ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Safety in Pervasive Computing

Deliverable number:	D13
Workpackage:	Resource-Oriented Specialization (WP4)
Preparation date:	1 November 2003
Due date:	1 May 2004
Classification:	Public
Lead participant:	Univ. of Southampton
Partners contributed:	Tech. Univ. of Madrid (UPM), Roskilde Univ

Project funded by the European Community under the "Information Society Technologies" (IST) Programme (1998–2002).

Short description:

The aim of task 6.1 is to study safety conditions which are applicable to pervasive computing and develop a framework for certifying that the execution of a code is safe according to the safety conditions established via a predefined policy. We present a deliverable D13 which consists of four main parts.

The first part, D13.1, presents a framework for the verification of CLP programs based on abstract interpretation. This deliverable was included already in the previous period report. We introduce a practical verification framework which can be used to approximate at compile-time a wide range of properties, from directional types to variable independence, determinacy or termination, always safely, and with a significant degree of precision. Our proposed approach takes advantage, within the context of program verification and debugging, of significant advances in static program analysis techniques and the resulting concrete tools, which have been shown useful for other purposes such as optimization, and are thus likely to be present in compilers. The focus is put on compile-time checking of assertions which is conceptually more powerful than run-time checking albeit it is also more complex. The deliverable pursues that the results of compile-time checking are valid for *any* query which satisfies the existing *entry* declarations, hence compile-time checking can be used both to detect that an assertion is violated and to prove that an assertion holds for any valid query, i.e., the assertion is validated. The main problem with compile-time checking is that it requires the existence of suitable static analyses which are capable of proving the properties of interest. Our system is generic in that any program property (for which a suitable analysis exists in the system) can be used for debugging. Currently CiaoPP can infer types, modes and other variable instantiation properties, constraint independence, non-failure of predicates, determinacy, bounds on computational cost, bounds on sizes of terms in the program, and other properties. Thus, we have available a wide range of properties which are amenable to be compile-time checked and that will become useful later to define safety policies in the context of mobile code certification.

An additional attachment, D13.2, elaborates on a refined analysis for non-failure of goals. This deliverable is an improved version of the corresponding one included in the first period report. Non-failure analysis aims at inferring that predicate calls in a program will never fail. This type of information has many applications in functional/logic programming. Clearly, non-failure is a property which is very useful for detecting program errors. It is moreover essential for estimating other complex properties such as lower bounds on the computational costs of goals, which can be used in turn to attest that the execution of a piece of code is not only safe but it is also efficient. Essentially, this is done by enhancing the code with cost certificates which attest

that the execution of the code will not take more than a given amount of time (or that it will not consume more than a given amount of memory). The non-failure property is also used for granularity control of parallel/distributed tasks, as we discuss in the last deliverable. It is also useful in the context of program parallelization, and instrumental in partial evaluation and other program transformations, and has also been used in query optimization. In this deliverable, we re-cast the non-failure analysis proposed by Debray et al. as an abstract interpretation, which not only allows to investigate it from a standard and well understood theoretical framework, but has also several practical advantages. It allows us to incorporate non-failure analysis into a standard, generic abstract interpretation engine. The analysis thus benefits from the fixpoint propagation algorithm, which leads to improved information propagation. Also, the analysis takes advantage of the multi-variance of the generic engine, so that it is now able to infer separate non-failure information for different call patterns. Moreover, the implementation is simpler, and allows to perform non-failure and covering analyses alongside other analyses, such as those for modes and types, in the same framework. Finally, besides the precision improvements and the additional simplicity, our implementation (in the Ciao/CiaoPP multiparadigm programming system) also shows better efficiency. We believe that more precise inference of this property helps improving the verification and safety tasks.

In the third part, D13.3, we address the issue of safety of mobile code and discuss some applications in pervasive systems. When developing software for deployment on Smart Cards (and similar ambient computing devices), several issues related to safety arise: 1) Pervasive computing is characterized by having a relatively large number of *untrusted* computing devices which interact with environment by means of sensors, with other devices or, possibly, with the user. Thus, when modeling such a system, it is not realistic to consider one device in isolation: it will receive plenty of mobile data from the environment. In this context, the *safety* of the deployed software is crucial, as the cost of recalling unfit devices can be prohibitive. 2) It is essential to simplify the (safety) verification process and reduce its resource usage. Indeed, Smart Cards typically provide less than 4Kb of RAM while it is possible to use only up to 128Kb for storing the application and static data. Such resource considerations tend to dominate the development process for pervasive systems, forcing developers to write low-level code from scratch, as mobile system developers have found in their own experience. *Proof-Carrying Code* (PCC) is a general approach to mobile code safety in which programs are augmented 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. 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 the previous period we proposed Abstraction-Carrying Code (ACC), a novel approach which uses abstract interpretation as enabling technology. ACC, like recent approaches to mobile code safety, such as PCC, involves associating safety information to programs. Essentially, ACC is a framework for ensuring the safety of mobile code in which programs are augmented with *abstractions* as certificates (or proofs of safety). The *abstraction* (or abstract model) of the program is computed by standard static analyzers. The safety policy is specified by using an expressive assertion language defined over abstract domains. In particular, we rely on an expressive class of safety properties which have been discussed in the two former deliverables. We have developed a framework for checking the validity of these assertions based on the use of abstract interpretation analysis techniques which infer properties of programs (i.e. abstract approximations) and compare them with assertions using the abstract comparison operators already defined in the abstract domains. The validity of the abstraction on the consumer side is checked in a single-pass by a specialized abstract-interpreter. We argue that the large body of applications of abstract interpretation to program verification (see D13.1) is amenable to the overall PCC scheme. We believe that ACC is of interest for bringing the automation and expressiveness which is inherent in the abstract interpretation techniques to the area of mobile code safety.

In this period we have extended ACC along two lines:

- *Implementation and evaluation*: although global analysis is routinely used as a practical tool, it is in general unacceptable (and specifically in a pervasive environment) to run the whole analyzer to validate the certificate since it involves considerable cost. We have designed a very efficient, highly specialized abstract interpretation-based checking algorithm. While our approach is general, we have developed it for concreteness in the context of constraint logic programming. We have implemented and benchmarked ACC within the `Ciao` system preprocessor. The algorithm has been integrated within `CiaoPP`. With this, we have proceeded to benchmark and experimentally evaluate a practical incarnation of the ACC approach in the integrated tool. Our experimental results show that the checking phase is indeed faster than the proof generation phase, and that the sizes of certificates are reasonable.

- *Resource-aware ACC*: In addition to having some assurance of the correctness and safety characteristics of the code received, in a pervasive computing platform an essential issue is to also have some assurance of the kind of load the particular code is going to pose. We have proposed a method whereby this can be specified by means of *cost certificates*. To this end, we use the assertion language available in `CiaoPP` which allows specifying complex programs properties (including safety and resource-related properties). A receiver

can now reject code that does not adhere to a particular safety policy involving *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). We rely on the compile-time tools described above (and available in the integrated tool) for the certification of programs with resource consumption assurances and the efficient checking of such certificates. We have implemented this type of resource-aware ACC on the `Ciao` system and the integrated tool. Essentially, the fact that abstract interpretation techniques allow inferring very rich information will allow us to generate certificates which specify complex program properties including traditional safety issues but also *resource*-related properties.

In the last part, D13.4, we present a unified abstract interpretation-based approach to resource-aware distributed and mobile computing and discuss its implementation in the context of a multi-paradigm programming system. Distributed parallel execution systems speed up applications by splitting tasks into processes whose execution is assigned to different receiving nodes in a high-bandwidth network. On the distributing side, a fundamental problem is grouping and scheduling such tasks such that each one involves sufficient computational cost when compared to the task creation and communication costs and other such practical overheads. For this purpose, we show that some of the properties discussed in deliverables D13.1 and D13.2, especially lower bounds on cost and upper bounds on data sizes, can be used to perform high-level optimizations such as resource-aware task granularity control. On the receiving side, an important issue is to 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, which can be specified by means of *certificates* using ACC as discussed in deliverable 13.3. In essence, in this deliverable, we show that our proposals can contribute to bringing increased flexibility, expressiveness and automation of important resource-awareness aspects in the area of mobile and distributed computing. The deliverable presents in a tutorial way a number of general solutions to these problems, and illustrate them through their implementation in the `Ciao` program development environment. In particular, we focus on the facilities for parallel and distributed execution of `Ciao`, on its assertion language for specifying complex programs properties (including safety and resource-related properties), its compile-time and run-time tools for performing automated parallelization and resource control, as well as on the certification of programs with resource consumption assurances and efficient checking of such certificates.

Attachments:

Part I (D13.1) — Abstract Verification and Debugging of Constraint Logic Programs (published at *Recent Advances in Constraints*, number 2627 in LNCS, pages 1–14. Springer-Verlag, January 2003).

Part II (D13.2) — Multivariant Non-Failure Analysis via Standard Abstract Interpretation (published at *Proc. of 7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, LNCS, Num. 2998, pages 100-116, Springer-Verlag, April 2004).

Part III (D13.3) — Abstraction-Carrying Code (published at *Proc. of 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, LNAI, Num. 3452, pages 380-397, Springer-Verlag, March 2005).

Part IV (D13.4) — Some Techniques for Automated, Resource-Aware Distributed and Mobile Computing in a Multi-Paradigm Programming System (published at *Proc. of EURO-PAR 2004*, LNCS, Num. 3149, pages 21-37, Springer-Verlag, August 2004).

Contents

Part I

Abstract Verification and Debugging of Constraint Logic Programs

1 Background

The technique of Abstract Interpretation [CC77] has allowed the development of sophisticated program analyses which are provably correct and practical. The semantic approximations produced by such analyses have been traditionally applied to *optimization* during program compilation. However, recently, novel and promising applications of semantic approximations have been proposed in the more general context of program *verification* and *debugging* [Bou93, CLMV96, BDD+97].

In the case of Constraint Logic Programs (CLP), a comparatively large body of approximation domains, inference techniques, and tools for abstract interpretation-based semantic analysis have been developed to a powerful and mature level (see, e.g., [MH92, CV94, GdW94, BCHP96, dlBHB+96, HBPLG99] and their references). These systems can approximate at compile-time a wide range of properties, from directional types to variable independence, determinacy or termination, always safely, and with a significant degree of precision.

Our proposed approach takes advantage, within the context of program verification and debugging, of these significant advances in static program analysis techniques and the resulting concrete tools, which have been shown useful for other purposes such as optimization, and are thus likely to be present in compilers. This is in contrast to using traditional proof-based methods (e.g., for the case of CLP, [AM94, AP93, Der93, Fer87, Vet94]), developing new tools and procedures (such as specific concrete [BDM97, DNTM88, DNTM89] or abstract [CLMV96, CLMV99] diagnosers and declarative debuggers), or limiting error detection to run-time checking (e.g., [Vet94]).

2 An Approach Based on Semantic Approximations

We now briefly describe the basis of our approach [BDD+97, HPB99, PBH00c]. 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

Property	Definition
P is partially correct w.r.t. $\mathcal{I}$	$[\![P]\!] \subseteq \mathcal{I}$
P is complete w.r.t. $\mathcal{I}$	$\mathcal{I} \subseteq [\![P]\!]$
P is incorrect w.r.t. $\mathcal{I}$	$[\![P]\!] \not\subseteq \mathcal{I}$
P is incomplete w.r.t. $\mathcal{I}$	$\mathcal{I} \not\subseteq [\![P]\!]$

Table 1: Set theoretic formulation of verification problems

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., $[\![P]\!] = \mathrm{lfp}(S_P)$. 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. In Table 1 we define classical verification problems in a set-theoretic formulation as simple relations between $[\![P]\!]$ and $\mathcal{I}$.

Using the exact 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 and interesting approach is to approximate the semantics. This is interesting, among other reasons, because a well understood technique already exists, abstract interpretation, which provides *safe* approximations of the program semantics. Our first objective is to present the implications of the use of *approximations* of both the intended and actual semantics in the verification and debugging process.

2.1 Approximating Program Semantics

We start by recalling some basic concepts from abstract interpretation. In this technique, a program is interpreted over a non-standard domain called the *abstract* domain D_α which is simpler than the *concrete* domain D, 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.

The concrete and abstract domains are related via a pair of monotonic mappings: *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$, which relate the two domains by a Galois insertion (or a Galois connection) [CC77]. We will denote by $[\![P]\!]_\alpha$ the result of abstract interpretation

for a program P. 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 $[\![P]\!]_{\alpha^-}$.

2.2 Abstract Verification and Debugging

The key idea in our approach is to use the abstract approximation $[\![P]\!]_\alpha$ directly in verification and debugging tasks. As we will see, 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.

A number of approaches have already been proposed which make use to some extent of abstract interpretation in verification and/or debugging tasks. Abstractions were used in the context of algorithmic debugging in [LS88]. Abstract interpretation for debugging of imperative programs has been studied by Bourdoncle [Bou93], and for the particular case of algorithmic debugging of logic programs by Comini et al. [CLV95] (making use of partial specifications) and [CLMV96].

In our approach we actually compute the abstract approximation $[\![P]\!]_\alpha$ of the actual semantics of the program $[\![P]\!]$ and compare it directly to the (also approximate) intention (which is given in terms of *assertions* [PBH00b]), following almost directly the scheme of Table 1. This approach can be very attractive in programming systems 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.

For now, we assume that the program specification is given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program is most easily done in the same domain, since then the operators on the abstract lattice, that are typically already defined in the analyzer, can be used to perform this comparison. Thus, for comparison we need in principle $\alpha([\![P]\!])$. Using abstract interpretation, we can usually only compute instead $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$. Thus, it is interesting to study the implications of comparing $\mathcal{I}_\alpha$ and $[\![P]\!]_\alpha$.

In Table 2 we propose (sufficient) conditions for correctness and completeness w.r.t. $\mathcal{I}_\alpha$, which can be used when $[\![P]\!]$ is approximated. Several instrumental conclusions can be drawn from these relations.

Analyses which over-approximate the actual semantics (i.e., those denoted as $[\![P]\!]_{\alpha^+}$), are

Property	**Definition**	**Sufficient condition**
P is partially correct w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \subseteq [\![P]\!]_{\alpha^-}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \not\subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^-} \not\subseteq \mathcal{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \not\subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \not\subseteq [\![P]\!]_{\alpha^+}$

Table 2: Validation problems using approximations

specially suited for proving partial correctness and incompleteness with respect to the abstract specification $\mathcal{I}_\alpha$. It will also be sometimes possible to prove incorrectness in the extreme case in which the semantics inferred for the program is incompatible with the abstract specification, i.e., when $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset$. We also note that it will only be possible to prove completeness if the abstraction is *precise*, i.e., $[\![P]\!]_\alpha = \alpha([\![P]\!])$. According to Table 2 only $[\![P]\!]_{\alpha^-}$ can be used to this end, and in the case we are discussing $[\![P]\!]_{\alpha^+}$ holds. Thus, the only possibility is that the abstraction is precise.

On the other hand, if analysis under-approximates the actual semantics, i.e., in the case denoted $[\![P]\!]_{\alpha^-}$, it will be possible to prove completeness and incorrectness. In this case, partial correctness and incompleteness can only be proved if the analysis is precise.

If analysis information allows us to conclude that the program is incorrect or incomplete w.r.t. $\mathcal{I}_\alpha$, an (abstract) symptom has been found which ensures that the program does not satisfy the requirement. Thus, debugging should be initiated to locate the program construct responsible for the symptom.

More details about the theoretical foundation of our approach can be found in [BDD+97, PBH00c].

3 A Practical Framework and its Implementation

Using the ideas outlined above, we have developed a framework [HPB99, PBH00a] capable of combined static and dynamic validation, and debugging for CLP programs, using semantic approximations, and which can be integrated in an advanced program development environment comprising a variety of co-existing tools [DHM00].

This framework has been implemented as a generic preprocessor composed of several tools. Figure 1 depicts the overall architecture of the system. Hexagons represent the different tools involved and arrows indicate the communication paths among the different tools.

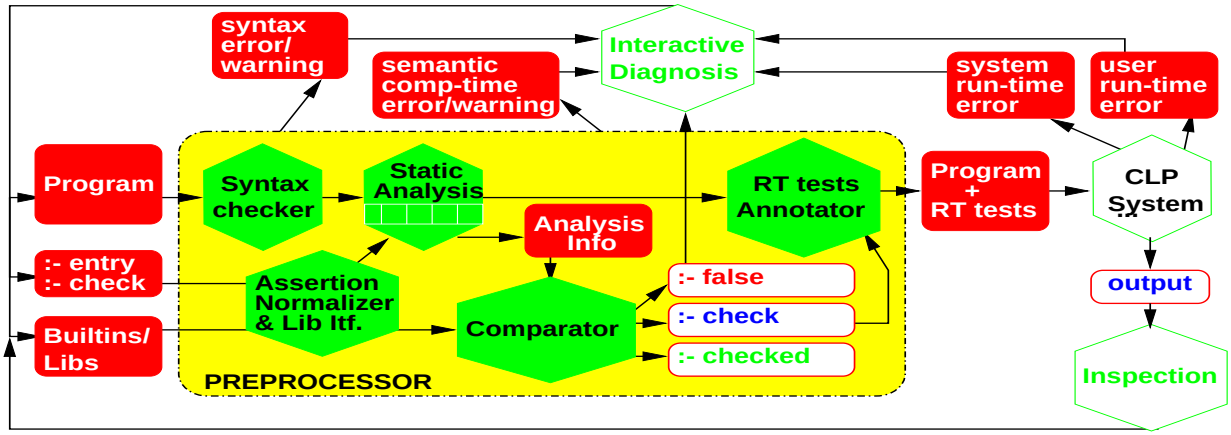

Figure 1: Architecture of the Preprocessor

Program verification and detection of errors is first performed at compile-time by using the sufficient conditions shown in Table 2. I.e., by inferring properties of the program via abstract interpretation-based static analysis and comparing this information against (partial) specifications written in terms of assertions. Such assertions are linguistic constructions which allow expressing properties of programs.

