*TLP* **18** (2): 167–223, 2018. © Cambridge University Press 2018
doi:10.1017/S1471068418000042

# *Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption**

P. LOPEZ-GARCIA
*IMDEA Software Institute, Madrid, Spain*
*Spanish Council for Scientific Research (CSIC), Madrid, Spain*
(*e-mail:* `pedro.lopez@imdea.org`)

L. DARMAWAN
*IMDEA Software Institute, Madrid, Spain*
(*e-mail:* `luthfi.darmawan@imdea.org`)

M. KLEMEN and U. LIQAT
*IMDEA Software Institute, Madrid, Spain*
*Universidad Politécnica de Madrid (UPM), Madrid, Spain*
(*e-mails:* `maximiliano.klemen@imdea.org, umer.liqat@imdea.org`)

F. BUENO
*Universidad Politécnica de Madrid (UPM), Madrid, Spain*
(*e-mail:* `bueno@fi.upm.es`)

M. V. HERMENEGILDO
*IMDEA Software Institute, Madrid, Spain*
*Universidad Politécnica de Madrid (UPM), Madrid, Spain*
(*e-mail:* `manuel.hermenegildo@imdea.org`)

*submitted 16 February 2017; revised 28 March 2018; accepted 29 March 2018*

## Abstract

Many applications require conformance with specifications that constrain the use of resources, such as execution time, energy, bandwidth, etc. We present a configurable framework for static resource usage verification where specifications can include data size-dependent resource usage functions, expressing both lower and upper bounds. Ensuring conformance with respect to such specifications is an undecidable problem. Therefore, to statically check such specifications, our framework infers the same type of resource usage functions, which safely approximate the actual resource usage of the program, and compares them against the specification. We review how this framework supports several languages and compilation output formats by translating them to an intermediate representation based on Horn clauses and using the configurability of the framework to describe the resource semantics of the input language. We provide a detailed formalization and extend the framework so that both resource usage specification and analysis/verification output can include preconditions expressing intervals

* This research was partially funded by EU FP7 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft*, and TIN2015-67522-C3-1-R *TRACES* projects, and the Madrid M141047003 *N-GREENS* program.

for the input data sizes for which assertions are intended to hold, proved, or disproved. Most importantly, we also extend the classes of functions that can be checked. We also report on and provide results from an implementation within the Ciao/CiaoPP framework, as well as on a practical tool built by instantiating this framework for the verification of energy consumption specifications for imperative/embedded programs. Finally, we show as an example how embedded software developers can use this tool, in particular, for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

*KEYWORDS*: static analysis, resource usage analysis and verification, Horn clause-based analysis and verification, energy consumption, program verification and debugging.

## 1 Introduction and motivation

The conventional understanding of software correctness is the 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, ranging from those running on devices with limited resources (e.g., the ones used in *Internet of Things* applications, sensors, smart watches, smart phones, portable/implantable medical devices, or mission critical systems), to large data centers and high-performance computing systems, it is also important and sometimes essential to ensure conformance with respect to specifications expressing non-functional global properties such as energy consumption, maximum execution time, memory usage, 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., a portable or implantable medical device, an autonomous space vehicle, or even a mobile phone), if the application makes the device run out of batteries earlier than required, making the whole system useless in practice. In general, high-performance embedded systems must control, react to, and survive in a given environment, and this in turn establishes constraints about the system's performance parameters including energy consumption and reaction times. Therefore, a mechanism is necessary in these systems in order to prove correctness with respect to specifications about such non-functional global properties.

In the previous work, we have developed a general approach to the automated verification based on a novel combination of assertion-based partial specifications, static analysis, run-time checking, and testing (Bueno *et al.*, 1997; Hermenegildo *et al.*, 1999; Puebla *et al.*, 2000b; Hermenegildo *et al.*, 2005; Mera *et al.*, 2009), and which has been implemented in the CiaoPP framework. In addition to different functional properties (supported by "pluggable" *abstract domains*[1]), such as types,

[1] By pluggable abstract domains, we refer to the fact that in CiaoPP, new abstract domains can be integrated easily as modules implementing a well-defined interface. This interface connects each

modes, or groundness, this framework can also deal with a large class of properties related to resource usage, including upper and lower bounds on execution time, memory, energy, and, in general, user-definable resources (the latter in the sense of Navas *et al.*, 2007, 2009). Such bounds are given as functions on input data sizes (see Navas *et al.*, 2007 for the different metrics that can be used to measure data sizes, such as list length, term depth, or term size).

In order to make our framework parametric with respect to programming languages and program representations at different compilation levels, each *input language* supported (e.g., Java source, Java bytecode, XC source, Ciao, LLVM intermediate representation—LLVM IR, or Instruction Set Architecture—ISA) is translated into an *intermediate program representation which is based on Horn clauses* Méndez-Lojo *et al.* (2007)—see Figure 1. All analysis and verification is performed on this Horn clause-based representation, that we will refer to as "HC IR" from now on. That is, given program $p$ in an input language $L_p$ plus a definition of the semantics of $L_p$, $p$ is translated into a set of Horn clauses capturing the semantics of the program, $[\![p]\!]$, or an abstraction of it, $[\![p]\!]_\alpha$ (see Section 2 for a description of this notation). A Horn clause (HC) is a first-order predicate logic formula of the form $\forall(S_1 \wedge \ldots \wedge S_n \rightarrow S_0)$, where all variables in the clause are universally quantified over the whole formula, and $S_0, S_1, \ldots, S_n$ are atomic formulas, also called literals. It is usually written $S_0\ :-\ S_1, \ldots, S_n$. This HC IR consists of a set of connected code *blocks*, each block represented by a *Horn clause*: $< block\_id > (< params >)\ :-\ S_1,\ \ldots\ , S_n$. Each such block has an entry point, that we call the *head* of the block (to the left of the $:-$ symbol), with a number of parameters $< params >$, and a sequence of steps (the *body*, to the right of the $:-$ symbol). Each of these $S_i$ steps (or *literals*) is either (the representation of) a *call* to another (or the same) block or an operation. Such operations depend on the input language represented, i.e., they can be bytecode instructions (from a Java bytecode program), ISA instructions (from an ISA program), calls to built-ins or constraints (from a logic program), LLVM instructions, etc. The semantics of each bytecode, instruction, built-in, etc. is provided compositionally to the analyzers by means of *trust* assertions (see Section 2.2). In the case of resources, the set of these assertions constitutes the *resource model* (see Figs. 1 and 4). The HC IR representation offers a good number of features that make it very convenient for analysis such as supporting naturally static single assignment (SSA) and recursive forms, making all variable scoping explicit, reducing the semantics of all constructs (loops, conditionals, switches, etc.) to a simple form, etc. Méndez-Lojo *et al.* (2007).

The CiaoPP analyzers handle the HC IR uniformly, regardless of its origin. In particular, the resource analysis infers resource usage functions in terms of input data sizes, for all the predicates in the HC IR program, which are then reflected back to the input language or representation also as assertions. This analysis can infer different classes of resource usage functions such as, e.g., polynomial, exponential,

abstract domain to the built-in abstract interpretation algorithms (the "fixpoints"), giving rise to different program analyzers. The same interface also connects the domains to other parts of the system that are based on abstractions, such as, e.g., the abstract partial evaluators.

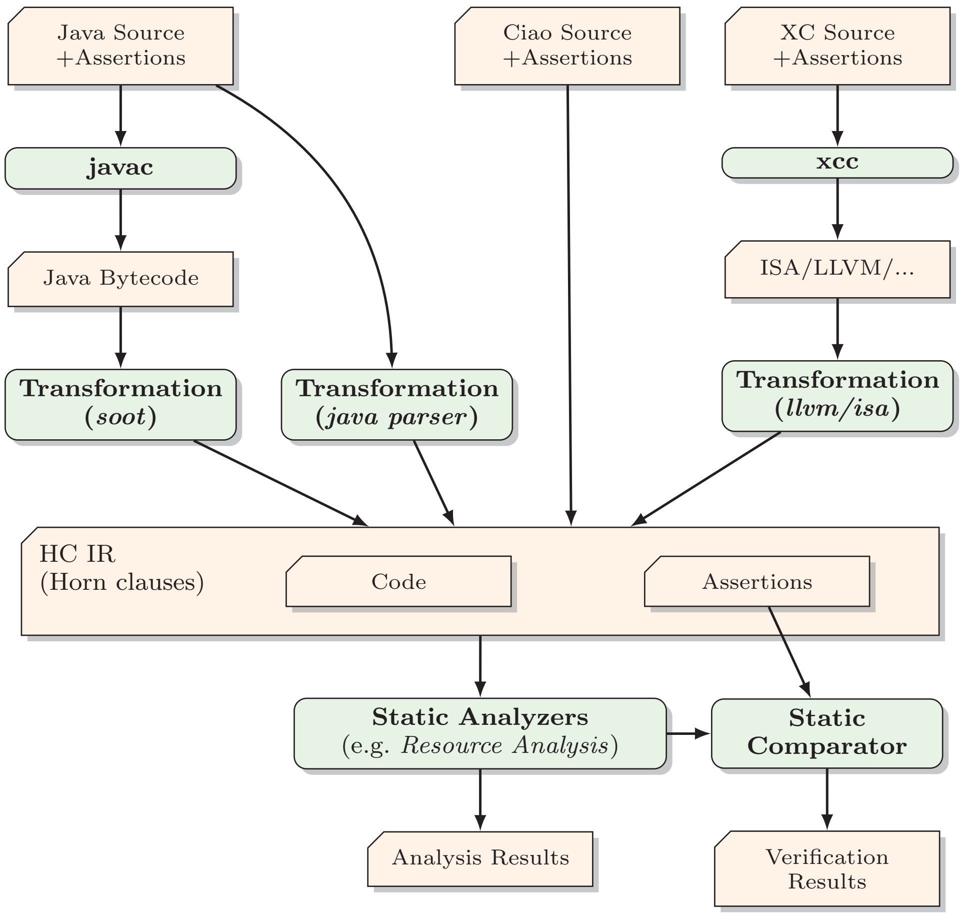


Fig. 1. Overview of the framework for analysis and verification of different input programming languages, using Horn clauses as intermediate language.

summation, or logarithmic, using the techniques of Debray *et al.* (1990), Debray *et al.* (1997), Debray and Lin (1993), Navas *et al.* (2007), Navas *et al.* (2008), and Serrano *et al.* (2014). *Verification* implies comparing specifications (in our case, the resource consumption specifications, given in the form of assertions) against analysis results. Our focus in this paper is on this comparison process, rather than on the resource analysis, which is described in Navas *et al.* (2007), Serrano *et al.* (2014) and references therein. We do not cover the debugging aspect either, i.e., process of finding the cause of an assertion violation. Since both static analysis and verification are in general undecidable our techniques used are necessarily approximate. Nevertheless, such approximations are *safe*, in the sense that they are guaranteed to be correct considering all possible executions, i.e., they provide correct answers or return "unknown."

```
#pragma check fact(n)
        : (1 <= n) ==> (6.0 <= energy_nJ <= 2.3*n+9.0)

int fact(int N) {
  if (N <= 0) return 1;
  return N * fact(N - 1);
}
```

Fig. 2. An XC source (factorial) function.

