as $[\![P]\!]_\alpha$.

Typically, abstract interpretation guarantees that $[\![P]\!]_\alpha$ is an *over*-approximation of the abstract semantics of the program itself, $\alpha([\![P]\!])$. Thus, we have that $[\![P]\!]_\alpha \supseteq \alpha([\![P]\!])$, which we will denote as $[\![P]\!]_{\alpha+}$. Alternatively, the analysis can be designed to safely *under*-approximate the actual semantics, and then we have that $[\![P]\!]_\alpha \subseteq \alpha([\![P]\!])$, which we denote as $[\![P]\!]_{\alpha-}$.

2.3 AI-based Program Verification

The key idea of our verification (and static debugging) framework, as implemented in CiaoPP, is to use the abstract approximation $[\![P]\!]_\alpha$ computed by the analysis engines directly in verification and debugging 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) intended semantics.

Herein, 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:

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

However, using abstract interpretation, we can usually only compute $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$. Thus, we are interested in studying the implications of comparing $\mathcal{I}_\alpha$ and $[\![P]\!]_\alpha$. Analyses which over-approximate the actual semantics (i.e., those denoted above as $[\![P]\!]_{\alpha+}$), 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:

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

Program verification and detection of errors are performed in CiaoPP at compile-time by using the above sufficient condition, i.e., the abstract approximation $[\![P]\!]_\alpha$ of the actual semantics of the program $[\![P]\!]$ is computed via abstract interpretation-based static analysis and then this information is compared against (also approximate) partial specifications $\mathcal{I}_\alpha$, written in terms of ("check") *assertions* [24]. This framework is partially depicted in Figure 1, see also [6, 25, 15, 23] for a fuller description. The assertions used are linguistic constructions which allow expressing properties of programs, as we will see later. Each part of $\mathcal{I}_\alpha$ (each assertion, or even certain parts of an assertion) can be the subject of comparison against $[\![P]\!]_\alpha$. The result of the comparison can be that the (part of the) assertion is met and then it is validated ("checked"). If all assertions are checked the program is verified. Alternatively, it may possible to show during the comparison that the condition above is false, in which case an error in the program has been detected (a "false" assertion). Finally, due to the approximating nature of abstract interpretation it may also be the case that an assertion cannot be proved or disproved, in which case a more precise or specific analysis domain must be used or the analysis guided via "trust" assertions. In any case the static checking is provably *safe* in the sense that all errors flagged are definite violations of the specifications and if a part of the specification is validated then it is indeed valid for all possible executions.

Note that this approach is very attractive in programming systems, like CiaoPP, where the compiler already performs such program analysis in order to use the resulting information to, e.g., optimize the generated code. I.e., in these cases the compiler will compute $[\![P]\!]_\alpha$ anyway.

As it appears in Figure 1, in principle the analyzer is domain–independent, which 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 during the generation of the safety assertions. Different domains give analyzers which provide different types of information and degrees of accuracy. The core of each generic abstract interpretation-based engine is an algorithm for efficient fixed-point computation [20, 21, 7, 17, 26].

