Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 104–113
http://www.floc-conference.org/ICLP-home.html

A Framework for Verification and Debugging of Resource Usage Properties
RESOURCE USAGE VERIFICATION

PEDRO LOPEZ-GARCIA [1,2] AND LUTHFI DARMAWAN [3] AND FRANCISCO BUENO [3]

E-mail address: pedro.lopez@imdea.org,luthfi@clip.dia.fi.upm.es,bueno@fi.upm.es

[1] IMDEA Software, Madrid, Spain

[2] Spanish Research Council (CSIC), Spain

[3] Technical University of Madrid (UPM), Spain

Abstract. We present a framework for (static) verification of general resource usage program properties. The framework extends the criteria of correctness as the conformance of a program to a specification expressing non-functional global properties, such as upper and lower bounds on execution time, memory, energy, or user defined resources, given as functions on input data sizes. A given specification can include both lower and upper bound resource usage functions, i.e., it can express intervals where the resource usage is supposed to be included in. We have defined an abstract semantics for resource usage properties and operations to compare the (approximated) intended semantics of a program (i.e., the specification) with approximated semantics inferred by static analysis. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential or logarithmic functions). A novel aspect of our framework is that the static checking of assertions generates answers that include conditions under which a given specification can be proved or disproved. For example, these conditions can express intervals of input data sizes such that a given specification can be proved for some intervals but disproved for others. We have implemented our techniques within the Ciao/CiaoPP system in a natural way, so that the novel resource usage verification blends in with the CiaoPP framework that unifies static verification and static debugging (as well as run-time verification and unit testing).

1. Introduction and Motivation

The conventional understanding of software correctness is absence of errors or bugs, expressed in terms of conformance of all possible executions of the program with a functional specification (like type correctness) or behavioral specification (like termination or possible sequences of actions). However, in an increasing number of computing applications additional observables play an essential role. For example, embedded systems must control

1998 ACM Subject Classification: D.1.6 [**Programming Techniques**]: Logic Programming; D.2.4 [**Software Engineering**]: Software/Program Verification—Assertion Checkers, Formal Methods; D.2.5 [**Software Engineering**]: Testing and Debugging. General Terms: Performance, Verification.

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

This research has been partially funded by the EU 7th. FP NoE *S-Cube* 215483, FET IST-231620 *HATS*, MICINN TIN-2008-05624 *DOVES* and CM project P2009/TIC/1465 *PROMETIDOS*.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.104

and react to the environment, which also establishes constraints about the system's behavior such as resource usage and reaction times. Therefore, it is necessary for these systems to extend the criteria for correctness with new aspects which include non-functional global properties such as maximum execution time and usage of memory, energy, or other types of resources.

In this paper we propose techniques that extend the capacity of debugging and verification systems based on static analysis [3, 2, 6], when dealing with a quite general class of properties related to resource usage, including upper and lower bounds on execution time, memory, energy, and user-defined resources (the latter in the sense of [8]). Such bounds are given as functions on input data sizes (see [8] for the different metrics that can be used to measure data sizes, such as list length, term depth, or term size). The proposed extension has been implemented in the CiaoPP framework, that unifies static verification and static debugging (as well as run-time verification and unit testing). For example, For example, it extends the capacity of CiaoPP to certify programs with resource consumption assurances and also to efficiently check such certificates.

We define an abstract semantics for resource usage properties and 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. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential or logarithmic functions). In traditional static checking, for each property of (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 that we propose 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.