Classical examples of assertions are type declarations (e.g., in the context of (C)LP those used by [HL94, SHC96, BCC+02]). However, herein we are interested in supporting a much more powerful setting in which assertions can be of a much more general nature, stating additionally other properties, some of which cannot always be determined statically for all programs. These properties may include properties defined by means of user programs and extend beyond the predefined set which may be natively understandable by the available static analyzers. Also, in the proposed framework only a small number of (even zero) assertions may be present in the program, i.e., the assertions are *optional*. In general, we do not wish to limit the programming language or the language of assertions unnecessarily in order to make the validity of the assertions statically decidable (and, consequently, the proposed framework needs to deal throughout with *approximations*). We also propose a concrete language of assertions which allows writing this kind of (partial) specifications for CLP [PBH00b].

The assertion language is also used by the preprocessor to express both the information inferred by the analysis and the results of the comparisons performed against the specifications.[1] As can be derived from Table 2, these comparisons can result in proving statically (i.e., at compile-time) that the assertions hold (i.e., they are validated) or that they are violated, and thus bugs

[1] Interestingly, the assertions are also quite useful for generating documentation automatically (see [Her00]).

have been detected. User-provided assertions (or *parts* of assertions) which cannot be statically proved nor disproved are optionally translated into run-time tests. Both the static and the dynamic checking are provably *safe* in the sense that all errors flagged are definite violations of the specifications.

The practical usefulness of the framework is illustrated by what is arguably the first and most complete implementation of these ideas: CiaoPP,[2] the Ciao system preprocessor [PBH00a, HBPLG99]. Ciao is a public-domain, next-generation constraint logic programming system, which supports ISO-Prolog, but also, selectively for each module, extensions and restrictions such as, for example, pure logic programming, constraints, functions, objects, or higher-order. Ciao is specifically designed to a) be highly extensible and b) support modular program analysis, debugging, and optimization. The latter tasks are performed in an integrated fashion by CiaoPP.

CiaoPP, which incorporates analyses developed by several groups in the LP and CLP communities, uses abstract interpretation to infer properties of program predicates and literals, including types, modes and other variable instantiation properties, constraint independence, non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc. It processes modules separately, performing incremental analysis. CiaoPP can find errors at compile-time (or perform partial verification) by checking how programs call system libraries. This is possible since the expected behaviour of system predicates is also given in terms of assertions This allows detecting errors in user programs even if they contain no assertions. Also, the preprocessor can detect errors as well by checking the assertions present in the program or in other modules used by the program. As already mentioned, assertions are completely optional. Nevertheless, if the program is not correct, the more assertions are present in the program the more likely it is for errors to be automatically detected. Thus, for those parts of the program which are potentially buggy or for parts whose correctness is crucial, the programmer may decide to invest more time in writing assertions than for other parts of the program which are more stable. In addition, CiaoPP also performs program transformations and optimizations such as multiple abstract specialization, parallelization (including granularity control), and inclusion of run-time tests for assertions which cannot be checked completely at compile-time.

Finally, the implementation of the preprocessor is generic in that it can be easily customized to different CLP systems and dialects and in that it is designed to allow the integration of additional analyses in a simple way. As a particularly interesting example, the preprocessor has been adapted for use with the CHIP CLP(FD) system. This has resulted in CHIPRE, a preprocessor for CHIP which has been shown to detect non-trivial programming errors in CHIP programs. In the next section we show an example of a debugging session with CHIPRE. More information

[2] A demonstration of the system was performed at the meeting.

on the system can be found in [PBH00a].

4 A Sample Debugging Session with CHIPRE

In this section we will show some of the capabilities of our debugging framework through a sample session with CHIPRE, an implemented instance of the framework. Consider Figure 2, which contains a tentative version of a CHIP program for solving the *ship* scheduling problem, a typical CLP(FD) benchmark.

Often, the results of static analysis are good indicators of bugs, even if no assertion is given. This is because "strange" results often correspond to bugs. An important observation is that plenty of static analyses, such as modes and regular types, compute over-approximations of the success sets of predicates. Then, if such an over-approximation corresponds to the empty set then this implies that such predicate never succeeds. Thus, unless the predicate is dead-code, this often indicates that the code for the predicate is erroneous since every call either fails finitely (or raises an error) or loops. If analysis is goal-dependent and thus also computes an over-approximation of the calling states to the predicate, predicates which are dead-code can often be identified by having an over-approximation of the calling states which corresponds to the empty set.

We now preprocess the current version of our example program using *regular type* [YS87, DZ92, GdW94, GP02, VB02] analysis. Our implementation of regular types is goal-dependent and thus computes over-approximations of both the success set and calling states of all predicates. In addition, our analysis also computes over-approximations of the values of variables at each program point. Once analysis information is available, the preprocessor automatically checks the consistency of the analysis results and we get the following messages:

```
WARNING: Literal set_precedences(L, Sis, Dis)
           at solve/7/1/5 does not succeed!
WARNING: Literal set_pre_lp(l, array_starts, Array_duration)
           at set_precedences/3/1/4 does not succeed!
```

The first warning message refers to a literal (in particular, the 5th literal in the 1st clause of `solve/7`) which calls the predicate `set_precedences/3`, whose success type is empty. Also, even if the success type of a predicate is not empty, i.e., there may be some calls which succeed, it may be possible to detect that at a certain program point the given call to the predicate cannot succeed because the type of the particular call is incompatible with the success type of the predicate. This is the reason for the second warning message. Note that this kind of reasoning can only be made if (1) the static analysis used infers properties which are *downwards closed*,

```
solve(Upper,Last,N,Dis,Mis,L,Sis):-
        length(Sis, N),
        Sis :: 0..Last,
        Limit :: 0..Upper,
        End :: 0..Last,
        set_precedences(L, Sis, Dis),
        cumulative(Sis, Dis, Mis, unused, unsed, Limit, End, unused),
        min_max(labeling(Sis), End).

labeling([]).
labeling([H|T]):-
        delete(X,[H|T],R,0,most_constrained),
        indomain(X),
        labeling(R).

set_precedences(L, Sis, Dis):-
        Array_starts=..[starts|Sis], % starts(S1,S2,S3,...)
        Array_durations=..[durations|Dis], % durations(D1,D2,D3,...)
        initialize_prec(L,Array_starts),
        set_pre_lp(l, array_starts, Array_durations).

set_pre_lp([], _, _).
set_pre_lp([After#>=Before|R], Array_starts, Array_durations):-
        arg(After, Array_starts, S2),
        arg(Before, Array_starts, S1),
        arg(Before, Array_durations, D1),
        S2 #>= S1 + D1,
        set_pre_lp(R, Array_starts, array_durations).

initialize_prec(_,_).
```

Figure 2: A tentative *ship* program in CHIP

i.e., once they hold they keep on being valid during forward execution and (2) analysis computes descriptions at each program point which, as already mentioned, is the case with our regular type analysis. Note that the predicate `set_pre_lp/3` can only succeed if the value at the first argu-

ment is compatible with a list. However, the call `set_pre_lp(l, array_starts, Array_duration)` has the constant `l` at the first argument position. This is actually a bug, as the constant `l` should instead be the variable `L`. Once we correct this bug, in subsequent preprocessing of the program both warning messages disappear. In fact, the first one was also a consequence of the same bug which propagated to the calling predicates of `set_precedences/3`.

4.1 Aiding the Analyzer

In the ship program, all initial queries to the program are intended to be to the `solve` predicate. However, the compiler has no way to automatically determine this. Thus, in the absence of more information, the most general possible calls have to be assumed for *all* predicates in the program.[3] One way to alleviate this is to provide *entry* assertion(s) which are assumed to cover all possible initial calls to the program. Even the simplest entry declaration which can be given for predicate `solve`, i.e., '`:- entry solve/7.`', is very useful for goal-dependent static analysis. Since it is the only `entry` assertion, the only calls to the rest of the predicates in the program are those generated during computations of `solve/7`. This allows analysis to start from the predicate `solve/7` only, instead of from all predicates. Reducing the number (and generality) of starting points for goal-dependent analysis by means of `entry` declarations often leads to increased precision and reduced analysis times. However, analysis will still make no assumptions regarding the arguments of the calls to `solve/7` since there is no further information available. This could be improved using a more accurate entry declaration such as the following:

```
:- entry solve/7 : int * int * int * list(int) * list(int) * list * term.
```

It gives the types of the seven arguments, and describes more precisely the valid input data. Note that the assertion above also specifies a *mode* for the calling patterns. The first three arguments are *required* to be instantiated to integers. The forth and fifth must be fully instantiated to lists of integers. The sixth argument is (only) required to be instantiated to a list skeleton. Finally, the seventh argument can be any possible term. Note that, by default, our assertion language interprets properties in assertions as *instantiation* properties. However, the assertion language also allows the use of *compatibility* properties if so desired [PBH00b].

4.2 Assertions for System Predicates

Consider a new version of the *ship* program, after correcting the typo involving `L` and introducing the (simple) entry declaration '`:- entry solve/7.`'. When preprocessing the program the

[3] Note that this can be partly alleviated with a strict module system such as that of Ciao [CH00], in which only exported predicates of a module can be subject to initial queries.

following messages are issued:

```
ERROR: Builtin predicate
        cumulative(Sis,Dis,Mis,unused,unsed,Limit,End,unused)
        at solve/7/1/6 is not called as expected (argument 5):
         Called:    ^unsed
         Expected:  intlist_or_unused

ERROR: Builtin predicate arg(After,Array_starts,S2)
        at set_pre_lp/3/2/1 is not called as expected (argument 2):
         Called:    ^array_starts
         Expected:  struct
```

Which indicate that the program is still definitely incorrect. Note that the preprocessor could not detect this without the extra precision allowed by the `entry` assertion. In error messages involving regular types, one important issue is not to confuse term constructors with type constructors. In order to improve the readability and conciseness of the error messages, the marker `^` is used to distinguish terms (constants) from regular types (which represent regular sets of terms). By default, values represent regular types. However, if they are marked with `^` they represent constants. In our example, `intlist_or_unused` is a type since it is not marked with `^` whereas `^unsed` is a constant. Note that though it is always possible to define a regular type which contains a single constant such as `unsed` and distinguish terms from types by the context in which the value appears, we opt by introducing the marker `^` ("quote") since in our experience this improves readability of error messages. Note that defining such type explicitly instead would require inventing a new name for it and providing the definition of the type together with the error message.

Coming back to the pending error messages, the first message is due to the fact that the constant `unused` has been mistakingly typed as `unsed` in the fifth argument of the call to the CHIP builtin predicate `cumulative/8`. As indicated in the error message, this predicate requires the fifth argument to be of type `intlist_or_unused` which was defined when writing assertions for the system predicates in CHIP and which indicates that such argument must be either the constant `unused` or a list of integers.

The automatic detection of this error at compile-time has been possible because the CHIP builtins have been provided with assertions that describe their intended use. Though system predicates are in principle considered correct under the assumption that they are called with valid input data, it is often useful to check that they are indeed called with valid input data. In fact, existing CLP systems perform this checking at run-time. The existence of such assertions

allows checking the calls to system predicates at compile-time in addition to run-time in CLP systems which originally do not perform compile-time checking.

In the second message we have detected that we call the CHIP builtin predicate `arg/3` with the second argument bound to `array_starts` which is a constant (as indicated by the marker `^`) and thus of arity zero. This is incompatible with the expected call type `struct`, i.e., a structure with arity strictly greater than zero. In the current version of CHIP, this will generate a run-time error, whereas in other systems such as Ciao and SICStus, this call would fail but would not raise an error. Though we know the program is incorrect, the literal where the error is flagged, `arg(After, Array_starts, S2)` is apparently correct. We correct the first error and leave detection of the cause for the second error for later.

The different behaviour of seemingly identical builtin predicates (such as `arg/3` in the example above) in different systems further emphasizes the benefits of describing builtin predicates by means of assertions. They can be used for easily customizing static analysis for different systems, as assertions are easier to understand by naive users of the analysis than the hardwired internal representation used in ad-hoc analyzers for a particular system.

4.3 Assertions for User-Defined Predicates

Up to now we have seen that the preprocessor is capable of issuing a good number of error and warning messages even if the user does not provide any `check` assertions (assertions that the system should check to hold). We believe that this is very important in practice. However, adding assertions to programs can be useful for several reasons. One is that they allow further semantic checking of the programs, since the assertions provided encode a partial specification of the user's intention, against which the analysis results can be compared. Another one is that they also allow a form of diagnosis of the error symptoms detected, so that in some cases it is possible to automatically locate the program construct responsible for the error.

Consider again the pending error message from the previous iteration over the ship program. We know that the program is incorrect because (global) type analysis tells us that the variable `Array_starts` will be bound at run-time to the constant `array_starts`. However, by just looking at the definition of predicate `set_pre_lp` it is not clear where this constant comes from. This is because the cause of this problem is not in the definition of `set_pre_lp` but rather in that the predicate is being used incorrectly (i.e., its precondition is violated). We thus introduce the following `calls` assertion, which describes the expected calls to the predicate:

```
:- calls set_pre_lp(A,B,C): (struct(B),struct(C)).
```

In this assertion we require that both the second and third parameters of the predicate, i.e., `B`

and `C` are structures with arity greater than zero, since in the program we are going to access the arguments in the structure of `B` and `C` with the builtin predicate `arg/3`.

The next time our ship program is preprocessed, having added the `calls` assertion, besides the pending error message of above regarding `arg/3`, we also get the following one:

```
ERROR: false assertion at set_precedences/3/1/4
       unexpected call (argument 2):
         Called:   ^array_starts
         Expected: struct
```

This message tells us the exact location of the bug, the fourth literal of the first clause for predicate `set_precedences/3`. This is because we have typed the constant `array_starts` instead of the variable `Array_starts` in such literal.

Thus, as shown in the example above, user-provided `check` assertions may help in locating the actual cause for an error. Also, as already mentioned, and maybe more obvious, user-provided assertions may allow detecting errors which are not easy to detect automatically otherwise.

After correcting the bug located in the previous example, preprocessing the program once again produces the following error message:

```
ERROR: false assertion at set_pre_lp/3/2/5
       unexpected call (argument 3):
         Called:   ^array_durations
         Expected: struct
```

which would not be automatically detected by the preprocessor without user-provided assertions. The obvious correction is to replace `array_durations` in the recursive call to `set_pre_lp` in its second clause with `Array_durations`.

After correcting this bug, preprocessing the program with the given assertions does not generate any more messages. Besides, the user provided `calls` assertion would have been proved by analysis.

Additionally, if some part of an assertion for a user-defined predicate has not been proved nor disproved during compile-time checking, it can be checked at run-time in the classical way, i.e., run-time tests are added to the program which encode in some way the given assertions. Introducing run-time tests by hand into a program is a tedious task and may introduce additional bugs in the program. In the preprocessor, this is performed automatically upon user's request.

Compile-time checking of assertions is conceptually more powerful than run-time checking. However, it is also more complex. Since the results of compile-time checking are valid for *any*

query which satisfies the existing `entry` declarations, compile-time checking can be used both to detect that an assertion is violated and to prove that an assertion holds for any valid query, i.e., the assertion is validated. The main problem with compile-time checking is that it requires the existence of suitable static analyses which are capable of proving the properties of interest. For conciseness, we have shown the possibilities of our system using only a (regular) type analysis. However, the system is generic in that any program property (for which a suitable analysis exists in the system) can be used for debugging. As mentioned before, currently CiaoPP can infer types, modes and other variable instantiation properties, constraint independence, non-failure of predicates, determinacy, bounds on computational cost, bounds on sizes of terms in the program, and other properties.

More info: For more information, full versions of selected papers and technical reports, and/or to download Ciao and other related systems please access `http://www.clip.dia.fi.upm.es/`.

Part II

Multivariant Non-Failure Analysis via Standard Abstract Interpretation

5 Introduction

Non-failure analysis involves detecting at compile time that, for any call belonging to a particular (possibly infinite) class of calls, a predicate will never fail. As an example, consider a predicate defined by the following two clauses:

```
abs(X, Y) :- X >= 0, Y is X.
abs(X, Y) :- X < 0, Y is -X.
```

and assume that we know that this predicate will always be called with its first argument bound to an integer, and the second argument a free variable. Obviously, for any particular call, one or the other of the tests `X >= 0` and `X < 0` may fail; however, taken together, one of them will always succeed. Thus, we can infer that calls to the predicate will never fail.

Being able to determine statically that a predicate will not fail has many applications. It is essential for determining lower bounds on the computational cost of goals since without such information a lower bound of almost zero (corresponding to an early failure) must often be assumed [DLGHL97]. Detecting non-failure is also very useful in the context of parallelism because it allows avoiding unnecessary speculative parallelism and ensuring no-slowdown properties for the parallelized programs (in addition to using the lower bounds mentioned previously to perform granularity control) [GPA+01]. Non-failure information is also instrumental in partial evaluation and other program transformations, such as reordering of calls, and has also been used in query optimization in deductive databases [DL90]. It is also useful in program debugging, where it allows verifying user assertions regarding non-failure of predicates [HBPLG99, HPBLG03]. Finally, similar techniques can be used to detect the absence of errors or exceptions when running particular predicates.

A practical non-failure analysis has been proposed by Debray *et al.* [DLGH97]. In a similar way to the example above, this approach relies on first inferring mode and type information, and then testing that the constraints in the clauses of the predicate are entailed by the types of the input arguments, which is called a *covering* test. Covering cannot be inferred by examining the constraints of each clause separately: it is necessary to collect them together and examine the behavior of the predicate as a whole. Furthermore, non-failure of a given predicate depends on

non-failure of other predicates being called and also possibly on the constraints in such predicates.