3. THE ACC FRAMEWORK

The idea of Abstraction Carrying Code (ACC) [1, 2] is a

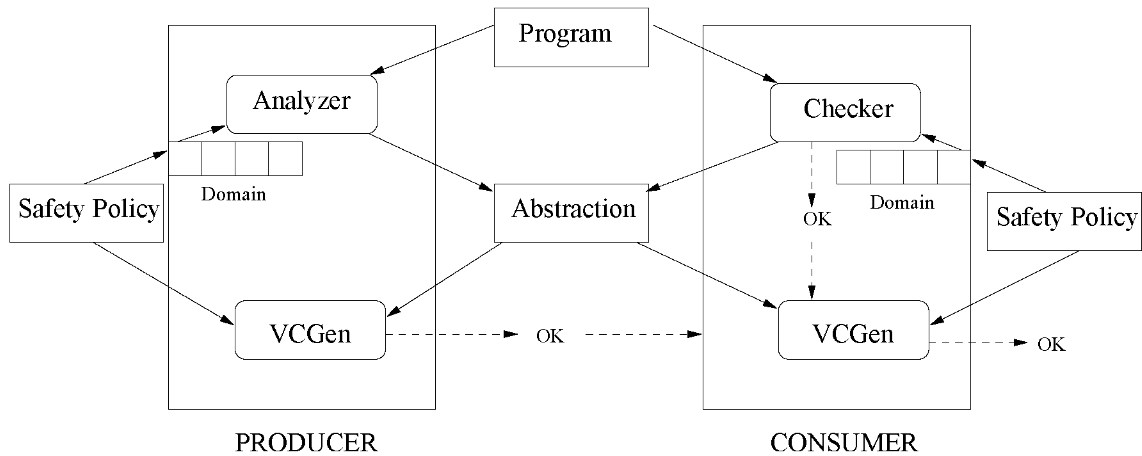

Figure 2: The Abstraction-Carrying Code Scheme

natural extension of our approach to abstract interpretation-based program verification. Figure 2 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 mentioned in Sect. 1.

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 [13, 5, 21, 18, 7, 14]. In particular, our implementation is based on PLAI [21, 20], 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 *checking* process performed by the consumer is illustrated on the right hand side of Fig. 2. 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 defined as a very simplified "one-pass" analyzer.* The computation of the Analyzer can be understood as:

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

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,

$$\text{Checker} \equiv \textit{analysis_step}$$

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].

4. INFERRING RESOURCE BOUNDS

As mentioned before, abstract interpretation-based program analysis techniques allow inferring much richer information than, for example, traditional types. This information will allow specifying *safety policies* involving not only traditional safety issues (e.g., that the code will not write on specific areas of the disk) but also *resource*-related issues (e.g., that it will not compute for more than a given amount of time, or that it will not take up an amount of memory or other resources above a certain threshold) and, thus, achieving further expressiveness.

In our approach, such cost bounds (upper or lower) are expressed as functions on the sizes of the input arguments and yield bounds on the number of execution steps required by the computation. Various metrics are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. Types, modes, and size measures are first automatically inferred by the analyzers and then used in the size and cost analysis.

We illustrate through a simple example the fundamental intuition behind our lower bound cost estimation technique. Consider the naive reverse program in Figure 3. The `entry` assertion states information on the *entry points* to the program module. It states that outside calls to `nrev` must be performed with a totally instantiated list (i.e., a ground list of terms) in the first argument and a free variable in the second one (the output), i.e., it will indeed be used as a function. Assume also that the cost unit is the number of procedure calls. With these assumptions the exact cost function of procedure `append` is $\texttt{Cost}_{append}(x, y) = x + 1$, where x and y are the sizes (lengths) of the first and second input lists respectively. Note that this cost function does not depend on the size of the second argument of `append` really. Also, based on this cost function, the exact cost function of procedure `nrev` is $\texttt{Cost}_{nrev}(n) = 0.5\ n^2 + 1.5\ n + 1$, where n is the size (length) of the input list.

In order to obtain a lower-bound approximation of the previous cost functions, `CiaoPP` first performs the following analyses (all using abstract interpretation techniques):

- A mode (and sharing) analysis. This determines which arguments (or parts of them) are inputs and which are outputs for each constructor operation, procedure and procedure call, as well as the dependencies between any variables (pointers) in the data structures passed via these arguments.

```
:- module(reverse, [nrev/2],
   [assertions,functions,regtypes,nativeprops]).
:- entry nrev/2 : {list, ground} * var.

nrev( [] )         := [].
nrev( [H|L] )      := ~append( nrev(L), [H] ).

append([],X)       := X.
append([H|X],Y)    := [ H | append(X,Y) ].
```

Figure 3: The naive reverse program.

- A type analysis. This infers the types for all program variables. Note that type declarations are not compulsory in the language, so the relevant type definitions may also have to be inferred.

- A determinacy analysis. It requires the results of type and mode analysis, and which detects which procedures and procedure calls are deterministic.