Example 1.1. Consider an assertion which declares an upper bound (`ub`) on the resource usage, in terms of resolution steps, of the classical `fibonacci` program such as:

```
:- check comp fib(N,F): (int(N), var(F)) + steps_ub( exp(2, int(N))-1000 ).
```

meaning that the computation of any call to `fib(N,F)` with the first argument bound to an integer and the second one a free variable should take at most $2^x - 1000$ resolution steps, x being the size of the first argument (i.e., the actual value of `N`, since it has to be an integer number). This is true only for $x \geq 10$, and maybe programmers have tried the program only with big numbers, and then generalized their observations in the above assertion. We will see how the CiaoPP system, with our approach, is able to inform the programmer that this idea is wrong. Indeed, as we will see, the output of our assertion checking implementation within the CiaoPP system is:

```
:- false comp fib(N,F): (int(N), var(F)) + steps_ub( exp(2,int(N))-1000 ).
   in interval [0, 10] for int(N).
:- true comp fib(N,F): (int(N), var(F)) + steps_ub( exp(2,int(N))-1000 ).
   in interval [11, +inf] for int(N).
```

meaning that the system has proved that the assertion is false for values of the input argument `N` in the interval $[0, 10]$, and true for `N` in the interval $[11, \infty)$. This is because in the interval $[0, 10]$, the *lower bound* on resolution steps inferred by the analysis is greater than the upper bound expressed in the assertion, and in the interval $[11, \infty)$, the *upper bound* inferred by the analysis is less than the upper bound in the assertion.

In our approach, user specifications (i.e., assertions) can include for example lower and upper bounds, and even asymptotic values of the resource usage of the computation (given as functions on input data sizes). Moreover, a given specification can include both lower and upper bound resource usage functions, i.e., it can express intervals where the resource usage is supposed to be included in.

The most 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 ours 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.

In the following, we describe, in Section 2, how to extend and use the CiaoPP verification framework, that we take as starting point, for the verification of general resource usage program properties. In Section 3 we explain the technique that we have developed for resource usage function comparison. Section 4 summarizes our conclusions.

2. A Framework for Verification of Resource Usage Properties

The verification and debugging framework of CiaoPP [6] 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. In the framework, both semantics are given in the form of (*safe*) approximations. The abstract approximation $[\![P]\!]_\alpha$ of the concrete semantics $[\![P]\!]$ of the program is actually computed and compared directly to the (also approximate) specification, which is safely assumed to be also given as an abstract value I_α. Program verification is then performed by comparing I_α and $[\![P]\!]_\alpha$. We refer the reader to [3, 5, 6] for a detailed description of the foundations, such as conditions for safely prove partial correctness or incorrectness, and implementation issues of the framework. In this paper we concentrate on defining the main elements of the framework required for its application to resource usage properties.

Resource usage semantics. Given a program p, let 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, $p(\bar{t}) \in C_p$ 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 $[\![P]\!] : C_p \mapsto R$ where R is the set of real numbers (note that depending on the type of resource we can take other 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:

- A *resource usage bound function* for p is a monotonic arithmetic function, $\Psi : S \mapsto R_\infty$, for a given subset $S \subseteq R^k$, where R is the set of real numbers, k is the number of input arguments to predicate p and R_∞ 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.
- A *resource usage interval function* for p is an arithmetic function, $\Phi : S \mapsto RI$, where S is defined as before and 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 [8].

Example 2.1. 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.

```
:- module(reverse, [nrev/2], [assertions]).
:- use_module(library('assertions/native_props')).
:- entry nrev(A,B) : ( ground(A), list(A), var(B) ).
nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), append(R1,[H],R).
```

Figure 1: A module for naive reverse.

Intended meaning. The intended approximated meaning I_α of a program is an abstract semantic object with the same kind of tuples: $(p(\bar{v}) : c(\bar{v}), \Phi, \mathit{input}_p, \mathit{size}_p)$, which are given in the form of assertions. The basic form of resource usage assertions is:[1]

```
:- 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 2.2. In the program of Figure 1 one could use the assertion:

```
:- comp nrev(A,B): ( ground(A), list(A), 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 $n = \mathit{length}(A)$. 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 $(p(\bar{v}) : c(\bar{v}), \Phi, \mathit{input}_p, \mathit{size}_p)$, where $p(\bar{v}) : c(\bar{v})$ is defined by *Pred* and *Precond*, and Φ is defined by *ResUsage*. 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 2.3. The assertion in Example 2.2 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

[1] Assertions may be prefixed with a *status* indicating that it is to be checked, or that it has been already checked, detected to be false or detected to be true. Omitting this prefix means "to be checked" [9].