While [DLGH97] proposed the basic ideas behind non-failure analysis, only a simple, monovariant algorithm was proposed for propagating the non-failure information. In our experience since that proposal, we have found a need to improve it in several ways. First, information propagation needs to be improved, which leads us to a fixpoint propagation algorithm. Furthermore, the analysis really needs to be *multi–variant*, which means that it should be able to infer separate non-failure (and covering) information for different call patterns for a given predicate in a program. This is illustrated by the following example which, although simple, captures the very common case where the same (library) procedure is called from a program (in different points) for different purposes:

Example 1 Consider the (exported) predicate `mv`/3 (which uses the library predicate `qsort`/2), defined for the sake of discussion as follows:

```
mv(A,B,C):- qsort(A,B), !, C = B.
mv(A,B,C):- append(A,B,D), qsort(D, C).
```

Assume the following entry assertion for `mv`/3:

```
:- entry mv(A,B,C) : (list(A, num), list(B, num), var(C)).
```

which means that the predicate `mv(A,B,C)` will be called with `A` and `B` bound to lists of numbers, and `C` a free variable. A multi–variant non-failure analysis would infer two call patterns for predicate `qsort`/2:

1. The call pattern `qsort(A,B): (list(A,num), list(B,num))`, for which the analysis infers that it *can fail* and is *not covered*, and
2. the call pattern `qsort(A,B): (list(A,num), var(B))`, for which the analysis infers that it will *not fail* and *is covered*.

This in turn allows the analysis to infer that the predicate `mv`/3 will *not fail* and *is covered* (for the call pattern expressed by the entry assertion).

However, a monovariant analysis only considers one call pattern per predicate. In particular, for predicate `qsort`/2, the call pattern used is `qsort(A,B): (list(A,num), term(B))`[4] (which is the result of "collapsing" all call patterns which can appear in the program, so that precision is lost), for which it infers that `qsort`/2 *can fail* and *is not covered*. This causes the analysis to infer that the predicate `mv`/3 *can fail* (since the calls to `qsort`/2 in both clauses of predicate `mv`/3 are detected as failing) and *is covered*. □

[4] `term(B)` means that argument `B` can be bound to any term.

In order to address the different shortcomings of [DLGH97] in this paper we start by casting the ideas behind non-failure and covering analysis as an abstract interpretation [CC77]. This then allows us to incorporate non-failure analysis into a (somewhat modified) standard, generic abstract interpretation engine. This has several advantages. First of all, the analysis is now based on a standard and well studied theoretical framework. But, most importantly, being able to take advantage of standard and well developed analysis engines allows us to obtain a simpler and more efficient implementation, with better propagation of information, performing an efficient fixpoint. The non-failure and covering analyses can be performed alongside other abstract interpretation based analyses, such as those for modes and types, in the same framework. Furthermore, the analysis that we obtain is *multi–variant* (on calls and successes) thus inferring separate non-failure (and covering) information for different call patterns for a given predicate in a program. Finally, the abstract domain for non-failure can be easily enhanced to define a domain for determinacy of predicates.

Abstract Interpretation [CC77] is often proposed as a means for inferring properties of programs at compile–time. It was shown by Bruynooghe [Bru87], Jones and Sondergaard [JS87], Debray [DW88], and Mellish [Mel86] that this technique can be extended to flow analysis of programs in logic programming languages, and several frameworks or particular analyses have evolved since (e.g. [MU87, ST84, Wae88, WHD88]). Abstract interpretation formalizes the relation between analysis and semantics, and, therefore, it is inherently semantics sensitive, different semantic definition styles yielding different approaches to program analysis. For logic programs we distinguish between two main approaches, namely *bottom–up* analysis and *top–down* analysis. We also distinguish between *goal dependent* and *goal independent* analyses. In this paper we use a goal dependent framework, since non-failure analysis is inherently goal dependent. In [Bru91], Bruynooghe describes a framework for the goal-dependent, top–down abstract interpretation of logic programs. We use the PLAI/CiaoPP framework [HBPLG99, HPBLG03], which follows [Bru91], but incorporates a number of optimizations and efficient fixpoint algorithms, described in [MH90, MH92, HPMS00].

6 Preliminaries

We will denote $\mathcal{C}$ the universal set of constraints. We let $\theta \downarrow_L$ be the constraint θ restricted to the variables of the syntactic object L. We denote constraint entailment by $\models$, so that $c_1 \models c_2$ denotes that c_1 entails c_2.

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 *constraint logic program*, or *program*, is a finite set of rules. The *definition* of an atom A in program P, $defn_P(A)$, is the set of variable renamings of rules in P such that each renaming has A as a head and has distinct new local (but not head) variables.

The operational semantics of a program is in terms of its "derivations" which are sequences of reductions between "states". A *state* $\langle G \mid \theta \rangle$ consists of a goal G and a constraint store (or *store* for short) θ. A state $\langle L :: G \mid \theta \rangle$, where L is a literal and :: denotes concatenation of sequences, can be *reduced* as follows:

1. If L is a constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle G \mid \theta \wedge L \rangle$.
2. If L is an atom, it is reduced to $\langle B :: G \mid \theta \rangle$ for some rule $(L$:- $B) \in defn_P(L)$.

assuming for simplicity that the underlying constraint solver is complete. We use $S \rightsquigarrow_P S'$ to indicate that in program P a reduction can be applied to state S to obtain state S'. Also, $S \rightsquigarrow_P^* S'$ indicates that there is a sequence of reduction steps from state S to state S'. A *derivation* from state S for program P is a sequence of states $S_0 \rightsquigarrow_P S_1 \rightsquigarrow_P ... \rightsquigarrow_P S_n$ where S_0 is S and there is a reduction from each S_i to S_{i+1}. Given a non-empty derivation D, we denote by *curr_goal(D)* and *curr_store*(D) the first goal and the store in the last state of D, respectively. E.g., if D is the derivation $S_0 \rightsquigarrow_P^* S_n$ with $S_n = \langle g :: G \mid \theta \rangle$ then $curr\_goal(D) = g$ and $curr\_store(D) = \theta$. A *query* is a pair (L, θ) where L is a literal and θ a store of an initial state $\langle L \mid \theta \rangle$. The set of all derivations from Q for P is denoted $derivations(P, Q)$. We will denote sets of queries by $\mathcal{Q}$. We extend $derivations$ to $\mathcal{Q}$ as follows: $derivations(P, \mathcal{Q}) = \bigcup_{Q \in \mathcal{Q}} derivations(P, Q)$.

The observational behavior of a program is given by its "answers" to queries. A finite derivation from a query (L, θ) for program P is *finished* if the last state in the derivation cannot be reduced. A finished derivation from a query (L, θ) is *successful* if the last state is of the form $\langle nil \mid \theta' \rangle$, where nil denotes the empty sequence. The constraint $\theta' \downarrow_L$ is an *answer* to (L, θ). We denote by $answers(P, Q)$ the set of answers to query Q. A finished derivation is *failed* if the last state is not of the form $\langle nil \mid \theta \rangle$. Note that $derivations(P, \mathcal{Q})$ contains not only finished derivations but also all intermediate derivations. A query Q *finitely fails* in P if $derivations(P, Q)$ is finite and contains no successful derivation.

Abstract Interpretation. Abstract interpretation [CC77] is a technique for static program analysis in which execution of the program is simulated on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). For this study, we restrict 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* $\alpha : 2^D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to 2^D$, 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 defined so that the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in a precise sense:

$$\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$$
$$\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcup \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cup \gamma(\lambda_2) = \gamma(\lambda')$$
$$\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcap \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cap \gamma(\lambda_2) = \gamma(\lambda')$$

Goal dependent abstract interpretation takes as input a program P, an abstract domain D_α, and a description $\mathcal{Q}_\alpha$ of the possible initial queries to P, given as a set of abstract queries. An *abstract query* is a pair (L, λ), where L is an atom (for one of the exported predicates) and $\lambda \in D_\alpha$ describes the initial stores for L. A set $\mathcal{Q}_\alpha$ represents the set of queries $\gamma(\mathcal{Q}_\alpha)$, which is defined as $\gamma(\mathcal{Q}_\alpha) = \{(L, \theta) \mid (L, \lambda) \in \mathcal{Q}_\alpha \wedge \theta \in \gamma(\lambda)\}$. Such an abstract interpretation computes a set of triples $Analysis(P, \mathcal{Q}_\alpha, D_\alpha) = \{\langle L_p, \lambda^c, \lambda^s \rangle \mid p \text{ is a predicate of } P\}$, where L_p is a (program) atom for predicate p. Note that, the analysis being multivariant (on calls), it may compute several tuples of the form $\langle L_p, \lambda^c, \lambda^s \rangle$ for different call patterns $\langle L_p, \lambda^c \rangle$ of each predicate p (including different program atoms L_p). If p is detected to be dead code then $\lambda^c = \lambda^s = \bot$. As usual in abstract interpretation, $\bot$ denotes the abstract constraint such that $\gamma(\bot) = \emptyset$, whereas $\top$ denotes the most general abstract constraint, i.e., $\gamma(\top) = D$.

7 The Abstract Interpretation Framework

PLAI is an analysis system based on the abstract interpretation framework of Bruynooghe [Bru91] with the optimizations described in [MH90]. The framework works on an abstraction of the (SLD) AND-OR trees of the execution of a program for given entry points. The abstract AND-OR graph makes it possible to provide information at each program point, a feature which is crucial for many applications (such as, e.g., reordering, automatic parallelization, or garbage collection).

Program points and abstract substitutions are related as follows. Consider a clause h `:-` $p_1, \ldots, p_n$. Let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal p_i, $1 \leq i \leq n$ in this clause. Then λ_i and λ_{i+1} are, respectively, the *abstract call substitution* and the *abstract success substitution* for the subgoal p_i. For this same clause, λ_1 is the *abstract entry substitution* and λ_{n+1} is the *abstract exit substitution*. Entry and exit substitutions are denoted respectively β_{entry} and β_{exit} when projected on the variables of the clause head.

Computing the *success* substitution from the *call* substitution is done as follows (see Figure 3(a)). Given a call substitution λ_{call} for a subgoal p, let $h_1, \ldots, h_m$ be the heads of clauses which unify with p. Compute the entry substitutions $\beta 1_{entry}, \ldots, \beta m_{entry}$ for these clauses. Compute their exit substitutions $\beta 1_{exit}, \ldots, \beta m_{exit}$ as explained below. Compute the success substitutions $\lambda 1_{success}, \ldots, \lambda m_{success}$ from the corresponding exit substitutions. At this point,

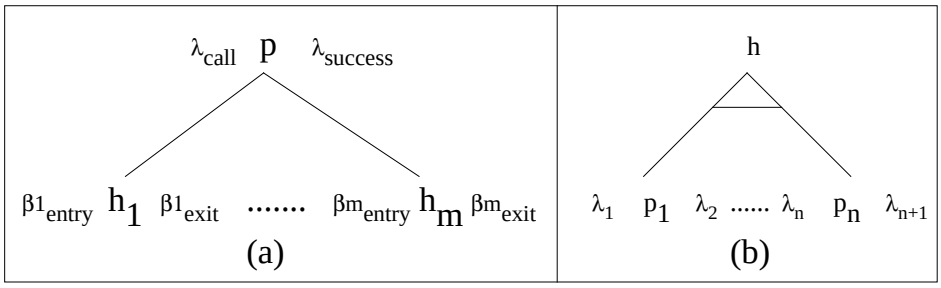

Figure 3: Illustration of the Top–Down Abstract Interpretation Process

all different success substitutions can be considered for the rest of the analysis, or a single success substitution $\lambda_{success}$ for subgoal p computed by means of an aggregation operation for $\lambda 1_{success}, \ldots, \lambda m_{success}$. This aggregator is usually the LUB (least upper bound) of the abstract domain. In the first case the analysis is *multi-variant* on successes, in the second case it is not.

Computing the *exit* substitution from the *entry* substitution is straightforward (see Figure 3(b)). Given a clause h :- $p_1, \ldots, p_n$ and an entry substitution β_{entry} for the clause head h, λ_1 is the call substitution for p_1. This one is computed simply by adding to β_{entry} an abstraction for the free variables in the clause. The success substitution λ_2 for p_1 is computed as explained above (essentially, by repeating this same process for the clauses which match p_1). Similarly, $\lambda_3, \ldots, \lambda_{n+1}$ are computed. The exit substitution B_{exit} for this clause is precisely the projection onto h of λ_{n+1}.

If, from a different subgoal in the program, a different entry substitution is computed for an already analyzed clause, different call substitutions will appear (for p_1 and possibly the other subgoals). These substitutions can be collapsed using the LUB operation, or a different node in the graph can be computed. In the latter solution, different nodes exist in the graph for each call substitution and subgoal, thus yielding an analysis which is *multi–variant* on calls.

Note that the framework itself is domain independent. To instantiate it, a particular analysis needs to define an abstract domain and abstract unification, and the $\sqsubseteq$ relation, which in turn defines $\sqcup$ (LUB). Abstract unification is divided into two in the framework, so that it is required to define: (1) how to compute the entry substitution for a clause C given a subgoal p (which unifies with the head of C) and its call substitution; and (2) how to compute the success substitution for a subgoal p given its call substitution and the exit substitution for a clause C whose head unifies with p. We formalize this with functions *entry_to_exit* and *call_to_success* in Figure 4. The domain dependent functions used there are:

- $call\_to\_entry(p(\bar{u}), C, \lambda)$ which gives an abstract substitution describing the effects on $vars(C)$ of unifying $p(\bar{u})$ with $head(C)$ given an abstract substitution λ describing $\bar{u}$,

- $exit\_to\_success(\lambda, p(\bar{u}), C, \beta)$ which gives an abstract substitution describing $\bar{u}$ accordingly to β (which describes $vars(head(C))$) and the effects of unifying $p(\bar{u})$ with $head(C)$ under the abstract substitution λ describing $\bar{u}$,

- $extend(\lambda, \lambda')$ which extends abstract substitution λ to incorporate the information in λ' in a way that it is still consistent,

- $project\_in(\bar{v}, \lambda)$ which extends λ so that it refers to all of the variables $\bar{v}$,

- $project\_out(\bar{v}, \lambda)$ which restricts λ to only the variables $\bar{v}$.

$entry\_to\_exit(C, \beta_{entry}) \equiv$
  $A_1 := project\_in(vars(C), \beta_{entry})$;
  For $i := 1$ to $length(C)$ do
    A_{i+1} := $call\_to\_success(q_i(\bar{u}_i), A_i)$);
  return $project\_out(vars(head(C)), A_{n+1})$;

$call\_to\_success(p(\bar{u}), \lambda_{call}) \equiv$
  $\lambda := project\_out(\bar{u}, \lambda_{call})$; $\lambda' := \bot$;
  For each clause C which matches $p(\bar{u})$ do
    $\beta_{exit} := entry\_to\_exit(C, call\_to\_entry(p(\bar{u}), C, \lambda))$;
    $\lambda' := \lambda' \sqcup exit\_to\_success(\lambda, p(\bar{u}), C, \beta_{exit})$;
  od;
  return $extend(\lambda_{call}, \lambda')$;

Figure 4: The Top–Down Framework

In the presence of recursive predicates, analysis requires a fixpoint computation. In [MH90, MH92] a fixpoint algorithm was proposed for the framework that localizes fixpoint computations to only the strongly connected components of (mutually) recursive predicates. Additionally, an initial approximation to the fixpoint is computed from the non-recursive clauses of the recursive predicate. Fixpoint convergence is accelerated by updating this value with the information from every clause analyzed in turn. The algorithm is (schematically) shown in Figure 5. For a complete description see [MH90, MH92].

$call\_to\_success\_recursive(p(\bar{u}), \lambda_{call}) \equiv$
  $\lambda := project\_out(\bar{u}, \lambda_{call})$; $\lambda' := \bot$;
  For each non-recursive clause C which matches $p(\bar{u})$ do
    $\beta_{exit} := entry\_to\_exit(C, call\_to\_entry(p(\bar{u}), C, \lambda))$;
    $\lambda' := \lambda' \sqcup exit\_to\_success(\lambda, p(\bar{u}), C, \beta_{exit})$;
  od;
  $\lambda'' := fixpoint(p(\bar{u}), \lambda, \lambda')$;
  return $extend(\lambda_{call}, \lambda'')$;

$fixpoint(p(\bar{u}), \lambda, \lambda') \equiv$
  $\lambda'' := \lambda'$;
  For each recursive clause C which matches $p(\bar{u})$ do
    $\beta_{exit} := entry\_to\_exit(C, call\_to\_entry(p(\bar{u}), C, \lambda))$;
    $\lambda'' := \lambda'' \sqcup exit\_to\_success(\lambda, p(\bar{u}), C, \beta_{exit})$;
  od;
  If $\lambda'' = \lambda'$ then return λ''
  else return $fixpoint(p(\bar{u}), \lambda, \lambda'')$;

Figure 5: The Fixpoint Computation

8 Abstract Framework, Domain, and Operations for Non-Failure Analysis

In the non-failure analysis, the covering test is instrumental. In fact, covering can be seen as a notion that characterizes the fact that execution of a query will not finitely fail, i.e., if it has finished derivations then at least one is successful. Note that, as in [DLGH97], non-failure does not imply success: a predicate that is non-failing may nevertheless not produce an answer because it does not terminate.

Definition 8.1 [Covering] Given computation state $\langle g :: G \mid \theta \rangle$ in the execution of program P, define the *global answer constraint* of goal g in store θ as:

$$c = \vee\{\ curr\_store(D'_i) \mid D'_i \in derivations(P, \langle g, \theta \rangle) \text{ and is maximal } \}$$

Let $\bar{u}$ denote the variables of g already constrained in θ, call them the *input* variables. We say that g is *covered* in θ iff $\theta{\downarrow_{\bar{u}}} \models c{\downarrow_{\bar{u}}}$.