- A non-failure analysis. This also requires the results of type and mode analysis, and can detect procedures and goals that can be guaranteed not to fail, i.e., to produce at least one solution or not terminate. The need for a non-failure analysis, stems from an interesting problem with estimating lower bounds: in general it is necessary to account for the possibility of failure of a call to the procedure (because of, e.g., an inadmissible argument) leading to a trivial lower bound of 0.

- Inference of size metrics for relevant arguments. It is based on the type information.

The results of these analyses ($[\![P]\!]_\alpha$ for these domains), as produced by `CiaoPP` in the form of assertions are shown in Figure 4. It is beyond the scope of this paper to fully explain all the (generally abstract interpretation-based) techniques involved in inferring this information (see, e.g., [16, 11, 12] and their references). However, we will sketch in the following how, once all this information is obtained, the work done by (recursive) clauses is determined.

To this end, it is first necessary to be able to estimate the size of input arguments in the procedure calls in the body of the procedure, relative to the sizes of the input arguments to the procedure, using the inferred metrics. The size of an output argument in a procedure call depends, in general, on the size of the input arguments in that call. For this reason, for each output argument we use an expression which yields its size as a function of the input data sizes. For this, we use an abstraction of procedure definitions called a data dependency graph, built using all the abstract information inferred previously. The following steps are then performed:

- The data dependency graphs are used to determine the relative sizes of variable bindings at different program points.

```
:- true pred nrev(A,B)      :  ( list(A), var(B) )
                            => ( list(A), list(B) )
                            + ( not_fails, covered, is_det, mut_exclusive ).

:- true pred nrev(A,B)      :  ( ground(A), var(B), mshare([[B]]) )
                            => ( ground(A), ground(B) ).

:- true pred append(A,B,C) :  ( list(A,term), t84(B), var(C) )
                            => ( list(A,term), t84(B), list1(C,term)
                            +  ( not_fails, covered, is_det, mut_exclusive ).

:- true pred append(A,B,C) :  ( ground(A), ground(B), var(C), mshare([[C]]) )
                            => ( ground(A), ground(B), ground(C) ).

:- regtype t84/1. t84 := [_].
```

Figure 4: CiaoPP compiler output (types, modes, determinacy, non-failure).

- The size information is used to set up difference equations representing the computational cost of procedures.
- Abstractions (lower and upper bounds) of the solutions of these difference equations are then obtained which provide the lower/upper-bound procedure cost and data size functions.

Let us see in more detail the steps performed in order to infer the cost function for the nrev example above. During size and cost analysis, the call graph of the program is traversed in reverse topological order. Thus, we first, consider the procedure append. Let $\mathtt{Size}^3_{\mathtt{append}}(x,y)$ denote the size of the output argument (the third) as a function of the size of the two first (input) arguments x and y. With the previous analysis information we determine the directionality of all the data manipulation operations and that the size measure to use is list length. We also determine the size relationship which says that the size of the first input list to the recursive call is the size of the first input list of the procedure head minus one. Also, the size of the second input list to the recursive call equals the size of the second input list of the procedure head. With this, the following difference equation can be set up for append:

$\mathtt{Size}^3_{\mathtt{append}}(0,y) = y$ (boundary condition from base case),
$\mathtt{Size}^3_{\mathtt{append}}(x,y) = 1 + \mathtt{Size}^3_{\mathtt{append}}(x-1,y)$.

The solution obtained for this difference equation is:

$$\mathtt{Size}^3_{\mathtt{append}}(x,y) = x + y$$

Let $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{p}}(n)$ denote a lower bound on the cost (number of resolution steps) of a call to procedure p with an input of size n. Given all the assumptions above, and the size relations obtained, the following difference equation can be set up for the cost of append:

$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{append}}(0,y) = 1$ (boundary condition from base case),
$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{append}}(x,y) = 1 + \mathtt{Cost}^{\mathtt{L}}_{\mathtt{append}}(x-1,y)$.

