Interval-based Resource Usage Verification: Formalization and Prototype

Pedro Lopez-Garcia[1,2], Luthfi Darmawan[1], Francisco Bueno[3], and Manuel Hermenegildo[1,3]

[1] IMDEA Software Institute, Madrid, Spain
[2] Spanish National Research Council (CSIC), Spain
[3] Technical University of Madrid, Spain
{pedro.lopez,luthfi.darmawan,manuel.hermenegildo}@imdea.org
bueno@fi.upm.es

Abstract. In an increasing number of applications (e.g., in embedded, real-time, or mobile systems) it is important or even essential to ensure conformance with respect to a specification expressing the use of some resource, such as execution time, energy, or user-defined resources. In previous work we have presented a novel framework for data size-dependent, static resource usage verification (which can also be combined with run-time tests). Specifications can include both lower and upper bound resource usage functions. In order to statically check such specifications, both upper- and lower-bound resource usage functions (on input data sizes) approximating the actual resource usage of the program are automatically inferred and compared against the specification. The outcome of the static checking of assertions can express *intervals* for the input data sizes such that a given specification can be proved for some intervals but disproved for others. After an overview of the approach, in this paper we provide a number of novel contributions: we present a more complete formalization and we report on and provide results from an implementation within the Ciao/CiaoPP framework (which provides a general, unified platform for static and run-time verification, as well as unit testing). We also generalize the checking of assertions to allow preconditions expressing intervals within which the input data size of a program is supposed to lie (i.e., intervals for which each assertion is applicable), and we extend the class of resource usage functions that can be checked.

Key words: Cost Analysis, Resource Usage Analysis, Resource Usage Verification, Program Verification and Debugging.

1 Introduction and Motivation

The conventional understanding of software correctness is the conformance to a functional or behavioral specification, i.e., with respect to what the program is supposed to compute or do. However, in an increasing number of applications, particularly those running on devices with limited resources, it is also important and sometimes essential to ensure conformance with respect to specifications expressing the use of some resource (such as execution time, energy, or user-defined resources). For example, in a real-time application, a program completing an action later than required is as erroneous as a program not computing the correct answer. The same applies to an embedded application in a battery-operated device (e.g., in the medical or mobile phone domains) which makes the device run out of batteries earlier than required, thus making the whole system useless.

In [13] we proposed techniques that extended the capacity of debugging and verification systems based on static analysis [4, 2, 11] when dealing with a quite general class of properties related to resource usage. This includes upper and lower bounds on execution time, energy, and user-defined resources (the latter in the sense of [19, 18]). Such bounds are given as functions on input data sizes (see [19] for some metrics that can be used for data sizes, such as list-length, term-depth or term-size). For example, the techniques of [13] extended the capacities already present in CiaoPP for certifying programs with resource consumption assurances and also for checking such certificates [10, 11], in terms of both power and efficiency. We also defined an abstract semantics for resource usage properties and described operations to compare the (approximated) intended semantics of a program (i.e., the specification, given as assertions in the program [20]) with approximated semantics inferred by static analysis, all for the case of resources, beyond [21]. These operations include the comparison of arithmetic functions (in particular, for [13], polynomial and exponential functions).

In traditional static checking-based verification (e.g., [4]), for each property or (part of) an assertion, the possible outcomes are *true* (property proved to hold), *false* (property proved not to hold), and *unknown* (the analysis cannot prove true or false). However, it is very common that cost functions have intersections, so that for a given interval of input data sizes, one of them is smaller than the other one, but for another interval it is the other way around. Consequently, a novel aspect of the resource verification and debugging approach proposed in [13] is that the *answers* of the checking process go beyond the three classical outcomes and typically include conditions under which the truth or falsity of the property can be proved. Such conditions can be parameterized by attributes of inputs, such as input data size or value *ranges*. For example, it may be possible to say that the outcome is true if the input data size is in a given range and false if it is in another one.

Consider for example the naive reverse program in Figure 1, with the classical definition of predicate `append`. The assertion (see [20] for more details on the Ciao assertion language):

```
:- check comp nrev(A,B) + (cost(lb, steps, length(A)),
                           cost(ub, steps, 10*length(A))).
```

is a resource usage specification to be checked by CiaoPP. It uses the `cost/3` property for expressing a resource usage as a function on input data sizes (third argument) for a particular resource (second argument), approximated in the way expressed by the first argument (e.g., `lb` for lower bounds and `ub` for upper bounds). The assertion expresses both an upper and a lower bound for

```
:- module(rev, [nrev/2], [assertions,regtypes,
                          nativeprops,predefres(res_steps)]).

:- entry nrev(A,B) : (list(A, gnd), var(B)).
:- check comp nrev(A,B)
     + (cost(lb, steps, length(A)), cost(ub, steps, 10*length(A))).

nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), append(R1,[H],R).
```

Fig. 1. A module for the naive reverse program.

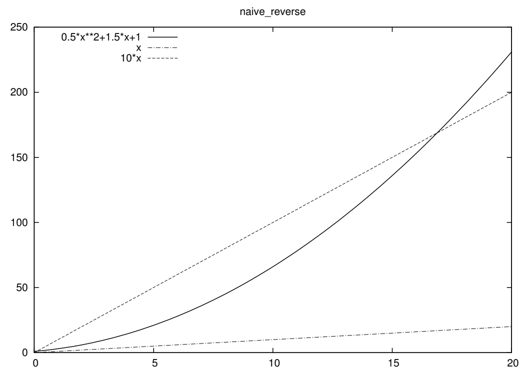

Fig. 2. Resource usage functions for program naive reverse.

the number of resolution steps performed by `nrev(A,B)`, given as functions on the length of the input list `A`. In other words, it specifies that the resource usage (given in number of resolution steps) of `nrev(A,B)` lies in the interval $[length(A), 10 \times length(A)]$.

Each Ciao assertion can be in a *verification status* [4, 20], marked by prefixing the assertion itself with keywords such as `check`, `checked`, `false`, or `true`. This specifies respectively whether the assertion is provided by the programmer and is to be checked, it is the result of processing an input assertion and proving it correct or false, or it is the output of static analysis and thus correct (safely approximated) information. Omitting this prefix means `check`, i.e., to be checked.