*Example 1*
Assume that we are interested in verifying specifications about *energy consumption*. Consider, for example, the recursive factorial function definition `fact` in Figure 2, written in the XC C-style language (Watt, 2009). The ISA program corresponding to it is generated using the XC compiler, XCC (left-hand side of Fig. 3). The resulting ISA program is passed to a translator (see Fig. 1) that generates the associated Horn clauses (right-hand side of Fig. 3). Such HC IR program, together with the information contained in the energy models at the ISA level (represented also by using assertions, see Fig. 4 for a simple example), is passed to the resource analysis (as represented in Fig. 1), which outputs the energy consumption analysis results and the verification results for all procedures in the HC IR program. More specifically, the energy model provides the information on the energy consumed by basic operations (ISA instructions in this case). This information is taken (trusted) by the static analyzer, which propagates it, during the abstract interpretation of the program, through code segments, conditionals, loops, recursions, etc., mimicking the actual execution of the program with symbolic "abstract" data instead of concrete data, in order to infer energy consumption functions for higher-level entities, such as procedures and functions in the program. The analysis of recursive procedures gives rise to recurrence equations, whose closed form solutions are the resource usage functions, which depend on input data sizes, resulting from the analysis. The XC assertion:

```
#pragma check fact(n) : (1 <= n) ==> (6.0 <= energy_nJ <= 2.3*n+9.0).
```

is a resource usage specification that also gets translated into the HC IR representation to be checked by CiaoPP (the Ciao assertion language (Puebla *et al.*, 2000a; Hermenegildo *et al.*, 2012)), as shown in lines 1–3 in Fig. 3 (right):[2]

```
:- check pred fact(N,Ret) : intervals(nat(N),[i(1,inf)])
                + costb(energy_nJ,6.0,2.3*nat(N)+9.0).
```

The assertion expresses that the cost of `fact(N,Ret)`, in terms of the resource "energy in nano-Joules,"[3] must lie in the interval $[6.0, 2.3 * \texttt{nat(N)} + 9.0]$ nJ. In the HC IR representation, the return values of functions are represented as additional arguments (`Ret` as second argument to `fact`). The assertion uses the `costb/3` property for expressing both a lower and an upper bound, in the second and

[2] See Section 6 for further details on specifications in XC syntax and Section 2.2 for their counterpart in the HC IR.
[3] 1 nano-Joule = $10^{-9}$ Joules.

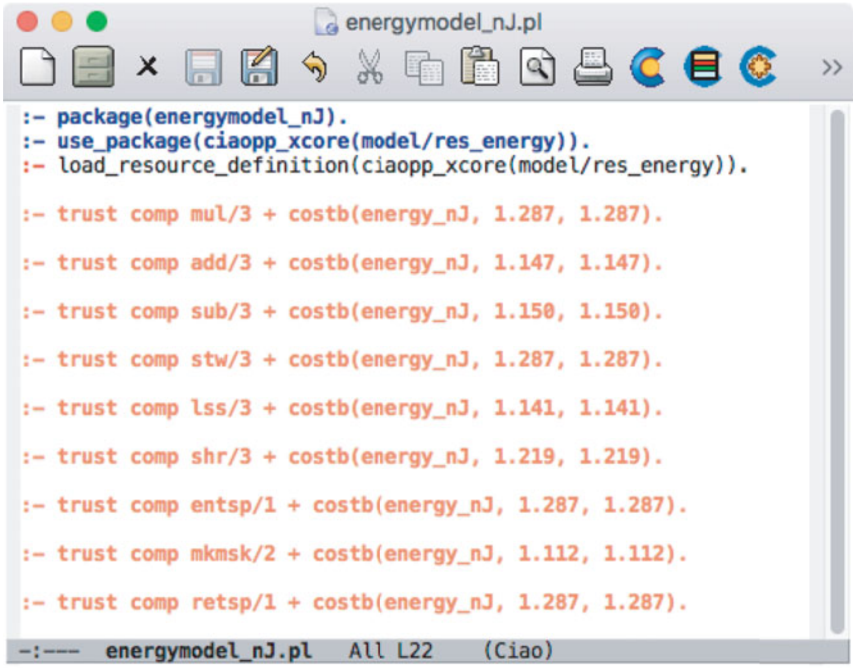
```
.
.
.
.

<fact>:
001: entsp 0x2
002: stw   r0, sp[0x1]
003: ldw   r1, sp[0x1]
004: ldc   r0, 0x0
005: lss   r0, r0, r1
006: bf    r0, <008>



007: bu    <010>
010: ldw   r0, sp[0x1]
011: sub   r0, r0, 0x1
012: bl    <fact>

013: ldw   r1, sp[0x1]
014: mul   r0, r1, r0
015: retsp 0x2


008: mkmsk r0, 0x1
009: retsp 0x2
```

```
:- check pred fact(N, Ret)
   : intervals(nat(N),[i(1,inf)])
   + costb(energy_nJ,6.0,
           2.3*nat(N)+9.0).

fact(R0,R0_3) :-
   entsp(0x2),
   stw(R0,Sp0x1),
   ldw(R1,Sp0x1),
   ldc(R0_1,0x0),
   lss(R0_2,R0_1,R1),
   bf(R0_2,0x8),
   fact_aux(R0_2,Sp0x1,R0_3,R1_1).

fact_aux(1,Sp0x1,R0_4,R1) :-
   bu(0x0A),
   ldw(R0_1,Sp0x1),
   sub(R0_2,R0_1,0x1),
   bl(fact),
   fact(R0_2,R0_3),
   ldw(R1,Sp0x1),
   mul(R0_4,R1,R0_3),
   retsp(0x2).

fact_aux(0,Sp0x1,R0,R1) :-
   mkmsk(R0,0x1),
   retsp(0x2).
```

Fig. 3. ISA program for Figure 2 (left) and its Horn-clause representation (right).

third arguments, respectively, on a cost given in terms of a particular resource, in the first argument. The `intervals/2` property specifies the set of input sizes, under a particular metric, for which the assertion has to be checked. The first argument indicates the input argument that is being considered, together with the corresponding size metric. The second argument indicates the set of values as a union of intervals, represented by a list of `i/2` properties, which in this example contains only one interval, $(1, \infty)$. It provides bounds on the energy to be consumed by `fact(N,Ret)` given as functions on the size of the input argument `N`. Since such argument is numeric, the size metric used is its "non-negative value," defined as $\texttt{nat(N)} \overset{\text{def}}{=} \max(0, \texttt{N})$. The $nat(N)$ size metric is applied to a numeric variable $N$, not to arithmetic expressions. However, our size analysis understands arithmetic expressions, and can give the size of an output argument as an arithmetic function that depends on the $nat(N)$ values of variables that represent input arguments.

As mentioned before, the verification of resource usage specifications is performed by comparing the abstract intended semantics (i.e., the resource usage specifications) with the safe approximation of the concrete semantics inferred by the resource analysis. We say that a program property $\phi^\#$ is a *safe approximation* of a property $\phi$, if the set of program traces, where $\phi$ holds is included in the set of program traces where $\phi^\#$ holds. The idea of using safe approximations is further explained in Section 2. In our original work on resource usage verification, reported, e.g., in Hermenegildo *et al.* (2005) and previous papers, for each property expressed in 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 for the cost functions involved in the comparisons to have intersections, so that for some input data sizes one of them is smaller than the



```
:- package(energymodel_nJ).
:- use_package(ciaopp_xcore(model/res_energy)).
:- load_resource_definition(ciaopp_xcore(model/res_energy)).

:- trust comp mul/3 + costb(energy_nJ, 1.287, 1.287).

:- trust comp add/3 + costb(energy_nJ, 1.147, 1.147).

:- trust comp sub/3 + costb(energy_nJ, 1.150, 1.150).

:- trust comp stw/3 + costb(energy_nJ, 1.287, 1.287).

:- trust comp lss/3 + costb(energy_nJ, 1.141, 1.141).

:- trust comp shr/3 + costb(energy_nJ, 1.219, 1.219).

:- trust comp entsp/1 + costb(energy_nJ, 1.287, 1.287).

:- trust comp mkmsk/2 + costb(energy_nJ, 1.112, 1.112).

:- trust comp retsp/1 + costb(energy_nJ, 1.287, 1.287).
```

Fig. 4. A simple energy model, expressed in the Ciao assertion language.

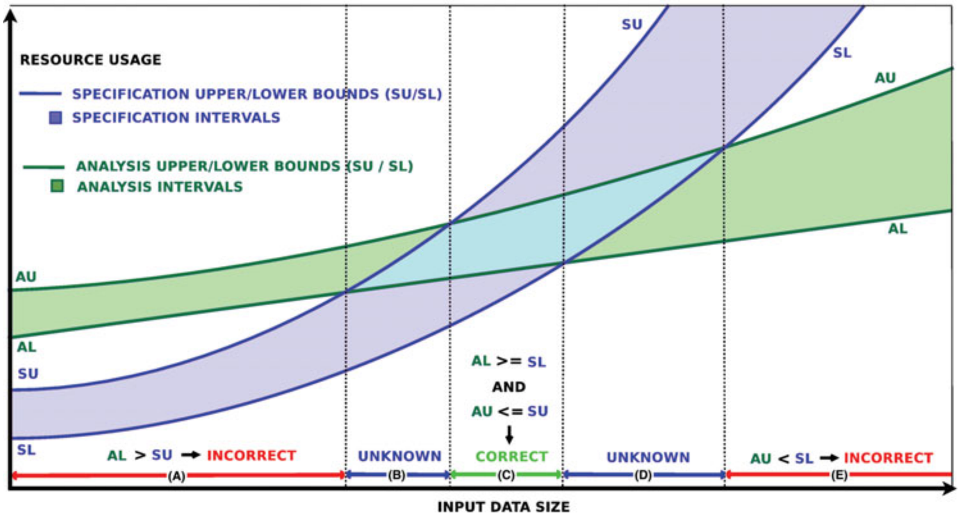


Fig. 5. Interval-based resource usage verification.

other one, and for others, it is the other way around. The first major contribution of this paper is to generalize our approach so that the answers of the comparison process can now 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 sizes or value ranges. In particular, the outcome of the comparison process can now be that the original specification holds for input data sizes that lie within a given set of intervals, does not hold for other intervals, and the result may be unknown for some others. This is illustrated in Figure 5. We can see that the specification gives both a lower and upper bound cost function, so that for any input data size $n$ (ordinate axis), the specification expresses that the resource usage of the computation with input data of that size must lie in the interval determined by both functions (which depend on $n$). Similarly, the bound cost functions inferred by the static analysis determine a resource usage interval for any $n$, in which the resource usage of the computation (with input data of size $n$) is granted to lie. We can see that in the (input data size) interval $C$ in the ordinate axis, the program is correct (i.e., it meets the specification), because for any $n$ in such interval, the resource usage intervals inferred by the analysis are included in those expressed by the specification. In contrast, the program is incorrect in the data size intervals $A$ and $E$ because the resource usage intervals inferred by the analysis and those expressed by the specification are disjoint. In interval $A$, this is proved by the sufficient condition that says that the lower bound cost function inferred by the analysis is greater than the upper bound cost function expressed in the specification (in that interval). A similar reasoning applies to the interval $E$ (using the upper bound of the analysis and the lower bound of the specification). However, nothing can be ensured for the intervals $B$ and $D$. This is because for any data size $n$ in such intervals, the resource usage of the computation for some input data of size $n$ may lie within the interval expressed by the specification; but for other input data of the same size, the resource usage may lie outside the interval expressed by the specification.