It is not difficult to show that, in a pure language, where failure can only be caused by constraint store inconsistency, covering is a sufficient condition for non-failure. Indeed, if g

is covered in θ, i.e., $\theta \downarrow_{\bar{u}} \models c \downarrow_{\bar{u}}$, then one of the disjunctions in (the projection of) c is entailed. This corresponds to a (maximal) derivation of $\langle g, \theta \rangle$, and this derivation cannot be failed, since, if it were, it would be inconsistent, and no inconsistent constraint can be entailed by a consistent one. Therefore, either such derivation is infinite, or, if finite, it is successful. Thus:

If g is *covered* in θ then $\langle g, \theta \rangle$ does not finitely fail.

A key issue in non-failure analysis will thus be how to approximate the current store and the global answer constraint so that covering can be effectively and accurately approximated. In [DLGH97] such an approximation is defined in the following terms: A goal is non-failing if there is a subset of clauses of the predicate which do not fail and which match the input types of the goal. This "matching" is the so-called *covering test*, and basically amounts to the analysis being able to gather, for each such clause, enough constraints on the input variables of the goal to be able to prove that, for each of the variables, any element in the corresponding type satisfies at least the constraint gathered for one clause. An analysis for non-failure thus needs to traverse the clauses of a predicate to check non-failure of the clause body goals, collect constraints that approximate the global answer constraint, and finally check that they cover the input types of the original goal. In the rest of this section, we show how to accommodate the abstract interpretation based framework of the previous section to perform these tasks, and define an abstract domain suitable for them.

8.1 Abstract Domain

The abstractions for non-failure analysis are made of four components. The first two are (abstractions of) constraints that represent the current store and the global answer constraint for the current goal. This is the core part of the domain. The other two components carry the results of the covering test, specifying if the current constraint store covers the global answer constraint, and if this implies that the computation may fail or not. The covering and non-failure information is represented by values of the set $\mathcal{B} = \{\top, \bar{0}, \bar{1}, \bot\}$, where $\bar{0}$ and $\bar{1}$ are not comparable in the ordering. For covering, $\bar{0}$ is interpreted as "not covered" and $\bar{1}$ as covered. For non-failure, $\bar{0}$ is interpreted as "not failing" and $\bar{1}$ as failing.

Definition 8.2 [Abstract Domain] Let $\mathcal{C}^{\alpha_1}$ and $\mathcal{C}^{\alpha_2}$ be abstract domains for $\mathcal{C}$. The abstract domain for non-failure is the set

$$\mathcal{F} = \{(s, c, o, f) \mid s \in \mathcal{C}^{\alpha_1}, c \in \mathcal{C}^{\alpha_2}, o \in \mathcal{B}, f \in \mathcal{B}\}$$

The ordering in domain $\mathcal{F}$ is induced from that in $\mathcal{B}$, so that (overloading $\sqsubseteq$):

$$(s_1, c_1, o_1, f_1) \sqsubseteq (s_2, c_2, o_2, f_2) \text{ iff } f_1 \sqsubseteq f_2$$

In an element $(s, c, o, f) \in \mathcal{F}$, components s and c are abstractions α_1 and α_2 of the constraint domain $\mathcal{C}$. The usual approximations used (e.g., in [DLGH97]) are types (and modes) for s, and a finite set of (concrete) constraints for c.

Definition 8.3 [Abstraction Function] The abstraction of a derivation D in the execution of program P, such that $curr\_store(D) = \theta$ and $curr\_goal(D) = g$, and the input variables and global answer constraint of g in θ are respectively $\bar{u}$ and c, is $\alpha(D) = (\theta^{\alpha_1}, c^{\alpha_2}, o, f)$, where:

$$f = \begin{cases} \bar{1} & \text{if } D \text{ is failed} \\ \bar{0} & \text{otherwise} \end{cases} \quad \text{and } o = \begin{cases} \bar{1} & \text{if } \theta^{\alpha_1}\downarrow_{\bar{u}} \models^{\alpha} c^{\alpha_2}\downarrow_{\bar{u}} \\ \bar{0} & \text{otherwise} \end{cases}$$

It is easy to show that such an abstraction is correct, provided that α_1 and α_2 are also correct abstractions, and that the corresponding abstract covering test ($\models^{\alpha}$) correctly approximates Definition 8.1. For α_1 we have already mentioned the use of type and mode information. One possibility for α_2 is to use only those constraints appearing explicitly in the clause bodies of the predicate whose covering test is to be performed (the current goal g in the derivation).

Example 2 Consider the following (contrived) predicates:

```
p(X,Y,Z):- X =< Y, q(X,Z).
q(X,Y):- X =< Y.
```

The global answer constraint for `p(X,Y,Z)` is `X =< Y` $\wedge$ `X =< Z`, but it can be approximated simply by `X =< Y`, the only constraint in the definition of `p`/3. □

One rationale for the above choice might be that collecting all constraints in derivations may not be possible during a compile-time analysis (since such constraints are only known during execution), or may lead to non-termination of the analysis. However, the first problem can be alleviated by proper abstractions of the tests (such as a depth-k abstraction, in a way similar to [DRRS93]), and the second problem only occurs for recursive predicates. Thus, the most simple solution to the termination problem is to avoid collecting constraints in recursive calls.[5]

Example 3 The global answer constraint for the predicate `sorted`/1 defined below includes a constraint for each two elements in the input list, the length of which is not in general known at compile-time.

```
sorted([]).
sorted([_]).
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```

[5]Note that this does not imply that recursive calls are simply ignored. They need to be considered to check that they are indeed non-failing, even though their global answer constraint is not computed.

□

Our solution to this problem[6] is to collect only constraints that refer literally to the predicate arguments in the program clause head, which also excludes in general (but not always) the constraints arising from recursive calls.

Example 4 Consider again the predicate `sorted`/1 defined in the previous example. We collect constraints only for the clause head argument `[X,Y|L]`, which amounts to only one constraint: `X =< Y` (since the recursive call does not provide constraints for the head arguments that appear literally in the program).

Consider, on the other hand, predicate `p`/3 of Example 2. In this case the complete global answer constraint for `p(X,Y,Z)` will be collected: `X =< Y` $\wedge$ `X =< Z`, since the two single constraints can be "projected" onto the clause head. □

Note that such a solution yields an under-approximation of the global answer constraints. Given the use of type and mode information, which are in general over-approximations, we have that, for any element $(s, c, o, f) \in \mathcal{F}$, given current constraint store θ and global answer constraint ω, $s = \theta^{\alpha_1}$ is an over-approximation of θ, and $c = \omega^{\alpha_2}$ is an under-approximation of ω. In this situation, it is not difficult to prove that $\theta^{\alpha_1} \downarrow_{\bar{u}} \models^{\alpha} \omega^{\alpha_2} \downarrow_{\bar{u}}$ correctly approximates covering: $\theta \downarrow_{\bar{u}} \models \omega \downarrow_{\bar{u}}$.

8.2 Abstract Operations

Abstract values $(s, c, o, f) \in \mathcal{F}$ are built during analysis in the following way: f is carried along during the abstract computation by the abstract operations below, o is computed from the covering test, c is collected as explained above, and for s, type and mode analysis is performed. Thus, our analysis is in fact three-fold: it carries on mode, type, and non-failure analyses simultaneously. We focus now on the abstract operations for non-failure, given that those for types and modes are standard:

- $call\_to\_entry(p(\bar{u}), C, \lambda)$ solves head unification $p(\bar{u}) = head(C)$, and checks that it is consistent with the c component of λ. If it is not, it returns $\bot$, otherwise, the resulting abstraction.

  If $p(\bar{u}) \in \mathcal{C}$, i.e., if it happens to be a constraint itself, then no clause C exists, and $p(\bar{u})$ itself is added to the c component. In this case the following $exit\_to\_success$ function is not called.

[6] However, we plan to investigate other solutions. In particular, the use of a depth-k abstraction seems to be a very promising one.

- $exit\_to\_success(\lambda, p(\bar{u}), C, \beta)$ adds the equations resulting from unification $p(\bar{u}) = head(C)$ to the c component of β and projects it onto $vars(\bar{u})$.

  It is the projection performed here that gets rid of useless constraints, like in the case of Example 4. Constraints that cannot be projected onto the (goal) variables $\bar{u}$ are simply dropped in the analysis.

- $\lambda \uplus \lambda'$ adds abstraction λ to the set λ' if λ is non-failing.

- $extend(\lambda, \lambda')$ performs the covering test for λ' (a set of abstractions); if it is successful, the c component of λ' is merged with that of λ.

  This operation uses the covering algorithm described in [DLGH97], which takes the global answer constraint c and a *type assignment* for the input variables appearing in c. Given a finite set of variables V, a type assignment over V is a mapping from V to a set of types. This is computed from the type information in the first component of λ. Input variables are determined from the mode information in that same component. The global answer constraint is obtained as the disjunction of the c components of each abstraction in λ'.

8.3 Adapting the Analysis Framework

The framework described in the previous section is not adequate for non-failure analysis. The main reason for this is that the aggregation function for the successive exit abstractions of the different clauses is not the LUB anymore. In non-failure analysis, the constraints for each clause need to be gathered together, and a covering test on the set of constraints needs to be performed. Another difference is that the covering test should only consider constraints from clauses that are not guaranteed to fail altogether;[7] therefore the aggregator must be able to discriminate abstract substitutions on this criterion.

We have adapted the definition of the *call_to_success* function to reflect the aggregation operator. The adapted definition is shown in Figure 6. Note that, as a result of this, λ' in the algorithm is not anymore an abstract substitution, but a set of them. This is input to *extend*, which is in charge of the covering test.

When fixpoint computation is required, adapting the framework is a bit more involved. Basically, since the aggregation operator is not LUB, fixpoint detection cannot be performed right after the success substitution has been computed. Normally, it is the LUB that is used for updating the successive approximations to the fixpoint value, and fixpoint detection works by simply comparing the initial and the final values for the success substitution. In non-failure analysis,

[7] Note how this information could be used to improve the results of other analyses.

$call\_to\_success(p(\bar{u}), \lambda_{call}) \equiv$
  $\lambda := project\_out(\bar{u}, \lambda_{call})$; $\lambda' := \emptyset$;
  For each clause C which matches $p(\bar{u})$ do
    $\beta_{exit} := entry\_to\_exit(C, call\_to\_entry(p(\bar{u}), C, \lambda))$;
    $\lambda' := \lambda' \uplus exit\_to\_success(\lambda, p(\bar{u}), C, \beta_{exit})$;
  od;
  return $extend(\lambda_{call}, \lambda')$;

Figure 6: The Top–Down Framework for Non-Failure Analysis

the covering test must be performed first, and only after this one has been performed, the test for the fixpoint can be done. The resulting algorithm is shown in Figure 7. It is basically a simpler fixpoint iterator over the function *call_to_success* abandoning the sophisticated fixpoint computation of Figure 5.

$call\_to\_success\_recursive(p(\bar{u}), \lambda_{call}) \equiv$
  $\lambda := project\_out(\bar{u}, \lambda_{call})$;
  return $fixpoint(p(\bar{u}), \lambda, \bot)$;

$fixpoint(p(\bar{u}), \lambda, \lambda') \equiv$
  $\lambda'' := call\_to\_success(p(\bar{u}), \lambda)$;
  If $\lambda'' = \lambda'$ then return λ''
  else return $fixpoint(p(\bar{u}), \lambda, \lambda'')$;

Figure 7: The Fixpoint Computation for Non-Failure Analysis

A Running Example We now illustrate our analysis by means of a detailed example on how it will proceed. Consider the program (fragment) below:

```
qsort(As,Bs):- qsort(As,Bs,[]).

qsort([X|L],R,R2) :-
    partition(L,X,L1,L2), qsort(L2,R1,R2), qsort(L1,R,[X|R1]).
qsort([],R,R).
```

