Abstraction-Carrying Code

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

[1] DSIP, Universidad Complutense Madrid
[2] Facultad de Informática, Technical University of Madrid
[3] Depts. of Comp. Sci. and El. and Comp. Eng., U. of New Mexico

Abstract. *Proof-Carrying Code* (PCC) is a general approach to mobile code safety in which programs are augmented with a certificate (or proof). The practical uptake of PCC greatly depends on the existence of a variety of enabling technologies which allow both to prove programs correct and to replace a costly verification process by an efficient checking procedure on the consumer side. In this work we propose *Abstraction-Carrying Code* (ACC), a novel approach which uses abstract interpretation as enabling technology. We argue that the large body of applications of abstract interpretation to program verification is amenable to the overall PCC scheme. In particular, we rely on an expressive class of safety policies which can be defined over different abstract domains. 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 believe that ACC brings the expressiveness, flexibility and automation which is inherent in abstract interpretation techniques to the area of mobile code safety. We have implemented and benchmarked ACC within the `Ciao` system preprocessor. The experimental results show that the checking phase is indeed faster than the proof generation phase, and that the sizes of certificates are reasonable.

1 Introduction

One of the most important challenges which computing research faces today is the development of security techniques for verifying that the execution of a program (possibly) supplied by an untrusted source is *safe*, i.e., it meets certain properties according to a predefined *safety policy*. Proof-Carrying Code (PCC) [15] 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.

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. In Typed Assembly Languages [13], the certificate is a type annotation of the assembly language program and the checking process involves a form of type checking. Each of the different approaches possess their own set of stronger and weaker points. Depending on the particular safety property and the available computing resources in the consumer, some approaches are more suitable than others. In some cases the priority is to reduce the size of the certificate as much as possible in order to fit in small devices or to cope with scarce network access (as in, e.g., Oracle-based PCC [17] or Tactic-based PCC [1]), whereas in other cases the priority is to reduce the checking time (as in, e.g., standard PCC [15] or lightweight bytecode verification [11]). As a result of all this, a successful certificate infrastructure should have a wide set of enabling technologies available for the different requirements.

In this work we propose *Abstraction-Carrying Code* (ACC), a novel approach which uses *abstract interpretation* [5] as enabling technology to handle the above practical (and difficult) challenges. 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, thus obtaining safe approximations of the behavior of the program. The technique allows inferring much richer information than, for example, traditional types. This includes 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 above domains as enabling technology for PCC. In particular, ACC has the following three fundamental elements:

1. An expressive class of safety policies based on "abstract"—i.e. symbolic—properties over different abstract domains. Our framework is parametric w.r.t. the abstract domain(s) of interest, which gives us generality and expressiveness.
2. A fixpoint static analyzer is used to automatically infer an abstract model (or simply *abstraction*) about the mobile code which can then be used to prove that the code is safe w.r.t. the given policy in a straightforward way.

We identify the particular *subset* of the analysis results which is sufficient for this purpose.

3. A simple, easy-to-trust (analysis) checker verifies the validity of the information on the mobile code. It is indeed a specialized abstract interpreter whose key characteristic is that it does not need to iterate in order to reach a fixpoint (in contrast to standard analyzers).

While ACC is a general approach, for concreteness we develop herein 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. Also for concreteness, we build on the algorithms of (and report on an implementation on) `CiaoPP` [8], the abstract interpretation-based preprocessor of the `Ciao` multi-paradigm (Constraint) Logic Programming system. `CiaoPP` uses modular, incremental abstract interpretation as a fundamental tool to obtain information about programs. 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, resource usage control, and program verification. We report on our extension of the framework to incorporate ACC and on how this instantiation of ACC already shows promising results.

2 An Assertion Language to Specify the Safety Policy

The purpose of a *safety policy* is to specify precisely the conditions under which the execution of a program is considered safe. We propose the use of (a subset of) the high-level *assertion* language [18] available in `CiaoPP` to define an expressive class of safety policies in the context of *constraint logic programs*.

2.1 Preliminaries and Notation

We assume familiarity with constraint logic programming [10] (CLP) and the concepts of abstract interpretation [5] which underlie most analyses in CLP. The remaining of this section introduces some notation and recalls preliminary concepts on these topics.

Terms are constructed from variables (e.g., X), functors (e.g., f) and predicates (e.g., p). We denote by $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i) = t_i$ for all $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$) and $\sigma(X) = X$ for any other variable X, where t_i are terms. A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} \equiv \rho^{-1}\rho \equiv id$. We say that a renaming ρ is a *renaming substitution* of term t_1 w.r.t. term t_2 if $t_2 = \rho(t_1)$.

A *constraint* is essentially a conjunction of expressions built from predefined predicates. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *literal* is either an atom or a constraint. A *goal* is a finite sequence of literals. A *rule* is of the form H `:-` B where H, the *head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *CLP program*, or *program*, is a finite set of rules.