Furthermore, intervals can now also appear in specifications, i.e., our approach can check specifications that include preconditions expressing intervals of input data sizes. In that case, the data size intervals automatically generated by the system are subintervals of the ones given in the specification by the user.

*Example 2*

Continuing with Example 1, using the techniques proposed herein (and the prototype implemented), the outcome of static checking for the assertion in Figures 2 and 3 is the following set of assertions:

```
:- false pred fact(N,Ret) : intervals(nat(N),[i(1,1),i(13, inf)])
           + ( costb(energy_nJ, 6.0, 2.3*nat(N)+9.0) ).

:- checked pred fact(N,Ret) : intervals(nat(N),[i(2,12)])
             + ( costb(energy_nJ, 6.0, 2.3*nat(N)+9.0) ).
```

meaning that the specification does not hold for values of $n$ belonging to the interval $[1,1] \cup [13,\infty]$, and that it does hold for values of $n$ in the interval $[2,12]$,

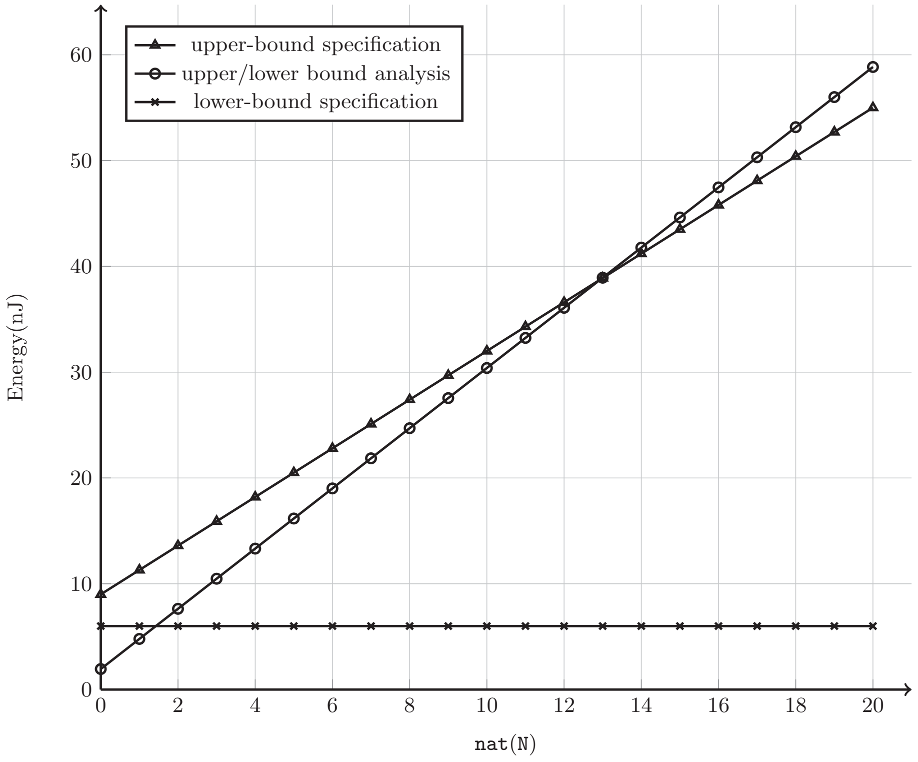


Fig. 6. Resource usage functions for the factorial program: Specification and analysis results.

where $n$ = nat(N). In order to produce that outcome, first, CiaoPP's resource analysis infers the upper and lower bound functions for the energy consumption of the factorial program, which in this particular case are both the same: the function $(2.845\ n + 1.94)$ nJ, which obviously implies that this is the *exact* cost function for `fact/2`. It is depicted as a continuous line in Figure 6. Thus, the resource usage of the computation of `fact/2` with input data of a given size $n$ is granted to lie in the resource usage interval $[2.845\ n + 1.94, 2.845\ n + 1.94]$.[4] These functions are then compared against the specification resource functions, depicted in Figure 6 as dashed lines. For any value $n$ (ordinate axis) of the input data size in the interval [2, 12], the resource usage interval inferred by CiaoPP (i.e., $[2.8\ n + 1.9, 2.8\ n + 1.9]$) is included in the resource usage interval expressed by the specification, namely, $[6.0, 2.3\ n+9.0]$. Therefore, after performing the resource usage function comparison, using the techniques that we present, CiaoPP's output indicates that the assertion is *checked* in that data size interval. Conversely, the assertion is reported as *false* for $n = 1$ or $n \in [13,\ \infty]$, because for this interval the lower bound resource usage function inferred by the analysis is greater than the upper

[4] As mentioned before, we refer the reader to (Navas *et al.*, 2007; Serrano *et al.*, 2014) for more details on the user-definable version of the resource analysis and references.

bound resource usage function expressed in the specification (and consequently, the corresponding resource usage intervals are disjoint).

The process of checking of resource usage specifications against the analysis information obviously involves the comparison of arithmetic functions. In our previous work (again, see Hermenegildo *et al.*, 2005 and references therein), the approach to cost function comparison was relatively simple, basically consisting on performing function normalization and then using some syntactic and asymptotic comparison rules. The second major contribution of this work is to provide stronger techniques for this purpose, extending the types of functions that can be dealt with in the specifications and in the analysis results to a much larger class. We also provide the benchmarking results for the proposed interval based, function comparison techniques.

As a final contribution, and in order to illustrate the usefulness of the techniques developed, we report on a specialization of the proposed framework for a practical application: verifying energy consumption specifications, i.e., comparing inferred energy bound functions and specifications. We study the particular case of programs written in the XC language and running on the XMOS XS1-L architecture, already illustrated in the previous examples. However, using our Horn-clause translation approach, the proposed approach and its implementation in CiaoPP are general and can be applied to the resource verification of other programming languages and architectures. We also illustrate through a case study, how embedded software developers can use the tool developed, in particular, for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

This paper unifies, improves, and extends our previous work in Lopez-Garcia *et al.* (2012, 2010, and 2015), especially by adding operations that allow dealing with a richer set of usage functions, including summation, exponential, and logarithmic cost functions, as well as multi-variable functions (see Section 4). We also present a more detailed formalization than in Lopez-Garcia *et al.* (2012) and López-García *et al.* (2010).

The overall contributions of this work can be summarized as follows:

- We have developed a configurable framework for static resource usage verification, where specifications can include data size-dependent resource usage functions, expressing both lower and upper bounds.
- We have extended the criteria of software correctness to resource usage specifications. In particular, we have defined a resource usage semantics and its approximation, and devised sufficient conditions for program correctness/incorrectness based on such semantics.
- We have defined operations to check such sufficient conditions that compare the (possibly abstract) intended semantics of a program with approximated semantics inferred by static analysis. Such comparison can deal with a rich class of resource usage functions (polynomial, summation, exponential, logarithmic), as well as multi-variable functions.

- Our framework produces a refined output of the assertion checking process, that may determine a partition of the set of possible input values (by inferring intervals for input data sizes), in place of a unique interval of values. Each subinterval of such partition may correspond to different outcomes of the verification.
- Our framework also deals with *specifications* containing assertions that include preconditions expressing intervals for the input data sizes.
- We have implemented a prototype and provided experimental results.
- We have specialized our framework for its application to the energy consumption verification of imperative (XC) programs.

In the rest of the paper, Section 2 provides an overview of the foundations of the CiaoPP verification framework, and of the Ciao assertion language used for specifications. Then, Section 3 describes how this traditional framework is extended for the data size, *interval-dependent* verification of resource usage properties, presenting also the formalization of the framework. In particular, we define an abstract semantics for resource usage properties and operations to compare the (approximated) intended semantics of a program with approximated semantics inferred by static analysis. Section 4 presents our extended techniques for the comparison of (arithmetic) resource usage functions. Section 5 reports on the implementation of our techniques within the Ciao/CiaoPP system, providing experimental results. Section 6 describes a specialization of the framework for its application to the energy consumption analysis of XC programs, and explains how embedded software developers can use this tool in the case study already mentioned. Finally, Section 7 discusses related work and Section 8 summarizes our conclusions.

## 2 Basics of the verification framework

This section summarizes some relevant parts of our previous work in Hermenegildo *et al.* (2005) and previous papers (Bueno *et al.*, 1997; Hermenegildo *et al.*, 1999; Puebla *et al.*, 2000b), that together form the basis for the resource usage verification techniques described in the following sections, which are the contributions of this paper. The section is based mostly on Bueno *et al.* (1997), which provides a basic introduction to abstract verification from a conceptual point of view. A more detailed description of the verification framework can be found in Puebla *et al.* (2000b).

As mentioned before, the verification framework of CiaoPP uses analyses, based on the abstract interpretation technique, which are provably correct and also practical, in order to statically compute safe approximations of the program semantics. These safe approximations are compared with specifications, in the form of assertions that are written by the programmer, in order to prove such specifications correct or incorrect. In the following, we restrict ourselves to the important class of fixpoint semantics. Under these assumptions, the meaning of a program $p$, i.e., its *concrete semantics*, denoted by $[\![p]\!]$, is the least fixpoint of a monotonic operator associated with the program $p$, denoted $S_p$, i.e., $[\![p]\!] = \mathrm{lfp}(S_p)$. Such operator is a function defined on a domain $D$, which we assume to be a complete lattice. We will refer to $D$ as the *concrete* domain. We will assume for simplicity that the elements of $D$ are sets and that the order relation in $D$ is set inclusion.

In the abstract interpretation technique, a domain $D_\alpha$ is defined, called the *abstract* domain, which also has a lattice structure and is *simpler* than the domain $D$. In particular, $D$ is finite or, if the lattice contains infinite ascending chains, the abstract domain defines operations that accelerate the convergence of the fixpoint computation, ensuring termination. 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 connection (Cousot and Cousot, 1977). Abstract operations over $D_\alpha$ 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 $D_\alpha$, is computed (or approximated) by interpreting the abstraction of the program $p$ over the abstract domain $D_\alpha$. One of the fundamental results of abstract interpretation is that an abstract semantic operator $S_p^\alpha$ for a program $p$ can be defined which is correct w.r.t. $S_p$ in the sense that $\gamma(\text{lfp}(S_p^\alpha))$ is an approximation of $[\![p]\!]$, and, if certain conditions hold, then the computation of $\text{lfp}(S_p^\alpha)$ (i.e., the analysis of $p$) 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 $p$, its abstract semantics, as $[\![p]\!]_\alpha$.

Typically, abstract interpretation guarantees that $[\![p]\!]_\alpha$ is a safe *over*-approximation of the abstraction of the concrete 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 abstraction of the concrete semantics of $p$, i.e., to meet the condition $[\![p]\!]_\alpha \subseteq \alpha([\![p]\!])$. In this case, we use the notation $[\![p]\!]_{\alpha^-}$ to express that the result of the analysis, $[\![p]\!]_\alpha$, meets such a condition.

Program verification compares the *concrete 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 concrete or intended semantics for automatic verification is in general not realistic, since the concrete semantics is typically only partially known, infinite, too expensive to compute, etc. Since the technique of abstract interpretation allows computing safe approximations of the program semantics, the key idea of the CiaoPP approach (Bueno *et al.*, 1997; Hermenegildo *et al.*, 1999; Puebla *et al.*, 2000b) is to use the abstract approximation $[\![p]\!]_\alpha$ directly in program verification tasks (and in an integrated way with other techniques such as run-time checking and with the use of assertions).