The *outcome* of the static checking of the previous assertion is the following set of assertions:

```
:- false comp nrev(A, B) : intervals(length(A),[i(0,0),i(17, inf)])
          + ( cost(lb,steps,length(A)), cost(ub,steps,10*length(A)) ).

:- checked comp nrev(A, B) : intervals(length(A),[i(1,16)])
            + ( cost(lb,steps,length(A)), cost(ub,steps,10*length(A)) ).
```

meaning that the assertion is false for values of $length(A)$ belonging to the interval $[0, 0] \cup [17, \infty]$, and true for values of $length(A)$ in the interval $[1, 16]$. In order to produce that outcome, CiaoPP's resource analysis infers both upper and lower bounds for the number of resolution steps of the naive reverse program of arity 2 (`nrev/2`), which are compared against the specification. In this particular case, the upper and lower bounds inferred by the analysis are the same, namely the function $0.5 \times length(A)^2 + 1.5 \times length(A) + 1$ (which implies that this is the exact resource usage function for `nrev/2`). We refer the reader to [19] for more details on the (user-definable version of the) resource analysis and references.

As we can see in Figure 2, the resource usage function inferred by CiaoPP lies in the resource usage interval expressed by the specification, namely: $[length(A), 10 \times length(A)]$, for $length(A)$ belonging to the data size interval $[1, \ 16]$. Therefore, CiaoPP says that the assertion is *checked* in that data size interval. However for $length(A) = 0$ or $length(A) \in [17, \ \infty]$, the assertion is *false*. This is because the resource usage interval inferred by the analysis is disjoint with the one expressed in the specification. This is determined by the fact that the lower bound resource usage function inferred by the analysis is greater that the upper bound resource usage function expressed in the specification.

Elaborating further on our contributions, in this paper we extend our previous work [13] in several ways. We present (a) a more detailed formalization of the resource usage verification framework (including more accurate definitions, e.g., concretization functions, and making more explicit its relation with a more general verification framework). We also (b) extend the framework to deal with *specifications* containing assertions that include preconditions expressing intervals, and (c) extend the class of resource usage functions that can be checked (summatory functions). Finally, (d) we report on a prototype implementation and provide experimental results.

In order to illustrate (b) above, consider that often in a system the possible input data belong to certain value ranges. We extend the model to make it possible to express *specifications* whose applicability is restricted to intervals of input data sizes (previously this capability was limited to the output of the analyzer). This is useful to reduce false negative errors during static checking which may be caused by input values that actually never occur. To this end (and also to allow the system to express inferred properties in a better way w.r.t. [13]) we extended the Ciao assertion language with a new property `intervals/2`, for expressing interval preconditions (used already previously in the system output assertions). Consider the previous example, and assume now that the possible length of the input list is in interval $[1, \ 10]$. In this case, we can add a precondition to the specification expressing an interval for the input data size as follows:

```
:- check comp nrev(A,B) : intervals(length(A),[i(1,10)])
          + (cost(lb, steps, length(A)), cost(ub, steps, 10*length(A))).
```

As we can see in Figure 2, this assertion is true because for input values A such that $length(A) \in [1, \ 10]$, the resource usage function of the program inferred by analysis lies in the specified resource usage interval $[length(A), 10 \times length(A)]$. In general, the outcome of the static checking of an assertion with a precondition expressing an interval for the input data size can be different for different subintervals of the one expressed in the precondition.

In the rest of the paper Section 2 recalls the CiaoPP verification framework and Section 3 describes how it is extended for the verification of resource usage properties, presenting also the formalization of the framework. Section 4 then explains our resource usage function comparison technique. Section 5 reports on the implementation of our techniques within the Ciao/CiaoPP system, providing experimental results, and finally Section 7 summarizes our conclusions.

2 Foundations of the Verification Framework

Our work on data size-dependent, static resource usage verification presented in [13] and in this paper builds on top of the previously existing framework for static *verification* and *debugging* [4, 10, 21], which is implemented and integrated in the CiaoPP system [11]. Our initial work on resource usage verification reported, e.g., in [11] and previous papers, was based on a different type of cost function comparison, basically consisting on performing function normalization and then using some syntactic comparison rules. Also, the outcome of the assertion checking was the classical one (true, false, or unknown), and did not produce intervals of input data sizes for which the verification result is different.

The verification and debugging framework of CiaoPP uses analyses, based on the abstract interpretation technique, which are provably correct and also practical, in order to statically compute semantic safe approximations of programs. These safe approximations are compared with (partial) specifications, in the form of assertions that are written by the programmer, in order to detect inconsistencies or to prove such assertions correct.

Semantics associate a meaning to a given program, and captures some properties of the computation of the program. We restrict ourselves to the important class of fixpoint semantics. Under these assumptions, the meaning of a program p, i.e., its *actual semantics*, denoted $[\![p]\!]$, is the (least) fixpoint of a monotonic operator associated to the program p, denoted S_p, i.e. $[\![p]\!] = \mathrm{lfp}(S_p)$. Such operator is a function defined on a semantic domain D, usually a complete lattice. We assume then that the actual semantic of a given program p is a set of semantic objects and the semantic domain D is the lattice of ordered sets by the inclusion relation.

In the abstract interpretation technique, a domain D_α is defined, called the *abstract* domain, which also has a lattice structure and is simpler than the *concrete* domain D. 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 [8]. Abstract operations over D_α are also defined for each of the (concrete) operations over D. The abstraction of a program p is obtained by replacing the (concrete) operators in p by their abstract counterparts. The *abstract semantics* of a program p, i.e., its semantics w.r.t. the abstract domain D_α, is computed (or approximated) by interpreting the abstraction of the program p over the abstract domain D_α. One of the fundamental results of abstract interpretation is that an abstract semantic operator S_p^α for a program p can be defined which is correct w.r.t. S_p in the sense that

Property	**Definition**
p is partially correct w.r.t. I	$[\![p]\!] \subseteq I$
p is complete w.r.t. I	$I \subseteq [\![p]\!]$
p is incorrect w.r.t. I	$[\![p]\!] \not\subseteq I$
p is incomplete w.r.t. I	$I \not\subseteq [\![p]\!]$

Table 1. Set theoretic formulation of verification problems

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

Typically, abstract interpretation guarantees that $[\![p]\!]_\alpha$ is an *over*-approximation of the abstraction of the actual semantics of p ($\alpha([\![p]\!])$), i.e., $\alpha([\![p]\!]) \subseteq [\![p]\!]_\alpha$. When $[\![p]\!]_\alpha$ meets such a condition we denote it as $[\![p]\!]_{\alpha^+}$. Alternatively, the analysis can be designed to safely *under*-approximate the actual semantics. In this case, we have that $[\![p]\!]_\alpha \subseteq \alpha([\![p]\!])$, and $[\![p]\!]_\alpha$ is denoted as $[\![p]\!]_{\alpha^-}$.