Example 1. The main predicate, `create_streams/2`, of the following CLP program receives a list of numbers which correspond to certain file names, and returns in the second argument the list of file handlers (*streams*) associated to the (opened) files:

```
create_streams([],[]).
create_streams([N|NL],[F|FL]):-
        number_codes(N,ChInN), app("/tmp/",ChInN,Fname),
        safe_open(Fname,write,F), create_streams(NL,FL).

safe_open(Fname,Mode,Stream):-
        atom_codes(File,Fname), open(File,Mode,Stream).
```

The call `number_codes(N,ChInN)` receives the number `N` and returns in `ChInN` the list of the ASCII codes of the characters comprising a representation of `N`. Then, it uses the well-known list concatenation predicate `app/3`. The call `atom_codes(File,Fname)` receives in `Fname` a list of ASCII codes and returns the atom `File` made up of the corresponding characters. Also, a call such as `open(File,Mode,Stream)` opens the file named `File` and returns in `Stream` the stream associated with the file. The argument `Mode` can have any of the values: `read`, `write`, or `append`.[4]

A distinguishing feature of our approach is that a class of safety policies can be defined for the different *abstract domains* available in the system. In particular, safety properties are expressed as *substitutions* in the context of an abstract domain (D_α) which is simpler than the selected *concrete domain* (D). An abstract value is a finite representation of a, possibly infinite, set of actual values in the concrete domain. Our approach relies on the abstract interpretation theory [5], 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 $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x$ 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. In this framework an *abstract property* is defined as an abstract substitution which allows us to express properties, in terms of an abstract domain, that the execution of a program must satisfy. The description domain we use in our examples is the following *regular type* domain [6].

*Example 2 (*regular type domain*).* We refer to the *regular type* domain as *eterms*, since it is the name it has in `CiaoPP`. Abstract substitutions in *eterms* [21], over a set of variables V, assign a *regular type* to each variable in V. We use in our examples `term` as the most general type (i.e., `term` $\equiv \top$ corresponds

[4] Predicates `number_codes/2`, `atom_codes/2`, and `open/3` are ISO-standard Prolog predicates, and thus they are available in `CiaoPP`.

to all possible terms). 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)`. Also, `list(T)` $\sqsubseteq$ `list` $\sqsubseteq$ `term` for any type `T`. The least general substitution $\bot$ assigns the empty set of values to each variable.[5]

Apart from predefined types, in the *eterms* domain, one can have user-defined regular types declared by means of *Regular Unary Logic* programs [7]. For instance, in the context of mobile code, it is a safety issue whether the code tries to access files which are not related to the application in the machine consuming the code. A very simple safety policy can be to enforce that the mobile code only accesses temporary files. In a UNIX system this can be controlled (under some assumptions) by ensuring that the file resides in the directory `/tmp/`. The following regular type `safe_name` defines this notion of safety:[6]

```
:- regtype safe_name/1.
safe_name("/tmp/"||L) :- list(L,alphanum_code).

:- regtype alphanum_code/1.
alphanum_code(X):- member(X,"abcdefghijklmnopqrstuvwzyz").
alphanum_code(X):- member(X,"ABCDEFGHIJKLMNOPQRSTUVWXYZ").
alphanum_code(X):- member(X,"0123456789").
```

The abstract property made up of substitution {`X`$\mapsto$`safe_name`} expresses that `X` is bound to a string which starts by the prefix "`/tmp/`" followed by a list of alpha-numerical characters. In the following, we write simply `safe_name(X)` to represent it.

2.2 The Safety Policy

Assertions are syntactic objects which allow expressing a wide variety of high-level properties of (in our case CLP-) programs. Examples are assertions which state information on *entry points* to a program module, assertions which describe properties of built-ins, assertions which provide some type declarations, cost bounds, etc. The original assertion language [18] available in `CiaoPP` is composed of several assertion schemes. Among them, we simply consider the two following schemes for the purpose of this paper, which intuitively correspond to the traditional pre- and postcondition on procedures.

calls$(B, \{\lambda^1_{Pre}; \ldots; \lambda^n_{Pre}\})$**:** They express properties which should hold in *any* call to a given predicate similarly to the traditional precondition. B is a *predicate descriptor*, i.e., it has a predicate symbol as main functor and all arguments are distinct free variables, and λ^i_{Pre}, $i = 1, \ldots, n$, are abstract properties about execution states. The resulting assertion should be interpreted as "in all activations of B *at least* one property λ^i_{Pre} should hold in the calling state."

[5] Let us note that certain abstract domains assign a different meaning to $\bot$. In these cases, a distinguished symbol (i.e., an extra $\bot$) can always be added to represent unreachable points.