A solution for this difference equation is (as expected):

$$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{append}}(x,y) = x + 1$$

Now, continuing the traversal of the call graph of the program in reverse topological order, we consider the procedure nrev. As before, let $\mathtt{Size}^2_{\mathtt{nrev}}(n)$ denote the size of the output argument (the second) as a function of the size of its first (input) argument n. Once we have again determined that the size measure to use is list length, and the size relationship which says that the size of the input list to the recursive call is the size of the input list of the procedure head minus one, the following difference equation can be set up for nrev/2:

$\mathtt{Size}^2_{\mathtt{nrev}}(0) = 0$ (boundary condition from base case),
$\mathtt{Size}^2_{\mathtt{nrev}}(n) = \mathtt{Size}^3_{\mathtt{append}}(\mathtt{Size}^2_{\mathtt{nrev}}(n-1), 1)$.

which, using the previously inferred size function for append in a normalization algorithm, can be rewritten as:

$\mathtt{Size}^2_{\mathtt{nrev}}(0) = 0$ (boundary condition from base case),
$\mathtt{Size}^2_{\mathtt{nrev}}(n) = \mathtt{Size}^2_{\mathtt{nrev}}(n-1) + 1$.

A solution for this difference equation is:

$$\mathtt{Size}^2_{\mathtt{nrev}}(n) = n$$

Now, given the size relations obtained, the following difference equation can be set up for the cost of nrev/2:

$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(0) = 1$ (boundary condition from base case),
$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(n) = 1 + \mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(n-1) +$
$\qquad \mathtt{Cost}^{\mathtt{L}}_{\mathtt{append}}(\mathtt{Size}^2_{\mathtt{nrev}}(n-1), 1)$.

which, using the inferred size function for nrev can be rewritten as:

$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(0) = 1$ (boundary condition from base case),
$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(n) = 1 + \mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(n-1) + \mathtt{Cost}^{\mathtt{L}}_{\mathtt{append}}(n-1, 1)$.

and, finally, using the inferred cost function for append, the difference equation can be rewritten as:

$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(0) = 1$ (boundary condition from base case),
$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(n) = 1 + n + \mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(n-1)$.

```
:- true pred nrev(A,B)      :  ( list(A), var(B) )
                            => ( list(A), list(B),
                                 size_lb(A,length(A)), size_lb(B,length(A)),
                                 size_ub(A,length(A)), size_ub(B,length(A))
                            +  ( not_fails, covered, is_det, mut_exclusive,
                                 steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1),
                                 steps_ub(0.5*exp(length(A),2)+1.5*length(A)+1)).

:- true pred nrev(A,B)      :  ( ground(A), var(B), mshare([[B]]) )
                            => ( ground(A), ground(B) ).

:- true pred append(A,B,C) :  ( list(A,term), t84(B), var(C) )
                            => ( list(A,term), t84(B), list1(C,term),
                                 size_lb(A,length(A)), size_lb(B,length(B)),
                                 size_lb(C,length(B)+length(A)),
                                 size_ub(A,length(A)), size_ub(B,length(B)),
                                 size_ub(C,length(B)+length(A)) ).
                            +  ( not_fails, covered, is_det, mut_exclusive,
                                 steps_lb(length(A)+1),
                                 steps_ub(length(A)+1) ).

:- true pred append(A,B,C) :  ( ground(A), ground(B), var(C), mshare([[C]]) )
                            => ( ground(A), ground(B), ground(C) ).

:- regtype t84/1. t84 := [_].
```

Figure 5: `CiaoPP` compiler output (including sizes and cost).

A solution for this difference equation is (as expected):

$$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{nrev}}(n) = 0.5\ n^2 + 1.5\ n + 1$$

In our approach, sometimes the solutions of the difference equations need to be in fact approximated by a lower bound (i.e., an abstraction which is a safe approximation) when the exact solution cannot be found. The upper bound cost estimation case is very similar to the lower bound one, although simpler, since we do not have to account for the possibility of failure.