Both program verification and debugging compare the *actual semantics* $[\![p]\!]$ of a program p with an *intended semantics* for the same program, which we will denote by I. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. In Table 1 we summarize the classical understanding of some verification problems in a set-theoretic formulation as simple relations between $[\![p]\!]$ and I. Using the exact actual or intended semantics for automatic verification and debugging is in general not realistic, since the exact semantics can be typically only partially known, infinite, too expensive to compute, etc. On the other hand the technique of abstract interpretation allows computing *safe* approximations of the program semantics. The key idea of the CiaoPP approach [4, 10, 21] is to use the abstract approximation $[\![p]\!]_\alpha$ directly in program verification and debugging tasks (and in an integrated way with other techniques such as run-time checking and with the use of assertions).

Abstract Verification and Debugging. In the CiaoPP framework the abstract approximation $[\![p]\!]_\alpha$ of the concrete semantics $[\![p]\!]$ of the program is actually computed and compared directly to the (also approximate) intention (which is given in terms of *assertions* [20]), following almost directly the scheme of Table 1. We safely assume that the program specification is given as an abstract value $I_\alpha \in D_\alpha$ (where D_α is the abstract domain of computation). Program verification is then performed by comparing I_α and $[\![p]\!]_\alpha$. Table 2 shows sufficient conditions for correctness and completeness w.r.t. I_α, 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 specially suited for proving partial correctness and incompleteness with respect to the abstract specification I_α. It will also be sometimes possible to prove incorrectness in the case in which the semantics inferred for the program

Property	Definition	Sufficient condition
P is partially correct w.r.t. I_α	$\alpha(\llbracket p \rrbracket) \subseteq I_\alpha$	$\llbracket p \rrbracket_{\alpha+} \subseteq I_\alpha$
P is complete w.r.t. I_α	$I_\alpha \subseteq \alpha(\llbracket p \rrbracket)$	$I_\alpha \subseteq \llbracket p \rrbracket_{\alpha-}$
P is incorrect w.r.t. I_α	$\alpha(\llbracket p \rrbracket) \not\subseteq I_\alpha$	$\llbracket p \rrbracket_{\alpha-} \not\subseteq I_\alpha$, or $\llbracket p \rrbracket_{\alpha+} \cap I_\alpha = \emptyset \wedge \llbracket p \rrbracket_{\alpha+} \neq \emptyset$
P is incomplete w.r.t. I_α	$I_\alpha \not\subseteq \alpha(\llbracket p \rrbracket)$	$I_\alpha \not\subseteq \llbracket p \rrbracket_{\alpha+}$

Table 2. Verification problems using approximations.

is incompatible with the abstract specification, i.e., when $\llbracket p \rrbracket_{\alpha^+} \cap I_\alpha = \emptyset$. On the other hand, we use $\llbracket p \rrbracket_{\alpha-}$ to denote the (less frequent) case in which analysis under-approximates the actual semantics. In such case, it will be possible to prove completeness and incorrectness.

Since most of the properties being inferred are in general undecidable at compile-time, the inference technique used, abstract interpretation, is necessarily approximate, i.e., possibly imprecise. Nevertheless, such approximations are also always guaranteed to be safe, in the sense that they are never incorrect.

3 Extending the Framework to Data Size-Dependent Resource Usage Verification

As mentioned before, our data size-dependent resource usage verification framework is characterized by being able to deal with specifications that include both lower and upper bound resource usage functions (i.e., specifications that express intervals where the resource usage is supposed to be included in), and, in an extension of [13], that include preconditions expressing intervals within which the input data size of a program is supposed to lie. We start by providing a more complete formalization than that of [13].

3.1 Resource usage semantics

Given a program p, let $\mathcal{C}_p$ be the set of all calls to p. The concrete resource usage semantics of a program p, for a particular resource of interest, $\llbracket p \rrbracket$, is a set of pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of terms, $p(\bar{t}) \in \mathcal{C}_p$ is a call to predicate p with actual parameters $\bar{t}$, and r is a number expressing the amount of resource usage of the computation of the call $p(\bar{t})$. Such a semantic object can be computed by a suitable operational semantics, such as SLD-resolution, adorned with the computation of the resource usage. We abstract away such computation, since it will in general be dependent on the particular resource r refers to. The concrete resource usage semantics can be defined as a function $\llbracket p \rrbracket : \mathcal{C}_p \to \mathcal{R}$, i.e., where $\mathcal{R}$ is the set of real numbers (note that depending on the type of resource we can take another set of numbers, e.g., the set of natural numbers). In other words, the concrete (semantics) domain D is $2^{\mathcal{C}_p \times \mathcal{R}}$, and $\llbracket p \rrbracket \subseteq \mathcal{C}_p \times \mathcal{R}$.

We define an abstract domain D_α whose elements are sets of pairs of the form $(p(\bar{v}) : c(\bar{v})), \Phi)$, where $p(\bar{v}) : c(\bar{v})$, is an abstraction of a set of calls and Φ is an abstraction of the resource usage of such calls. We refer to such pairs as *call-resource* pairs. More concretely, $\bar{v}$ is a tuple of variables and $c(\bar{v})$ is an abstraction representing a set of tuples of terms which are instances of $\bar{v}$. The abstraction $c(\bar{v})$ is a subset of the abstract domains present in the CiaoPP system expressing instantiation states. An example of $c(\bar{v})$ (in fact, the one used in Section 5 in our experiments) is a combination of properties which are in the domain of the regular type analysis (*eterms*) [22] and properties such as groundness and freeness present in the *shfr* abstract domain [17].

