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] The 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 resource usages, such as execution time, memory, energy, or user-defined resources. In previous work we have presented a novel framework for data size-aware, static resource usage verification. 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 which 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 full 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 the conformance with respect to specifications expressing resource usages (such as execution time, memory, energy, or user defined resources). For example, in a real-time application, a program completing an action later than expected is as erroneous as a program not computing the correct answer. The same applies to an embedded application in a battery operated device (for example in the medical or mobile phone domains) which makes the device to run-out of batteries earlier than expected, and thus, making the whole system useless.

In [12] we proposed techniques that extended the capacity of debugging and verification systems based on static analysis [4, 2, 10] when dealing with a quite general class of properties related to resource usage. This includes upper and lower bounds on execution time, memory, energy, and user-defined resources (the later in the sense of [15]). Such bounds are given as functions on input data sizes (see [15] for the different metrics that can be used to measure data sizes, such as list-length, term-depth or term-size). For example, the techniques of [12] extend the capacity of CiaoPP to certify programs with resource consumption assurances and also to efficiently check such certificates. 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) with approximated semantics inferred by static analysis, all for the case of resources. Thus, these operations include the comparison of arithmetic functions (in particular, for [12], 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. Thus, a novel aspect of the *resource verification and debugging* approach proposed in [12] is that the *answers* of the checking process go beyond these 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:

```
:- 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 naive reverse.

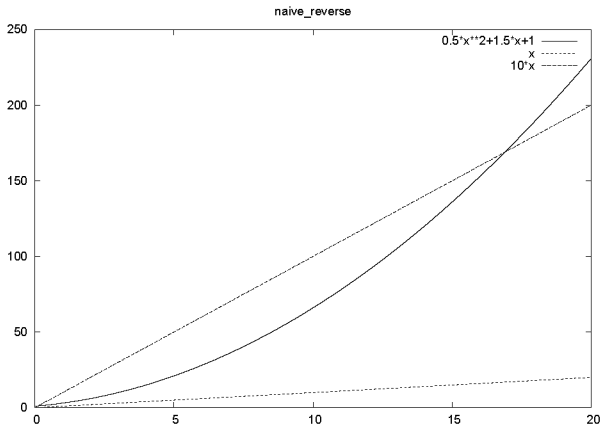

Fig. 2. Resource usage functions for program naive reverse.

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*, 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, or is the result of processing an input assertion and proving it correct or false, or is the output of static analysis and thus correct (safely approximated) information. Omitting this prefix means "to be checked" [16].

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

```
:- false comp nrev(A,_1)
         : intervals(length(A),[i(0,0),i(17, inf)])
         + ( cost(lb,steps,length(A)), cost(ub,steps,10*length(A)) ).

:- checked comp nrev(A,_1)
         : 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 `nrev/2`, which are compared against the specification. In this particular case, the upper and lower bound inferred by the analysis are the same, namely the function $0.5 \times length(A)^2 + 1.5 \times length(A) + 1$ (which means that it is the exact resource usage function for `nrev/2`).

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.

In this paper we extend our previous work [12] in several ways: (a) presenting a complete formalization of the resource usage verification framework, and (b) reporting on a prototype implementation and experimental results. We also (c) extend the framework to deal with *specifications* containing assertions that include preconditions expressing intervals, and (d) extend the class of resource usage functions that can be checked (e.g., summatory and logarithm functions).

In order to illustrate (c) 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. [12]) we have extended the Ciao assertion language with the new `intervals/2` property, for expressing such preconditions, used already in the previous output. Consider the previous example, and assume now that the possible length of the input list to be reversed 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.

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. The method in [1] also differs from the one presented here 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. However, in this work we go beyond these syntactic comparison rules. Moreover, we present an application for which cost function comparison is instrumental and which is not covered in the cited work: verification of resource usage properties. This implies extending the criteria of correctness and defining a resource usage (abstract) semantics and conditions under which a program is correct or incorrect with respect to an (approximated) intended semantics.

The structure of the rest of the paper is the following: Section 2 recalls the CiaoPP verification framework and Section 3 describes how it is used and extended for the verification of general resource usage program properties, and presents the formalization of the framework. Section 4 then briefly explains the technique that we have developed for resource usage function comparison. Section 5 reports on the implementation of our techniques within the Ciao/CiaoPP system, providing experimental results, and finally Section 6 summarizes our conclusions.

2 Foundations of the Verification Framework

Our work on data size-aware, static resource usage verification presented in [12] and in this paper builds on top of the previously existing framework for static *verification* and *debugging* [17], which is implemented and integrated in the CiaoPP system [10]. Our initial work on resource usage verification reported, e.g., in [10] 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 abstract interpretation-based analyses, which are provably correct and also practical, in order to statically compute semantic approximations of programs. These semantic 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.