[6] The `regtype` declarations are used to define new regular types in `CiaoPP`.

success$(B, [\lambda_{Pre},]\lambda_{Post})$**:** This assertion schema is used to describe a *postcondition* which must hold on all success states for a given predicate. B is a predicate descriptor, and λ_{Pre} and λ_{Post} are abstract properties about execution states. λ_{Pre} is optional and must be evaluated w.r.t. the store at the calling state to the predicate while condition λ_{Post} is evaluated at the success state. If the optional λ_{Pre} is present, then λ_{Post} is only required to hold in those success states which correspond to call states satisfying λ_{Pre}. Note that several success assertions with different λ_{Pre} may be given.

Therefore, abstract properties λ_{Pre} and λ_{Post} in assertions allow us to express conditions, in terms of an *abstract domain*, that the execution of a program must satisfy. Each condition is an abstract substitution corresponding to the variables in some atom. In existing approaches, safety policies usually correspond to some variants of type safety (which may also control the correct access of memory or array bounds [16]). In our system, the (co-)existence of several domains allows expressing a wider range of properties using the assertion language. They include a wide class of safety policies based on modes, types, non-failure, termination, determinacy, non-suspension, non-floundering, cost bounds, and their combinations.

In the `CiaoPP` preprocessor, the assertion language allows us to define the safety policy for the run-time system in the presence of foreign functions, built-ins, etc. In general, it is the task of the compiler designer to define the safety policies associated to the predefined system predicates. In addition to these assertions, the user can optionally provide further assertions manually for user-defined predicates.

Example 3. The following assertion for predicate `safe_open`:

```
calls(safe_open(Fname,_,_), {safe_name(Fname)})
```

provides a simple way to guarantee that all calls to `open` are safe. It can be read as "the calling conventions for predicate `safe_open` require that the first argument be a `safe_name`". Meanwhile the following assertion for `open` is predefined in our system:

```
success(open(X,Y,Z), ⊤, {constant(X),io_mode(Y),stream(Z)})
```

It requires, upon success, the first variable to be of type `constant`, the second a proper `io_mode` and the last one of type `stream`.

In contrast to traditional approaches, assertions are not compulsory for every predicate. Thus, the user can decide how much effort to put into writing assertions: the more of them there are, the more complete the partial correctness of the program is described and more possibilities to detect problems. Indeed, pre- and post-conditions are frequently provided by programmers since they are often easy to write and very useful for generating program documentation. Nevertheless, the analysis algorithm is able to obtain safe approximations of the program behavior even if no assertions are given. This is not always the case in other approaches such as classical program verification, in which loop invariants are actually required. Such invariants are hard to find and existing automated techniques are generally not sufficient to infer them, so that often they have to be provided by hand.

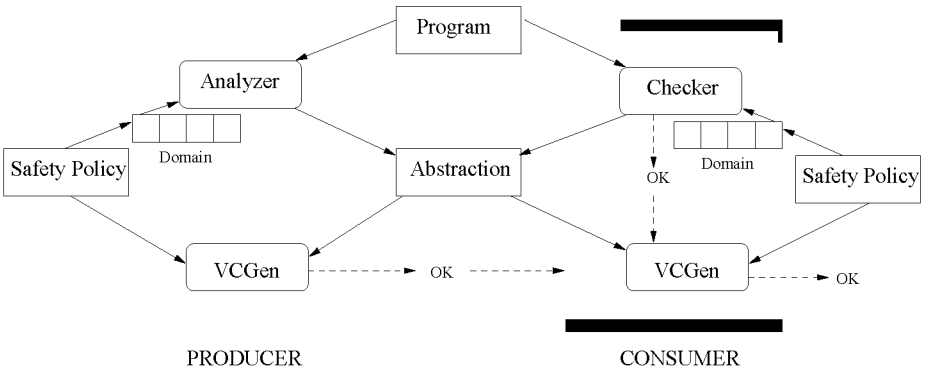

Fig. 1. Abstraction-Carrying Code in CiaoPP

3 Certifying Programs by Static Analysis

Fig. 1 presents an overview of ACC as performed in the CiaoPP system. This section introduces the *certification* process (sketched to the left of the figure) carried out by the producer, i.e., the generation of a certificate to attest the adherence of the program to the safety policy. The whole certification method is based on the following idea: *an abstraction of the program computed by abstract interpretation-based analyzers can play the role of certificate for attesting program safety.* Our certification process is carried out in the following phases. We start from an initial program P. Firstly, the Safety Policy is defined by means of a set of assertions AS in the context of an abstract domain D_α, as introduced in Sect. 2, among a repertoire of Domains available in the system. Secondly, a standard Analyzer is run, which returns an *abstraction* of P's execution in terms of the abstract domain D_α. Let us note that the analyzer 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 during the generation of the safety assertions. Then, a verification condition generator, VCGen extracts, from the initial assertions and the abstraction, a *Verification Condition* (VC) which can be proved only if the execution of the code does not violate the safety policy. If VC can be proved (marked as OK in Fig. 1), then the certificate (i.e., the abstraction) is sent together with the program P to the code consumer. Sections 3.1 and 3.2 give further details on the Abstraction and the VCGen process, respectively.