For illustration purposes, the concrete output from `CiaoPP` obtained after performing this process for the `nrev` program is presented in Figure 5. This output includes the *assertion* (simplified for brevity):

```
:- true pred nrev(A,B)
   : ( list(A), var(B) )
  => ( list(A), list(B),
       size_lb(B, length(A))
     )
   + ( not_fails, is_det,
steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1)).
```

Such a "`pred`" assertion specifies 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 procedure, as well as some global properties of its execution. The assertion above, with a "`true`" prefix, expresses that the compiler has proved that procedure `nrev` will produce as output a list of numbers `B`, whose length is at least (`size_lb`) equal to the length of the input list, that the procedure will never fail (i.e., an output value will be computed for any possible input), that it is deterministic (only one solution will be produced as output for any input), and that a lower bound on its computational cost (`steps_lb`) is $0.5\ length(A)^2 + 1.5\ length(A) + 1$ execution steps (where the cost measure used in the example is again the number of procedure calls, but it can be any other arbitrary measure).

This simple example illustrates type inference, non-failure and determinism analyses, as well as lower-bound argument size and computational cost inference. As can be observed in Figure 5, the same cost and size results are actually obtained from the upper bounds analyses (indicating that in this case the results are exact, rather than approximations). Note that obtaining a non-infinite upper bound on cost also implies proving *termination* of the procedure.

5. ACC AND RESOURCES

In this section, we illustrate in a tutorial fashion through an example the concepts of abstract verification and ACC, and, in particular, their application to the problem of resource-aware security in mobile code. Resource-aware ACC becomes interesting for example when developing software to be deployed by devices with a bounded amount of computing resources, such as in pervasive computing [27].

The fundamental idea is that the information obtained in $[\![P]\!]_\alpha$ will allow us to verify *safety policies* which may involve *resource*-related issues, such as that code will not compute for more than a given amount of time, or that it will not take up an amount of memory or other resources above a certain threshold. To this end, a very interesting feature of `CiaoPP` is the possibility of stating assertions, including assertions about the efficiency of the program, which the system will try to verify or falsify using the resource-related information in $[\![P]\!]_\alpha$. We show that, thanks to this functionality, `CiaoPP` can certify programs with resource consumption assurances and also efficiently check such certificates.

```
:- calls   nrev(A,B)  : list(A).                                    % A1
:- success nrev(A,B)  : list(A) => num(B).                          % A2
:- comp    nrev(_,_)  + ( not_fails, is_det, terminates ).          % A3
:- comp    nrev(_,_)  + seff(free).                                 % A4
:- comp    nrev(A,_)  + steps_ub( o(exp(length(A),2)) ).            % A5
```

Figure 6: Some assertions for the `nrev/2` program.

```
:- checked calls   nrev(A,B)  : list(A).                                    % A1
:- false   success nrev(A,B)  : list(A) => num(B).                          % A2
:- checked comp    nrev(_,_)  + ( not_fails, is_det, terminates ).          % A3
:- checked comp    nrev(_,_)  + seff(free).                                 % A4
:- checked comp    nrev(A,_)  + steps_ub( o(exp(length(A),2)) ).            % A5
```

Figure 7: `CiaoPP` compiler output (assertion checking).

Regarding the general problem of verification and static debugging, the technique used in `CiaoPP` is in fact capable of detecting many errors without even adding assertions to programs, because of the existence of assertions (specifications) in the system libraries. Buggy programs often violate such library assertions and this will be flagged. Nevertheless, "`check`" assertions can be added to a program in order to state its partial specification $\mathcal{I}_\alpha$. For our example, let us assume that the assertions shown in Figure 6 (the `check` prefix, meaning that such assertions are part of the specification and must be checked, is assumed when no prefix is given, as in the example) are given as specification for the `nrev` program. The properties used in these assertions, such as `ground`, `not_fails`, `terminates`, costs and types, are imported in the example from system libraries.