(which will never occur on 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. During verification / debugging within our framework, we need to compare an abstract semantics inferred by analysis with an intended abstract semantics. We give here some ideas about how to do it, and refer the reader to [7] for a complete formalization of the abstract semantics and comparison operations.

Given a program p and an intended resource usage semantics I, where $I : C_p \mapsto R$, we say that p is partially correct w.r.t. I if for all $p(\bar{t}) \in C_p$ we have that $(p(\bar{t}), r) \in I$, where r is precisely the amount of resource usage of the computation of the call $p(\bar{t})$. 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 $p(\bar{t}) \in C_p$ 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 $\bar{s} = size_p(input_p(\bar{t}))$. Finally, we say that p is partially correct with respect to I_α if:

- For all $p(\bar{t}) \in C_p$, 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.

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, respectively, such that $c_I(\bar{v}) \sqsubseteq c(\bar{v})$,[2] and for all $\bar{n} \in S$ ($S \subseteq R^k$), $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ and $\Phi_I(\bar{n}) = [\Phi^l_I(\bar{n}), \Phi^u_I(\bar{n})]$. We have that:

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

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

However, for simplicity, in this paper we assume that one of the endpoints of the interval is always the maximum (resp., minimum) of the possible values, i.e., $\forall \bar{n}\ (\Phi^u_I(\bar{n}) = \infty)$ (resp., $\Phi^l_I(\bar{n}) = -\infty$ or $\Phi^l_I(\bar{n}) = 0$, depending on the resource). Thus, one of the resource usage bound function comparisons in each of the two cases above is always trivial. Therefore, we will be faced with only one such comparison, between two resource usage bound functions, each denoting either a lower bound (l) or an upper bound (u).

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 allows us to give intervals of input data sizes for which a program p is partially correct (or incorrect).

3. Resource Usage Bound Function Comparison

Given two resource usage bound functions (one of them inferred by the static analysis and the other one given in an assertion/specification present in the program), $\Psi_1(n)$ and $\Psi_2(n)$, $n \in R$ 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 $\Psi_1(n) < \Psi_2(n)$.

[2] 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.

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 and logarithmic), it is also desirable to be able to compare all of these functions. For polynomial functions there exist powerful algorithms for obtaining roots. For the other functions, we have to approximate them using polynomials. In this case, we should guarantee that the error falls in the safe side when comparing the corresponding resource usage bound functions.

3.1. Comparing Polynomial Functions

There are general methods for finding roots of polynomial equations. Root equation finding of polynomial functions can be done analytically until polynomial order four. For higher order polynomial functions, numerical methods must be used. According to the fundamental theorem of algebra, a polynomial equation of order m has m roots, whether real or complex numbers. Numerical methods exist that allow computing all these roots (although the complex numbers are not needed in our approach).

For this purpose in our implementation we have used the GNU Scientific Library [4], 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.

3.2. Approximation of Non-Polynomial Functions

There are two non-polynomial resource usage functions that the CiaoPP analyses can infer: exponential and logarithmic. For approximating these functions we use Taylor series.

Exponential function approximation using polynomials. This approximation is carried out using these formulae:

$$e^x \approx \Sigma_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \qquad for\ all\ x$$

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

In our implementation these series are limited up to order 8. This decision has been taken based on experiments we have carried out that show that higher orders do not bring a significant difference in practice. Also, in our implementation, the computation of the factorials is done separately and the results are kept in a table in order to reuse them.

Logarithmic function approximation using polynomials. Unfortunately this approximation cannot be done in a straightforward way as previously. A Taylor series for this function for whole interval does not exist, the series only holds for interval $-1 < x < 1$. One possibility to work within this restriction is using range reduction [10].