```
partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right):- E < C,  partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):- E >= C, partition(R,C,Left,Right1).
```

Let the abstract call pattern for atom `qsort(As,Bs,[])` be

$$(\{list(As, num), var(Bs)\}, true, \overline{1}, \overline{0}).$$

Upon entering the first clause defining `qsort/3`, the result of $call\_to\_entry$ (restricted to the head variables) is

$$(\{num(X), list(L, num), var(R), [](R2)\}, true, \overline{1}, \overline{0})\ ^{8}$$

plus, additionally, $\{var(R1), var(L1), var(L2)\}$ for the free variables in the clause. Once projected, this gives the call pattern for the first literal in that clause:

$$(\{list(L, num), num(X), var(L1), var(L2)\}, true, \overline{1}, \overline{0}).$$

We omit the analysis of the partition predicate. After the fixpoint computation for this predicate, however, we will have a set of three abstract elements corresponding to the abstraction of the three clauses. For brevity, we express such set as a single abstraction where it is the c component that is a set, instead.[9] Note that this is possible because all other components (types, modes, covering, non-failure) of the abstractions in the set are the same. Thus, we have:

$$\begin{array}{lll} (& \{ & list(L, num), num(X), list(L1, num), list(L2, num)\ \}, \\ & \{ & L = [] \wedge L1 = [] \wedge L2 = [], \quad L = [E|\_] \wedge E < X \wedge L1 = [E|\_], \\ & & L = [E|\_] \wedge E >= X \wedge L2 = [E|\_]\ \}, \qquad\qquad \overline{1}, \overline{0}\ ). \end{array}$$

This is now extended (by abstract function $extend$) to the corresponding program point of the clause of `qsort`. First, the covering test is performed, and it succeeds, since $list(L, num), num(X)$ covers indeed the global answer constraint projected onto the input variables:

$$(L = [E|\_] \wedge (E < X \vee E >= X)) \vee L = [].$$

Therefore, computation is still covered and non-failing. This, together with the projection of the c component onto the variables of the first clause of `qsort`, yields success abstraction for partition:

$$\begin{array}{lll} (& \{ & num(X), list(L, num), var(R), var(R1), [](R2), \\ & & list(L1, num), list(L2, num)\ \}, \qquad\qquad true, \overline{1}, \overline{0}\ ) \end{array}$$

where the c component is still $true$ since the projection onto the clause variables factors out the previously computed global answer constraint. Now, analysis will proceed into call `qsort(L2,R1,R2)` with

$$(\{list(L2, num), var(R1), [](R2)\}, true, \overline{1}, \overline{0}).$$

[8] To be concise, we denote with $[](A)$ that the type of A is that of the empty lists.

[9] This very same "trick" is used in the implementation.

Since this is basically the same call pattern that we started with, no new fixpoint computation is started in this case.[10] On the other hand, a new fixpoint computation is started for the second recursive call `qsort(L1,R,[X|R1])` with

$$(\{list(L1, num), var(R), num(X), list(R1, num)\}, true, \overline{1}, \overline{0}).$$

This is a new call pattern for the `qsort` predicate, which initiates a new fixpoint computation. The fixpoint value obtained in this computation is the same abstraction, except for the type of R which on output is a list. Finally, $exit\_to\_success$ now lifts this result to the original goal `qsort(As,Bs,[])` giving:

$$(\{list(As, num), list(Bs, num)\}, As = [\_|\_], \overline{1}, \overline{0}).$$

The analysis of the non-recursive clause immediately gives:

$$(\{[](As), [](Bs)\}, As = [] \wedge Bs = [], \overline{1}, \overline{0}),$$

and $extend$ computes the covering test for the set of the above two abstractions with the initial input abstraction, in which the input types are $list(As, num)$. Certainly, this type covers the (projected) global answer constraint $As = [\_|\_] \vee As = []$. Thus, the goal is still covered and non-failing.

Finally, since the abstraction now computed is only the result of a first iteration of the fixpoint computation, a new iteration is started. The result in this case is the same, and fixpoint computation finishes with that very same result.

9 Implementation Results

We have constructed a prototype implementation in (Ciao) Prolog by adapting the framework of the PLAI implementation and defining the abstract operations for non-failure analysis that we have described in this paper. Most of these abstract operations have been implemented by reusing code of the implementation in [DLGH97], such as for example, the covering algorithm. We have incorporated the prototype in the Ciao/CiaoPP multiparadigm programming system [HBPLG99, HPBLG03, BLGPH04] and tested it on the benchmarks used in the non-failure analysis of Debray *et al.* [DLGH97], plus some benchmarks exhibiting paradigmatic behaviours, plus a last group with those used in the cardinality analysis of Braem *et al.* [BCMH94]. These two analyses are the closest related previous work that we are aware of. Some relevant results of these tests for non-failure analysis are presented in Table 3. **Program** lists the program names, **N** the number of predicates in the program, **F** and **C** are the number of non-failing predicates detected by the non-failure analysis in [DLGH97], and the cardinality analysis in [BCMH94],

[10] Here, we save the reader from some more fi xpoint iterations that will be taking place. However, the results are as indicated.

respectively.

Note that our multi–variant analysis can infer several variants (call patterns) for the same predicate, where some of them may be non-failing (resp. covered) and the other ones can be failing (resp. not covered). For instance, in the case of the program *Mv* in Table 3 (also described in Example 1), which has 4 predicates (`mv`/3, `qsort`/2, `partition`/4 and `append`/3), the analysis infers one variant for `mv`/3, which is non-failing and covered, 2 variants for `qsort`/2 (one of them which is non-failing and covered, and the other one which is failing and not covered), one variant for `partition`/4, which is non-failing and covered, and 3 variants for `append`/3 (2 of them which are non-failing and covered, and the other one which is failing and not covered). For this reason, and in order to make the results comparable, column **AF** shows two figures (both corresponding to the analysis presented in this paper): the number of predicates such that **all of their variants** (call patterns) are detected as non-failing, and (between parenthesis) the number of predicates such that **some of their variants** are detected as non-failing (this second figure is omitted if it is equal to the first one).

Similarly, **ACov** shows two figures (both corresponding to the analysis presented in this paper): the number of predicates detected to cover **all** of their (calling) types (variants), and (between parenthesis), the number of predicates detected to cover **some** of their (calling) types. **Cov** is the number of predicates detected to cover their (calling) types by the analysis in [DLGH97].

$\mathbf{T}_{AF}$ and $\mathbf{T}_{F}$ are the total time (in milliseconds) required by the analysis presented in this paper and the analysis in [DLGH97] respectively (both of which include the time required to derive the modes and types). The timings were taken on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 1Gb of RAM memory, running Red Hat Linux 8.0, and averaging several runs and eliminating the best and worst values. Ciao version 1.9.111 and CiaoPP-1.0 were used.

Analysis time averages (per predicate) are also provided in the last row of the table. From these numbers, it is clear that the new implementation based on the abstract interpretation engine is more efficient than the previous one. It is also more precise, as shown for example in the benchmarks *Mv*, *Zebra*, *Family*, *Blocks*, *Reach*, and *Plan*.

10 Conclusions

We have described a non-failure analysis based on abstract interpretation, which extends the previous proposal of Debray et al. Our analysis improves in precision, and enjoys a clear theoretical setting, and a simpler implementation. Also, the implementation is more efficient. The abstract domain underlying the analysis can be easily modified to cater for a determinacy analysis. Such

Program	**N**	**AF**	**F**	**C**	**ACov**	**Cov**	$\mathbf{T}_{AF}$	$\mathbf{T}_{F}$	$\frac{T_{AF}}{T_F}$
Hanoi	2	2	2	N/A	2	2	33	242	0.14
Fib	1	1	1	N/A	1	1	17	22	0.77
Tak	1	1	1	N/A	1	1	9	11	0.82
Subs	1	1	1	N/A	1	1	5	33	0.15
Reverse	2	2	2	N/A	2	2	17	29	0.59
Mv	4	2 (4)	1	N/A	2 (4)	2	54	102	0.53
Zebra	6	2	1	N/A	5 (6)	4	1008	1100	0.92
Family	3	3	1	N/A	3	2	10	18	0.56
Blocks	7	1 (2)	0	N/A	4 (5)	4	30	59	0.51
Reach	2	2	0	N/A	2	1	19	30	0.63
Bid	20	5 (8)	5	N/A	14 (17)	14	3089	3369	0.92
Occur	4	1 (3)	1	N/A	1 (3)	1	69	78	0.88
Plan	16	5 (8)	3	0	11 (13)	10	2626	4128	0.64
Qsort	3	3	3	0	3	3	29	65	0.45
Qsort2	5	3	3	0	3	3	33	76	0.43
Queens	5	2 (3)	2	0	3 (4)	3	60	74	0.81
Pg	10	2 (3)	2	0	6 (9)	6	412	477	0.86
Mean							38 (/p)	58 (/p)	0.67 (/p)

Table 3: Accuracy and efficiency of the non-failure analysis (times in mS).

an analysis, provided with a depth-k abstraction, would be the abstract interpretation counterpart of determinacy analyses such as that of [DRRS93]. We are currently working on the verification of this proposition.

The implemented analysis we have described in this paper is currently integrated in CiaoPP, and is being used for lower-bounds cost analysis, granularity control, and program debugging. Arguably, although our presentation covers strictly constraint logic programming, the technique could be easily applied to functional logic languages with similar results, as is indeed the case in the Ciao system, where the analysis presented works without modification for Ciao's functional subset or for combinations of functions and predicates.

Part III

Abstraction-Carrying Code

11 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) [Nec97] 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 [Nec97] 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 [AF99] to define a representation of lemmas and definitions which helps keep the proofs small. A recent proposal [BL02] uses temporal logic to specify security policies in PCC. In Typed Assembly Languages [MWCG99], 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 [NR01] or Tactic-based PCC [AGH+04]), whereas in other cases the priority is to reduce the checking time (as in, e.g., standard PCC [Nec97] or lightweight bytecode verification [Ler03]). 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* [CC77] 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. Figure 8 presents an overview of ACC as performed in our system. 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 aforementioned challenges:

1. The first element 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. Our framework is parametric w.r.t. the abstract domain(s) of interest, which gives us generality and expressiveness.

2. The next element at the producer's side is a fixpoint static Analyzer which automatically infers 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. 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

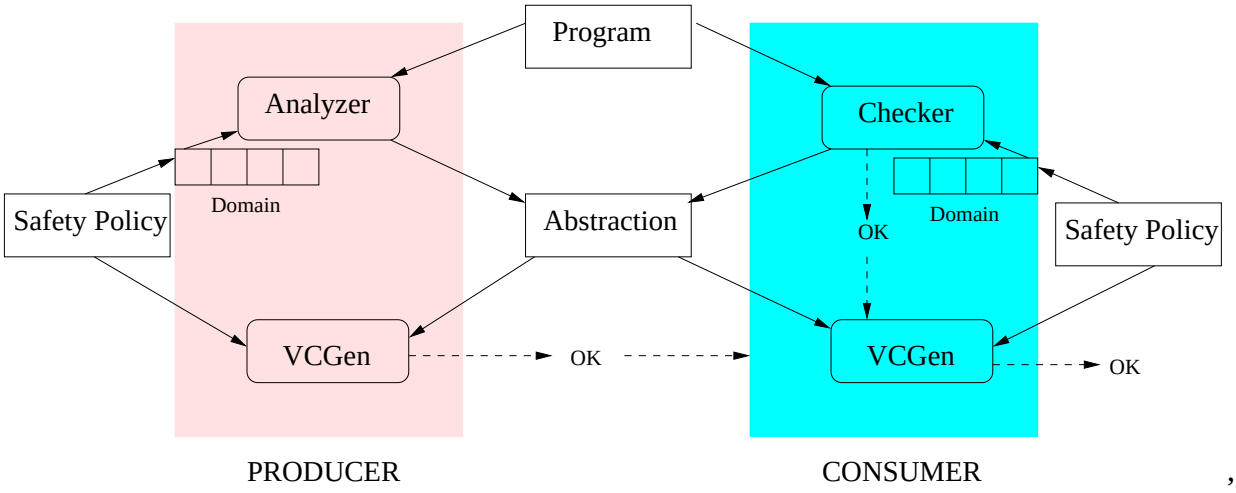

Figure 8: Abstraction-Carrying Code in `CiaoPP`

if the execution of the code does not violate the safety policy. As in standard PCC methods, this process is performed also by the consumers in order to have a trustworthy VC.

4. Finally, a simple, easy-to-trust (analysis) checker at the consumer's side 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` [HPBLG03], the abstract interpretation-based preprocessor of the `Ciao` multi-paradigm CLP system. `CiaoPP` uses modular, incremental abstract interpretation as a fundamental tool to obtain information about programs. In `CiaoPP`, the semantic approximations thus produced have been applied to perform high- and low-level optimizations during program compilation, including transformations such as multiple abstract specialization, parallelization, resource usage control, and program verification. More recently, novel and promising applications of such semantic approximations are being applied in the more general context of program development. We report on our extension of the framework to incorporate ACC and on how this instantiation of ACC already shows promising results. Our approach is highly flexible because it inherits the parametricity on the abstract domain and inference power of the abstract interpretation engines used in (C)LP.

The paper is organized as follows. Section 12 introduces some notation and preliminary notions on CLP and abstract interpretation. Section 13 describes the assertion language which is used to define our safety policy. Section 14 formalizes the certification process performed in `CiaoPP` to generate an abstraction of the program. In Section 15, we present the verification condition generator which attests compliance of the abstraction with respect to the safety policy. In Section 16, we introduce an abstract interpretation-based checker which validates the safety certificate in the consumer. Section 17 reports some experiments performed in the `CiaoPP`-based implementation. In Section 18, we sketch promising applications of our framework within a pervasive computing environment. Finally, Section 19 discusses the work presented in this paper and related work.

12 Preliminaries

We assume familiarity with constraint logic programming [JM94] (CLP) and the concepts of abstract interpretation [CC77] which underlie most analyses in CLP. The remaining of this section introduces some notation and recalls preliminary concepts on these topics.

12.1 Constraint Logic Programming

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 12.1 (CLP Program) *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`*. Predicates* `number_codes/2`*,* `atom_codes/2`*, and* `open/3` *are ISO-standard Prolog predicates, and thus they are available in* `CiaoPP`*.*

12.2 Abstract Interpretation

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 [CC77], 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 [DZ92].

Example 12.2 (*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 *[VB02], 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 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.*[11]

Apart from predefined types, in the $eterms$ *domain, one can have user-defined regular types declared by means of* Regular Unary Logic *programs [FSVY91]. 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:*[12]

```
:- 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.*

13 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 [PBH00b] available in `CiaoPP` to define an expressive class of safety policies in the context of *constraint logic programs*.

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 [PBH00b]

[11] 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.

[12] The `regtype` declarations are used to defi ne new regular types in `CiaoPP`.

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."

$success(B, [\lambda_{Pre},]\lambda_{Post})$**:** 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 [NL98]). 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 13.1 (Safety Policy) *Figure 9 shows the assertions which are relevant to the program in our running example. The first four rows correspond to* `calls` *assertions, whereas the last three are* `success` *assertions. Out of the four* `calls`*, the first three are predefined in the system. The last user-defined assertion for predicate* `safe_open` *provides a simple way to*

calls(number_codes(X,Y), {(num(X);list(Y,numcodes))})
calls(atom_codes(X,Y), {(constant(X);string(Y))})
calls(open(X,Y,_Z), {constant(X),io_mode(Y)})
calls(safe_open(Fname,_,_), {safe_name(Fname)})
success(number_codes(X,Y), ⊤, {num(X),list(Y,numcodes)})
success(atom_codes(X,Y),⊤, {constant(X),string(Y)})
success(open(X,Y,Z), ⊤, {constant(X),io_mode(Y),stream(Z)})

Figure 9: Assertions for the example

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`*". The assertion for* `open` *is predefined in our system and it requires that, 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.

14 Certifying Programs by Static Analysis

This section introduces (part of) the *certification* process, as sketched to the left of Figure 8, carried out by the producer, namely the generation of a certificate to attest the adherence of the program to the safety policy. The generation of the verification condition from the certificate is discussed in the next section.

14.1 Using Analysis Results as Certificates

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, as introduced in Sect. 13. 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, a standard Analyzer is run. Our 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.*

Global program analysis is becoming a practical tool in constraint logic program compilation in which information about calls, answers, and the effect of the constraint store on variables at different program points is computed statically [HWD92, VD92, MH92, SCWY91, BdlBH94]. Essentially, an analyzer returns an *abstraction* of P's execution in terms of the abstract domain D_α. The underlying theory, formalized in terms of abstract interpretation [CC77], and the related implementation techniques are well understood for several general types of analysis and, in particular, for top-down analysis of Prolog [Deb89, Deb92, Bru91, MH92, MSJ94, CV94]. Several generic analysis engines, such as the one implemented in the `CiaoPP` system [HPMS00, MH92, MH90], GAIA [CV94], and the CLP($\mathcal{R}$) analyzer facilitate construction of such top-down analyzers. These generic engines have the description domain and functions on this domain as parameters. As it appears in Figure 8, in principle 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. 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 [MH90, MH92, CDMV93].

In order to analyze a program, traditional (goal dependent) abstract interpreters for (C)LP programs receive as input, in addition to the program and the abstract domain, 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 $CP \in D_\alpha$.[13] Given a program P and a call pattern CP in the context of an abstract domain

[13] 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).

D_α, this analyzer constructs an *and–or graph* (or analysis graph) for Q which can be viewed as a finite representation of the (possibly infinite) set of (possibly infinite) AND-OR trees explored by the concrete execution of Q in P [Bru91]. The analysis graph corresponds to (or approximates) the abstract semantics of the program and entry [Bru91]. 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.

The important point here is that the analysis graph computed by an abstract interpretation-based analyzer represents an *abstract model* (or abstraction) of the program. The following section recalls the concrete analysis algorithm of [HPMS00] which computes an analysis graph and identifies the fragment of the information stored in such graph which is sufficient in order to play the role of safety certificate.

14.2 The Analysis Algorithm

The program analysis graph is implicitly represented in the algorithm implemented in `CiaoPP` [HPMS00] by means of two data structures, the *answer table* and the *dependency arc table*. Given the information in these it is straightforward to construct the graph and the associated program point annotations. The answer table contains entries of the form A : *CP* $\mapsto$ *AP* where A is always a base form. This corresponds to an OR-node in the analysis graph of the form $\langle A$: *CP* $\mapsto$ *AP*$\rangle$. It is interpreted as "the answer pattern for calls to A satisfying precondition (or call substitution), *CP*, accomplishes postcondition (or success substitution), *AP*." A dependency arc is of the form H_k : $CP_0 \Rightarrow [CP_1]\ B_{k,i}$: CP_2. This is interpreted as follows: if the rule with H_k as head is called with description CP_0 then this causes literal $B_{k,i}$ to be called with description CP_2. The remaining part CP_1 is the program annotation just before $B_{k,i}$ is reached and contains information about all variables in rule k. CP_1 is not really necessary, but is included for efficiency. Dependency arcs represent the arcs in the program analysis graph from atoms in a rule body to an atom node.

Intuitively, the analysis algorithm is just a graph traversal algorithm which places entries in the answer table and dependency arc table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixed-point algorithms, we use a priority queue. Thus, the third, and final, structure used in our algorithms is a *prioritized event queue*. Events are of three forms:

- $newcall(A$: *CP*$)$ which indicates that a new calling pattern for atom A with description

CP has been encountered.

- $arc(R)$ which indicates that the rule referred to in R needs to be (re)computed from the position indicated.
- $updated(A : CP)$ which indicates that the answer description to calling pattern A with description *CP* has been changed.

The generic analysis algorithm of [HPMS00] is given in Figure 10. It is defined in terms of five abstract operations on the description domain D_α of interest:

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

Apart from the parametric description domain-dependent functions, the algorithm has several other undefined functions. The functions add_event and next_event respectively add an event to the priority queue and return (and delete) the event of highest priority.

When an event being added to the priority queue is already in the priority queue, a single event with the maximum of the priorities is kept in the queue. When an arc $H_k : CP \Rightarrow [CP'']B_{k,i} : CP'$ is added to the dependency arc table, it replaces any other arc of the form $H_k : CP \Rightarrow [\_]B_{k,i} : \_$ in the table and the priority queue. Similarly when an entry $H_k : CP \mapsto AP$ is added to the answer table, it replaces any entry of the form $H_k : CP \mapsto \_$. Note that the underscore (_) matches any description, and that there is at most one matching entry in the dependency arc table or answer table at any time.

The function initial_guess returns an initial guess for the answer to a new calling pattern. The default value is $\bot$ but if the calling pattern is more general than an already computed call then its current value may be returned.

The algorithm centers around the processing of events on the priority queue in main_loop, which repeatedly removes the highest priority event and calls the appropriate event-handling function. When all events are processed it calls remove_useless_calls. This procedure traverses the dependency graph given by the dependency arcs from the initial calling patterns S and marks

```
analyze(S)
  foreach A : CP ∈ S
    add_event(newcall(A : CP))
  main_loop()

main_loop()
  while E := next_event()
    if (E = newcall(A : CP))
      new_calling_pattern(A : CP)
    elseif (E = updated(A : CP))
      add_dependent_rules(A : CP)
    elseif (E = arc(R))
      process_arc(R)
  endwhile
  remove_useless_calls(S)

new_calling_pattern(A : CP)
  foreach rule A_k :- B_{k,1}, ..., B_{k,n_k}
    CP_0 :=
      Aextend(CP, vars(B_{k,1}, ..., B_{k,n_k}))
    CP_1 := Arestrict(CP_0, vars(B_{k,1}))
    add_event(arc(
      A_k : CP ⇒ [CP_0] B_{k,1} : CP_1))
  AP := initial_guess(A : CP)
  if (AP ≠ ⊥)
    add_event(updated(A : CP))
  add A : CP ↦ AP to answer table

add_dependent_rules(A : CP)
  foreach arc of the form
      H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
      in graph
    where there exists renaming σ
      s.t. A : CP = (B_{k,i} : CP_2)σ
    add_event(arc(
      H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2))

process_arc(H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2)
  if (B_{k,i} is not a constraint)
    add H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
    to dependency arc table
  W := vars(A_k :- B_{k,1}, ..., B_{k,n_k})
  CP_3 := get_answer(B_{k,i} : CP_2, CP_1, W)
  if (CP_3 ≠ ⊥ and i ≠ n_k)
    CP_4 := Arestrict(CP_3, vars(B_{k,i+1}))
    add_event( arc(
      H_k : CP_0 ⇒ [CP_3] B_{k,i+1} : CP_4))
  elseif (CP_3 ≠ ⊥ and i = n_k)
    AP_1 := Arestrict(CP_3, vars(H_k))
    insert_answer_info(H : CP_0 ↦ AP_1)

get_answer(L : CP_2, CP_1, W)
  if (L is a constraint)
    return Aadd(L, CP_1)
  else
    AP_0 := lookup_answer(L : CP_2)
    AP_1 := Aextend(AP_0, W)
    return Aconj(CP_1, AP_1)

lookup_answer(A : CP)
  if (there exists a renaming σ s.t.
    σ(A : CP) ↦ AP in answer table)
    return σ^{-1}(AP)
  else
    add_event(newcall(σ(A : CP)))
    where σ is a renaming s.t.
    σ(A) is in base form
    return ⊥

insert_answer_info(H : CP ↦ AP)
  AP_0 := lookup_answer(H : CP)
  AP_1 := Alub(AP, AP_0)
  if (AP_0 ≠ AP_1)
    add (H : CP ↦ AP_1) to answer table
    add_event(updated(H : CP))
```

Figure 10: Fixpoint Analyzer

those entries in the dependency arc and answer table which are reachable. The remainder are removed.