We refer to Φ as a *resource usage interval function* for p, defined as follows:

Definition 1. *A* resource usage bound function *for p is a monotonic arithmetic function, $\Psi_p : S \mapsto \mathcal{R}_\infty$, for a given subset $S \subseteq \mathcal{R}^k$, where $\mathcal{R}$ is the set of real numbers, k is the number of input arguments to predicate p, and $\mathcal{R}_\infty$ is the set of real numbers augmented with the special symbols ∞ and $-\infty$. We use such functions to express lower and upper bounds on the resource usage of predicate p depending on its input data sizes.*

Definition 2. *A* resource usage interval function *for p is an arithmetic function, $\Phi : S \mapsto \mathcal{RI}$, where S is defined as before and $\mathcal{RI}$ is the set of intervals of real numbers, such that $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ for all $\bar{n} \in S$, where $\Phi^l(\bar{n})$ and $\Phi^u(\bar{n})$ are* resource usage bound functions *that denote the lower and upper endpoints of the interval $\Phi(\bar{n})$ respectively for the tuple of input data sizes $\bar{n}$.[1] We require that Φ be well defined so that $\forall \bar{n}\ (\Phi^l(\bar{n}) \leq \Phi^u(\bar{n}))$.*

In order to relate the elements $p(\bar{v}) : c(\bar{v})$ and Φ in a call-resource pair as the one described previously, we assume the existence of two functions $input_p$ and $size_p$ associated to each predicate p in the program. Assume that p has k arguments and i input arguments ($i \leq k$). The function $input_p$ takes a k-tuple of terms $\bar{t}$ (the actual arguments of a call to p) and returns a tuple with the input arguments to p. This function is generally inferred by using existing mode analysis, but can also be given by the user by means of assertions. The function $size_p(\bar{w})$ takes a i-tuple of terms $\bar{w}$ (the actual input arguments to p) and returns a tuple with the sizes of those terms under a given metric. The metric used for measuring the size of each argument of p is automatically inferred (based on type analysis information), but again can also be given by the user by means of assertions [19].

Example 1. Consider for example the naive reverse program in Figure 1, with the classical definition of predicate `append`. The first argument of `nrev` is declared input, and the two first arguments of `append` are consequently inferred to be also input. The size measure for all of them is inferred to be *list-length.* Then, we have that:
$input_{nrev}((x, y)) = (x)$, $input_{app}((x, y, z)) = (x, y)$,
$size_{nrev}((x)) = (length(x))$ and $size_{app}((x, y)) = (length(x), length(y))$.

[1] Although $\bar{n}$ is typically a tuple of natural numbers, we do not restrict the framework to this case.

We define the concretization function $\gamma : D_\alpha \mapsto D$ as follows:

$$\forall X \in D_\alpha, \gamma(X) = \bigcup_{x \in X} \gamma_1(x)$$

where γ_1 is another concretization function, applied to call-resource pairs x's of the form $(p(\bar{v}) : c(\bar{v}), \Phi)$. We define:

$$\gamma_1(x) = \{(p(\bar{t}), r) \mid \bar{t} \in \gamma_m(c(\bar{v})) \wedge \bar{n} = size_p(input_p(\bar{t})) \wedge r \in [\Phi^l(\bar{n}), \Phi^u(\bar{n})]\}$$

where γ_m is the concretization function of the mode/type abstract domain.

The definition of the abstraction function $\alpha : D \mapsto D_\alpha$ is straightforward, given the definition of the concretization function γ above.

Intended meaning. As already said, the intended semantics is an expression of the user's expectations, and is typically only partially known. For this and other reasons it is in general not realistic to use the exact intended semantics. Thus, we define the intended approximated semantics I_α of a program as a set of *call-resource* pairs $(p(\bar{v}) : c(\bar{v}), \Phi)$, identical to those previously used in the abstract semantics definition. However, the former are provided by the user using the Ciao/CiaoPP assertion language, while the latter are automatically inferred by CiaoPP's analysis tools. In particular, each one of such pairs is represented as a resource usage assertion for predicate p in the program.

The most common syntactic schema of a resource usage assertion and its correspondence to the *call-resource* pair it represents is the following:

$$\boxed{\texttt{:- comp } p(\bar{v}) \texttt{ : } c(\bar{v}) \texttt{ + } \Phi\texttt{.}}$$

which expresses that for any call to predicate p, if (precondition) $c(\bar{v})$ is satisfied in the calling state, then the resource usage of the computation of the call is in the interval represented by Φ. Note that $c(\bar{v})$ is a conjunction of program execution state properties, i.e., properties about the terms to which program variables are bound to, or instantiation states of such variables. We use the comma (,) as the symbol for the conjunction operator. If the precondition $c(\bar{v})$ is omitted, then it is assumed to be the "top" element of the lattice representing calls, i.e., the one that represents any call to predicate p. The syntax used to express the resource usage interval function Φ is a conjunction of `cost/3` properties (already explained). Assuming that $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$, where $\bar{n} = size_p(input_p(\bar{v}))$, Φ is represented in the resource usage assertion as the conjunction:

$$(\texttt{cost}(\text{lb}, r, \Phi^l(\bar{n})), \texttt{cost}(\text{ub}, r, \Phi^u(\bar{n})))$$

We use Prolog syntax for variable names (variables start with uppercase letters).

Example 2. In the program of Figure 1 one could use the assertion:

```
:- comp nrev(A,B): ( list(A, gnd), var(B) )
                 + ( cost(lb, steps, 2 * length(A)),
                     cost(ub, steps, 1 + exp(length(A), 2) )).
```

to express that for any call to `nrev(A,B)` with the first argument bound to a ground list and the second one a free variable, a lower (resp. upper) bound on the number of resolution `steps` performed by the computation is $2 \times length(A)$ (resp. $1 + length(A)^2$).

In this example, p is $nrev$, $\bar{v}$ is `(A, B)`, $c(\bar{v})$ is `( list(A, gnd), var(B) )`, $\bar{n} = size_{nrev}(input_{nrev}((A, B))) = (length(A))$, where the functions $size_{nrev}$ and $input_{nrev}$ are those defined in Example 1, and the interval $\Phi_{rev}(\bar{n})$ approximating the number of resolution steps is $[2 \times length(A), 1 + length(A)^2]$ (in other words, we are assuming that $\Phi^l_{nrev}(x) = 2 \times x$ and $\Phi^u_{nrev}(x) = 1 + x^2$). If we omit the `cost` property expressing the lower bound (`lb`) on the resource usage, the minimum of the interval is assumed to be zero (since the number of resolution steps cannot be negative). If we assume that the resource usage can be negative, the interval would be $(-\infty, 1+n^2]$. Similarly, if the upper bound (`ub`) is omitted, the upper limit of the interval is assumed to be ∞.

Example 3. The assertion in Example 2 captures the following concrete semantic pairs:

```
( nrev([a,b,c,d,e,f,g],X), 35 )        ( nrev([],Y), 1 )
```

but it does not capture the following ones:

```
( nrev([A,B,C,D,E,F,G],X), 35 )        ( nrev(W,Y), 1 )
( nrev([a,b,c,d,e,f,g],X), 53 )        ( nrev([],Y), 11 )
```

Those in the first line above are not captured because they correspond to calls which are outside the scope of the assertion, i.e., they do not meet the assertion's precondition $c(\bar{v})$: the leftmost one because `nrev` is called with the first argument bound to a list of unbound variables (denoted by using uppercase letters), and the other one because the first argument of `nrev` is an unbound variable. The concrete semantic pairs on the second line will never occur during execution because they violate the assertion, i.e., they meet the precondition $c(\bar{v})$, but the resource usage of their execution is not within the limits expressed by Φ.