3.1 Using Analysis Results as Certificates

A key idea in our certification process is that the certificate is automatically generated by an abstract interpretation-based analyzer (or simply static analyzer). In particular, the *goal dependent* (a.k.a. goal oriented) analyzer of [9], which is the one implemented in the CiaoPP system, plays the role of Analyzer. This analysis algorithm (we simply write *Analysis* for short in the following) receives as input, in addition to the program P and the abstract domain D_α, a set of *calling patterns* CP. A calling pattern is a description of the calling modes

(or entries) into the program. For simplicity, we assume that P comes enhanced with its entries CP. In particular, a set of calling patterns Q consists of a set of pairs of the form $\langle A : CP \rangle$ where A is a predicate descriptor and CP is an abstract substitution (i.e., a condition of the run-time bindings) of A expressed as $CP \in D_\alpha$. In principle, calling patterns are only required for exported predicates. The analysis algorithm is able to generate them automatically for the remaining internal predicates. Nevertheless, they can still be automatically generated by assuming $\top$ (i.e., no initial data) for all exported predicates (although the idea is to improve this information in the initial calling patterns).

In order to compute $Analysis(P, Q, D_\alpha)$, traditional (goal dependent) abstract interpreters for (C)LP programs construct an *and–or graph* (or analysis graph) which corresponds to (or approximates) the abstract semantics of the program [2]. The graph has two sorts of nodes: *or–nodes* and *and–nodes*. Or–nodes correspond to literals whilst and–nodes to rules. Both kinds of nodes are interleaved in the graph and connected as follows. An or–node has arcs to those and–nodes which correspond to the rules whose head unifies with the literal it represents. An and–node for a rule H :- $B_1, \ldots, B_n$ has n arcs to the or–nodes which corresponds to the literals B_i in the body of the rule. Due to space limitations, and given that it is now well understood, we do not describe here algorithm $Analysis(P, Q, D_\alpha)$ (details can be found in, e.g., [9]). Nevertheless, the checking algorithm of Sect. 4 illustrates how an and–or graph is traversed.

The analysis graph computed by `CiaoPP`'s analyzer represents an *abstract model* (or abstraction) of the program. It is represented by means of two data structures in the output: the *answer table* and the *arc dependency table*. The following definition introduces the notion of analysis table (similar definitions can be found, e.g., in [2, 9]). Informally, it says that its entries are of the form $\langle A : CP \mapsto AP \rangle$ which should be interpreted as "the answer pattern for calls to A satisfying precondition (or call substitution), CP, accomplishes postcondition (or success substitution), AP."

Definition 1 (AT – analysis answer table). *Let P be a program. Let Q be a set of calling patterns expressed in the abstract domain D_α. We define an* analysis answer table*, AT, as the set of entries $\langle A_j : CP_j \mapsto AP_j \rangle$, $\forall j = 1..n$ computed by $Analysis(P, Q, D_\alpha)$[9] where, in each entry, A_j is an atom and CP_j and AP_j are, respectively, the abstract call and success substitutions.*

Intuitively, the answer table contains the answer patterns for all literals in the or–nodes of the graph while the arc dependency table keeps detailed information about dependencies among or–nodes in the graph. A central idea in this work is that, for certifying program safety, it suffices to send the information stored in the analysis answer table. In contrast to the original generic algorithm [9], a simple analysis checker can be designed for validating the answer table without requiring the use of the arc dependency table at all (as we show in Sect. 4). The theory of abstract interpretation guarantees that the answer table is a safe approximation of the runtime behavior (see [2, 9] for details).

Example 4. Take the calling pattern $\langle$`create_streams(X,Y)`, {`list(X,num)`}$\rangle$, which indicates that calls to `create_streams` are performed with a list of num-

bers in the first argument. The answer table computed by CiaoPP contains (among others) these entries:

$\langle$create_streams(A,B) : {list(A,num)} $\mapsto$ {list(A,num),list(B,stream)}$\rangle$
$\langle$safe_open(A,B,C) : {sf(A),B = write} $\mapsto$ {sf(A),B = write,stream(B)}$\rangle$

The first entry should be interpreted as: all calls to predicate create_streams provide as input a list of numbers in the first argument and, upon success, they yield lists of numbers and streams, respectively, in each of its two arguments. In the second entry, it is interesting to note that CiaoPP creates the auxiliary type:

```
sf("/tmp/"||A):-list(A,numcodes).
```

to represent lists of numbers starting by the prefix "/tmp/". We use the notation B = write to denote that the system generates a new type for B whose only element is constant write.