### *2.1 Abstract verification*

In the CiaoPP framework, the abstraction $[\![p]\!]_\alpha$ of the concrete semantics $[\![p]\!]$ of the program is actually computed and compared directly to the abstract intended

Table 1. *Set theoretic formulation of verification problems*

| 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 not partially correct w.r.t. $I$ | $[\![p]\!] \not\subseteq I$ |
| $p$ is incomplete w.r.t. $I$ | $I \not\subseteq [\![p]\!]$ |

Table 2. *Verification problems using approximations*

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

semantics, which is given in terms of *assertions* (Puebla *et al.*, 2000a), following almost directly the scheme of Table 1. A program specification $I_\alpha$ is an abstract value $I_\alpha \in D_\alpha$, where $D_\alpha$ is the abstract domain of computation. Program verification is then performed by comparing $I_\alpha$ and $[\![p]\!]_\alpha$. Table 2 shows sufficient conditions for correctness and completeness w.r.t. $I_\alpha$, which can be used when $[\![p]\!]$ is approximated. Several instrumental conclusions can be drawn from these relations.

Analyses, which over-approximate the concrete semantics (i.e., those denoted as $[\![p]\!]_{\alpha^+}$), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification $I_\alpha$. 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 concrete semantics. In such case, it will be possible to prove completeness and incorrectness.

Since most of the properties being inferred are in general undecidable, the technique used to infer such properties, in our case abstract interpretation, is necessarily approximate. Nevertheless, such approximations are also always guaranteed to be safe, in the sense that they are never incorrect, i.e., they are strict over- (conversely under-) approximations of a property for the set of all possible program behaviors.

## 2.2 *Expressing $I_\alpha$: A relevant subset of the Ciao assertion language*

In order to instantiate the language used to express the intended semantics, $I_\alpha$, and, in particular, resource usage properties, we introduce the assertion language that we will use throughout the paper. These assertions are part of the Ciao assertion language. For brevity, we only introduce here the class of "`pred`" assertions, since they suffice for our purposes. We refer the reader to Puebla *et al.* (2000a); Hermenegildo *et al.*

(2005, 2012) and references therein for a full description of the Ciao assertion language.

*Pred assertions:* These assertions follow the schema:

$$\boxed{\texttt{:- pred } \textit{Pred} \texttt{ [: } \textit{Precond}\texttt{] [=> } \textit{Postcond}\texttt{] [+ } \textit{Comp-Props}\texttt{].}}$$

where *Pred* is a predicate symbol applied to distinct free variables, and *Precond* and *Postcond* are logic formulae about execution states. An execution state is defined by the set of variable/value bindings associated with a given execution step. The assertion indicates that in any call to *Pred*, if *Precond* holds in the calling state and the computation of the call succeeds, then *Postcond* should hold in the success state. Also, the set of *Precond*s for all the *pred* assertions for a given *Pred* describes all the possible call states, i.e., for any call state for a predicate, there must be at least one *pred* assertion for that predicate whose *Precond* holds in that state.

A new property we introduced in this work and used throughout the paper is the following (see Section 3.2 for further details):

$$\boxed{\textbf{intervals}(Size_A\textbf{, [}Int_1\textbf{, \dots, }Int_n\textbf{])}}$$

which expresses that the size $Size_A$ for a given argument $A$ belongs to some of the intervals in the list $[Int_1, \dots, Int_n]$, where $Int_j = \texttt{i}(Lo, Up)$, $j \geqslant 1$ and $\{Lo, Up\} \in \mathbb{R} \cup \{\texttt{inf}\}$. Finally, the *Comp-Props* field (appearing after the "+" operator) is used to describe properties of the whole computation for calls to predicate *Pred* that meet *Precond*. In our application, the *Comp-Props* are precisely the resource usage properties. As already shown in Example 1, another global non-functional property we introduce in this paper is `costb/3`, which expresses such resource usages, and follows the schema:

$$\boxed{\textbf{costb(}\textit{Res\_Name}\textbf{, }\textit{Low\_Arith\_Expr, Upp\_Arith\_Expr)}}$$

where *Res_Name* is a user-provided identifier for the resource the assertion refers to *Low_Arith_Expr* and *Upp_Arith_Expr* are arithmetic functions that map input data sizes to resource usages, representing, respectively, lower and upper bounds on the resource consumption. Similarly to `costb/3`, the `cost/3` property allows expressing only one resource usage function on input data sizes that follows this schema:

$$\boxed{\textbf{cost(}\textit{Bound\_Type}\textbf{, }\textit{Res\_Name}\textbf{, }\textit{Arith\_Expr}\textbf{)}}$$

where *Res_Name* is the same as in `costb/3`, *Arith_Expr* is similar to *Low_Arith_Expr* and *Upp_Arith_Expr* in `costb/3`, but it can be either upper or lower bound depending on the value of *Bound_Type,* which are `lb` for lower bounds and `ub` for upper bounds. This is illustrated in Example 6.

*Example 3*
Figure 7 shows an assertion for a typical `append/3` predicate. The assertion states that for any call to predicate `append/3` with the first and second arguments bound to lists and the third one unbound, where the length of the first list lies in the interval $[1, \infty]$, it holds that if the call succeeds, then the third argument will also be bound to a list. It also states that `length(A) + 1` is both a lower and upper

```
:- pred append(A, B, C)
   : ( list(A), list(B), var(C),
       intervals(length(A),[1,inf])
     )
  => list(C)
   + costb(steps, length(A)+1, length(A)+1).
```

Fig. 7. An example Ciao *resource* assertion for `append/3`.

bound on the number of resolution steps required to execute any of such calls. The property `length/1` represents a size metric, in particular, the length of a list. In this case, the assertion expresses an exact cost, since the lower- and upper-bound cost functions coincide.

*Assertion status:* Each assertion has an associated *status*, marked with one of the following prefixes, placed just before the `pred` keyword: `check` (indicating that the assertion is to be checked), `checked` (the assertion has been checked and proved correct by the system), `false` (it has been checked and proved incorrect by the system; a compile-time error is reported in this case), `trust` (the assertion provides information coming from the programmer in order to guide the analyzer, and it will be trusted), or `true` (the assertion is a result of static analysis and thus correct, i.e., it is a safe approximation of the concrete semantics). The default status, i.e., if no status appears before `pred`, is `check`.

## 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 the classical model (Bueno *et al.*, 1997; Hermenegildo *et al.*, 2005) and (López-García *et al.*, 2010), that include preconditions expressing intervals within which the input data size of a program is supposed to lie (Lopez-Garcia *et al.*, 2012).

We start by providing a formalization of our data size-dependent resource usage verification framework, assuming that the programs that we are dealing with are written in the HC IR language (i.e., they are logic programs). However, as mentioned before, the techniques apply to other languages, by applying our transformation to Horn clauses. Furthermore, the concepts are in fact also applicable directly to other languages, with some adaptations and changes in terminology.

### *3.1 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 (not necessarily ground), $p(\bar{t}) \in C_p$ is a call to $p$ 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 (Selective Linear Definite clause 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 $r$ it refers to. The concrete resource usage semantics can be defined as a relation $[\![p]\!] \subseteq C_p \times \mathbb{R}$, where $\mathbb{R}$ 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). Such relation is usually a function. In other words, the domain $D$ of the concrete semantics is $2^{C_p \times \mathbb{R}}$, so that $[\![p]\!] \in D$. Recall that, as described in Section 2, $D$ is a complete lattice, and the abstract domain, $D_\alpha$ has also a lattice structure. The concretization and abstractions functions ($\gamma$ and $\alpha$, respectively) are mappings that relate both domains, altogether composing a Galois connection Cousot and Cousot (1977).

We define an abstract domain $D_\alpha$ 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 $\Phi$ is an abstraction of the resource usage of such calls. We refer to such pairs as *call-resource* pairs. Specifically, $\bar{v}$ is a tuple of variables and $c$ is a property on terms, so that $p(\bar{v}) : c(\bar{v})$ represents the set of all calls $p(\bar{t})$ such that $\bar{v} = \bar{t} \rightarrow c(\bar{v})$ holds.

The abstraction $c(\bar{v})$ is some subset of the abstract domains available for the analyzer, i.e., those loaded in the CiaoPP system, expressing program states. An example of $c(\bar{v})$ (in fact, the one used in Section 5 in our experiments) is a combination of properties that are in the domain of the regular type analysis, *eterms* Vaucheret and Bueno (2002), and properties such as groundness and freeness present in the *shfr* abstract domain (Muthukumar and Hermenegildo, 1992). For conciseness, we refer to such combination as the mode/type abstract domain. A regular type is a set of terms, which is the language accepted by a (possibly non-deterministic) *finite tree automaton*, although regular types can be expressed using several type representations. Internally, the *eterms* regular type analysis Vaucheret and Bueno (2002) uses a representation based on *regular term grammars*, equivalent to Dart and Zobel (1992) but with some adaptations. This analysis produces abstractions, represented by using *regular term grammars*, that over-approximate the set of terms that can occur at all program points. Such abstractions are presented to the user in the form of predicates, as will be illustrated later.

We refer to $\Phi$ 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, $\Psi_p : S \mapsto \mathbb{R}_\infty$, for a given subset $S \subseteq \mathbb{N}^k$, where $\mathbb{N}$ is is the set of natural numbers, $k$ is the number of input arguments to predicate $p$, and $\mathbb{R}_\infty$ is the set of real numbers augmented with the special symbols $\infty$ 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, $\Phi : S \mapsto RI$, 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*

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

:- entry nrev(A,B) : (list(A, gnd), var(B)).
:- check pred nrev(A,B)
     + costb(steps, length(A), 10*length(A)).

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

Fig. 8. A module for the naive reverse program.

*bound functions* that denote the lower and upper endpoints of the interval $\Phi(\bar{n})$, respectively, for the tuple of input data sizes $\bar{n}$.[5] We require that $\Phi$ be well defined so that $\forall \bar{n}\ (\Phi^l(\bar{n}) \leqslant \Phi^u(\bar{n}))$.

Intuitively, $\Phi$ defines a resource usage band, and $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ is resource usage interval.

In order to relate the elements $p(\bar{v}) : c(\bar{v})$ and $\Phi$ in a call-resource pair as the one described previously, we assume the existence of two functions $input_p$ and $size_p$ associated with each predicate $p$ in the program. Assume that $p$ has $k$ arguments and $i$ input arguments ($i \leqslant k$). 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 analysis that infer groundness, freeness and sharing information, 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 (Navas *et al.*, 2007).

*Example 4*
Consider, for example, the naive reverse (Ciao) Prolog program in Figure 8, with the classical definition of predicate `append`. The first argument of `nrev/2` 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:

$$input_{nrev}((x, y)) = (x), input_{app}((x, y, z)) = (x, y),$$
$$size_{nrev}((x)) = (length(x)) \text{ and } size_{app}((x, y)) = (length(x), length(y)).$$

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

$$\forall E \in D_\alpha, \gamma(E) = \bigcup_{e \in E} \gamma_1(e)$$

where $\gamma_1$ is another concretization function, applied to call-resource pairs $e$'s of the form $(p(\bar{v}) : c(\bar{v}), \Phi)$. We define:

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

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

where $\gamma_m$ is the concretization function of the mode/type abstract domain. We use the subscript $m$ as a short name for such a mode/type domain for conciseness. The concretization function $\gamma_1$ returns a set of concrete pairs $(p(\bar{t}), r)$. As already stated, each such set is an element of the concrete domain $D = 2^{C_p \times \mathbb{R}}$, where $\bar{t}$ 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 complete computation of the call $p(\bar{t})$.