3.2 Comparing Abstract Semantics: Correctness

The definition of partial correctness has been given by the condition $[\![p]\!] \subseteq I$ in Table 1. However, we have already argued that we are going to use an approximation I_α of the intended semantics I, where I_α is given as a set of *call-resource* pairs of the form $(p(\bar{v}) : c(\bar{v}), \Phi)$.

Definition 3. *We say that p is partially correct with respect to a call-resource pair $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ if for all $(p(\bar{t}), r) \in [\![p]\!]$ (i.e., $p(\bar{t}) \in \mathcal{C}_p$ and r is the amount of resource usage of the computation of the call $p(\bar{t})$), it holds that: if $\bar{t} \in \gamma_m(c_I(\bar{v}))$ then $r \in \Phi_I(\bar{n})$, where $\bar{n} = size_p(input_p(\bar{t}))$ and γ_m is the concretization function of the mode/type abstract domain.*

Lemma 1. *p is partially correct with respect to I_α, i.e. $[\![p]\!] \subseteq \gamma(I_\alpha)$ if:*

- *For all $(p(\bar{t}), r) \in [\![p]\!]$, there is a pair $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in I_α such that $\bar{t} \in \gamma_m(c_I(\bar{v}))$, and*
- *p is partially correct with respect to every pair in I_α.*

As mentioned before, we use a safe over-approximation of the program semantics $[\![p]\!]$, that is automatically computed by static analyses, and that we denote $[\![p]\!]_{\alpha+}$. The description of how the resource usage bound functions appearing in $[\![p]\!]_{\alpha+}$ are computed is out of the scope of this paper, and it can be found in [19] and its references. We assume for simplicity that the computed abstract semantics $[\![p]\!]_{\alpha+}$ is a set made up of a single call-resource pair $(p(\bar{v}) : c(\bar{v}), \Phi)$. The safety of the analysis can then be expressed as follows:

Lemma 2 (Safety of the static analysis). *Let* $[\![p]\!]_{\alpha+} = \{(p(\bar{v}) : c(\bar{v}), \Phi)\}$. *For all* $(p(\bar{t}), r) \in [\![p]\!]$, *it holds that* $\bar{t} \in \gamma_m(c_I(\bar{v}))$, *and* $r \in \Phi_I(\bar{n})$, *where* $\bar{n} = size_p(input_p(\bar{t}))$.

Definition 4. *Given two resource usage interval functions* Φ_1 *and* Φ_2, *such that* $\Phi_1, \Phi_2 : S \mapsto \mathcal{RI}$, *where* $S \subseteq \mathcal{R}^k$, *we define the inclusion relation* $\sqsubseteq_S$ *and the intersection operation* $\sqcap_S$ *as follows:*

- $\Phi_1 \sqsubseteq_S \Phi_2$ *iff for all* $\bar{n} \in S$ $(S \subseteq \mathcal{R}^k)$, $\Phi_1(\bar{n}) \subseteq \Phi_2(\bar{n})$.
- $\Phi_1 \sqcap_S \Phi_2 = \Phi_3$ *iff for all* $\bar{n} \in S$ $(S \subseteq \mathcal{R}^k)$, $\Phi_1(\bar{n}) \cap \Phi_2(\bar{n}) = \Phi_3(\bar{n})$.

Consider a pair $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in the intended meaning I_α, and the pair $(p(\bar{v}) : c(\bar{v}), \Phi)$ in the computed abstract semantics $[\![p]\!]_{\alpha+}$ (for simplicity, we assume the same tuple of variables $\bar{v}$ in all abstract objects).

Definition 5. *We say that* $(p(\bar{v}) : c(\bar{v}), \Phi) \sqsubseteq (p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ *if* $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$ *and* $\Phi \sqsubseteq_S \Phi_I$.

Note that the condition $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$ is checked using the CiaoPP capabilities for comparing program state properties such as types and modes, using the appropriate definition of the comparison operator $\sqsubseteq_m$. Such a condition is needed to ensure that we select resource analysis information that can safely be used to verify the assertion corresponding to the pair $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

Definition 6. *We say that* $(p(\bar{v}) : c(\bar{v}), \Phi) \sqcap (p(\bar{v}) : c_I(\bar{v}), \Phi_I) = \emptyset$ *if:*

$$c_I(\bar{v}) \sqsubseteq_m c(\bar{v}) \text{ and } \Phi \sqcap_S \Phi_I = \Phi_\emptyset,$$

where $\Phi_\emptyset$ *represents the constant function identical to the empty interval.*

Lemma 3. *If* $(p(\bar{v}) : c(\bar{v}), \Phi) \sqsubseteq (p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ *then* p *is partially correct with respect to* $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

Proof. If $(p(\bar{v}) : c(\bar{v}), \Phi) \sqsubseteq (p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ then $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$ (Definition 5). For all $(p(\bar{t}), r) \in [\![p]\!]$, it holds that: if $\bar{t} \in \gamma_m(c_I(\bar{v}))$ then $\bar{t} \in \gamma_m(c(\bar{v}))$ (because $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$), and thus $r \in \Phi(\bar{n})$, where $\bar{n} = size_p(input_p(\bar{t}))$ (because of the safety of the analysis, Lemma 2). Since $\Phi \sqsubseteq_S \Phi_I$ (Definition 5), we have that $r \in \Phi_I(\bar{n})$.

Lemma 4. *If* $(p(\bar{v}) : c(\bar{v}), \Phi) \sqcap (p(\bar{v}) : c_I(\bar{v}), \Phi_I) = \emptyset$ *and* $(p(\bar{v}) : c(\bar{v}), \Phi) \neq \emptyset$ *then* p *is incorrect w.r.t.* $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

In order to prove partial correctness or incorrectness we compare call-resource pairs by using Lemmas 3 and 4 (thus ensuring the sufficient conditions given in Table 2). This means that whenever $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ we have to determine whether $\Phi \sqsubseteq_S \Phi_I$ or $\Phi \sqcap_S \Phi_I = \Phi_\emptyset$. To do this in practice, we compare resource usage bound functions in the way expressed by Corollary 1 below.