In order to increase accuracy, analyzers are usually *multivariant* on calls (see, e.g., [9]). Indeed, though not visible in this example, CiaoPP incorporates a multivariant analysis, i.e., more than one triple $\langle A : CP_1 \mapsto AP_1\rangle$,..., $\langle A : CP_n \mapsto AP_n\rangle$ $n > 1$ with $CP_i \neq AP_i$ for some i, j may be computed for the same predicate descriptor A.

It is important to note that our approach would work directly in other programming paradigms, such as imperative or functional programming (the latter already covered in our current system), 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 these paradigms.

3.2 The Verification Condition

In the next step, the verification condition generator (VCGen in Fig. 1) extracts, from the initial assertions and answer table, a *Verification Condition* (VC) which can be proved only if the execution of the code does not violate the safety policy.

Definition 2 (VC – verification condition). *Let AT be an analysis answer table computed for a program P and a set of calling patterns Q in the abstract domain D_α. Let S be an assertion. Then, the verification condition, $VC(S, AT)$, for S w.r.t. AT is defined as follows:*

$$VC(S,AT) ::= \begin{cases} \bigwedge_{\langle A:CP\mapsto AP\rangle\in AT} (\rho(CP) \sqsubseteq \lambda^1_{Prec} \vee \ldots \vee \rho(CP) \sqsubseteq \lambda^n_{Prec}) \\ \qquad\qquad \text{if } S = calls(B, \{\lambda^1_{Prec}; \ldots; \lambda^n_{Prec}\}) \\ \bigwedge_{\langle A:CP\mapsto AP\rangle\in AT} \rho(CP) \sqcap \lambda_{Prec} = \bot \vee \rho(AP) \sqsubseteq \lambda_{Post} \\ \qquad\qquad \text{if } S = success(B, \lambda_{Prec}, \lambda_{Post}) \end{cases}$$

where ρ is a variable renaming substitution of A w.r.t. B.

If AS is a finite set of assertions, then its verification condition, $V(AS, AT)$, is the conjunction of the verification conditions of the elements of AS.

Roughly speaking, the VC generated according to Def. 2 is a conjunction of boolean expressions (possibly containing disjunctions) whose validity ensures the consistency of a set of assertions w.r.t. the answer table computed by *Analysis*. It distinguishes two different cases depending on the kind of assertion. For *calls* assertions, the VC requires that at least one precondition λ^i_{Prec} be a safe approximation of all existing abstract calling patterns for the atom B. In the case of *success* assertions, there are two cases for them to hold. The first one indicates that the precondition is never satisfied and, thus, the assertion trivially holds (and the postcondition does not need to be tested). The second corresponds to the case in which the success substitutions computed by analysis for the predicate are more particular than the one required by the assertion.

Example 5. Consider the entry for predicate `safe_open` in the answer table of Ex. 4 and the *calls* assertion of Ex. 3 for the same predicate. According to Def. 2, the VC is: $\texttt{B} = \texttt{write}, \texttt{sf(X)} \sqsubseteq \texttt{safe\_name(X)}$ whose validity can be easily proved in our system since $\texttt{sf} \sqsubseteq \texttt{safe\_name}$. This allows `CiaoPP` to infer that calls to `open` performed within this program satisfy the simple safety policy discussed in Ex. 1. The complete example includes further assertions for the different predicates and its corresponding VCs. We do not include them here due to space limitations.

Therefore, upon creating the answer table and generating the VC, the validity of the whole boolean condition is checked by resolving each conjunct separately. Note that each conjunct consists of comparisons of pairs of abstract substitutions, which simply return either true or false but do not compute any substitution. This validation may yield three different possible status: i) the VC is indeed checked and the AT is considered a valid abstraction (marked as OK), 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. The latter case happens because some properties are undecidable and the analyzer performs approximations in order to always terminate. Therefore, it may not be 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 shown in the picture, in both the ii) and iii) cases, the certification process needs to be restarted until achieving a VC which meets i).

The following theorem states the soundness of the VC. Intuitively, it amounts to saying that if the VC holds, then the execution of the program will preserve all safety assertions. Following the notation of [15], we write $\rhd VC$ when VC is valid.

Theorem 1 (Soundness of the Verification Condition). *Let AT be an analysis answer table for a program P and a set of calling patterns Q in an abstract domain D_α (as defined in Def. 1). Let AS be a set of assertions. Let $VC(AS, AT)$ be the verification condition for AS w.r.t. AT (generated as stated in Def. 2). If $\rhd VC(AS, AT)$, then P satisfies all assertions in AS for all computations described by Q.*

This result derives from the fact that the static analysis algorithm of [9] computes a safe approximation of the stores reached during computation.