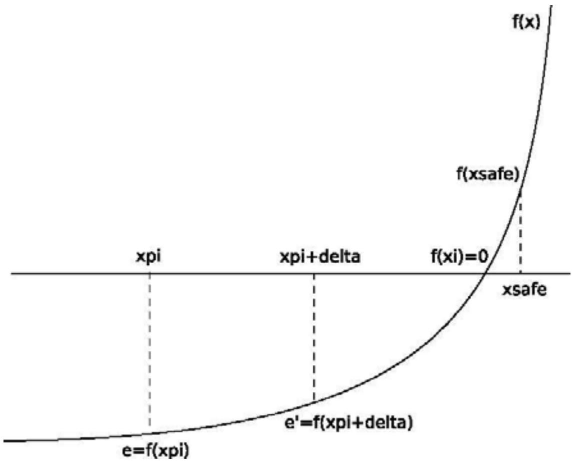

Figure 2: Case 1. $x_i > xp_i$ (since $e' > e$). A safe approximate root found is x_{safe}.

3.3. Safety of the Approximation

Since the roots obtained for function comparison are in some cases approximations of the real roots, we must guarantee that their values are safe, i.e., that they can be used for verification purposes, in particular, for safely checking the conditions in (1) and (2) (page 109.) Assume for example that we are going to safely check whether $\Phi^u(x) \leq \Phi^u_I(x)$ (where Φ^u and Φ^u_I are resource usage bound functions, the former is a result of program analysis and the latter an assertion declared in the program). In this case, we define $f(x) = \Phi^u_I(x) - \Phi^u(x)$, so that we can safely say that if $f(x) > 0$, then $\Phi^u(x) \leq \Phi^u_I(x)$. Assume also that Φ^l_I is not given in the assertion, meaning that the specification does not state any lower bound for the resource usage (i.e., the lower endpoint of any resource usage interval is $-\infty$, which means that $\Phi^l_I(x) \leq \Phi^l(x)$ is always true). We can then safely state that the assertion is true for all x such that $f(x) > 0$. In the same way, if we define $f(x) = \Phi^l(x) - \Phi^u_I(x)$ we can safely say that if $f(x) > 0$ then, $\Phi^u_I(x) < \Phi^l(x)$, proving that the assertion is false for all x such that $f(x) > 0$. We can reason similarly in the comparisons involving a lower bound in the assertion (Φ^l_I). Thus, we focus exclusively on safely determining values for x such that $f(x) > 0$, where $f(x)$ is conveniently defined in each case. Let us see how it can be performed.

In general, we approximate $f(x)$ using a polynomial $P(x)$, so that $f(x) = P(x) + -e$, with e being an approximation error. Let the roots of equation $f(x) = 0$ be $x_0, ..., x_n$. Using a root finding algorithm on equation $P(x) = 0$, we obtain the roots $xp_0, ..., xp_n$, so that we have $P(xp_i) = 0$, and therefore $f(xp_i) \in [-e, +e]$. Then, we have to determine, for each approximated root xp_i, $1 \leq i \leq n$, a value ε such that $f(xp_i{+}\varepsilon) > 0$ and $x_i \in [xp_i{-}\varepsilon, xp_i{+}\varepsilon]$. We do this by first determining the relative position of xp_i and x_i (i.e., whether xp_i is "to the right" or "to the left" of x_i) and then starting an iterative process that increments (or decrements) xp_i by some δ until we have that, after m iterations, $f(xp_i + m\ \delta) > 0$.

Determining the relative position of the exact root. To determine the relative position of the exact root and its approximated value we use the gradient of $f(x)$ around $x = xp_i$. For determining the gradient we use the values of $e = f(xp_i)$ and $e' = f(xp_i + \delta')$, with $\delta' > 0$ a relatively small number. Whether the approximated root is greater or less than the exact root depends on the following conditions:

(1) if $e < 0$ and $e' > e$ then $x_i > xp_i$
(2) if $e > 0$ and $e' > e$ then $x_i < xp_i$
(3) if $e > 0$ and $e' < e$ then $x_i > xp_i$
(4) if $e < 0$ and $e' < e$ then $x_i < xp_i$

From Figure 2 we can see the rationale behind the first case (the other cases follow an analogous reasoning). If $e' > e$ then $f(x)$ is increasing, but, since $e < 0$, then $f(x) > 0$ can only occur for values of x greater than xp_i. Therefore, $x_i > xp_i$. In such cases we use a positive value of δ for the iterative process. When $x_i < xp_i$ we use a negative value of δ.