Definition 7 (Input-size set). *The* input-size set *of a call-resource abstract pair* $(p(\bar{v}) : c(\bar{v}), \Phi)$ *is the set* $S = \{\bar{n} \mid \exists\, \bar{t} \in \gamma_m(c(\bar{v})) \wedge \bar{n} = size_p(input_p(\bar{t}))\}$. *The input-size set is represented as an interval (or a union of intervals).*

Corollary 1. *Let* $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ *be a pair in the intended abstract semantics* I_α *(given in a specification), and* $(p(\bar{v}) : c(\bar{v}), \Phi)$ *the pair in the abstract semantics* $[\![p]\!]_{\alpha+}$ *inferred by analysis. Let* S *be the input-size set of* $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$. *Assume that* $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$. *Then, we have that:*

1. *If for all* $\bar{n} \in S$, $\Phi_I^l(\bar{n}) \leq \Phi^l(\bar{n})$ *and* $\Phi^u(\bar{n}) \leq \Phi_I^u(\bar{n})$, *then* p *is partially correct with respect to* $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.
2. *If for all* $\bar{n} \in S$, $\Phi^u(\bar{n}) < \Phi_I^l(\bar{n})$ *or* $\Phi_I^u(\bar{n}) < \Phi^l(\bar{n})$, *then* p *is incorrect with respect to* $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

When Φ_I^u (resp., Φ_I^l) is not present in a specification, we assume that $\forall \bar{n}$ $(\Phi_I^u(\bar{n}) = \infty)$ (resp., $\Phi_I^l = -\infty$ or $\Phi_I^l(\bar{n}) = 0$, depending on the resource). With this assumption, one of the resource usage bound function comparisons in the sufficient condition 1 (resp., 2) above is always true (resp., false) and the truth value of such conditions depends on the other comparison.

If none of the conditions 1 or 2 in Corollary 1 hold for the input-size set S of the pair $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$, our proposal is to compute subsets S_j, $1 \leq j \leq a$, of S for which either one holds. Thus, as a result of the verification of $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ we produce a set of pairs (corresponding to assertions in the program) $(p(\bar{v}) : c_I^j(\bar{v}), \Phi_I)$, $1 \leq j \leq a$, whose input-size set is S_j.

For the particular case where resource usage bound functions depend on one argument, the element $c_I^j(\bar{v})$ (in the assertion precondition) is of the form $c_I(\bar{v}) \wedge d_j$, where d_j defines an interval for the input data size n to p. This allows us to give intervals d_j of input data sizes for which a program p is partially correct (or incorrect).

The definition of *input-size set* can be extended to deal with data size intervals d_j's in a straightforward way:

$$S_j = \{n \mid \exists\, \bar{t} \in \gamma_m(c(\bar{v})) \wedge n = size_p(input_p(\bar{t})) \wedge n \in d_j\}.$$

From the practical point of view, in order to represent properties like $n \in d_j$, we have extended the Ciao assertion language with the new `intervals(A, B)` property, which expresses that the input data size `A` is included in some of the intervals in the list `B`. To this end, in order to show the result of the assertion checking process to the user, we group all the $(p(\bar{v}) : c_I^j(\bar{v}), \Phi_I)$ pairs that meet the above sufficient condition 1 (applied to the set S_j) and, assuming that $d_{f_1}, \ldots, d_{f_b}$ are the computed input data size intervals for such pairs, an assertion with the following syntactic schema is produced as output:

$$\boxed{\texttt{:- checked comp:}c_I^j(\bar{v})\texttt{,intervals(}size_p(input_p(\bar{v}))\texttt{,[}d_{f_1},\ldots,d_{f_b}\texttt{]) + }\Phi_I\texttt{ .}}$$

Similarly, the pairs meeting the sufficient condition 2 are grouped and the following assertion is produced:

$$\boxed{\texttt{:- false comp: }c_I^j(\bar{v})\texttt{,intervals(}size_p(input_p(\bar{v}))\texttt{,[}d_{g_1},\ldots,d_{g_e}\texttt{]) + }\Phi_I\texttt{ .}}$$

Finally, if there are intervals complementary to the previous ones w.r.t. S (the input-size set of the original assertion), say $d_{h_1},\ldots,d_{h_q}$, the following assertion is produced:

$$\boxed{\texttt{:- check comp:}c_I^j(\bar{v})\texttt{,intervals(}size_p(input_p(\bar{v}))\texttt{, [}d_{h_1},\ldots,d_{h_q}\texttt{]) + }\Phi_I\texttt{ .}}$$

The description of how the input data size intervals d_j's are computed is given in Section 4. While we have limited the discussion to cases where resource usage bound functions depend on one argument, the approach can be extended to the multi-argument case. Indeed, we have ongoing work to this end, using techniques from constraint programming.

Dealing with Preconditions Expressing Input Data Size Intervals. In order to allow checking assertions which include preconditions expressing intervals within which the input data size of a program is supposed to lie (i.e., using the `intervals(A, B)` property), we replace the concretization function γ_m by an extended version γ'_m. Given an abstract call-resource pair: $(p(\bar{v}) : c_I(\bar{v}) \wedge d, \Phi_I)$, where d represents an interval (or the union of severals intervals) for the input data sizes to p, we define:

$$\gamma'_m(c_I(\bar{v}) \wedge d) = \{\bar{t} \mid \bar{t} \in \gamma_m(c_I(\bar{v})) \wedge size_p(input_p(\bar{t})) \in d\}.$$

We also extend the definition of the $\sqsubseteq_m$ relation accordingly. With these extended operations, all the previous results in Section 3 are applicable.

4 Resource Usage Bound Function Comparison

As stated in [13, 14], fundamental to our approach to verification is the operation that compares two resource usage bound functions, one of them inferred by the static analysis and the other one given in an assertion present in the program (i.e., given as a specification). Given two of such functions, $\Psi_1(n)$ and $\Psi_2(n)$, where n is in the input-size set of the assertion, the objective of this operation is to determine intervals for n in which $\Psi_1(n) > \Psi_2(n)$, $\Psi_1(n) = \Psi_2(n)$, or $\Psi_1(n) < \Psi_2(n)$. The fact that it is possible to restrict the input-size set of assertions (using preconditions with intervals as already seen), facilitates the function comparison operation.