4 Checking Safety in the Consumer

The checking process performed by the consumer is illustrated in 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 on 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. Thus, in the *validation* process, a code consumer not only checks the validity of the answer table but it also (re-)generates a trustworthy VC. The re-generation of VC (and its corresponding validation) is identical to the process already discussed in the previous section. Therefore, this section describes only the former part of the validation process, i.e., algorithm check.

Although global analysis is now routinely used as a practical tool, it is still unacceptable to run the whole *Analysis* 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 *Analysis* algorithm can be understood as: $Analysis = fixpoint(analysis\_step)$. I.e., a process which repeatedly performs a traversal of the analysis graph (denoted by *analysis_step*) until the computed information does not change. 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, `check` $\equiv$ *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 answer table is valid, one single execution of *analysis_step* validates the certificate.

The next definition presents our *abstract interpretation-based checking* algorithm. It receives as an additional input a Certificate (which is the analysis fixpoint). In a single traversal, it constructs a program analysis graph by using the information in Certificate. The algorithm is devised as a graph traversal procedure which places entries in a *local* answer table, AT, as new nodes in the program analysis graph are encountered. Thus, it handles two distinct answer tables: the local AT + the incoming Certificate. The final goal of the checking is to reconstruct the analysis graph and compare the results with the information

stored in Certificate. As long as Certificate is valid, both results coincide and, thus, the certificate is guaranteed to be valid w.r.t. the program.

Definition 3 (Analysis Checker). *Let P be a normalized*[7] *program and Q be a set of calling patterns in the abstract domain D_α. Let* Certificate *be a safety certificate as defined in Def. 1. The validation of* Certificate *is performed by the procedure* `check` *depicted in Figure 2. The algorithm uses a local answer table, AT, to compute the results (initially it does not contain any entry). Procedure* `check` *is defined in terms of five abstract operations [9] on the description domain D_α of interest:*

- Arestrict(CP, V) *performs the abstract restriction of a description CP to the set of variables in the set V, denoted $vars(V)$;*
- Aextend(CP, V) *extends the description CP to the variables in the set V;*
- Aadd(C, CP) *performs the abstract operation of conjoining the actual constraint C with the description CP;*
- Aconj(CP_1, CP_2) *performs the abstract conjunction of two descriptions;*
- Alub(CP_1, CP_2) *performs the abstract disjunction of two descriptions.*

Following the presentation of *Analysis* [9], we assume that the program P and the answer table are global parameters throughout the algorithm. The checking algorithm proceeds as follows. For each calling pattern in the set Q, the procedure process_node inspects all rules defining the considered atom. For each rule, it performs a left-to-right traversal of the atoms in the rule body. The processing of each atom $B_{k,i}$ in the rule body is handled by process_arc. We refer by CP_b to the description of the program point immediately *before* the atom $B_{k,i}$ and by CP_a to the description *after* processing the atom. Initially, the description CP_b takes the value of the initial description CP for the calling pattern $A : CP$ (extended to all the variables in the rule).[8] We use variables CPR_x to denote that description CP_x has been restricted, with $x \in \{a, b\}$. The procedure process_arc is aimed at computing the resulting description CP_a after processing a given atom $B_{k,i}$. It distinguishes two different cases:

- Constraints are simply abstractly added to the current description.
- If $B_{k,i}$ is an atom, then it inspects whether it has been processed before:
  - If the atom already has an entry in the answer table, we do not need to recompute it. Indeed, this could risk the termination of the algorithm.
  - Otherwise, we process it by executing procedure process_node. On return, and in the absence of errors, this processing will have placed an answer for $B_{k,i}$ in the answer table (and possibly for other related atoms as well).

  Either way, there will be an answer for the atom at this point. This answer is *conjoined*[8] with the description CP_b from the program point immediately before $B_{k,i}$ in order to obtain the description for the program point after it.

[7] For clarity of presentation, in the algorithm we assume that all rule heads are normalized, i.e., H is of the form $p(X_1, ..., X_n)$ where $X_1, ..., X_n$ are distinct free variables.

[8] Further insights on the operations on abstract substitutions (like extensions, restrictions, disjunctions etc.) can be found in [2].

```
check(Q, Certificate)
    foreach A : CP ∈ Q
        process_node(A : CP, Certificate)
    return Valid

process_node(A : CP, Certificate)
    if (∃ a renaming σ s.t. σ(A : CP ↦ AP) in Certificate)
        then add (A : CP ↦ AP) to AT
        else return Error
    foreach rule A_k ← B_{k,1}, ..., B_{k,n_k} in P
        W := vars(A_k, B_{k,1}, ..., B_{k,n_k})
        CP_b :=Aextend(CP, vars(B_{k,1}, ..., B_{k,n_k}))
        CPR_b := Arestrict(CP_b, B_{k,1})
        foreach B_{k,i} in the rule body i = 1, ..., n_k
            CP_a := process_arc(B_{k,i} : CPR_b, CP_b, W, Certificate)
            if (i <> n_k) then CPR_a := Arestrict(CP_a, var(B_{k,i+1}))
            CP_b := CP_a
            CPR_b := CPR_a
        AP_1 := Arestrict(CP_a, vars(A_k))
        AP_2 := Alub(AP_1, σ^{-1}(AP))
        if AP <> AP_2 then return Error

process_arc(B_{k,i} : CPR_b, CP_b, W, Certificate)
    if B_{k,i} is a constraint then CP_a := Aadd (B_{k,i}, CP_b)
    elseif (∄ a renaming σ s.t. σ(B_{k,i} : CPR_b ↦ AP') in AT)
        then process_node (B_{k,i} : CPR_b, Certificate)
        AP_1 := Aextend (ρ^{-1}(AP), W) where ρ is a renaming s.t.
                                        ρ(B_{k,i} : CPR_b ↦ AP) in AT
        CP_a := Aconj (CP_b, AP_1)
    return CP_a
```