Iterative process for computing the safe root. Once we have determined the relative position of the exact root and its approximated value, we first set up the appropriate sign for δ, where $|\delta|$ is a relatively small number: $\delta < 0$ if the iteration should go to the left ($x_i < xp_i$), or $\delta > 0$ if it should go to the right ($x_i > xp_i$). Then we iterate on the addition $xp_i = xp_i + \delta$ until $f(xp_i) > 0$ (if $e < 0$) or $f(xp_i) < 0$ (if $e > 0$). Such an iteration is apparent in the following pseudo-code:

```
$xsafe \leftarrow xp_i$
if $f(xp_i) < 0$ then
    while $f(xsafe) < 0$ do $xsafe \leftarrow xsafe + \delta$
    end while
else ($f(xp_i) > 0$)
    while $f(xsafe) > 0$ do $xsafe \leftarrow xsafe + \delta$
    end while
end if
return $xsafe$
```

Example 3.1. Consider again the assertion in Example 1.1 in Section 1, which declares an upper bound on resource usage given by function $\Phi^u_I(x) = 2^x - 1000$. Let the analysis infer a lower bound $\Phi^l(x) = 1.45 \times 1.62^x - 1$. Their intersection occurs at $x \approx$10.22. However, the root obtained by our root finding algorithm is $x \approx 10.89$. By doing an iterative approximation from 10.89 to the left, we finally obtain a safe approximate root of $x \approx$10.18.

Note that usually (as in the above example), resource usage functions are on variables which range on natural numbers. Because of this, the iterative approximation process for safe roots can be substituted by simply taking the closest natural number to the left or right of the approximated root (depending on the gradient) to obtain a safe value. In the previous example, we will simply take 10, without any iteration.

It turns out that the analysis also infers an upper bound given by function $\Phi^u(x) = 1.45 \times 1.62^x - 1$. Thus, the output of our assertion checking for the `fibonacci` program will be that of Example 1.1, showing extra conditions (an interval of integers) on which the assertion can be proved false, on one hand, and another condition (the rest of the range of the positive integer numbers) on which it can be proved true, on the other hand.

4. Conclusions

We have proposed a method for extending how a framework for verification/debugging (implemented in the CiaoPP system) deals with specifications about the resource usage of programs. We have provided a formalization which blends in with the previous framework for verification of functional or program state properties. A key aspect of the framework is to be able to compare mathematical functions. We have proposed a method which is safe, in the sense that the results of verification/debugging cannot go wrong. In the case where the resource usage functions being compared depend on one variable (which represents some input argument size) our method reveals particular numerical intervals for such variable, if they exist, which might result in different answers to the verification problem: a given specification might be proved for some intervals but disproved for others. Our current method computes such intervals with precision for polynomial and exponential resource usage functions, and in general for functions that can be approximated by polynomials. Moreover, we have proposed an iterative post-process to safely tune up the interval bounds by taking as starting values the previously computed roots of the polynomials.

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)*, Lecture Notes in Computer Science. Springer, 2009. To appear.

[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. 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.

[4] 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. Library and Manual also available at http://www.gnu.org/software/gsl/.

[5] 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.

[6] 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.

[7] P. López-García, L. Darmawan, F. Bueno, and M. Hermenegildo. Towards a Framework for Resource Usage Verification and Debugging in the CiaoPP System. Technical Report CLIP1/2010.0, Technical University of Madrid (UPM), School of Computer Science, UPM, February 2010. Available at http://cliplab.org/papers/resource-verif-10-tr.pdf.

[8] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *ICLP'07*, number 4670 in LNCS, pages 348–363, 2007.

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

[10] Jyri Ylostalo. Function approximation using polynomials. *Signal Processing Magazine*, 23:99–102, September 2006.