Our approach consists in defining $f(n) = \Psi_1(n) - \Psi_2(n)$ and finding the roots of the equation $f(n) = 0$. Assume that the equation has m roots, $n_1, \ldots, n_m$. These roots are intersection points of $\Psi_1(n)$ and $\Psi_2(n)$. We consider the intervals $S_1 = [0, n_1)$, $S_2 = (n_1, n_2)$, $S_m = \ldots (n_{m-1}, n_m)$, $S_{m+1} = (n_m, \infty)$. For each interval S_i, $1 \leq i \leq m$, we select a value v_i in the interval. If $f(v_i) > 0$ (respectively $f(v_i) < 0$), then $\Psi_1(n) > \Psi_2(n)$ (respectively $\Psi_1(n) < \Psi_2(n)$) for all $n \in S_i$.

Since our resource analysis is able to infer different types of functions, such as polynomial, exponential, logarithmic and summatory, it is also desirable to be able to compare all of these functions.

For polynomial functions there exist powerful algorithms for obtaining roots, e.g., the one implemented in the GNU Scientific Library (GSL) [9], which is the one we are currently using in our implementation, and which offers a specific polynomial function library that uses analytical methods for finding roots of polynomials up to order four, and uses numerical methods for higher order polynomials. For the other functions, we safely approximate them using polynomials such that they bound (from above or below as appropriate) such functions. In this case, we should guarantee that the error falls in the safe side when comparing the corresponding resource usage bound functions.

Exponential functions are approximated by using the expression:

$$a^x = e^{x\ ln\ a} \approx 1 + x\ ln\ a + \frac{(x\ ln\ a)^2}{2!} + \frac{(x\ ln\ a)^3}{3!} + \ldots$$

which approximates the functions near $x = 0$. Since it is possible to restrict the input-size set of assertions (by using preconditions with intervals), we can approximate exponential functions near the minimum of such input-size set to improve the accuracy of the approximation. We use *finite calculus* rules to decompose summatory functions into polynomials or functions that can we know how to approximate by polynomials. We refer the reader to [14] for a full description of how the approximations above are performed.

5 Implementation and Experimental Results

In order to assess the accuracy and efficiency (as well as the scalability) of the resource usage verification techniques presented in this paper, we have implemented and integrated them in a seamless way within the Ciao/CiaoPP framework, which unifies static and run-time verification, as well as unit testing [16].

Table 3 shows some experimental results obtained with our prototype implementation on an Intel Core2 Quad 2.5 GHz with quad core processor, 2GB of RAM memory, running Debian Squeeze, kernel 2.6.32-5-686. The column labeled **Program** shows the name of the program to be verified, the upper (**ub**) and lower (**lb**) bound resource usage functions inferred by CiaoPP's analyzers, the input arguments, and the size measure used.

The scalability of the different analyses required is beyond the scope of this paper (in the case of the core of the resource inference itself it follows generally from its compositional nature). Our study focuses on the scalability of the assertion comparison process. To this end, we have added a relatively large number of assertions to a number of programs that are then statically checked. Column **Program** shows an expression $\mathbf{AvT} = \frac{\mathbf{VTime}}{\mathbf{\#Asser}}$ giving the total time **VTime** in milliseconds spent by the verification of all such assertions (whose number is given by **#Asser**), and the resulting average time per assertion (**AvT**). A

Program	**ID**	**Assertion**	**Verif. Result**	**Time** (ms)	
				Tot	**Avg**
<u>*Fibonacci*</u> **lb,ub:** $1.45 * 1.62^x$ $+0.55 * -0.62^x - 1$ x = length(N) **AvT=** $\frac{964\ ms}{65\ a} = 15\frac{ms}{a}$	*A1*	:- comp fib(N,R) +cost(ub,steps, exp(2,int(N))-1000).	F in $[0, 10]$ T in $[11, \infty]$	88	29
	A2	:- comp fib(N,R) + (cost(ub,steps, exp(2,int(N))-1000), cost(lb,steps, exp(2,int(N))-10000)).	F in $[0, 10] \cup [15, \infty]$ T in $[11, 13]$ C in $[14, 14]$		
	A3	:- comp fib(N,R) :(intervals(int(N),[i(1,12)])) + (cost(ub,steps, exp(2,int(N))-1000), cost(lb,steps, exp(2,int(N))-10000)).	F in $[1, 10]$ T in $[11, 12]$		
<u>*Naive Reverse*</u> **lb,ub:** $0.5x^2 + 1.5x + 1$ x = length(A) **AvT=** $\frac{780\ ms}{54\ a} = 14\frac{ms}{a}$	*B1*	:- comp nrev(A,B) + (cost(lb,steps,length(A)), cost(ub,steps, exp(length(A),2))).	F in $[0, 3]$ T in $[4, \infty]$	44	22
	B2	:- comp nrev(A,_1) + (cost(lb, steps, length(A)), cost(ub, steps, 10*length(A))).	F in $[0, 0] \cup [17, \infty]$ T in $[1, 16]$		
<u>*Quick Sort*</u> **lb:** $x + 5$ **ub:** $(\sum_{j=1}^{x} j2^{x-j}) + x2^{x-1}$ $+2 * 2^x - 1$ x = length(A) **AvT=** $\frac{800\ ms}{56\ a} = 14\frac{ms}{a}$	*C1*	:- comp qsort(A,B) + cost(ub, steps, exp(length(A),2)).	F in $[0, 2]$ C in $[3, \infty]$	44	22
	C2	:- comp qsort(A,B) + cost(ub, steps, exp(length(A),3)).	C in $[0, \infty]$		
<u>*Client*</u> **ub:** $8x$ x = length(I) **AvT=** $\frac{180\ ms}{60\ a} = 3\frac{ms}{a}$	*D1*	:- comp main(Op, I, B) + cost(ub, bits_received, exp(length(I),2)).	C in $[1, 7]$ T in $[0, 0] \cup [8, \infty]$	20	7
	D2	:- comp main(Op, I, B) + cost(ub, bits_received, 10*length(I)).	T in $[0, \infty]$		
	D3	:- comp main(Op, I, B) : intervals(length(I), [i(1,10),i(100,inf)]) + cost(ub, bits_received, 10*length(I)).	T in $[1, 10] \cup [100, \infty]$		
<u>*Reverse*</u> **lb:** $428x + 694$ **ub:** $467x + 758$ x = length(A) **AvT=** $\frac{820\ ms}{60\ a} = 14\frac{ms}{a}$	*E1*	:- comp reverse(A, B) + (cost(ub, ticks, 500 * length(A))).	F in $[0, 9]$ C in $[10, 22]$ T in $[23, \infty]$	20	20
<u>*Palindrome*</u> **lb,ub:** $x2^{x-1} + 2 * 2^x - 1$ x=length(X) **AvT=** $\frac{616\ ms}{52\ a} = 12\frac{ms}{a}$	*F1*	:- comp palindrome(X,Y) + cost(ub,steps, exp(length(X),2)).	F in $[0, \infty]$	32	16
	F2	:- comp palindrome(X,Y) + cost(ub,steps, exp(length(X),3)).	F in $[0, 2] \cup [5, \infty]$ T in $[3, 4]$		
<u>*Powerset*</u> **ub:** $0.5 * 2^{x+1}$ x = length(A) **AvT=** $\frac{564\ ms}{52\ a} = 11\frac{ms}{a}$	*G1*	:- comp powset(A,B) + cost(ub,output_elements, exp(length(A),4)).	C in $[0, 1] \cup [17, \infty]$ T in $[2, 16]$	32	16
	G2	:- comp powset(A,B) + cost(ub,output_elements, length(A)*exp(2,length(A))).	C in $[0, 1]$ T in $[2, \infty]$		

Table 3. Results of the interval-based static assertion checking integrated into CiaoPP.