Fig. 2. Abstract Interpretation-based Checking in `CiaoPP`

The computed result is used to process the next literal in the rule when $B_{k,i}$ is not the last literal. Otherwise, the computed result constitutes indeed the computed answer for the rule. The answer is *combined*[8] with the corresponding answer supplied by the certification process in Certificate. If Certificate is valid, the comparison should hold; otherwise the process prompts an error and the program is not safe to run.

The following theorem ensures that algorithm check is able to validate safety certificates which are stored in a valid analysis answer table.

Theorem 2 (partial correctness). *Let P be a program, let Q be a set of calling patterns in an abstract domain D_α. Let* Certificate *be a safety certificate for P and Q as stated in Def. 1. Then,* check$(Q,$ Certificate$)$ *terminates and validates* Certificate *in P.*

The theorem can be demonstrated by showing that check is a simplified version of *Analysis* [9] in two main aspects. Regarding the efficiency, our point to justify an efficient behavior of `check` for validating an answer table is that it performs a

	Analysis			Checking			Speedup		Source	Byte Code		Certificate	
Bench	P_A	An	T_A	P_C	Ch	T_C	A/C	T_A/T_C	Source	ByteC	B/S	Cert	C/S
aiakl	2	87	89	2	71	72	1.2	1.2	1555	3805	2.4	3090	2.0
ann	22	452	474	18	254	272	1.8	1.7	12745	43884	3.4	24475	1.9
bid	4	56	60	4	35	38	1.6	1.6	4945	10376	2.1	5939	1.2
boyer	9	143	151	7	85	92	1.7	1.6	11010	32522	3.0	12300	1.1
browse	3	14	17	3	12	15	1.2	1.2	2589	8467	3.3	1661	0.6
deriv	2	86	88	1	19	20	4.6	4.4	957	4221	4.4	288	0.3
grammar	2	10	12	2	9	11	1.1	1.1	1598	3182	2.0	1259	0.8
hanoiapp	2	25	26	2	16	18	1.5	1.5	1172	2264	1.9	2325	2.0
mmatrix	1	13	14	1	10	11	1.3	1.3	557	1053	1.9	880	1.6
occur	2	16	18	2	10	12	1.7	1.6	1367	6903	5.0	1098	0.8
progeom	2	13	15	2	9	11	1.5	1.4	1619	3570	2.2	2148	1.3
read	9	792	801	8	488	497	1.6	1.6	11843	24619	2.1	25359	2.1
qplan	13	1411	1424	11	962	973	1.5	1.5	9983	33472	3.4	20509	2.1
qsortapp	1	20	21	1	12	14	1.6	1.5	664	1176	1.8	2355	3.5
query	5	11	15	4	9	12	1.2	1.3	2090	8833	4.2	531	0.3
rdtok	8	141	149	6	43	49	3.3	3.1	13704	15354	1.1	6533	0.5
serialize	2	40	42	2	17	19	2.3	2.2	987	3801	3.9	1779	1.8
warplan	8	173	181	7	108	115	1.6	1.6	5203	23971	4.6	15305	2.9
witt	16	196	212	14	72	86	2.7	2.5	17681	41760	2.4	19131	1.1
zebra	3	94	97	3	90	92	1.1	1.0	2284	5396	2.4	4058	1.8
Overall							1.63	1.61	1		2.66		1.44

Table 1. Checking Time and Certificate Size

single graph traversal. Indeed, for a regular type domain, [4] demonstrates that directional type-checking for logic programs is fixed-parameter linear. The next section reports experimental evidence of efficiency issues.