These `check` assertions can be seen as integrity constraints: if their properties do not hold at the corresponding program points (procedure call, procedure exit, etc.), the program is incorrect. `Calls` assertions specify properties of all calls to a predicate, while `success` assertions specify properties of exit points for all calls to a predicate. Properties of successes can be restricted to apply only to calls satisfying certain properties upon entry by adding a "`:`" field to `success` assertions. Finally, `Comp` assertions specify *global* properties of the execution of a predicate. These include complex properties such as determinacy or termination and are in general not amenable to run-time checking. They can also be restricted to a subset of the calls using "`:`".

Concretely, the first three assertions in Figure 6 state that (`A1`) `nrev` should always be called with a list (including all recursive calls), that (`A2`) the output is a number (obviously wrong), that (`A3`) the procedure will never fail, that it should be deterministic, and that it will terminate.

In addition, and directly related to our resource-awareness objective, assume that we know that the consumer will only accept tasks of polynomial (actually, at most quadratic) complexity, and only those which are purely computational, i.e., tasks that have no side effects. This safety policy can be expressed at the producer side for this particular program using the assertions `A5` and `A4`, respectively, of Figure 6. More concretely, `A4` states that it should be verified that the computation is pure in the sense that it does not produce any side effects (such as opening a file, etc.). `A5` states that it should be verified that if the predicate is called with a list in the first argument and a free variable in the second one, then there is an upper bound for the cost of this predicate in $O(n^2)$, i.e., quadratic in n, where n is the length of the first list (represented as `length(A)`).

We are assuming that the code will be accepted at the receiving end, provided all assertions can be checked, i.e., that the intended semantics expressed in the assertions determines the safety condition. Such a policy can be agreed a priori or exchanged dynamically. Stating the policy in this form will allow us to ensure during program development that we produce a program that adheres to our specifications and also to the known safety policy of the consumer.

Indeed, during compilation of the `nrev` program, `CiaoPP` will check the assertions above (representing $\mathcal{I}_\alpha$) by comparing them with the assertions inferred by the types, modes, non-failure, determinism, and upper- and lower-bound cost analysis (representing $[\![P]\!]_\alpha$) and given in Figure 5. The result of compile-time checking the intended semantics (assertions in Fig. 6) against this output appears in Fig. 7 (refer also to the output of the comparator in Figure 1). Note that a number of initial assertions have been marked as `checked`, i.e., they have been *validated*. If all assertions had been moved to this `checked` status, then the program would have been *verified*.

However, assertion `A2` has been detected to be false (`A2` was obviously put in just to illustrate this). This indicates a violation of the specification given (in this case it is the specification that is in error), which is also flagged by `CiaoPP` during compilation as a compile-time error. It may also happen that a given assertion cannot be proved or disproved. The assertion will then remain in `check` status, and this will result in a verification warning ("alarm").

Let us now assume that the erroneous assertion `A2` is taken out. Then, after compilation all assertions are moved to the `checked` status, and the program is *verified*. This means that all calls to `nrev` performed within this program satisfy the resource-aware safety policy, i.e., the safety condition is met and the code is indeed safe to run, for now on the producer side.

Following the ACC scheme, (a subset of) the assertions in Fig. 5 (i.e., the analysis results) is used as the abstract *cost and safety certificate* to be used to check for a safe and efficient use of procedure `nrev` on the receiving side.[1] On the consumer side, a receiver using our method will use this

[1]As mentioned before the exact details of how this subset is selected are provided in [2].

abstract certificate in order to accept/reject code depending on whether it adheres or not to some specification.

First of all, the code receiver proceeds to validate the certificate. This implies running the checker over the program assuming the information in Fig. 5 in the relevant points and checking that it is indeed a fixpoint (and later a solution to the recurrence equations, for the case of cost analysis). This process clearly involves less effort that creating the certificate, since only a single pass over the program is required (and checking that an expression is a solution is typically cheaper than obtaining such solution).

If the certificate is not valid, the code is clearly discarded. If the certificate is valid, it is compared against the (local) specifications. The code will be accepted only if all assertions involved can be turned to "checked".