The function new_calling_pattern initiates processing of the rules in the definition of atom

A, by adding arc events for each of the first literals of these rules, and determines an initial answer for the calling pattern and places this in the table. The function add_dependent_rules adds arc events for each dependency arc which depends on the calling pattern $(A : CP)$ for which the answer has been updated. The function process_arc performs the core of the analysis. It performs a single step of the left-to-right traversal of a rule body. If the literal $B_{k,i}$ is an atom, the arc is added to the dependency arc table. The current answer for the call $B_{k,i} : CP_2$ is conjoined with the description CP_1 from the program point immediately before $B_{k,i}$ to obtain the description for the program point after $B_{k,i}$. This is either used to generate a new arc event to process the next literal in the rule if $B_{k,i}$ is not the last literal; otherwise the new answer for the rule is combined with the current answer in insert_answer_info. The function get_answer processes a literal. If it is a constraint, it is simply abstractly added to the current description. If it is an atom, the current answer to that atom for the current description is looked up; then this answer is extended to the variables in the rule the literal occurs in and conjoined with the current description. The functions lookup_answer and insert_answer_info lookup an answer for a calling pattern in the answer table, and update the answer table entry when a new answer is found, respectively. The function lookup_answer also generates $newcall$ events in the case that there is no entry for the calling pattern in the answer table.

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 this analysis algorithm, a simple 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. 16). The theory of abstract interpretation guarantees that the answer table is a safe approximation of the runtime behavior (see [Bru91, HPMS00, PH96] for details). The following example shows an answer table computed by `CiaoPP`.

Example 14.1 (Abstraction) *Take the initial calling pattern*

$$\langle \texttt{create\_streams}(\texttt{X},\texttt{Y}), \{\texttt{list}(\texttt{X},\texttt{num})\}\rangle$$

which indicates that calls to `create_streams` *are performed with a list of numbers in the first argument. The answer table computed by* `CiaoPP` *contains (among others) these entries:*

Predicate	*Calling Pattern*	*Success Pattern*
`create_streams(A,B)`	`list(A,num)`	`list(A,num)`,`list(B,stream)`
`number_codes(A,B)`	`num(A)`	`num(A),list(B,numcodes)`
`generate(A,B)`	`list(A,numcodes)`	`list(A,numcodes),sf(B)`
`app(A,B,C)`	`A="/tmp/", list(B,numcodes)`	`A="/tmp/", list(B,numcodes),sf(C)`
`safe_open(A,B,C)`	`sf(A),B=write`	`sf(A),B=write,stream(B)`
`atom_codes(A,B)`	`sf(B)`	`constant(A),sf(B)`
`open(A,B,C)`	`constant(A), B=write`	`constant(A),B=write, stream(C)`

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. 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/"`*. Moreover, we use the notation* $Var = constant$ *to denote that the system generates a new type whose only element is this constant, as it happens: for* `write`*, in the entries for* `safe_open` *and* `open` *and, for* `"/tmp/"`*, in the entry for* `app`*.*

Clearly, `sf` $\sqsubseteq$ `safe_name`*. This will allow* `CiaoPP` *to infer that calls to* `open` *performed within this program satisfy the simple safety policy discussed in Example 13.1. Therefore, the information stored in the answer table is sufficient to attest the safety policy.*

In order to increase accuracy, analyzers are usually *multivariant* on calls (see, e.g., [HPMS00]). 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.

15 The Verification Condition

As part of the certification process carried out by the code producer, the verification condition generator (VCGen in Fig. 8) extracts, from the initial assertions and 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. 8), then the certificate (i.e., the abstraction) is sent together with the program P to the code consumer. Sections 14.1 and 15 give further details on the Abstraction and the VCGen process, respectively.

Definition 15.1 (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:*

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

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. 15.1 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 15.2 (Verification Condition) *Consider the answer table generated in Example 14.1*

and the $calls$ and $success$ assertions of Figure 9. According to Def. 15.1, the VC is:

$$\begin{aligned}
(\texttt{num(X)} &\sqsubseteq (\texttt{num(X)}; \texttt{list(Y, numcodes)}) \wedge \\
\texttt{sf(Y)} &\sqsubseteq (\texttt{constant(X)}; \texttt{string(Y)}) \wedge \\
\texttt{constant(X)}, \texttt{Y} = \texttt{write} &\sqsubseteq \texttt{constant(X)}, \texttt{io_mode(Y)} \wedge \\
\texttt{sf(X)} &\sqsubseteq \texttt{safe_name(X)} \wedge \\
\texttt{num(X)}, \texttt{list(Y, numcodes)} &\sqsubseteq \texttt{num(X)}, \texttt{list(Y, numcodes)} \wedge \\
\texttt{constant(X)}, \texttt{sf(Y)} &\sqsubseteq \texttt{constant(X)}, \texttt{string(Y)} \wedge \\
\texttt{constant(X)}, \texttt{Y} = \texttt{write}, \texttt{stream(Z)} &\sqsubseteq \texttt{constant(X)}, \texttt{io_mode(Y)}, \texttt{stream(Z)})
\end{aligned}$$

Each conjunct corresponds to an assertion in Fig. 9 in the same order they appear there. Thus, the first four conjuncts are for the $calls$ assertions and the last three for the $success$ assertions. The validity of the whole conjunction can be easily proved by taking into account the following (trivial) relations between the elements in the domain:

$$\begin{aligned}
\texttt{sf(X)} &\sqsubseteq \texttt{string(X)} \\
\texttt{X} = \texttt{write} &\sqsubseteq \texttt{io_mode(X)}
\end{aligned}$$

Note that the first two conjuncts contain a disjunction in the right condition. In the second one, the condition $\texttt{sf(Y)} \sqsubseteq (\texttt{constant(X)}; \texttt{string(Y)})$ *holds because* $\texttt{sf(Y)} \sqsubseteq \texttt{string(Y)}$.

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 [Nec97], we write $\triangleright VC$ when VC is valid.

Theorem 15.3 (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 Figure 10). 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. 15.1). If $\triangleright VC(AS, AT)$, 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 [HPMS00] computes a safe approximation of the stores reached during computation.

16 Checking Safety in the Consumer

The checking process performed by the consumer is illustrated in the right hand side of Fig. 8. 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 validation of AT is carried out by the Analysis Checker. 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 16.1 (Analysis Checker) *Let P be a normalized*[14] *program and Q be a set of calling patterns in the abstract domain D_α. Let* Certificate *be an answer table (or safety certificate) as defined in Figure 10. The validation of* Certificate *is performed by the procedure* check *depicted in Figure 11. The algorithm uses a local answer table, AT, to compute the results (initially it does not contain any entry).*

Following the presentation of the analysis algorithm in Section 14.2, we assume that the program P and the answer table are global parameters throughout the algorithm. The checking algorithm proceeds as follows. As in the analysis algorithm, the procedure process_arc is aimed at computing the resulting description CP_a after processing a given atom $B_{k,i}$. The computed result is used to process the next literal in the rule when $B_{k,i}$ is not the last one. Otherwise, the computed result constitutes indeed the computed answer for the rule. The difference w.r.t. the analyzer is that the answer is *combined* 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.

Example 16.2 *Consider again the program of Ex. 12.1, now in normalized form:*

```
create_streams(X,Y):- X=[],Y=[].
create_streams(X,Y):- X=[N|NL], Y=[F|FL],
```

[14] 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.

check(Q, Certificate)
  foreach $A : CP \in Q$
    process_node($A : CP$, Certificate)
  return Valid

process_node($A : CP$, Certificate)
  if ($\exists$ a renaming σ s.t. $\sigma(A : CP \mapsto AP)$ in Certificate)
    then add ($A : CP \mapsto AP$) to AT
    else return Error
  foreach rule $A_k \leftarrow B_{k,1}, \ldots, B_{k,n_k}$ in P
    W := $vars(A_k, B_{k,1}, \ldots, B_{k,n_k})$
    CP_b :=Aextend(CP, $vars(B_{k,1}, \ldots, B_{k,n_k})$)
    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($AP_1, \sigma^{-1}(AP)$)
    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 ($\nexists$ a renaming σ s.t. $\sigma(B_{k,i} : CPR_b \mapsto AP')$ in AT)
    then process_node ($B_{k,i} : CPR_b$, Certificate)
    AP_1 := Aextend ($\rho^{-1}(AP), W$) where ρ is a renaming s.t.
                $\rho(B_{k,i} : CPR_b \mapsto AP)$ in AT
    CP_a := Aconj (CP_b, AP_1)
  return CP_a

Figure 11: Abstract Interpretation-based Checking in CiaoPP

```
number_codes(N,ChInN), generate(ChInN,Fname),
safe_open(Fname,write,F), create_streams(NL,FL).
```

the calling pattern $\langle$`create_streams(X,Y)`$,\{$`list(X,num)`$\}\rangle$ *and the answer table, denoted by* Certificate*, of Ex. 14.1. We describe the more representative steps that algorithm* check *performs in order to validate the answer table. First, procedure* `process_node` *looks up an answer for the initial calling pattern in* Certificate *and adds the entry*

$\langle$`create_streams(X,Y)` : `list(X,num)` $\mapsto$ AP = `list(X,num)`, `list(Y,streams)`$\rangle$

in the answer table AT *(note that, for short, we use* AP *to denote this particular answer pattern). Since there are two rules defining* `create_streams` *the outermost loop performs two iterations:*

Iter 1. *We start by describing the processing of the first rule (although the order is irrelevant). Since the first atom* `X=[]` *in the rule body is a constraint, its description is computed within procedure* process_arc *by adding its abstract description, i.e.,* $\{$`nil(X)`$\}$*, to the initial description* $\{$`list(X,num)`$\}$*, resulting in* $\{$`nil(X)`$\}$*. Similarly, the analysis for the second constraint adds* $\{$`nil(Y)`$\}$ *to the former description producing* $\{$`nil(X),nil(Y)`$\}$*. Upon exiting the innermost loop, the disjunction of this description with the answer stored in* Certificate *is calculated:*

`AP` := `Alub` $(\{$`nil(X)`, `nil(Y)`$\}$, `AP`$)$

since `nil(X)` $\sqcup$ `list(X,num)` *and the same happens for* `Y`*. Thus, the certificate holds for this rule.*

Iter 2. *In the second iteration, we find six atoms in the rule body. Thus, the innermost loop performs the following six steps. The first two traversals deal with the constraints for* `X` *and* `Y`*, and are similar to* **Iter 1***. They produce the calling pattern* $\{$`list(X,num),rt2(Y)`$\}$ *where the auxiliary regular type* `rt2` *is created by* `CiaoPP` *to represent a term whose top-level functor is a list constructed with* `F` *as head and* `FL` *as tail. For simplicity, we just write this description as* $\{$`list(X,num),Y=[F|FL]`$\}$ *in the following.*

The next atom, `number_codes`*, in the rule body is not a constraint, thus,* process_arc *checks whether it has been processed before. Since this is not the case, it recursively executes* process_node *in order to get an answer for it. By using its predefined definition, that* process_node *gives the answer* $\{$`num(N),list(ChInN,numcodes)`$\}$ *for it. This answer is conjoined with the description of the program point inmediately before the atom, i.e.:*

$\{$`list(X,num)`, `Y` = [`F`|`FL`], `num(N)`, `list(ChInN,numcodes)`$\}$:=
`Aconj`$(\{$`num(N)`, `list(ChInN,numcodes)`$\}, \{$`list(X,num)`, `Y` = [`F`|`FL`]$\})$

Similarly, nodes `generate` *and* `safe_open` *are processed producing the final descrip-*

tion after processing safe_open*, labeled as* CP*:*

$$CP = \{\texttt{list}(\texttt{X},\texttt{num}),\texttt{Y} = [\texttt{stream}|\texttt{FL}],\texttt{num}(\texttt{N}),\ \texttt{list}(\texttt{ChInN},\texttt{numcodes}),\texttt{sf}(\texttt{Fname}),\texttt{stream}(\texttt{F})\}$$

Finally, there is another call to create_streams*. Now,* process_node *finds out that* AT *already contains an answer pattern for this predicate. Then, both calling patterns are conjoined:* $\mathrm{AP} := \texttt{Aconj}(\mathrm{CP},\mathrm{AP})$ *. Upon return from* process_arc*, it performs the disjunction of the computed answer with the answer supplied by* Certificate*:* $\mathrm{AP} := \texttt{Alub}(\mathrm{AP},\mathrm{AP})$ *. Since the result* AP *coincides with the one in the certificate, the proof is validated and the algorithm terminates in a single graph traversal for the initial query.*

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

Theorem 16.3 (partial correctness) *Let* P *be a program, let* Q *be a set of calling patterns in an abstract domain* D_α*. Let* Certificate *be an answer table* P *and* Q *as defined in Figure 10. Then,* $\texttt{check}(Q,\mathsf{Certificate})$ *terminates and validates* Certificate *in* P*.*

The theorem can be demonstrated by showing that check is a simplified version of $Analysis$ [HPMS00] in two main aspects. One is that no control structure is needed in order to guarantee that a fixpoint is reached. This eliminates the need for the "event queue" of tha analysis algorithm in Fig. 10. The second is that since only one traversal of the analysis graph is to be performed, no detailed dependency information is required. This eliminates the need for the "dependency arc table" of the analysis algorithm. As a result, check is a suitable procedure for determining the validity of the certificate.

Another issue is the efficiency of the checking algorithm. Our point to justify an efficient behavior of check for validating an answer table is that it performs a single graph traversal. Indeed, for a regular type domain, [Cha00] demonstrates that directional type-checking for logic programs is fixed-parameter linear. The next section reports experimental evidence of efficiency issues.

17 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,

	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 4: Checking Time and Certificate Size

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 [MH91]. 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 [BCC+02] 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 programs has been used as benchmarks. They are the same ones used in [HPMS00], where they are described in some detail. For each benchmark, the columns for `Analysis` are the following: $\mathtt{P}_A$ 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 $\mathtt{T}_A$. Similarly, in the case of checking, three columns are shown. The preprocessing phase, $\mathtt{P}_C$, 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, $\mathtt{T}_C$ is the total time for checking. The columns under `Speedup` compare analysis and checking times. As can be seen in columns `A/C` and $\mathtt{T}_A/\mathtt{T}_C$, 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 [PH96]. 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).

18 Resource-Aware Abstraction Carrying Code

It is well-known that abstract interpretation techniques allow inferring much richer information than, for example, traditional types. This information will allow us 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. For instance, let us assume that the consumer will only accept purely computational tasks, i.e., tasks that have no side effects, and only those of polynomial (actually, at most quadratic) complexity. This safety policy can be expressed at the producer for this particular program using the following *success* assertions:

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

The first (computational –`comp`) assertion 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.). The second (also computational) assertion 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)`). Implicitly, we are assuming that the code will be accepted at the receiving end, provided all assertions can be checked, i.e., the intended semantics expressed in the above assertions determines the safety condition. This can be a policy agreed a priori or exchanged dynamically.

Abstract interpretation-based techniques are able to reason about computational properties which can be useful for controlling efficiency issues. `CiaoPP` can infer lower and upper bounds on the sizes of terms and the computational cost of predicates [DLGHL94, DLGHL97]. Cost bounds are expressed as functions on the sizes of the input arguments and yield the number of resolution steps. Various measures can be used for the "size" of the input, such as list-length, term-size, term-depth, integer-value, etc. For instance, the answer table computed by the analyzers available in `CiaoPP` contains, among others, the following information for the above program and entry:

```
:- true pred reverse(A,B)
   : ( list(A), var(B) )
```

```
=>( list(A), list(B))
+ ( not_fails, is_det, sideff(free),
  steps_ub(0.5*exp(length(A),2)+1.5*length(A)+1)).
```

which states that if the precondition holds (after ":"), then the output is also a list and, moreover, the procedure is deterministic and does not fail, it does not contain side-effects, and calls to this procedure take at most $0.5\ (length(A))^2 + 1.5\ length(A) + 1$ resolution steps.

Given this information, the verification condition is computed as stated in Definition 15.1:

$$\{\ \texttt{steps\_ub}(0.5 * \texttt{exp}(\texttt{length}(\texttt{A}), 2) + 1.5 * \texttt{length}(\texttt{A}) + 1)),\ \texttt{not\_fails}, \texttt{is\_det}, \texttt{sideff}(\texttt{free})\} \sqsubseteq \{\texttt{steps\_ub}(\texttt{o}(\texttt{exp}(\texttt{length}(\texttt{A}), 2)))\}$$

whose validity can be easly proved by showing that:

$$0.5\ (length(A))^2 + 1.5\ length(A) + 1 \sqsubseteq O(n^2).$$

This allows CiaoPP to infer that calls to `reverse` performed within this program satisfy the resource-aware safety policy discussed. The output shows that the "status" of the three `check` assertions has become `checked`, which means that they have been validated and thus the program is safe to run (according to the intended meaning):

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

Thus, we have verified that the safety condition is met and that the code is indeed safe to run (for now on the producer side). The analysis results above can themselves be used as the *cost and safety certificate* to attest a safe and efficient use of procedure `reverse` on the receiving side.

In the consumer side, a receiver could use our method to accept/reject code which adheres/does not adhere to some specification, including usage of computing resources (in time and/or space). For instance, 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 can be used for such particular node:

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

which expresses that the consumer node will not accept an implementation of `reverse` with complexity bigger than linear. In order to guarantee that the cost assertion holds, the certificate should contain upper bounds on computational cost. Then, the code receiver proceeds to validate the certificate. The task of checking that a given expression is an upper bound is definitely simpler than that of obtaining the most accurate possible upper bound. If the certificate is not valid, the code is discarded. If it is valid, the code will be accepted only if the upper bound in the certificate is lower or equal than that stated in the assertion. In our example, the certificate contains the (valid) information that `reverse` will take at most $0.5\ (length(A))^2 + 1.5\ length(A) + 1$ resolution steps. However, the assertion requires the cost to be at most $length(A) + 1$ resolution steps. A comparison between these cost functions does not allow proving that the code received by the consumer satisfies the efficiency requirements imposed (i.e. the assertion cannot be proved).[15] This means that the consumer will reject the code.