*Example 5*
Assume that $p$ is the predicate `nrev` in Figure 8, $\bar{v}$ is $(x, y)$, and $c(\bar{v})$ is the property defined as the conjunction $\texttt{list}(x) \wedge \texttt{var}(y)$, represented as $(\texttt{list}(x), \texttt{var}(y))$ in the assertions, since we use the comma (,) as the symbol for the conjunction operator. The property `list(_)` is a *regular type*, which can be inferred by CiaoPP by performing the analysis with the *eterms* abstract domain Vaucheret and Bueno (2002), and is represented as a predicate:

```
list([]).
list([H|R]) :- list(R).
```

The property `var(_)` can also be inferred by CiaoPP, with the *shfr* abstract domain (Muthukumar and Hermenegildo, 1992).

Under these assumptions, $\gamma_m(c(\bar{v}))$ is the infinite set:

$$\gamma_m(c(\bar{v})) = \gamma_m(\texttt{list}(x) \wedge \texttt{var}(y)) = \{([], y), ([a], y), ([a, b], y), ([a, b, c], y), \ldots\}.$$

Assume also that $input_{nrev}((x, y)) = (x)$ and $size_{nrev}((x)) = (length(x))$, as explained in Example 4. Let $\{e_\alpha\} \in D_\alpha$, such that:

$$e_\alpha \equiv ((nrev(x, y) : (\texttt{list}(x) \wedge \texttt{var}(y))), [\Phi^l_{nrev}, \Phi^u_{nrev}]),$$

where the resource usage bound functions $\Phi^l_{nrev}$ and $\Phi^u_{nrev}$ are defined as:

$$\Phi^l_{nrev}(n) = 2 \times n, \text{ and } \Phi^u_{nrev}(n) = 1 + n^2.$$

We have that $([a, b, c], y) \in \gamma_m(\texttt{list}(x) \wedge \texttt{var}(y))$ and $size_{nrev}(input_{nrev}([a, b, c], y)) = size_{nrev}([a, b, c]) = length([a, b, c]) = 3$. Thus, $\Phi^l_{nrev}(3) = 2 \times 3 = 6$ and $\Phi^u_{nrev}(3) = 1 + 3^2 = 10$, which means that any pair $(nrev([a, b, c], y), r)$ such that $r \in [6, 10]$, belongs to $\gamma_1(e_\alpha)$, e.g., $(nrev([a, b, c], y), 6) \in \gamma_1(e_\alpha)$ and $(nrev([a, b, c], y), 7) \in \gamma_1(e_\alpha)$. Therefore, we have that $\gamma_1(e_\alpha) = e$, where $e \in D$ is the infinite set:

$$e = \{(nrev([], y), 0), (nrev([], y), 1), (nrev([a], y), 2), (nrev([a, b], y), 4),$$
$$(nrev([a, b], y), 5), (nrev([a, b, c], y), 6), (nrev([a, b, c], y), 7), (nrev([a, b, c], y), 10) \ldots\}$$

Finally, $\gamma(\{e_\alpha\}) = \gamma_1(e_\alpha) = e$.

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

*Intended meaning*. As already mentioned, the intended semantics is an expression of the user's expectations, and is typically only partially known. For this reason, it is in general not realistic to use the exact intended semantics and we use an approximated intended semantics instead. We define the approximated intended semantics $I_\alpha$ 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 *call-resource* pairs defining the approximated intended semantics are provided by the user by means of the Ciao assertion language, introduced in Section 2.2, while the pairs corresponding to the approximated semantics of the program 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.

As mentioned in Section 2.2, we will be using `pred` assertions. The most common syntactic schema of a `pred` assertion that describes resource usage and its correspondence to the *call-resource* pair it represents is the following:

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

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 $\Phi$. 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. As already said, 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 $\Phi$ is a conjunction of `costb/3` or `cost/3` properties.

Assuming that $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$, where $\bar{n} = size_p(input_p(\bar{v}))$, $\Phi$ can be represented in the resource usage assertion as the conjunction:

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

or, alternatively, using the `costb/3` property:

$$\texttt{costb}(r, \Phi^l(\bar{n}), \Phi^u(\bar{n}))$$

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

*Example 6*
In the program of Figure 8 one could use the assertion:

```
:- pred 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. $1 + length(A)^2$). The property `list(_,_)` is represented as a higher order predicate:

```
list([], T).
list([H|R], T) :- T(H), list(R).
```

and the property `gnd(_)`, expressing "groundness," can also be inferred by CiaoPP, with the *shfr* abstract domain Muthukumar and Hermenegildo (1992).

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 4, 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 $(-\infty, 1 + n^2]$. This information can be given by the user when providing the assertions that constitute the definition of a particular resource and its cost model (which expresses the resource usage of basic elements of a program/language). A detailed description of our user-definable resource analysis framework is given in Navas *et al.* (2007). Similarly, if the upper bound (`ub`) is omitted, the upper limit of the interval is assumed to be $\infty$.

*Example 7*
The assertion in Example 6 is applicable for the following concrete semantic pairs:

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

but it is not applicable to 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 do not meet the assertion's precondition $c(\bar{v})$: the leftmost one because `nrev/2` 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/2` 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 $\Phi$.

### 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 $I_\alpha$ of the intended semantics $I$, where $I_\alpha$ is given as a set of *call-resource* pairs of the form $(p(\bar{v}) : c(\bar{v}), \Phi)$.

*Definition 3 (Input-size set)*
Let $e_\alpha$ be a call-resource abstract pair $(p(\bar{v}) : c(\bar{v}), \Phi)$. We define the *input-size set* of $e_\alpha$, denoted $input\_size\_set(e_\alpha)$ as the set $\{\bar{n} \mid \exists\, \bar{t} \in \gamma_m(c(\bar{v})) \wedge \bar{n} = size_p(input_p(\bar{t}))\}$. The input-size set is represented as an interval (or a union of intervals). We obviously require that $input\_size\_set(e_\alpha) \subseteq Dom(\Phi)$ for any call-resource abstract pair $e_\alpha$, where $Dom(\Phi)$ denotes the domain of function $\Phi$.

*Definition 4*
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., $p(\bar{t}) \in C_p$ and $r$ is the amount of resource usage of the computation of the call $p(\bar{t})$), it holds that: if $\bar{t} \in \gamma_m(c_I(\bar{v}))$ and $\bar{n} = size_p(input_p(\bar{t}))$, then $r \in \Phi_I(\bar{n})$, where $\gamma_m$ is the concretization function of the mode/type abstract domain.

*Lemma 1*
$p$ is partially correct with respect to $I_\alpha$, 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 $I_\alpha$ such that $\bar{t} \in \gamma_m(c_I(\bar{v}))$, and
- $p$ is partially correct with respect to every pair in $I_\alpha$.

Note that the notion of $p$ being partially correct with respect to a call-resource pair $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ is different from the notion of $p$ being partially correct with respect to a singleton set $\{(p(\bar{v}) : c_I(\bar{v}), \Phi_I)\}$, i.e., an intended semantics: if for all $(p(\bar{t}), r) \in [\![p]\!]$ it holds that $\bar{t} \notin \gamma_m(c_I(\bar{v}))$, then $p$ is partially correct with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ but $p$ is not partially correct with respect to $\{(p(\bar{v}) : c_I(\bar{v}), \Phi_I)\}$.

As mentioned before, we use a safe over-approximation of the program semantics $[\![p]\!]$, that we denote $[\![p]\!]_{\alpha+}$, and is automatically computed by the static analysis in Navas *et al.* (2007) and Serrano *et al.* (2014) as a set of *call-resource* pairs of the form $(p(\bar{v}) : c(\bar{v}), \Phi)$. For simplicity, we assume that $[\![p]\!]_{\alpha+}$ is a set made up of a single call-resource pair. 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 (Navas *et al.*, 2007; Serrano *et al.*, 2014) and references therein. The safety of such resource usage analysis can be expressed as follows.

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

Let $c_1(\bar{v})$ and $c_2(\bar{v})$ be two elements of the mode/type abstract domain already mentioned, each one representing a set of calls. The inclusion operator $\sqsubseteq_m$ is the order relation in such abstract domain, and meets the condition: $c_1(\bar{v}) \sqsubseteq_m c_2(\bar{v})$ if and only if $\gamma_m(c_1(\bar{v})) \subseteq \gamma_m(c_2(\bar{v}))$. In our case, we use the comparison operator $\sqsubseteq_m$ implemented in the CiaoPP system, which uses finer grain comparison operators for program state properties. In particular, it uses the type comparison operator of the *eterms* abstract domain (Vaucheret and Bueno, 2002) (based on adaptations of the type inclusion operations of Dart and Zobel (1992)) and the mode comparison operator of the *shfr* abstract domain (Muthukumar and Hermenegildo, 1992) (which represents *groundness* and *freeness* properties).

*Example 8*
Let $c_1(\bar{v})$ be $\texttt{list}(x,\texttt{gnd}) \wedge \texttt{var}(y)$, and $c_2(\bar{v})$ be $\texttt{list}(x) \wedge \texttt{var}(y)$. Note that the property $\texttt{list}(x, y)$, of arity 2, already defined in Example 6, is a higher-order predicate that succeeds if $x$ is a list whose elements meet the property $y$ of arity 1. The property $\texttt{gnd}(z)$, of arity 1, is true if $z$ is a ground term (i.e., it does

not contain any unbound variable). Thus, $\texttt{list}(x, \texttt{gnd})$ succeeds if $x$ is a list of ground elements. Note also that since the higher order predicate $\texttt{list}(x, y)$ assumes that the second argument $y$ is a first-order predicate of arity 1, we just need to specify the name of that predicate when calling $\texttt{list}(x, y)$. In this example, we just specify the name gnd, although $\texttt{list}(x, \texttt{gnd})$ will call $\texttt{gnd}(z)$ for all elements $z$ of $x$ when running. The property $\texttt{list}(x)$, of arity 1, defined in Example 5, is a first-order predicate that succeeds if $x$ is a list, and its elements can be any term, including ground terms, variables, etc. In other words, $\texttt{list}(x)$ does not impose any constraint on the type of the elements of $x$. Thus, in this example, we have that $c_1(\bar{v}) \sqsubseteq_m c_2(\bar{v})$, but $c_2(\bar{v}) \not\sqsubseteq_m c_1(\bar{v})$. Similarly, $(\texttt{list}(x, \texttt{gnd}) \wedge \texttt{var}(y)) \sqsubseteq_m (\texttt{gnd}(x) \wedge \texttt{var}(y))$, but $(\texttt{gnd}(x) \wedge \texttt{var}(y)) \not\sqsubseteq_m (\texttt{list}(x, \texttt{gnd}) \wedge \texttt{var}(y))$.

*Definition 5*
Let $\Phi_1$ and $\Phi_2$ be two resource usage interval functions, i.e., $\Phi_1 : Dom(\Phi_1) \mapsto RI$, and $\Phi_2 : Dom(\Phi_2) \mapsto RI$, where $Dom(\Phi_1) \subseteq \mathbb{R}^k$ and $Dom(\Phi_2) \subseteq \mathbb{R}^k$. Let $S$ be a set such that $S \subseteq Dom(\Phi_1)$ and $S \subseteq Dom(\Phi_2)$. We define the inclusion relation $\sqsubseteq_S$ and the intersection operation $\sqcap_S$ as follows:

- $\Phi_1 \sqsubseteq_S \Phi_2$ if and only if for all $\bar{n} \in S$, $\Phi_1(\bar{n}) \subseteq \Phi_2(\bar{n})$.
- We say that $\Phi_1 \sqcap_S \Phi_2 = \Phi_3$ if and only if for all $\bar{n} \in S$, $\Phi_1(\bar{n}) \cap \Phi_2(\bar{n}) = \Phi_3(\bar{n})$.

*Definition 6*
Let $e_I$ be a pair $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in the intended meaning $I_\alpha$, and $e_\alpha$ 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. We say that $e_\alpha \sqsubseteq e_I$ iff $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$ and $\Phi \sqsubseteq_S \Phi_I$, where $S = input\_size\_set(e_I)$.

Note that the condition $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$ is needed to ensure that we select resource analysis information that can safely be used to verify the assertion corresponding to the pair $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$. If $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$, then $input\_size\_set(e_I) \subseteq input\_size\_set(e_\alpha)$.

*Definition 7*
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.

*Theorem 1*
Let $e_\alpha = (p(\bar{v}) : c(\bar{v}), \Phi)$ and $[\![p]\!]_{\alpha+} = \{e_\alpha\}$. Let $e_I = (p(\bar{v}) : c_I(\bar{v}), \Phi_I)$. If $e_\alpha \sqsubseteq e_I$ then $p$ is partially correct with respect to $e_I$.

*Proof*
If $e_\alpha \sqsubseteq e_I$, then $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$ (by Definition 6), what implies that $\gamma_m(c_I(\bar{v})) \subseteq \gamma_m(c(\bar{v}))$ and hence $input\_size\_set(e_I) \subseteq input\_size\_set(e_\alpha)$. We are going to prove that the condition of Definition 4 holds. 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 $\gamma_m(c_I(\bar{v})) \subseteq \gamma_m(c(\bar{v}))$), and thus $r \in \Phi(\bar{n})$, where $\bar{n} = size_p(input_p(\bar{t}))$ (by Lemma 2). Since $\Phi \sqsubseteq_S \Phi_I$, where $S = input\_size\_set(e_I)$ (Definition 6), and $input\_size\_set(e_I) \subseteq input\_size\_set(e_\alpha)$, we have that $r \in \Phi_I(\bar{n})$. $\square$

Similarly, we have the following result:

*Theorem 2*
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 not partially correct w.r.t. $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

In order to prove or disprove program partial correctness, we compare call-resource pairs by using Theorems 1 and 2 (thus, ensuring the sufficient conditions given in Table 2). This means that whenever $c_I(\bar{v}) \sqsubseteq_m c(\bar{v})$, 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 the following Corollary 1 of Theorems 1 and 2.

*Corollary 1*
Let $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ be a pair in the intended abstract semantics $I_\alpha$ (given in a specification), and $[\![p]\!]_{\alpha+} = \{(p(\bar{v}) : c(\bar{v}), \Phi)\}$ the abstract semantics 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 $\forall \bar{n} \in S : (\Phi_I^l(\bar{n}) \leqslant \Phi^l(\bar{n}) \wedge \Phi^u(\bar{n}) \leqslant \Phi_I^u(\bar{n}))$, then $p$ is partially correct with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$;
2. if $\forall \bar{n} \in S : (\Phi^u(\bar{n}) < \Phi_I^l(\bar{n}) \vee \Phi_I^u(\bar{n}) < \Phi^l(\bar{n}))$, then $p$ is not partially correct with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

Note that the sufficient condition 1 (resp., 2) above implies that $\Phi \sqsubseteq_S \Phi_I$ (resp. $\Phi \sqcap_S \Phi_I = \Phi_\emptyset$, where, as already said, $\Phi_\emptyset$ represents the constant function identical to the empty interval. In practice, we also use the condition $(\forall \bar{n} \in S : \Phi^u(\bar{n}) < \Phi_I^l(\bar{n})) \vee (\forall \bar{n} \in S : \Phi_I^u(\bar{n}) < \Phi^l(\bar{n}))$, although it is stronger than condition 2. When $\Phi_I^u$ (resp., $\Phi_I^l$) 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.

*Inferring preconditions on data sizes for different verification outcomes.* 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 partition $S$ in a number of $n_S$ subsets $S_j$, $1 \leqslant j \leqslant n_S$, for which either condition holds. Thus, as a result of the verification of $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$, we produce a set of pairs $(p(\bar{v}) : c_I^j(\bar{v}), \Phi_I)$, $1 \leqslant j \leqslant n_S$, whose input-size set is $S_j$. Such pairs will be represented as assertions in the output of our implementation prototype.

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 (or is not) partially correct.

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 added to the Ciao assertion language a new `intervals(A, B)` property, which expresses that the input data size `A` belongs to 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 $d_{f_1}, \dots, d_{f_b}$ are the computed input data size intervals for such pairs, an assertion with the following syntactic schema is produced as output:

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

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

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

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

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

The description of how the input data size intervals $d_j$'s are computed is given in Section 4.

*Dealing with preconditions expressing input data size intervals.* So far, we have seen that a call-resource pair in the intended semantics $I_\alpha$ has the form $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$, where $c_I(\bar{v})$ is a conjunction of type and mode properties that is used to represent a set of calling data to $p$. In order to allow checking assertions that 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 also allow adding conjuncts to $c_I(\bar{v})$ that are constraints over the sizes of the data represented by $c_I(\bar{v})$. Such constraints can represent intervals for such data sizes. Accordingly, we replace the concretization function $\gamma_m$ by an extended version $\gamma'_m$. To this end, 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 several 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.

In the case where there are multi-variable resource usage bound functions, instead of intervals represented as pairs of numbers, we use arithmetic expressions that represent more general *size constraints* (see Section 4.7), usually inequalities. In this case, the interval $d$ above will be replaced by the set of values that satisfy such size constraints.

## 4 Resource usage bound function comparison

Fundamental to our approach to verification are the operations that compare two cost bound functions. In particular, sufficient conditions 1 and 2 of Corollary 1 for proving and disproving program correctness and incorrectness, respectively, involve comparisons of a cost bound function inferred by the static analysis with another given in a specification as an assertion present in the program.

Since our resource analysis is able to infer different types of functions (e.g., polynomial, exponential, summation, logarithmic, factorial, etc.), it is also desirable to be able to compare as many classes as possible of these functions.

Assume that we have to compare two cost functions $f(\bar{x})$ and $g(\bar{x})$ that depend on input data sizes $\bar{x} \in S$ for a given input-size set $S$. Also, given a function $f(\bar{x})$, let $f^l(\bar{x})$ and $f^u(\bar{x})$ denote a lower and an upper bound on $f(\bar{x})$, respectively, i.e., $\forall \bar{x} \in S : f^l(\bar{x}) \leqslant f(\bar{x})$ and $\forall \bar{x} \in S : f(\bar{x}) \leqslant f^u(\bar{x})$. In the cases in which the techniques we will describe in the following sections cannot be applied to give sound results for a given comparison, say $\forall \bar{x} \in S : f(\bar{x}) \leqslant g(\bar{x})$, then we replace any of the functions by an upper or lower bound on it, in a way that ensures obtaining sufficient conditions for such comparison. This is expressed by the following lemma whose proof is obvious.

*Lemma 3*
Let be $f(\bar{x})$ and $g(\bar{x})$ be cost functions and $S$ an input-size set. Then,

1. if any of the conditions:

$$\forall \bar{x} \in S : f^u(\bar{x}) \leqslant g^l(\bar{x}),$$
$$\forall \bar{x} \in S : f^u(\bar{x}) \leqslant g(\bar{x}), \text{ or}$$
$$\forall \bar{x} \in S : f(\bar{x}) \leqslant g^l(\bar{x})$$

   holds, then $\forall \bar{x} \in S : f(\bar{x}) \leqslant g(\bar{x})$ holds; and
2. if $\forall \bar{x} \in S : f^u(\bar{x}) \neq f(\bar{x})$ and $\forall \bar{x} \in S : g^l(\bar{x}) \neq g(\bar{x})$, then any of the conditions above is also a sufficient condition for $\forall \bar{x} \in S : f(\bar{x}) < g(\bar{x})$.

### *4.1 Single-variable cost function comparison*

We define two operations for comparing cost functions, namely, $<_f$ and $\leqslant_f$. The definition of $<_f$ $(\Psi_1, \Psi_2, S)$ is described in Figure 9 as a function. Function $\leqslant_f$ is similar to $<_f$, but it uses the condition $\Psi_1(n) \leqslant \Psi_2(n)$, which implies that there are endpoints of the intervals in Step 3 that are closed. As already said, $S$ is a subset of natural numbers, $S \subseteq \mathbb{N}$, and usually $S = \mathbb{N}$, which is extracted from the specification, taking into account its precondition. In general, $S$ is given as a union of intervals of natural numbers. However, the cost bound functions $\Psi_1$ and $\Psi_2$ are continuous functions defined over a subset of real numbers, i.e., $Dom(\Psi_i) \subseteq \mathbb{R}$ and $S \subset Dom(\Psi_i)$ for $i = 1, 2$. Thus, for simplicity, in the definition of $<_f$ and $\leqslant_f$, we first infer intervals of real numbers (see Steps 1–4 of Fig. 9), and, from them, we produce the intervals of natural numbers with the appropriate endpoints, as described in Steps 5 and 6. Note that, in Step 2, we ignore the negative roots of $f(x)$ because they cannot be endpoints of any interval of natural numbers. Since $\Psi_1$ and $\Psi_2$ are