ID	**Method**	**Intervals**			
		[1,12]	[1,100]	[1,1000]	[1,10000]
A3	**Root**	84	84	84	84
	Eval	80	84	132	644
D3	**Root**	19	19	19	19
	Eval	33	33	44	102

Table 4. Comparison of assertion checking times for two methods.

few of those assertions are shown as examples in column **Assertion** (**ID** is the assertion identifier.). Some assertions specify both upper and lower bounds (e.g., *A2* or *A3*), but others only specify upper bounds (e.g., *A1* or *C1*). Also, some assertions include preconditions expressing intervals within which the input data size of the program is supposed to lie (*A3* and *D3*). The column **Verif. Result** shows the result of the verification process for the assertions in column **Assertion**, which in general express intervals of input data sizes for which the assertion is true (`T`), false (`F`), or it has not been possible to determine whether it is true or false (`C`). Column **Tot** (under **Time**) shows the total time (in milliseconds) spent by the verification of the assertions shown in column **Assertion** and **Avg** shows the average time per assertion for these assertions.

Note that, as mentioned before, the system can deal with different types of resource usage functions: polynomial functions (e.g., program *Naive Reverse*), exponential functions (e.g., *Fibonacci*), and summatory functions (*Quick Sort*). In general, polynomial functions are faster to check than other functions, because they do not need additional processing for approximation. However the additional time to compute approximations is very reasonable in practice. Finally, note that the prototype was not able to determine whether the assertion *C2* in the *Quick Sort* program is true or false (this is because the root finding algorithm did not converge).

Table 4 shows assertion checking times (in milliseconds) for different input data size intervals (columns under **Intervals**) and for two methods: the one described so far (referred to as **Root**), and a simple method (**Eval**) that evaluates the resource usage functions for all the (natural) values in a given input data size interval and compares the results. Column **ID** refers to the assertions in Table 3. We can see that checking time grows quite slowly compared to the length of the interval, which grows exponentially.

6 Related Work

The closest related work we are aware of presents a method for comparison of cost functions inferred by the COSTA system for Java bytecode [1]. The method proves whether a cost function is smaller than another one *for all the values* of a given initial set of input data sizes. The result of this comparison is a boolean value. However, as mentioned before, in our approach the result is in general a set of subsets (intervals) in which the initial set of input data sizes is partitioned, so that the result of the comparison is different for each subset. Also, [1] differs in that comparison is syntactic, using a method similar to what was already being done in the CiaoPP system: performing a function normalization and then using some syntactic comparison rules. In this work we go beyond these syntactic comparison rules. Moreover, [1] only covers function comparisons while we have addressed the whole resource usage verification process. Note also that, although we have presented our work applied to logic programming, the CiaoPP system can also deal with Java bytecode [18, 15].

In a more general context, using abstract interpretation in debugging and/or verification tasks has now become well established. To cite some early work, abstractions were used in the context of algorithmic debugging in [12]. Abstract interpretation has been applied by Bourdoncle [3] to debugging of imperative programs and by Comini et al. to the algorithmic debugging of logic programs [6] (making use of partial specifications in [5]), and by P. Cousot [7] to verification, among others. The CiaoPP framework [4, 10, 11] was pioneering, offering an integrated approach combining abstraction-based verification, debugging, and run-time checking with an assertion language.

7 Conclusions

We have presented several extensions and improvements to our framework for verification/debugging within the CiaoPP system, dealing with specifications about the resource usage of programs. We have provided a more complete formalization and we have improved the resource usage function comparison method by extending the class of resource usage functions that can be compared and providing better algorithms, which in addition allow for the case when the assertions include preconditions expressing input data size intervals. We have also reported on a prototype implementation and provided experimental results, which are encouraging, suggesting that our extensions are feasible and accurate in practice.

References

1. E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *Proc. of FOPARA'09*, volume 6234 of *LNCS*, pages 1–17. Springer, 2010.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proc. of PLDI'03.* ACM Press, 2003.
3. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.
4. 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 WS on Automated Debugging–AADEBUG*, pages 155–170. U. Linköping Press, May 1997.
5. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.

6. M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *ILPS'95*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.
7. P. Cousot. Automatic Verification by Abstract Interpretation, Invited Tutorial. In *VMCAI*, number 2575 in LNCS, pages 20–24. Springer, January 2003.
8. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.
9. M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual.* Network Theory Ltd, 2009. Available at `http://www.gnu.org/software/gsl/`.
10. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, 1999.
11. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Comp. Progr.*, 58(1–2), 2005.
12. 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.
13. P. López-García, L. Darmawan, and F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties. In *Technical Communications of ICLP*, volume 7 of *LIPIcs*, pages 104–113. Schloss Dagstuhl, July 2010.
14. P. López-García, L. Darmawan, F. Bueno, and M. Hermenegildo. Towards Resource Usage Function Verification based on Input Data Size Intervals. Technical Report CLIP4/2011.0, UPM, 2011. Available at `http://cliplab.org/papers/resource-verif-11-tr.pdf`.
15. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.
16. E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *ICLP'09*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.
17. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2/3):315–347, July 1992.
18. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *BYTECODE'09*, volume 253 of *ENTCS*, pages 6–86. Elsevier, March 2009.
19. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *Proc. of ICLP'07*, volume 4670 of *LNCS*, pages 348–363. Springer, 2007.
20. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, 2000.
21. G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Proc. of LOPSTR'99*, LNCS 1817, pages 273–292. Springer-Verlag, March 2000.
22. C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *SAS'02*, number 2477 in LNCS, pages 102–116. Springer, 2002.