Resource-aware ACC becomes interesting when developing software to be deployed by devices with a bounded amount of computing resources [Wei91], like in pervasive computing. Indeed, pervasive computing is characterized by having a relatively large number of *untrusted* computing devices which interact. Thus, when modeling such a system, it is not realistic to consider one device in isolation: it will receive plenty of mobile data from the environment. In this context, the *safety* of the deployed software is crucial, as the cost of recalling unfit devices can be prohibitive. On the other hand, these platforms are becoming ever smaller and more powerful, and are embedded everywhere, even in living organisms. They can contain sophisticated models of our personal environment that help us to make everyday decisions; they have the power to do mathematical and logical reasoning in order to perform intelligent tasks. As a result, verification and validation techniques are necessary but have to keep pace with the huge requirements for intelligent, user-oriented applications that must run on devices with a minimum of computing resources. In this context, there is a large number of computing devices which may range from personal computers to PDAs, mobile phones, dedicated processors, smart cards, wearable computers and such like. As a result, time *efficiency* is an issue since often these devices have to operate on real-time tasks. Also, and possibly more importantly, memory efficiency is an issue. Therefore, compile-time (and run-time) tools for the certification of CLP programs with resource consumption assurances seem to play a very promising role. As we have seen throughtout the above example, one can use resource-aware ACC in order to decide if either the received software used is too large to fit in the device or needs too much memory to run. In these cases, it is simply not possible to use such software and the code should be rejected.

[15] Indeed, the lower bound cost analysis in fact disproves the assertion, which is clearly invalid.

In spite of being essential to verify safety, when developing software for deployment on Smart Cards (and similar ambient computing devices), it is important to simplify the (safety) verification process and reduce its resource usage. Indeed, Smart Cards typically provide less than 4Kb of RAM while it is possible to use only up to 128Kb for storing the application and static data. Such resource considerations tend to dominate the development process for pervasive systems, forcing developers to write low-level code from scratch, as mobile system developers have found in their own experience. We argue that ACC (as well as PCC techniques based on certificates which are computed outside the device) constitute a good scenario for the certification of software deployed in pervasive systems. They compute tamper-proof certificates which simplify code verification and pass them along with the code. In ACC, the burden on the consumer side is reduced by using a simple one-*traversal* checker, which is a very simplified and efficient abstract interpreter which does not need to compute a fixpoint. The benchmark results in Section 17 show that the speedup achieved by the checking is approximately 1.63 in just analysis time which, we believe, makes our approach practically applicable in pervasive contexts. A similar proposal is presented in [Ros98] to split the type-based bytecode verification of the KVM (an embedded variant of the JVM) in two phases, where the producer first computes the certificate by means of a type-based dataflow analyzer and then the consumer simply checks that the types provided in the code certificate are valid. This approach is extended in [KK04] to real world Java Software. As in our case, the validation can be done in a single, linear pass over the bytecode. However, these approaches are designed limited to types, whereas our approach supports a very rich set of domains especially well-suited for this purpose, including complex properties such as computational and memory cost, non-failure, determinacy, etc. (as we have seen in the examples in this section) and possibly even combining several of them.

19 Discussion and Related Work

The idea of using the results of abstract interpretation for program verification and debugging is not new. For instance, `CiaoPP` [HPBLG03] 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. Other approaches to abstract verification and debugging have also been proposed (see [CGLV00, HPBLG03] for further references). 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 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.

A type-level dataflow analysis of Java virtual machine bytecode is also the basis of most existing verifiers [LY97, Ler03], 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 [Ros98] 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 possibly even combining several of them. 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*.

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 [WN04] which formalises a simple assembly language with procedures and presents a safety policy for arithmetic overflow in Isabelle/HOL. We argue that our proposal brings the expressiveness, flexibility and automation which is inherent in the abstract interpretation techniques developed in logic programming to this area. 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) [SVB$^+$03]. MCC enables code consumers to try out different security policies of interest and select one that can be statically proved to be consistent with the model associated to the untrusted code. However, models are intended to describe low-level properties and their combination has not been studied, which differs from our idea of combining (high-level) abstract domains.

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). Clearly, in many cases the source code is simply not available to the consumer and even when there is a choice between object and source code, using object code means reducing the trusted computing base in the consumer since there is no need for a compiler. However, open-source code is becoming much more relevant these days (in fact, `Ciao` and `CiaoPP` are themselves GNU-licensed and available in source code for reviewing and modification). 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. This allows handling richer properties which in turn potentially allow more expressive safety policies.

Part IV

Some Techniques for Automated, Resource-Aware Distributed and Mobile Computing in a Multi-Paradigm Programming System

20 Introduction

Distributed parallel execution systems speed up applications by splitting tasks into processes whose execution is assigned to different nodes in a high-bandwidth network. GRID systems [FKNT99] in particular attempt to use for this purpose widely distributed sets of machines, often crossing several administrative domain boundaries. Many interesting challenges arise in this context.

A number of now classical problems have to be solved when this process is viewed from the *producer side*, i.e., from the point of view of the machine in charge of starting and monitoring a particular execution of a given application (or a part of such an application) by splitting the tasks into processes whose execution is assigned to different nodes (i.e., *consumers*) on receiving sides of the network. A fundamental problem involved in this process is detecting which tasks composing the application are independent and can thus be executed in parallel. Much work has been done in the areas of parallelizing compilers and parallel languages in order to address this problem. While obviously interesting, herein we will concentrate instead on other issues.

In this sense, a second fundamental problem, and which has also received considerable attention (even if less than the previous one), is the problem of grouping and scheduling such tasks, i.e., assigning tasks to remote processors, and very specially the particular issue of ensuring that the tasks involve sufficient computational cost when compared to the task creation and communication costs and other such practical overheads. Due to these overheads, and if the *granularity* of parallel tasks (i.e., the work necessary for their complete execution) is too small, it may happen that the costs are larger than the benefits of their parallel execution. Of course, the concept of small granularity is relative: it depends on the concrete system or set of systems where parallel programs are running. Thus, a *resource-aware* method has to be devised whereby the granularity of parallel tasks and their number can be controlled. We will call this the *task scheduling*

and granularity control problem. In order to ensure that effective speedup can be obtained from remote execution it is obviously desirable to devise a solution where load and task distribution decisions are made automatically, specially in the context of non-embarrassingly parallel and/or irregular computations in which hand-coded approaches are difficult and tedious to apply.

Interestingly, when viewed from the *consumer side*, and in an open setting such as that of the GRID and other similar overlay computing systems, additional and novel challenges arise. In more traditional distributed parallelism situations (e.g., on clusters) 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. In a more general setting, the administrative domain of the receiver can be completely different from that of the producer. Moreover, the receiver is possibly being used for other purposes (e.g., as a general-purpose workstation) in addition to being a party to the distributed computation. In this 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 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 more traditional safety issues (e.g., that it will not write on specific areas of the disk) or *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). 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.

Recent approaches to mobile code safety involve associating safety information in the form of a *certificate* to programs [Nec97, LY97, MWCG99, APH04]. 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 *verifier* 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. It appears interesting to devise 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*.

In this paper we present in a tutorial way a number of general solutions to these problems, and illustrate them through their implementation in the context of a multi-paradigm language and program development environment that we have developed, `Ciao` [BCC+02]. This system includes facilities for parallel and distributed execution, an assertion language for specifying

complex programs properties (including safety and resource-related properties), and compile-time and run-time tools for performing automated parallelization and resource control, as well as certification of programs and efficient checking of such certificates.

Our system allows coding complex programs combining the styles of logic, constraint, functional, and a particular version of object-oriented programming. Programs which include logic and constraint programming (CLP) constructs have been shown to offer a particularly interesting case for studying the issues that we are interested in [Her97]. These programming paradigms pose significant challenges to parallelization and task distribution, which relate closely to the more difficult problems faced in traditional parallelization. This includes the presence of highly irregular computations and dynamic control flow, non-trivial notions of independence, the presence of dynamically allocated, complex data structures containing pointers, etc. In addition, 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 including independence, bounds on data structure sizes, computational cost, etc.

After first reviewing our approach to solving the granularity control problem using program analysis and transformation techniques, we propose a technique for resource-aware security in mobile code based on safety certificates which express properties related to resource usage. Intuitively, we use the granularity information (computed by the cost analysis carried out to decide the distribution of tasks on the producer side) in order to generate so-called *cost certificates* which are packaged along with the untrusted code. The idea is that the receiving side can 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.

The rest of the paper proceeds as follows. After briefly presenting in Section 21 the basic techniques used for inferring complex properties in our approach, including upper and lower bounds on resource usage, Section 22 reviews our approach to the use of bounds on data structure sizes and computational cost to perform automatic granularity control. Section 23 then discusses our approach to resource-aware mobile code certification. Section 24 finally presents our conclusions.

21 Inferring Complex Properties Including Term Sizes and Costs

In order to illustrate our approach in a concrete setting, we will use `CiaoPP` [HPBLG03] throughout the paper. `CiaoPP` is a component of the `Ciao` programming environment which performs several tasks including *automated parallelization and resource control*, as well as *certification* of programs, and efficient *checking* of such certificates. `CiaoPP` uses throughout the now well-established technique of *abstract interpretation* [CC77]. This technique has allowed the development of very sophisticated global static program analyses which 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 program behavior. The technique allows inferring much richer information than, for example, traditional types. The fact that at the heart of `Ciao` lies an efficient logic programming-based kernel language allows the use in `CiaoPP` of the very large body of approximation domains, inference techniques and tools for abstract interpretation-based semantic analysis which have been developed to a powerful and mature level in this area (see, e.g., [Bru91, MH92, CC92, HPMS00] and their references) and which are integrated in `CiaoPP`. As a result of this, `CiaoPP` can infer at compile-time, always safely, and with a significance degree of precision, a wide range of *properties* such as data structure shape (including pointer sharing), bounds on data structure sizes, determinacy, termination, non-failure, bounds on resource consumption (time or space cost), etc.

All this information is expressed by the compiler using *assertions*: syntactic objects which allow expressing "abstract"—i.e. symbolic—properties over different abstract domains. In particular, we use the high-level assertion language of [PBH00b], which actually implements a two-way communication with the system: it allows providing information to the analyzer as well as representing its results.

As a very simple example, consider the following procedure `inc_all`/2, which increments all elements of a list by adding one to each of them (we use functional notation for conciseness):

```
inc_all([])    := [].
inc_all([H|T]) := [ H+1 | inc_all(T)].
```

Assume that analysis of the rest of the program has determined that this procedure will be called providing a list of numbers as input. The output from `CiaoPP` for this program then includes the following assertion:

```
:- true pred inc_all(A,B)
```

```
 : ( list(A,num), var(B) )
=> ( list(A,num), list(B,num), size_lb(B,length(A))
 + ( not_fails, is_det, steps_lb(2*length(A)+1)).
```

Such "`true pred`" assertions specify in a combined way properties of both: "`:`" the entry (i.e., upon calling) and "`=>`" the exit (i.e., upon success) points of all calls to the procedure, as well as some global properties of its execution. The assertion expresses that procedure `inc_all` 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 $2\ length(A) + 1$ execution steps (where the cost measure used in the example is 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. 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.

As can be seen from the example, in our approach 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 measures 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.

While it is beyond the scope of this paper to fully explain all the (generally abstract interpretation-based) techniques involved in this process (see, e.g., [HPBLG03, DLGHL94, DLGHL97] and their references), we illustrate through a simple example the fundamental intuition behind our lower bound cost estimation technique.

Consider again the simple `inc_all` procedure above and the assumption that type and mode inference has determined that it will be called providing a list of numbers as input. Assume again that the cost unit is the number of procedure calls. In a first approximation, and for simplicity, we also assume that the cost of performing an addition is the same as that of a procedure call. With these assumptions the exact cost function of procedure `inc_all` is $\texttt{Cost}_{inc\_all}(n) = 2\ n + 1$, where n is the size (length) of the input list.

In order to obtain a lower bound approximation of the previous cost function, `CiaoPP`'s analyses first determine, based on the mode and type information inferred, that the argument size metric to be used is list length. An interesting problem with estimating lower bounds is that 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. For this reason, the lower bound cost analyzer uses information inferred by non-failure analysis [DLGH97], which can detect procedures and goals that can be guaranteed not to fail, i.e., to produce at least one solution (which would indeed be the case for `inc_all`) or not terminate.

In general, in order to determine the work done by (recursive) clauses, it is 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. For this, we use an abstraction of procedure definitions called a data dependency graph. Our approach to cost analysis consists of the following steps:

1. Use data dependency graphs to determine the relative sizes of variable bindings at different program points.

2. Use the size information to set up difference equations representing the computational cost of procedures

3. Compute lower/upper bounds to the solutions of these difference equations to obtain estimates of task granularities.

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 the `inc_all` procedure let $\mathtt{Size}^2_{\mathtt{inc\_all}}(n)$ denote the size of the output argument (the second) as a function of the size of its first (input) argument n. Once we have 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 `inc_all`/2:

$\mathtt{Size}^2_{\mathtt{inc\_all}}(0) = 0$ (boundary condition from base case),

$\mathtt{Size}^2_{\mathtt{inc\_all}}(n) = 1 + \mathtt{Size}^2_{\mathtt{inc\_all}}(n-1)$.

The solution of this difference equation obtained is $\mathtt{Size}^2_{\mathtt{inc\_all}}(n) = n$.

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 `inc_all`/2:

$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(0) = 1$ (boundary condition from base case),

$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n) = 1 + \mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n-1)$.

The solution obtained for this difference equation is $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n) = 2\,n + 1$. In this case, the lower bound inferred is the exact cost (the upper bound cost analysis also infers the same function). In our approach, sometimes the solutions of the difference equations need to be in fact approximated by a lower bound (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.

22 Controlling Granularity in Distributed Computing

As mentioned in Section 20, and in view of the techniques introduced in Section 21, we now discuss the task scheduling and granularity control problem, assuming that the program is already parallelized.[16] The aim of such distributed granularity control is to replace parallel execution with sequential execution or vice-versa based on some conditions related to task size and overheads. The benefits from controlling parallel task size will obviously be greater for systems with greater parallel execution overheads. In fact, in many architectures (e.g. distributed memory multiprocessors, workstation "farms", GRID systems, etc.) such overheads can be very significant and in them automatic parallelization cannot in general be done realistically without granularity control. In some other architectures where the overheads for spawning goals in parallel are small (e.g. in small shared memory multiprocessors) granularity control is not essential but it can also achieve important improvements in speedup.

Granularity control has been studied in the context of traditional programming [KL88, MG89], functional programming [Hue93, HLA94], and also logic programming [Kap88, DLH90, ZTD+92, DL93, LGHD94, LGHD96]. In [LGHD96] we proposed a general granularity control model and reported on its application to the case of logic programs. This model proposes (efficient) conditions based on the use of information available on task granularity in order to choose between parallel and sequential execution. The problems to be solved in order to perform granularity control following this approach include, on one hand, estimating the cost of tasks, of the overheads associated with their parallel execution, and of the granularity control technique itself. On the other hand there is also the problem of devising, given that information, efficient compile-time and run-time granularity control techniques.

Performing accurate granularity control at compile-time is difficult because some of the information needed to evaluate communication and computational costs, as for example input data

[16] In the past two decades, quite significant progress has been made in the area of automatically parallelizing programs in the context of logic and constraint programs, and some of the challenges have been tackled quite effectively there –see, for example, [GPA+01, Her97, CC94] for an overview of this area.

size, is only known at run-time. A useful strategy is to do as much work as possible at compile-time, and postpone some final decisions to run-time. This can be achieved by generating at compile-time cost functions which estimate task costs as a function of input data size, which are then evaluated at run-time when such size is known. Then, after comparing costs of sequential and parallel executions (including all overheads), it is possible to determine which type of execution is profitable.

The approximation of these cost functions can be based either on some heuristics (e.g., profiling) or on a *safe* approximation (i.e. an upper or lower bound). We were able to show that if upper or lower bounds on task costs are available, under a given set of assumptions, it is possible to ensure that some parallel, distributed executions will always produce speedup (and also that some others are best executed sequentially). Because of these results, we will in general require the cost information to be not just an approximation, but rather a well-defined bound on the actual execution cost. In particular, we will use the techniques for inferring upper- and lower-bound cost functions outlined in the previous section.

Assuming that such functions or similar techniques for determining task costs and overheads are given, the remainder of the granularity control task is to devise a way to actually compute such costs and then dynamically control task creation and scheduling using such information. Again the approach of doing as much of the work as possible at compile-time seems advantageous. In our approach, a transformation of the program is performed at compile time such that the cost computations and spawning decisions are encoded in the program itself, and in the most efficient way possible. The idea is to perform any remaining computations and decisions at run-time when the parameters missing at compile-time, such as data sizes or node load are available. In particular, the transformed programs will perform (generally) the following tasks: computing the sizes of data that appear in cost functions; evaluating the cost functions of the tasks to be executed in parallel using those data sizes; safely approximating the spawning and scheduling overheads (often also a function of data sizes); comparing these quantities to decide whether to schedule tasks in parallel or sequentially; deciding whether granularity control should be continued or not; etc.