```
$<_f$ $(\Psi_1, \Psi_2, S)$
    Takes two single-variable cost bound functions, $\Psi_1$ and $\Psi_2$, and an input-size set $S$, $S \subseteq \mathbb{N}$.
    Returns a set $IS$ of intervals such that $\forall I \in IS : (\forall n \in I : (\Psi_1(n) < \Psi_2(n) \wedge n \in S))$.
    1. Let $f(x) = \Psi_2(x) - \Psi_1(x)$, and assume that $Dom(f) \subseteq \mathbb{R}$;
    2. Let $x_1, \ldots, x_m$ be the non-negative real roots of equation $f(x) = 0$, i.e.:
       $\forall i(1 \leq i \leq m) : (x_i \in \mathbb{R} \wedge x_i \geq 0 \wedge f(x_i) = 0)$;
    3. Let $IS_1 = \{[0, x_1), (x_1, x_2), \ldots, (x_{m-1}, x_m), (x_m, \infty)\}$;
    4. Let $IS_2 = \{I \mid I \in IS_1 \wedge f(v) > 0,$ for an arbitrary value $v \in I\}$;
    5. Let $IS_3 = \{[\lceil a \rceil, \lfloor b \rfloor] \mid (a, b) \in IS_2\}$;
    6. Let $IS = \{I \cap S \mid I \in IS_3\}$;
    7. return $IS$.
```

Fig. 9. A function for comparing two single-variable cost functions.

continuous, in Step 4, we have that $\forall(a,b) \in IS_2 : (\forall x \in (a,b) : (\Psi_1(x) < \Psi_2(x)))$. Then, in Step 5, we generate intervals of natural numbers, and it holds that for any interval of real numbers $(a,b) \in IS_2$, we have that $(\lceil a \rceil, \lfloor b \rfloor)$ is the largest interval of natural numbers included in $(a,b)$, and hence it holds that $\forall n \in [\lceil a \rceil, \lfloor b \rfloor] : (\Psi_1(n) < \Psi_2(n))$.

As already explained, given the input-size set $S$ of a call-resource pair in an intended semantics, which can also express data size intervals in the precondition, our goal is to partition $S$ in a number of $n_S$ subsets $S_j$ such that for any $S_j$, $1 \leqslant j \leqslant n_S$, either sufficient condition 1 or 2 of Corollary 1 holds. This can be done by using the comparison operators $<_f$ and $\leqslant_f$ described above, with the appropriate values for $\Psi_1$ and $\Psi_2$, and performing intersections or unions of the resulting intervals, depending on whether the condition is a conjunction or disjunction, respectively.

Consider again Step 2 of Figure 9. If $f(x)$ is a polynomial function, then there exist efficient algorithms for obtaining its roots. For the other functions (e.g., exponential, logarithmic, or summation), we have to approximate them using polynomials. We discuss this in the following sections, including a detailed description of the concept of "safety" of such approximations in Section 4.5.

### *4.2 Finding roots of polynomial functions*

According to the fundamental theorem of algebra, a polynomial equation of order $m$ has $m$ roots, whether real or complex numbers. General methods exist that allow computing all these roots, although in our approach we discard complex roots and negative real roots since they are not needed. All the roots of a polynomial equation can be obtained analytically until polynomial order four. Numerical methods must be used for polynomial orders greater than four. In our implementation, we have used the GNU Scientific Library (Galassi *et al.*, 2009) for this purpose. This library offers specific polynomial function root finding methods that are analytical or numerical depending on the polynomial order, as mentioned above.

```
hanoi(N,A,_B,C) :- N=1, print(A,C).
hanoi(N,A,B,C) :-
   N > 1,
   N1 is N - 1,
   hanoi(N1,A,C,B),
   print(A,C),
   hanoi(N1,B,A,C).
```

Fig. 10. A "Towers of Hanoi" program.

### *4.3 Finding roots of non-polynomial functions*

Two non-polynomial cost function classes that the CiaoPP analyses can infer are exponential and logarithmic. We approximate exponential functions with Taylor polynomials and for approximating logarithmic functions and we replace them with other functions that bound them from above or below. After finding the roots of the approximant polynomials by using the method described above, we apply a post-process for checking whether the original functions have additional roots, which is described in Section 4.4.

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

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

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

Our experiments show that in practice these series can typically be limited to order 8, since higher orders do not bring significant differences. Also, in the implementation, the computation of the factorials is done separately and the results are memoized in order to reuse them.

*Example 9*
Consider the program in Figure 10 that prints the shortest sequence of moves to solve the "Towers of Hanoi" problem with $N$ disks. The first argument of `hanoi/4` represents the number of disks to move, while the remaining ones represent the peg where the disks are, the auxiliary peg and the target peg, in that order.

Consider the following assertion:

```
:- check hanoi(N,_,_,_)
         : intervals(nat(N),[i(1,inf)])
         + costb(steps,2**(nat(N)-3) + 2, 2**(nat(N)-3) + 30).
```

which expresses that for any call to `hanoi(N,T1,T2,T3)`, a lower (resp. upper) bound on the number of resolution `steps` performed by the computation is $2^{n-3}+2$ (resp. $2^{n-3}+30$), where $n = \mathtt{nat(N)}$.

The analysis infers $2^{n+1} - 2$ as both upper and lower bound cost function for $n \geqslant 1$. The output of the assertion checking considering this result is (see Fig. 11):

```
:- false pred hanoi(N,_,_,_)
         : intervals(nat(N),[i(1,1),i(5,inf)])
         +  costb(steps,2**(nat(N)-3) + 2, 2**(nat(N)-3) + 30).
```

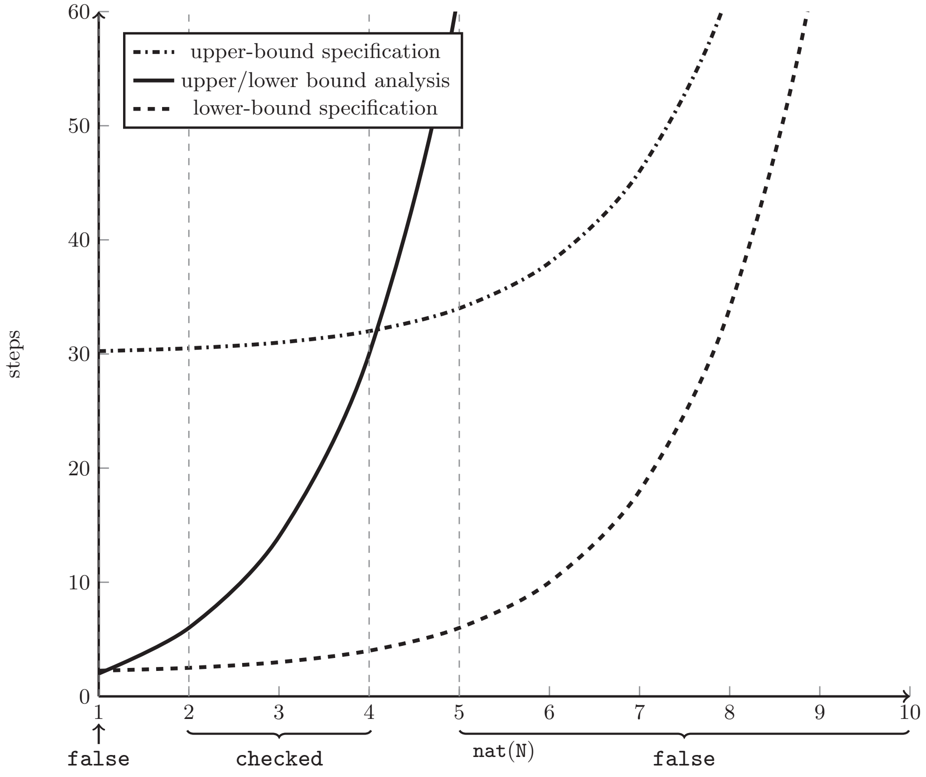


Fig. 11. Resource usage functions for `hanoi`: Specification and analysis results.

```
:- checked pred hanoi(N,_,_,_)
         : intervals(nat(N),[i(2,4)])
         +  costb(steps,2**(nat(N)-3) + 2, 2**(nat(N)-3) + 30).
```

which express that for $n \in [2,4]$, the specification given by the assertion is met, while for $n \in [1,1] \cup [5,\infty]$ it is never met. The real interval verifying $2^{n-3}+2 \leqslant 2^{n+1}-2 \leqslant 2^{n-3}+30$ is approximately $[1.09311, 4.09311]$, and the largest interval of natural numbers included in it, and in the interval expressed in the precondition of the specification, is $[\lceil 1.09311 \rceil, \lfloor 4.09311 \rfloor] = [2,4]$. Therefore, the result obtained from the comparison is exact, in the context of the specification and the $\mathbb{N}$ domain.

*Logarithmic function approximatio.* Assume that we have to perform the comparison $f(x) \leqslant g(x)$, where any of the two functions $f$ or $g$ is logarithmic. In this case, by Lemma 3, we can replace such functions by upper or lower bounds on them, depending on the case, to obtain sufficient conditions. For example, given the logarithmic function $log(h(x))$, our approach will use $h(x)$ as an upper bound on it.

Thus, $log(h(x)) \leqslant g(x)$ would be replaced by the sufficient condition $h(x) \leqslant g(x)$.

```
simple_log(N, N) :-
   N=<1,!.
simple_log(N, S) :-
   N>1,
   N1 is N/2,
   simple_log(N1,S1),
   S is S1 + N.
```

Fig. 12. A simple example with logarithmic cost.

*Example 10*
Consider the program in Figure 12 that calculates the sum $N+N/2+N/2^2+\ldots+1$, given $N$ as input. Consider the following assertion:

```
:- check pred simple_log(N,_) + costb(steps, 0, 3000).
```

in order to find intervals of possible sizes of $N$ for which the number of resolution steps of any call to `simple_log(N,_)` will be less or equal than $3,000$. Let $n = \texttt{nat(N)}$, the analysis infers that the cost of a call to this predicate will be upper/lower bounded by $log_2(\frac{n}{8}) + 4$. With this information, the assertion checking process returns the following two assertions:

```
:- check pred simple_log(N,_)
        : intervals(nat(N),[i(23969,inf)])
        + costb(steps,0,3000).