In our example, if we assume that the specification at the receiving end contains, e.g., (possibly a subset of) the assertions from Fig. 6 (except A2, of course), then the code would be accepted. Clearly, in order to guarantee that the cost assertion holds, the certificate has to contain upper bounds on computational cost.

In contrast, let us assume that a consumer with very limited computing resources is assigned to perform a computation using this code. Then, the following "check" assertion (instead of A5) could perhaps represent one of the resource-related requirements at this particular node:

```
:- check comp nrev(A,_)
         : list * var
         + steps_ub( o(length(A)) ).  % A5R
```

i.e., this consumer node will not accept an implementation of nrev with larger complexity than linear.

In this case, given that the certificate contains the (valid) information that nrev will take at least $0.5\ (length(A))^2 + 1.5\ length(A) + 1$ resolution steps, this will be found incompatible with the assertion A5R, which requires the cost to be in $O(length(A))$ resolution steps.

In our implementation in the Ciao system, these tasks are performed at the receiving end by a simplified version of the analysis framework of CiaoPP, that plays the role of efficient checker of certificates. In the case of the A5R the CiaoPP checker produces the following "complexity error:"

```
ERROR: false comp assertion:
  :- comp nrev(A,B) : true => steps_ub(o(length(A)))
  because in the computation the following holds:
  steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1)
```

thus flagging that the program does not satisfy the efficiency requirements imposed. This means of course that the consumer will reject the code.

Note that if we had replaced A5 with A5R during the compilation process at the producer end, this same error would have appeared during compilation, i.e., the compilation process would have flagged the "complexity error" at compile time (and reported this assertion as false in the output).

6. CONCLUSIONS

We have presented in a tutorial way *abstraction-carrying code* (ACC) as a novel enabling technology for PCC, which follows the standard strategy of associating safety certificates to programs but it is based throughout on the use of such abstract interpretation techniques. We argue that ACC is highly flexible due to the parametricity on the abstract domain inherited from the analysis engines used in (C)LP. Our approach differs from existing approaches to PCC in several aspects. In our case, the certificate is computed automatically on the producer side by an *abstract interpretation-based analyzer* and the certificate takes the form of a particular *subset* of the analysis results. 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. We have illustrated through an application of ACC for resource-aware security that our approach is inherently parametric and supports a very rich set of domains. We believe that ACC provides novel means for certifying security by enhancing mobile code with certificates which guarantee that the execution of the (in principle untrusted) code received from another node in the network is *safe* but also, as mentioned above, *efficient*, according to a predefined safety policy which includes properties related to *resource consumption*. We have illustrated the approach using the CiaoPP system. This system already uses a combination of abstract interpretation, abstract specialization, and a flexible assertion language, to perform program debugging, verification, and optimization with a wide variety of domains, and has been enhanced to produce certificates as dictated by the ACC scheme, as an integral part of the static debugging and verification performed during the program development process. A simplified version of the analysis framework of CiaoPP has also been developed that serves as an efficient checker of the certificates. The approach is currently being tested in a number of pervasive applications using an embedded version of the Ciao system.

8. 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)*, April 2004.

[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] A. Appel and A. Felty. Lightweight Lemmas in lambda-Prolog. In *Proc. of ICLP'99*, pages 411–425. MIT Press, 1999.

[4] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In *Proc. of CADE'02*, pages 31–46. Springer LNCS, 2002.

[5] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[6] 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.

[7] B. L. 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.

[8] P. Cousot. Types as Abstract Interpretations. In *ACM Symposium on Principles of Programming Languages*, pages 316–331. ACM Press, January 1997.

[9] 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.

[10] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[11] S. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[12] S. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[13] 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.

[14] 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.

[15] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[16] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[17] 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.

[18] K. Marriott, H. Søndergaard, and N. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[19] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[20] 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.

[21] 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.

[22] G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.

[23] 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.

[24] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[25] 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.

[26] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[27] M. Weiser. The computer for the twenty-first century. *Scientific American*, 3(265):94–104, September 1991.