As an example, consider the `inc_all` procedure of Section 21 and the program expression:

```
..., Y = inc_all(X) & M = r(Z), ...
```

which indicates that the procedure call `inc_all(X)` is to be made available for execution in parallel with the call to `r(Z)` (we assume that analysis has determined that `inc_all(X)` and `r(Z)` are independent, by, e.g., ensuring that there are no pointers between the data structures pointed to by `X,Y` and `Z,M`. From Section 21 we know that the cost function inferred for `inc_all` is

$\texttt{Cost}^{L}_{inc\_all}(n) = 2\,n + 1$. Assume also that the cost of scheduling a task is constant and equal to 100 computation steps. The previous goal would then be transformed into the following one:

```
..., ( 2*length(X)+1 > 100 -> Y = inc_all(X) & M = r(Z)
                            ;  Y = inc_all(X),  M = r(Z) ), ...
```

where (*if* -> *then* ; *else*) is syntax for an if-then-else and "," denotes sequential execution as usual. Thus, when $2{*}length(X){+}1$ (i.e., the lower bound on the cost of `inc_all(X)`) is greater than the threshold, the task is made available for parallel execution and not otherwise. Many optimizations are possible. In this particular case, the program expression can be simplified to:

```
..., ( length(X) > 50 -> Y = inc_all(X) & M = r(Z)
                       ;  Y = inc_all(X),  M = r(Z) ), ...
```

and, assuming that `length_gt(L,N)` succeeds if the length of `L` is greater than `N` (its implementation obviously only requires to traverse at most the n first elements of list), it can be expressed as:

```
..., ( length_gt(LX,50) -> Y = inc_all(X) & M = r(Z)
                         ;  Y = inc_all(X),  M = r(Z) ), ...
```

As mentioned before, scheduling costs are often also a function of data sizes (e.g., communication costs). For example, assume that the cost of executing remotely `Y = inc_all(X)` is $0.1\ (length(X) + length(Y))$, where $length(Y)$ is the size of the result, an upper bound on which (actually, exact size) we know to be $length(X)$. Thus, our comparison would now be:

$$\begin{aligned} 2\,length(X) + 1 > 0.1\ (length(X) + length(Y)) &\equiv \\ 2\,length(X) + 1 > 0.1\ (length(X) + length(X)) &\equiv \\ 2\,length(X) + 1 > 0.2\,length(X) &\cong \\ 2\,length(X) > 0.2\,length(X) &\equiv \\ 2 > 0.2 & \end{aligned}$$

Which essentially means that the task can be scheduled for parallel execution *for any input size*. Conversely, with a communication cost greater than $0.5(length(X)+length(Y))$ the conclusion would be that it would never be profitable to run in parallel.

These ideas have been implemented and integrated in the `CiaoPP` system, which uses the information produced by its analyzers to perform combined compile–time/run–time resource control. The more realistic example in Figure 12 (a quick-sort program coded using logic programming) illustrates additional optimizations performed by `CiaoPP` in addition to cost function simplification, which include improved term size computation and stopping performing granularity control below certain thresholds. The concrete transformation produced by `CiaoPP` adds

```
:- module(qsort, [qsort/2], [assertions]).

qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E < C,  partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E >= C, partition(R,C,Left,Right1).

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
```

Figure 12: A qsort program.

```
g_qsort([X|L],R) :-
        partition_o3_4(L,X,L1,L2,S1,S2),
        ( S2>7 -> (S1>7 -> g_qsort(L2,R2) & g_qsort(L1,R1)
                         ; g_qsort(L2,R2), s_qsort(L1,R1))
                ; (S1>7 -> s_qsort(L2,R2), g_qsort(L1,R1)
                         ; s_qsort(L2,R2), s_qsort(L1,R1))),
        append(R1,[X|R2],R).
g_qsort([],[]).
```

Figure 13: The qsort program transformed for granularity control

a clause: "`qsort(X1,X2) :- g_qsort(X1,X2).`" (to preserve the original entry point) and produces `g_qsort/2`, the version of `qsort/2` that performs granularity control (where `s_qsort/2` is the sequential version) is shown in Figure 13.

Note that if the lengths of the two input lists to the recursive calls to `qsort` are greater than a threshold (a list length of 7 in this case) then versions which continue performing granularity control are executed in parallel. Otherwise, the two recursive calls are executed sequentially. The executed version of each such call depends on its grain size: if the length of its input list is not greater than the threshold then a sequential version which does not perform granularity control is executed. This is based on the detection of a recursive invariant: in subsequent recursions this goal will not produce tasks with input sizes greater than the threshold, and thus, for all of them,

execution should be performed sequentially and, obviously, no granularity control is needed. Procedure `partition_o3_4/6`:

```
partition_o3_4([],_B,[],[],0,0).
partition_o3_4([E|R],C,[E|Left1],Right,S1,S2) :-
        E<C, partition_o3_4(R,C,Left1,Right,S3,S2), S1 is S3+1.
partition_o3_4([E|R],C,Left,[E|Right1],S1,S2) :-
        E>=C, partition_o3_4(R,C,Left,Right1,S1,S3), S2 is S3+1.
```

is the transformed version of `partition/4`, which "on the fly" computes the sizes of its third and fourth arguments (the automatically generated variables `S1` and `S2` represent these sizes respectively) [LGH95].

23 Resource-Aware Mobile Computing

Having reviewed the issue of granularity control, and following the classification of issues of Section 20 we now turn our attention to some resource-related issues on the receiver side. In an open setting, such as that of the GRID and other similar overlay computing systems, receivers must have some assurance that the received code is safe to run, i.e., that it adheres to some conditions (the *safety policy*) regarding what it will do. We follow current approaches to mobile code safety, based on the technique of *Proof-Carrying Code* (PCC) [Nec97], which as mentioned in Section 20 associate safety *certificates* to programs. A certificate (or proof) is created by the code supplier for each task at compile time, and packaged along with the untrusted mobile code sent to (or taken by) other nodes in the network. The consumer node who receives or takes the code+certificate package (plus a given task to do within that code) can then run a *checker* which by a straightforward inspection of the code and the certificate can verify the validity of the certificate and thus compliance with the safety policy. The key benefit of this approach is that the consumer is given by the supplier the capacity of ensuring compliance with the desired safety policy in a simple and efficient way. Indeed the (proof) checker used at the receiving side performs a task that should be much simpler, efficient, and automatic than generating the original certificate. For instance, in the first PCC system [Nec97], the certificate is originally a proof in first-order logic of certain *verification conditions* and the checking process involves ensuring that the certificate is indeed a valid first-order proof.

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

- allow expressing interesting safety *properties*,

- can be generated *automatically* and,
- are easy and *efficient* to check.

Our approach to mobile code safety [APH04] directly addresses these problems. It uses approximation techniques, generally based on abstract interpretation, and it has been implemented using the facilities available in `CiaoPP` and discussed in the previous sections. These techniques offer a number of advantages for dealing with the aforementioned issues. The expressiveness of the properties that can be handled by the available abstract domains (and which can be used in a wide variety of assertions) will be implicitly available to define a wide range of safety conditions covering issues like independence, types, freeness from side effects, access patterns, bounds on data structure sizes, bounds on cost, etc. Furthermore, the approach inherits the inference power of the abstract interpretation engines to automatically generate and validate the certificates. In the following, we review our standard mobile code certification process and discuss the application in parallel distributed execution.

Certification in the Supplier: The certification process starts from an initial program and a set of assertions provided by the user on the producer side, which encode the safety policy that the program should meet, and which are to be verified. Consider for example the following (naive) reverse program (where `append` is assumed to be defined as in Figure 12):

```
:- entry reverse/2 : list * var.
reverse( [] )    := [].
reverse( [H|L] ) := ~append( reverse(L), [H] ).
```

Let us assume also that we know that the consumer will only accept purely computational tasks, i.e., tasks that have no side effects, and only those of polynomial (actually, at most quadratic) complexity. This safety policy can be expressed at the producer for this particular program using the following assertions:

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

The first (computational –`comp`) assertion 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.). The second (also computational) assertion states that it should be verified that 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)`). Implicitly, we are assuming that the code will be accepted at the receiving end, provided all assertions can be checked, i.e., the intended semantics expressed in the above assertions determines the safety condition. This can be a policy agreed a priori or exchanged dynamically.

Note that, unlike traditional safety properties such as, e.g., type correctness, which can be regarded as platform independent, resource-related properties should take into account issues such as load and available computing resources in each particular system. Thus, for resource-related properties different nodes may impose different policies for the acceptance of tasks (mobile code).

Generation of the Certificate: In our approach, given the previous assertions defining the safety policy, the certificate is automatically generated by an *analysis engine* (which in the particular case of `CiaoPP` is based on the *goal dependent*, i.e., context-sensitive, analyzer of [HPMS00]). This analysis algorithm receives as input a set of entries (included in the program like the `entry` assertion of the example above) which define the base, boundary assumptions on the input data. These base assumptions can be checked at run-time on the actual input data (in our example the type of the input is stated to be a list). The computation of the analysis process terminates when a fixpoint of a set of equations is reached. Thus, the results of analysis are often called the *analysis fixpoint*.

Due to space limitations, and given that it is now well understood, we do not describe here the analysis algorithm (details can be found in, e.g., [Bru91, HPMS00]). The important point to note is that the certification process is based on the idea that the role of certificate can be played by a *particular and small subset of the analysis results* (i.e., of the analysis fixpoint) computed by abstract interpretation-based analyses.

For instance, the analyzers available in `CiaoPP` infer, among others, the following information for the above program and entry:

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

stating that the output is also a list, that the procedure is deterministic and will not fail, that it does not contain side-effects, and that calls to this procedure take at most $0.5\ (length(A))^2 + 1.5\ length(A) + 1$ resolution steps. In addition, given this information, the output shows that the

"status" of the three `check` assertions has become `checked`, which means that they have been validated and thus the program is safe to run (according to the intended meaning):

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

Thus, we have verified that the safety condition is met and that the code is indeed safe to run (for now on the producer side). The analysis results above can themselves be used as the *cost and safety certificate* to attest a safe and efficient use of procedure `reverse` on the receiving side.

In general the verification process requires first generating a *verification condition* [APH04] that encodes the information in the check assertions to be verified and then checking this condition against the information available from analysis. This validation may yield three different possible status: i) the verification condition is indeed checked and the fixpoint is considered a *valid certificate*, 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); and iii) it cannot be proved nor disproved. Case iii) occurs because the most interesting properties are in general undecidable. The analysis process in order to always terminate is based on approximations, and may not be able to infer precise enough information to verify the conditions. The user can then provide a more refined description of initial entries or choose a different, finer-grained, abstract domain. However, despite the inevitable theoretical limitations, the analysis algorithms and abstract domains have been proved very effective in practice. In both the ii) and iii) cases, the certification process needs to be restarted until achieving a verification condition which meets i). If it succeeds, the fixpoint constitutes a valid certificate and can be sent to the receiving side together with the program.

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

As it became apparent in the above example, the interesting point to note is that abstract interpretation-based techniques are able to reason about computational properties which can be

useful for controlling efficiency issues in a mobile computing environment and in distributed parallelism platforms. We consider the case of the receiver of a task in a parallel distributed system such as a GRID. This receiver (the code consumer) could use this method to reject code which does not adhere to some specification, including usage of computing resources (in time and/or space). Reconsider for example the previous reverse program and assume that a node with very limited computing resources is assigned to perform a computation using this code. Then, the following "check" assertion can be used for such particular node:

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

which expresses that the consumer node will not accept an implementation of `reverse` with complexity bigger than linear. In order to guarantee that the cost assertion holds, the certificate should contain upper bounds on computational cost. Then, the code receiver proceeds to validate the certificate. The task of checking that a given expression is an upper bound is definitely simpler than that of obtaining the most accurate possible upper bound. If the certificate is not valid, the code is discarded. If it is valid, the code will be accepted only if the upper bound in the certificate is lower or equal than that stated in the assertion. In our example, the certificate contains the (valid) information that `reverse` will take at most $0.5\ (length(A))^2 + 1.5\ length(A) + 1$ resolution steps. However, the assertion requires the cost to be at most $length(A) + 1$ resolution steps. A comparison between these cost functions does not allow proving that the code received by the consumer satisfies the efficiency requirements imposed (i.e. the assertion cannot be proved).[17] This means that the consumer will reject the code. Similar results would be obtained if the worst case complexity property `steps_ub( o(length(A)) )` was used in the above check assertion, instead of `steps_ub( length(A) + 1 )`.

Finally, and interestingly, note that the certificate can also be used to approximate the *actual costs of execution* and make decisions accordingly. Since the code receiver knows the data sizes, it can easily apply them to the cost functions (once they are verified) and obtain values that safely predict the time and space that the task received will consume.

24 Conclusions

We have presented an abstract interpretation-based approach to resource-aware distributed and mobile computing and discussed their implementation in the context of a multi-paradigm pro-

[17] Indeed, the lower bound cost analysis in fact disproves the assertion, which is clearly invalid.

gramming system. Our framework uses modular, incremental, abstract interpretation as a fundamental tool to infer resource and safety information about programs. We have shown this information, including lower bounds on cost and upper bounds on data sizes, can be used to perform high-level optimizations such as resource-aware task granularity control. Moreover, cost information and, in particular, upper bounds, inferred during the previous process are relevant to certifying and validating mobile programs which may have constraints in terms of computing resources (in time and/or space). In essence, we believe that our proposals can contribute to bringing increased flexibility, expressiveness and automation of important resource-awareness aspects in the area of mobile and distributed computing.

References

[AF99] A. Appel and A. Felty. Lightweight Lemmas in lambda-Prolog. In *Proc. of ICLP'99*, pages 411–425. MIT Press, 1999.

[AGH+04] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, LNCS. Springer, 2004. To appear.

[AM94] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

[AP93] K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 1(106):109–157, 1993.

[APH04] 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.

[BCC+02] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). The ciao system documentation series–TR, School of Computer Science, Technical University of Madrid (UPM), June 2002. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[BCMH94] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of prolog. In *Proc. International Symposium on Logic Programming*, pages 457–471, Ithaca, NY, November 1994. MIT Press.

[BDD+97] 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.

[BdlBH94] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[BDM97] J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.

[BL02] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In *Proc. of CADE'02*, pages 31–46. Springer LNCS, 2002.

[BLGPH04] F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP1/04, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2004.

[Bou93] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.

[Bru87] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[CC77] 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.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 and 3):103–179, 1992.

[CC94] J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[CDMV93] B. Le Charlier, O. Degimbe, L. Michael, and P. Van Hentenryck. Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, pages 15–26. Springer-Verlag, September 1993.

[CGLV00] M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract Interpretation based Verification of Logic Programs. *Electr. Notes Theor. Comput. Sci.*, 30(1), 2000.

[CH00] D. Cabeza and M. Hermenegildo. The Ciao Module System: A New Module System for Prolog. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[Cha00] W. Charatonik. Directional Type Checking for Logic Programs: Beyond Discriminative Types. In *Proc. of ESOP 2000*, pages 72–87. LNCS 1782, 2000.

[CLMV96] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.

[CLMV99] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.

[CLV95] M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.

[CV94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[Deb89] 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.

[Deb92] S. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.

[Der93] P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

[DHM00] P. Deransart, M. Hermenegildo, and J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming*. Number 1870 in LNCS. Springer-Verlag, September 2000.

[DL90] S.K. Debray and N.-W. Lin. Static Estimation of Query Sizes in Horn Programs. In *Third International Conference on Database Theory*, Lecture Notes in Computer Science 470, pages 515–528, Paris, France, December 1990. Springer-Verlag.

[DL93] S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[dlBHB+96] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, September 1996.

[DLGH97] S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[DLGHL94] S.K. 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.

[DLGHL97] S.K. 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.

[DLH90] S.K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[DNTM88] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.

[DNTM89] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[DRRS93] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Extracting Determinacy in Logic Programs. In *1993 International Conference on Logic Programming*, pages 424–438. MIT Press, June 1993.

[DW88] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

[DZ92] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[Fer87] G. Ferrand. Error diagnosis in logic programming. *J. Logic Programming*, 4:177–198, 1987.

[FKNT99] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[FSVY91] T. Früwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. LICS'91*, pages 300–309, 1991.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[GP02] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in LNCS, pages 243–261. Springer-Verlag, January 2002.

[GPA+01] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

[HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 Int'l. Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[Her97] M. Hermenegildo. Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming. In *Proceedings of EUROPAR'97*, volume 1300 of *LNCS*, pages 31–46. Springer-Verlag, August 1997.

[Her00] M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

[HL94] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.

[HLA94] L. Huelsbergen, J. R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. In *Proc. ACM Conf. on Lisp and Functional Programming*, June 1994.

[HPB99] 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.

[HPBLG03] 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.

[HPMS00] 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.

[Hue93] L. Huelsbergen. Dynamic Language Parallelization. Technical Report 1178, Computer Science Dept. Univ. of Wisconsin, September 1993.

[HWD92] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[JS87] N. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, 1987.

[Kap88] S. Kaplan. Algorithmic Complexity of Logic Programs. In *Logic Programming, Proc. Fifth International Conference and Symposium, (Seattle, Washington)*, pages 780–793, 1988.

[KK04] K. Klohs and U. Kastens. Memory Requirements of Java Bytecode Verification on Limited Devices. In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, 2004.

[KL88] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, January 1988.

[Ler03] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[LGH95] P. López-García and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.

[LGHD94] P. López-García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

[LGHD96] P. López-García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[LS88] Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington, August 1988. MIT.

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[Mel86] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.

[MG89] C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32, 1989.

[MH90] 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.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] 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.

[MSJ94] K. Marriott, H. Søndergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[MU87] H. Mannila and E. Ukkonen. Flow Analysis of Prolog Programs. In *Fourth IEEE Symposium on Logic Programming*, pages 205–214, San Francisco, California, September 1987. IEEE Computer Society.

[MWCG99] 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.

[Nec97] G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.

[NL98] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proc. of the '98 Conference on Programming Language Design and Implementation*. ACM Press, 1998.

[NR01] G.C. Necula and S.P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of POPL'01*, pages 142–154. ACM Press, 2001.

[PBH00a] 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.

[PBH00b] 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.

[PBH00c] 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.

[PH96] 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.

[Ros98] K. Rose, E. Rose. Lightweight bytecode verification. In *OOPSALA Workshop on Formal Underpinnings of Java*, 1998.

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

[ST84] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[SVB+03] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proc. of SOSP'03*, pages 15–28. ACM, 2003.

[VB02] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.

[VD92] P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[Vet94] E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Contraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.

[Wae88] A. Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 700–710, Seattle,Washington, August 1988.

[Wei91] M. Weiser. The computer for the twenty-first century. *Scientific American*, 3(265):94–104, September 1991.

[WHD88] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying Machine Code Safety: Shallow Versus Deep Embedding. In *17th Int. Conference on Theorem Proving in Higher Order Logics*, number 3223 in LNCS. Springer, 2004.

[YS87] E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.

[ZTD+92] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.