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 show classical 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

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

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 is to use the abstract approximation $[\![P]\!]_\alpha$ directly in program verification and debugging tasks.

A number of other approaches have also been proposed which make use to some extent of abstract interpretation in verification and/or debugging tasks. For example, abstractions were used in the context of algorithmic debugging in [11]. Abstract interpretation (generally for debugging of imperative programs) was studied by Bourdoncle [3], by Comini et al. for the particular case of algorithmic debugging of logic programs [6] (making use of partial specifications) [5], by P. Cousot [7], and others. Additional discussion and more details about the foundations and implementation issues of the CiaoPP approach can be found in [4, 9, 10].

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* [16]), 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 is incompatible with the abstract specification, i.e., when $[\![P]\!]_{\alpha^+} \cap I_\alpha = \emptyset$. On the other hand, we use $[\![P]\!]_{\alpha^-}$ 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

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

Table 2. Verification problems using approximations

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-Aware Resource Usage Verification

As mentioned before, our data size-aware 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 [12], 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 [12] in order to define all the elements of the CiaoPP framework for its application to data size-aware resource usage verification.

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, $[\![P]\!]$, 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 $[\![P]\!] : \mathcal{C}_p \rightarrow \mathcal{R}$ 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).

The abstract resource usage semantics is a set of 4-tuples:

$$(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$$

where $p(\bar{v}) : c(\bar{v})$, is an abstraction of a set of calls. $\bar{v}$ is a tuple of variables and $c(\bar{v})$ is an abstraction representing a set of tuples of terms which are instances of $\bar{v}$. $c(\bar{v})$ is an element of some abstract domain expressing instantiation states. Φ, is an abstraction of the resource usage of the calls represented by $p(\bar{v}) : c(\bar{v})$. We refer to it 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, $\Psi : S \mapsto \mathcal{R}_\infty$, 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 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 $\bar{n}$. Although $\bar{n}$ is typically a tuple of natural numbers, we do not want to restrict our framework. We require that Φ be well defined so that $\forall \bar{n}\ (\Phi^l(\bar{n}) \leq \Phi^u(\bar{n}))$.*

$input_p$ is a function that takes a tuple of terms $\bar{t}$ and returns a tuple with the input arguments to p. This function can be inferred by using existing mode analysis or can be given by the user by means of assertions. $size_p(\bar{t})$ is a function that takes a tuple of terms $\bar{t}$ 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 can be automatically inferred (based on type analysis information) or can be given by the user by means of assertions [15].

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 $size_{app}((x, y)) = (length(x), length(y))$.

In order to make the presentation simpler, we will omit the $input_p$ and $size_p$ functions in abstract tuples, with the understanding that they are present in all such tuples.

Intended meaning The intended approximate meaning I_α of a program is an abstract semantic object with the same kind of tuples: $(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$, which are given in the form of assertions. The basic form of a resource usage assertion is:

`:- comp` *Pred* [`:` *Precond*] `+` *ResUsage*`.`

which expresses that for any call to *Pred*, if *Precond* is satisfied in the calling state, then *ResUsage* should also be satisfied for the computation of *Pred*. *ResUsage* defines in general an interval of numbers for the particular resource usage of the computation of the call to *Pred* (i.e., *ResUsage* is satisfied by the computation of the call to *Pred* if the resource usage of such computation is in the defined interval).

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

```
:- comp nrev(A,B): ( list(A, gnd), var(B) )
                 +  resource(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, an upper bound (`ub`) on the number of resolution `steps` performed by the computation is $1 + n^2$, where $n = length(A)$. In this case, the interval approximating the number of resolution steps is $[0, 1 + n^2]$. Since the number of resolution steps cannot be negative, the minimum of the interval is zero. If we assume that the resource usage can be negative, the interval would be $(-\infty, 1 + n^2]$. If we had a lower bound (`lb`) instead of an upper bound in the assertion, the interval would be $[1 + n^2, \infty)$.

Such an assertion describes a tuple in I_α which is given by $(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$, where $p(\bar{v}) : c(\bar{v})$ is defined by *Pred* and *Precond*, and Φ is defined by *ResUsage*. For simplicity, we assume that *Pred* is actually $p(\bar{v})$ and that there is a syntactic correspondence from *Precond* to $c(\bar{v})$, and from *ResUsage* to Φ. The information about $input_p$ and $size_p$ is implicit in *ResUsage*. The concretization of I_α, $\gamma(I_\alpha)$, is the set of all pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of terms and $p(\bar{t})$ is an instance of *Pred* that meets precondition *Precond*, and r is a number that meets the condition expressed by *ResUsage* (i.e., r lies in the interval defined by *ResUsage*) for some assertion.

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

```
( 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 because they correspond to calls which are outside the scope of the assertion (i.e., they do not meet the precondition *Precond*); those on the second line (which will never occur during execution) because they violate the assertion (i.e., they meet the precondition *Precond*, but do not meet the condition expressed by *ResUsage*).