5 Experimental Results

In this section we show some experimental results aimed at studying two crucial points for the practicality of our proposal: the checking time as compared to the analysis time, and the size of certificates. We have implemented the checker as a simplification of the generic abstract interpretation system of `CiaoPP`. It should be noted that this is an efficient, highly optimized, state-of-the-art analysis system and which is part of a working compiler. Both the analysis and checker are parametric w.r.t. the abstract domain. In these experiments they both use the same implementation of the domain-dependent functions of the *sharing+freeness* domain [14]. We have selected this domain because the information it infers is very useful for reasoning about instantiation errors, which is a crucial aspect for the safety of logic programs. The whole system is implemented in Ciao 1.11#200 [3] with compilation to bytecode. All of our experiments have been performed on a Pentium 4 at 2.4GHz and 512MB RAM running GNU Linux RH9.0. The Linux kernel used is 2.4.25, customized with the *hrtime* patch to provide improved precision and resolution in time measurements.

Execution times are given in milliseconds and measure *runtime*. They are computed as the arithmetic mean of five runs. A relatively wide range of pro-

grams has been used as benchmarks. They are the same ones used in [9], where they are described in some detail. For each benchmark, the columns for `Analysis` are the following: P_A is the time required by the *preprocessing phase*, in which program clauses are processed and stored in the format required by the analyzer. The *analysis* time proper is shown in column `An`. The actual time needed for analysis –the sum of these two times– is shown in column T_A. Similarly, in the case of checking, three columns are shown. The preprocessing phase, P_C, includes asserting the certificate in addition to asserting the program to be analyzed. As the figures show, the overhead required for asserting the certificate is negligible. Column `Ch` is the time for executing the checking algorithm. Finally, T_C is the total time for checking. The columns under `Speedup` compare analysis and checking times. As can be seen in columns `A/C` and T_A/T_C, the checking algorithm is faster than the analysis algorithm in all cases. The actual speedup ranges from almost none, as in the case of zebra, to over four times faster in the case of deriv. The last row summarizes the results for the different benchmarks using a weighted mean, which places more importance on those benchmarks with relatively larger analysis times. We use as weight for each program its actual analysis time. We believe that this weighted mean is more informative than the arithmetic mean, as, for example, doubling the speed in which a large and complex program is analyzed (checked) is more relevant than achieving this for small, simple programs. Overall, the speedup is 1.63 in just analysis time, or 1.61 if we also take into account the preprocessing time. We believe that the achieved speedup is significant taking into account that `CiaoPP`'s analyzer for this domain is highly optimized and converges very efficiently. However, it is to be expected that, for other domains and implementations, the relative gains will be higher.

The second part of the table studies the size of the certificates, coded in compact (*fastread*) format, for the different benchmarks and compares it to the size of the source code for the same program and to the size of the corresponding bytecode. To make this comparison fair, we subtract 4180 bytes from the size of the bytecode for each program: the size of the bytecode for an empty program in this version of Ciao (minimal top-level drivers and exception handlers for any executable). The results show the size of the certificate to be quite reasonable. It ranges from 0.3 times the size of the source code (for deriv) to 3.5 (in the case of qsortapp). Overall, it is 1.44 times the size of the source code. We consider this acceptable since in general Prolog programs are quite compact (up to 10 times more compact than equivalent imperative programs). In fact, the size of source plus certificate is smaller (1+1.44) than that of the bytecode (2.66).

6 Discussion and Related Work

The main contribution of this work is to introduce, implement, and (preliminarily) benchmark *abstraction-carrying code* (ACC) as a novel enabling technology for PCC, which is based throughout on the use of 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.

A type-level dataflow analysis of Java virtual machine bytecode is also the basis of most existing verifiers [12, 11], and some are loosely based on abstract interpretation. These analyses allow proving that the program is correct w.r.t. type-related correctness conditions. In [19] a proposal is presented 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. As in our case, the second phase can be done in a single, linear pass over the bytecode. However, these approaches are designed limited to types, whereas our approach is inherently parametric and thus supports a very rich set of domains, and combinations of several of them. Let us note that the checker is part of the trusted computing base and, hence, the code consumer has to trust also the domain operations. Other approaches to PCC use logic-based verification methods as enabling technology, an example is [22] which formalises a simple assembly language with procedures and presents a safety policy for arithmetic overflow in Isabelle/HOL. The coexistence of several abstract domains in our framework is somewhat related to the notion of *models* to capture the security-relevant properties of code, as addressed in the work on Model-Carrying Code (MCC) [20].

Another difference between our work and other related work is that the instance that we have described is actually defined at the source-level, whereas in existing PCC frameworks the code supplier typically packages the certificate with the *object* code rather than with the *source* code (both are untrusted). Actually, both approaches are of interest from our point of view (and, in fact, our approach can also be applied to bytecode). Open-source code is becoming much more relevant these days (in fact, `Ciao` and `CiaoPP` are themselves GNU-licensed and available in source code). As a result, it is now realistic to expect that a relatively large amount of untrusted source code is available to the consumer. The advantages of open-source with respect to safety are important since it allows inspecting the code and applying powerful techniques for program analysis and validation which allow inferring information which may be difficult to observe in low-level, compiled code.