Partial correctness: comparing the abstract semantics.

Definition 3. *Given a program p and an intended resource usage semantics I, where $I : \mathcal{C}_p \mapsto \mathcal{R}$, we say that p is partially correct w.r.t. I if for all $p(\bar{t}) \in \mathcal{C}_p$ such that r is the amount of resource usage of the computation of the call $p(\bar{t})$, we have that $I(p(\bar{t})) = r$, i.e., $(p(\bar{t}), r) \in I$. This is equivalent to the condition $[\![P]\!] \subseteq I$ given in Table 1.*

Definition 4. *We say that p is partially correct with respect to a tuple of the form $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ if for all $p(\bar{t}) \in \mathcal{C}_p$ such that r is the amount of resource usage of the computation of the call $p(\bar{t})$, it holds that: if $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$ then $r \in \Phi_I(\bar{s})$, where $\bar{s} = size_p(input_p(\bar{t}))$.*

Definition 5. *Given an intended abstract resource usage semantics I_α expressed as a set of tuples of the form $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ (each tuple is expressed by an assertion in the program), we say that p is partially correct with respect to I_α if for all $p(\bar{t}) \in \mathcal{C}_p$ such that r is the amount of resource usage of the computation of the call $p(\bar{t})$, there is a tuple $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in I_α such that $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$ and $r \in \Phi_I(\bar{s})$, where $\bar{s} = size_p(input_p(\bar{t}))$.*

Lemma 1. *p is partially correct with respect to I_α if:*

- *For all $p(\bar{t}) \in \mathcal{C}_p$, there is a tuple $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in I_α such that $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$, and*
- *p is partially correct with respect to every tuple in I_α.*

Definition 6. *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 $\sqsubseteq_f$ and the intersection operation $\sqcap_f$ as follows:*

- *$\Phi_1 \sqsubseteq_f \Phi_2$ iff for all $\bar{n} \in S$ $(S \subseteq \mathcal{R}^k)$, $\Phi_1(\bar{n}) \subseteq \Phi_2(\bar{n})$.*
- *$\Phi_1 \sqcap_f \Phi_2 = \Phi_3$ 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 tuple $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in the intended meaning I_α, and a tuple $(p(\bar{v}) : c(\bar{v}), \Phi)$ in the computed abstract semantics $[\![P]\!]_{\alpha+}$ (for simplicity, we assume the same tuple of variables $\bar{v}$ in all abstract objects).

Definition 7. *We say that $(p(\bar{v}) : c(\bar{v}), \Phi) \sqsubseteq (p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ if $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ and $\Phi \sqsubseteq_f \Phi_I$.*

Note that the condition $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ can be checked using the CiaoPP capabilities for comparing program state properties such as types or variable sharing.

Definition 8. *We say that $(p(\bar{v}) : c(\bar{v}), \Phi) \sqcap (p(\bar{v}) : c_I(\bar{v}), \Phi_I) = \emptyset$ if $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ and $\Phi \sqcap_f \Phi_I = \Phi_\emptyset$, where $\Phi_\emptyset$ is the empty function defined as follows: $\Phi_\emptyset(\bar{n}) = \emptyset$ for all $\bar{n} \in S$ $(S \subseteq \mathcal{R}^k)$.*

Lemma 2. *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 $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ and $\Phi \sqsubseteq_f \Phi_I$. For all $p(\bar{t}) \in \mathcal{C}_p$ such that r is the amount of resource usage of the computation of the call $p(\bar{t})$, it holds that: if $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$ then $p(\bar{t}) \in \gamma(p(\bar{v}) : c(\bar{v}))$ (because $c_I(\bar{v}) \sqsubseteq c(\bar{v})$), and thus $r \in \Phi(\bar{s})$, where $\bar{s} = size_p(input_p(\bar{t}))$ (because of the safety of the analysis). Since $\Phi \sqsubseteq_f \Phi_I$, we have that $r \in \Phi_I(\bar{s})$.

Lemma 3. *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. $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.*

3.2 Comparing Resource Usage Interval Functions

During verification/debugging within the framework described in the previous section, we will need to compare abstract tuples following Table 2. Thus, whenever $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ we will have to determine whether $\Phi \sqsubseteq_f \Phi_I$ or $\Phi \sqcap_f \Phi_I = \Phi_\emptyset$.

Definition 9. *Given two resource usage bound functions Ψ_1 and Ψ_2 (as in Definition 1, $\Psi_1, \Psi_2 : S \mapsto \mathcal{RI}$, where $S \subseteq \mathcal{R}^k$), we define the $\leq_f$ relation as follows:*

$$\Psi_1 \leq_f \Psi_2 \text{ iff for all } \bar{n} \in S, \text{ it holds that } \Psi_1(\bar{n}) \leq \Psi_2(\bar{n})$$

where $\leq$ represents the standard relation between real numbers augmented with the special symbols ∞ and $-\infty$. Similarly, we define $<_f, >_f$ and $\geq_f$.

Lemma 4. *Given two resource usage interval functions Φ_1 and Φ_2, we have that:*

- $\Phi_1 \sqsubseteq_f \Phi_2$ *if* $\Phi_2^l \leq_f \Phi_1^l$ *and* $\Phi_1^u \leq_f \Phi_2^u$.
- $\Phi_1 \sqcap_f \Phi_2 = \Phi_\emptyset$ *if* $\Phi_1^u <_f \Phi_2^l$ *or* $\Phi_2^u <_f \Phi_1^l$.

Corollary 1. *Let $(p(\bar{v}) : c(\bar{v}), \Phi)$ and $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ be tuples expressing an abstract semantics $[\![P]\!]_{\alpha+}$ inferred by analysis and an intended abstract semantics I_α (given in a specification) respectively, such that $c_I(\bar{v}) \sqsubseteq c(\bar{v})$, and for all $\bar{n} \in S$ ($S \subseteq \mathcal{R}^k$), $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ and $\Phi_I(\bar{n}) = [\Phi_I^l(\bar{n}), \Phi_I^u(\bar{n})]$. 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 asumption, 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.

For the particular case where resource usage bound functions depend on one argument, the result of the resource usage bound function comparison in our approach is in general a set of intervals of input data sizes for which a function is less, equal, or greater than another. This is explained in Section 4 and allows us to give intervals of input data sizes for which a program p is partially correct (or incorrect).

3.3 Dealing with Preconditions Expressing Input Data Size Intervals

Given the formalization presented in the previous sections, note that it is now straightforward to allow checking assertions which include 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). All that is required is to modify some definitions.

From the practical view, we have extended the Ciao assertion language with the new `intervals(A, B)` property (to be used in the *Precond* field of assertions), which expresses that the input data size `A` is included in some of the intervals in the list `B`, and we have made the corresponding changes in the algorithms.

To give an example, the element $c(\bar{v})$ in a tuple representing an abstract semantics should be allowed to include the property `intervals(A, B)`. Also, the operation $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ in Definition 7 should be extended to deal with data size intervals. However, in practice we can follow a simpler approach. Even if we do have an abstract domain to infer the property `intervals(A, B)`, it is safe to work with a $c(\bar{v})$ expressing only instantiation states (since $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ is preserved). Also, in practice, we can perform the assertion checking ignoring the data size intervals in $c_I(\bar{v})$, and then "filter" the intervals of the verification outcome with the intervals initially present in $c_I(\bar{v})$ (using interval intersection operations).

4 Resource Usage Bound Function Comparison

As stated in [12, 13], 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 $\Psi_2(n)$, $n \in \mathcal{R}$, 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 $\Psi_1(n) < \Psi_2(n)$.

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 (e.g., 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 we are using which is implemented in the GNU Scientific Library (GSL) 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. We refer the reader to [13] for a full description of how such approximations are performed.

Benchmark	ID	Assertion	Verif. Result	Time (mS)
fibonacci lb,ub: $1.45 * 1.62^x$ $+0.55 * -0.62^x - 1$ x = length(N)	A1	:- comp fib(N,R) + cost(ub,steps,exp(2,int(N))-1000).	F in $[0, 10]$ T in $[11, \infty]$	60
	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 $[11, 14]$	80
	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 $[0, 10]$ T in $[11, 12]$	80
Naive reverse lb,ub: $0.5x^2 + 1.5x + 1$ x = length(A)	B1	:- comp nrev(A,B) + (cost(lb,steps,length(A)), cost(ub,steps,exp(length(A),2))).	F in $[0, 3]$ T in $[4, \infty]$	36
	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 $[1, 16]$	36
Quick sort lb: $x + 5$ ub: $(\sum_{j=1}^{x} j2^{x-j}) + x2^{x-1}$ $+2 * 2^x - 1$ x = length(A)	C1	:- comp qsort(A,B) + cost(ub, steps, exp(length(A),2))	F in $[0, 2]$ C in $(2, \infty)$	44
	C2	:- comp qsort(A,B) + cost(ub, steps, exp(length(A),3))	F in $[0, 1]$ C in $(1, \infty)$	52
Client ub: $8x$ x = length(I)	D1	:- pred main(Op, I, B) + cost(ub, bits_received, exp(length(I),2)).	C in $(0, 8)$ T in $[0, 0] \cup [8, \infty]$	28
	D2	:- pred main(Op, I, B) + cost(ub, bits_received, 10*length(I)).	T in $[0, \infty]$	24
	D3	:- pred main(Op, I, B) : intervals(length(I),[i(1,10),i(100,plusinf)]) + cost(ub, bits_received, 10*length(I)).	T in $[0, 10] \cup [100, \infty]$	28
Reverse lb,ub: $4x + 6$ x = length(A)	E1	:- pred reverse(A, B) + (cost(ub, ticks, 10 * length(A) - 20)).	F in $[0, 4]$ T in $[5, \infty]$	20

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

5 Implementation and Experimental Results

The resource usage verification techniques presented in this paper have been implemented and integrated in a seamless way within the Ciao/CiaoPP framework that unifies static and run-time verification, as well as unit testing [14].

As mentioned before, for the implementation of the resource usage function comparison operations we have used the GNU Scientific Library [8]. To this end we have defined a Ciao-GSL binding through the native code (C) interface. As mentioned before, we have implemented comparisons for polynomial, exponential, logarithmic and summatory functions, the latter two through safe approximations via polynomials.

We have also performed an experimental assesment of the accuracy and efficiency of the resource usage verification techniques. Table 3 shows some experimental results obtained with our prototype implementation. The column labeled `Benchmark` shows information about the program to be verified, which includes its name and the upper (`ub`) and lower (`lb`) bound resource usage functions inferred by CiaoPP's static analysers.

The columns `ID` and `Assertion` show several assertions expressing resource usages to be statically checked, which are written by the user together with the source code (`ID` is just an identifier to facilitate discussion.). We can see that some assertions only specify upper bounds (e.g., *A1*, *C1* or *C2*), and other assertions specify both upper and lower bounds (e.g., *A2*, *A3*, *B1* or *B2*). Note also that some assertions include preconditions expressing intervals within which the input data size of the program is supposed to lie (*A3* and *D3*). The column `Check Result` shows the result of the assertion checking process, which in general express intervals of input data sizes for which the assertion is true (`T`), false (`C`) or it has not been posible to determine whether it is true of false (`C`). Finally, the column labeled `Time` shows the resource verification times in milliseconds, on a Intel Centrino 1.5 GHz with one processor, 768Mb of RAM memory, running Debian Lenny, kernel 2.6.26-2-686.

Note that we can deal with different types of resource usage functions, as for example polynomial functions (see e.g. programs *naive reverse*, *client*, and *reverse*), exponential functions (see the *fibonacci* program), and summatory functions (as in the *quick sort* program).

We can see that 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.

Table 4 shows the results of an experiment that we have performed for the case where assertions include preconditions expressing input data size intervals. The experiment consists on comparing the method described so far (referred to as `Root` in Column `Method`) with a simple method (referred to as `Eval`) consisting on evaluating the resource usage functions (i.e., the ones inferred by analysis and the ones present in the assertions) for all the values in a given input data size interval (which is a finite set of natural numbers) and comparing the results. Column `ID` refers to the assertions in Table 3. Assertion checking

ID	Method	Intervals				
		[1,12]	[1,100]	[1,1000]	[1,10000]	[1,1000] ∪ [1001,10000]
A3	Root	84	84	84	84	84
	Eval	80	84	132	644	628
D3	Root	32	32	32	32	32
	Eval	36	36	48	116	112

Table 4. Comparison of assertion checking times for two methods dealing with preconditions expressing input data size intervals.

times (in milliseconds) are shown for different input data size intervals (columns under the `Intervals` label). We can see that checking time grows quite slowly compared to the length of the interval, which grows exponentially.

6 Conclusions

We have presented several extensions and improvements to our framework for verification/debugging (implemented in the CiaoPP system) dealing with specifications about the resource usage of programs, itself and extension of the CiaoPP framework for verification of functional or program state properties. We have provided a full 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 the first experimental results, which are encouraging, suggesting that our framework is 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 *1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, volume 6234 of *Lecture Notes in Computer Science*, 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.