:- checked pred simple_log(N,_)
        : intervals(nat(N),[i(0,23968)])
        + costb(steps,0,3000).
```

which expresses that for $n \leqslant 23,968$ the specification given by the assertion is met, while for $n > 23,969$ the assertion cannot be proved nor disproved. This result is correct but obviously it is an approximation.

### 4.4 Checking additional roots for non-polynomial functions

In this section, we describe a post-process that ensures the correctness of the function comparison approach that we have presented so far, for the cases in which there are functions that have been approximated by polynomials, e.g., exponential functions, for which generally the number of roots is unknown.

Consider the comparison operator $<_f$ described in Section 4.1, in particular, Step 1 of Figure 9, where we define $f(x) = \Psi_2(x) - \Psi_1(x)$. Assume that we approximate $f(x)$ by a polynomial $P(x)$ and find the non-negative real roots of $P(x)$, say $x'_1, \ldots, x'_k$. Then, $x'_1, \ldots, x'_k$ might not include all the non-negative real roots of $f(x)$, denoted $x_1, \ldots, x_m$ in Step 2.

To ensure that there is no other root of $f(x)$ inside any of the computed intervals for $P(x)$, i.e., $[0, x'_1), (x'_1, x'_2), \ldots, (x'_{k-1}, x'_k), (x'_k, \infty)$, we proceed as follows. We first consider all the intervals but the last one $(x'_k, \infty)$, i.e., let $IS' = \{[0, x'_1), (x'_1, x'_2), \ldots, (x'_{k-1}, x'_k)\}$, and $IS'' = \{[\lceil a \rceil, \lfloor b \rfloor] \mid (a, b) \in IS'\}$. First, we check that:

$$\forall I \in IS'' : (\forall n \in I : f(n) > 0)$$

by enumerating the finite number of values, i.e., natural numbers, in each interval $I$. It is always possible of course to give up and return *unknown* if this number is above a certain threshold, or use the procedure below.

However, in the last interval $(x'_k, \infty)$, we obviously have to use a different procedure to ensure whether a function is indeed *always* bigger than the other. Our procedure uses a set of syntactic rules to compare the two functions $\Psi_1(x)$ and $\Psi_2(x)$ together with a constraint $x > x'_k$, which expresses that the comparison only holds from the largest root to infinity. More specifically, we have implemented a modification of the comparison algorithm in (Albert *et al.*, 2010, 2015). Note that we only use such comparison algorithm for this very particular case, since it can be given constraints of the form $x > c$, where $c$ is a constant, which represents the interval $(c, \infty)$ in our approach. If such comparison returns *true*, then it is ensured that one of the functions to compare is greater than the other, in the context of the given constraints; otherwise, nothing can be ensured. Thus, such a comparison is complementary to ours for this particular case, i.e., checking the last interval already computed by our approach, when non-polynomial functions are approximated by polynomials. However, we do not use it for anything else, since, among other things, it cannot infer preconditions involving intervals for which one function is greater or smaller than the other, as our approach does.

In addition, we also use the derivatives of the functions, which tend to be simpler and easier to verify. In particular, we exploit the fact that if $\Psi_1(x) < \Psi_2(x)$ on $x = a$, then such functions will never intersect for all $x > a$ as long as their derivatives satisfy $\Psi'_1(x) < \Psi'_2(x)$ for all $x > a$.

Although our algorithm is not complete, it is correct in the sense that when checking $\Psi_1(x) < \Psi_2(x)$, if the algorithm returns *true*, then for some $x'_k$, such inequality holds for all $x \in (x'_k, \infty)$. If this cannot be ensured by our algorithm, then the algorithm returns *unknown*.

## *4.5 Safety of the approximation*

The roots obtained for function comparison are in some cases approximations of the actual roots. The errors in approximations come from two sources: (a) the numerical method for root calculation of polynomials, and (b) the difference between the original non-polynomial function and its polynomial approximant. In any case, we must guarantee that their values are safe, in the sense that they can be used for verification purposes, in particular, for proving sufficient conditions 1 and 2 in Corollary 1. In turn, such conditions depend on the comparison operators $<_f$ and $\leqslant_f$ already described. To this end, the concept of *safety* of the roots is meaningful in the context of a given comparison operator. Consider for example operator $<_f$, and Steps 1 and 2 of its definition in Figure 9, assuming that $x_1, \ldots, x_m$ are exact roots of function $f(x)$.

*Definition 8*
Let $f(x)$ be a continuous function such that $Dom(f) \subseteq \mathbb{R}$, and let $X = \{x_1, \ldots, x_m\}$ be the set of its exact non-negative real roots. Let $IS = \{[0, x_1), (x_1, x_2), \ldots, (x_{m-1}, x_m), (x_m, \infty)\}$ such that for any $I \in IS$ either $\forall x \in I$ :

$f(x) > 0$ or $\forall x \in I : f(x) < 0$. For any root $x_i \in X$, $1 \leqslant i \leqslant m$, there are two intervals that have $x_i$ as an endpoint: $(a, x_i)$, where $a = x_{i-1}$ or $a = 0$, and $(x_i, b)$, where $b = x_{i+1}$ or $b = \infty$. We then define the concept of safe approximation as follows. For any root $x_i \in X$, we say that $x'_i$ is a safe approximation of $x_i$ for interval $(a, x_i)$, where $a = x_{i-1}$ or $a = 0$, if $a < x'_i \leqslant x_i$. Similarly, $x'_i$, where $1 \leqslant i \leqslant m-1$, is a safe approximation of $x_i$ for interval $(x_i, x_{i+1})$ if $x_i \leqslant x'_i < x_{i+1}$, and $x'_m$ is a safe approximation of $x_m$ for interval $(x_m, \infty)$ if $x_m \leqslant x'_m$.

In the context of this definition, given any interval $I$ such that $\forall x \in I : f(x) > 0$, it is clear that if we replace any endpoint (or both) of $I$ by safe approximations for $I$, we obtain a smaller interval $I'$, and it holds that $\forall x \in I' : f(x) > 0$.

For example, in Step 4 of Figure 9, it holds that $\forall I \in IS_2 : (\forall x \in I : f(x) > 0)$, which implies that $\forall I \in IS_2 : (\forall x \in I : \Psi_1(x) < \Psi_2(x))$. Thus, if we replace the endpoints of the intervals in $IS_2$ by safe approximated roots for them, we can ensure that, if $IS$ is the result of $<_f (\Psi_1, \Psi_2, S)$, then $\forall I \in IS : (\forall n \in I : (\Psi_1(n) < \Psi_2(n)))$. A similar reasoning can be done for operator $\leqslant_f$.

When we say that we safely check a given condition, we mean that we possibly use safe approximated roots for building intervals for which our algorithm says that the condition holds, and thus such intervals may be smaller than the ones for which the condition actually holds. In addition, our verification approach works with the approximations of the concrete semantics and safely checks sufficient conditions to prove or disprove program partial correctness and incorrectness. This implies that our approach may infer stronger sufficient conditions.

Assume for example that we want to check whether $\forall x \in S : \Phi^u(x) \leqslant \Phi^u_I(x)$, where $\Phi^u$ and $\Phi^u_I$ are resource usage bound functions, the former is part of the result of program analysis and the latter appears in an assertion declared in the program. This check is part of the sufficient condition 1 in Corollary 1. In this case, we can use the operator $\leqslant_f (\Phi^u, \Phi^u_I, S)$, which defines $f(x) = \Phi^u_I(x) - \Phi^u(x)$. Assume that $\forall x \in S : f(x) \geqslant 0$. Then, it holds that $\forall x \in S : \Phi^u(x) \leqslant \Phi^u_I(x)$. Since $\leqslant_f$ may use safe approximated roots, it may return a set $S'$ smaller than $S$, i.e., $S' \subset S$. Assume also that $\Phi^l_I$ 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 0, which means that $\forall x \in S : \Phi^l_I(x) \leqslant \Phi^l(x)$ is true. Thus, if $\forall x \in S : f(x) \geqslant 0$, we can state that sufficient condition 1 of Corollary 1 holds. Similarly, assume that we use $<_f (\Phi^u_I, \Phi^l, S)$, which defines $f(x) = \Phi^l(x) - \Phi^u_I(x)$. Then, we can say that $\forall x \in S : \Phi^u_I(x) < \Phi^l(x)$ if $\forall x \in S : f(x) > 0$, proving that sufficient condition 2 of Corollary 1 holds. We can reason similarly in the comparisons involving a lower bound in the assertion, i.e., $\Phi^l_I$. Thus, we focus exclusively on checking that $\forall x \in S : f(x) > 0$ or $\forall x \in S : f(x) \geqslant 0$, where $f(x)$ is conveniently defined in each case.

We now focus on a method we propose for obtaining safe approximated roots. Assume that the exact roots of function $f(x)$ are $x_1, ..., x_m$, and that $x'_1, ..., x'_m$ are approximated roots obtained by using the techniques already explained, so that for each approximated root $x'_i$, $1 \leqslant i \leqslant m$, there is a value $\varepsilon$ such that $x_i \in [x'_i - \varepsilon, x'_i + \varepsilon]$. Consider an interval $I$ for which we need to ensure that $\forall x \in I : f(x) > 0$. Assume

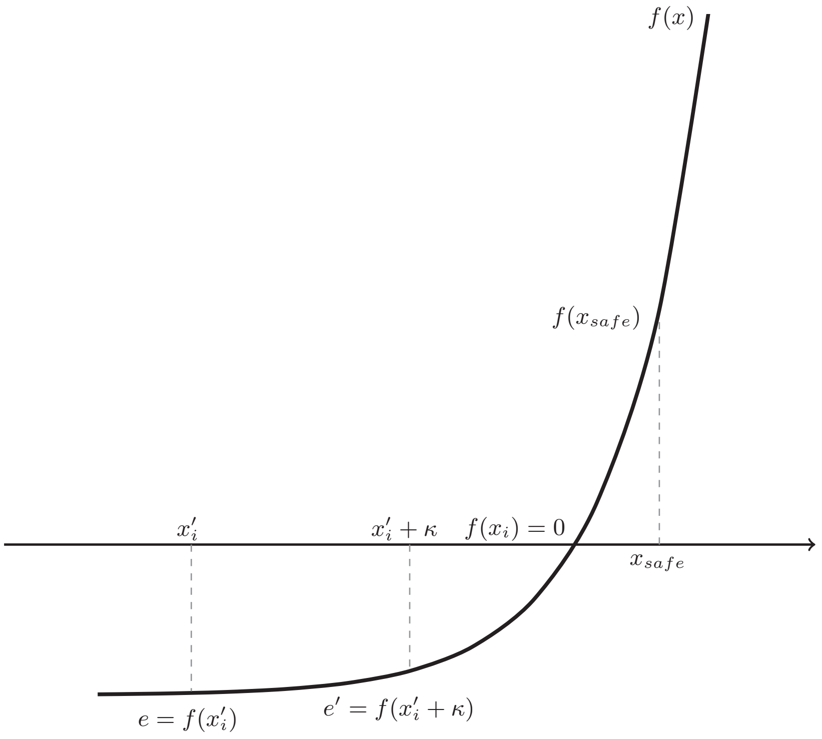


Fig. 13. Case 1. $x_i > x_i'$ (since $e' > e$). $x_{safe}$ is a safe approximated root of $x_i$.

that $I = (x_i', b)$ for some $1 \leqslant i \leqslant n$ and some endpoint $b$. In this case, the condition for $x_i'$ to be a safe root of $x_i$ for $I$ is $x_i \leqslant x_i'$. Then, we first determine the actual relative position of $x_i'$ and $x_i$, and, if it is not compatible with condition $x_i \leqslant x_i'$, i.e., if $x_i'$ is "to the left" of $x_i$, then we start an iterative process that increments $x_i'$ by some $0 < \delta < 1$ so that after $m$ iterations we have that $x_i'' = x_i' + m\ \delta$, and $x_i''$ is a safe root of $x_i$ for $I$. We can reason similarly for the case in which $I = (b, x_i')$. In this case, if $x_i'$ is "to the right" of $x_i$, then we start an iterative process that increments $x_i'$ by some $-1 < \delta < 0$, so that $x_i''$ is a safe root of $x_i$. This is explained in more detail in the rest of this section.

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

1. if $e < 0$ and $e' > e$ then $x_i > x_i'$
2. if $e > 0$ and $e' > e$ then $x_i < x_i'$
3. if $e > 0$ and $e' < e$ then $x_i > x_i'$
4. if $e < 0$ and $e' < e$ then $x_i < x_i'$

From Figure 13, we can see the rationale behind the first case. 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 $x_i'$. The other cases follow an analogous reasoning.

*Iterative process for computing the safe root.* Once we have determined the relative position of the exact root $x_i$ and its approximated value $x'_i$, we set up an appropriate value for $\delta$. If we have to ensure that $x_i \leqslant x'_i$ but it actually holds that $x_i > x'_i$, then we take $0 < \delta < 1$ so that we iterate on the addition $x''_i = x'_i + \delta$ until $f(x''_i) > 0$. In this case, the iteration goes to the right. Such an iteration is apparent in the following pseudo-code:

```
1: $x_{safe} \leftarrow x'_i$
2: **while** $f(x_{safe}) < 0$ **do** $x_{safe} \leftarrow x_{safe} + \delta$
3: **end while**
4: **return** $x_{safe}$
```

Conversely, if we have to ensure that $x'_i \leqslant x_i$ but it actually holds that $x'_i > x_i$, then we take $-1 < \delta < 0$ so that the iteration goes to the left.

Our approach ensures that there are no other roots of $f(x)$ between $x'_i$ and $x_{safe}$. As already said, we approximate $f(x)$ by a polynomial $P(x)$, and the techniques we use can find all the roots of polynomials. If $f(x)$ is not a polynomial, then $f(x)$ can have more roots than $P(x)$, but we use the techniques described in Section 4.4 to deal with this possible case and ensure that there are no additional roots inside the inferred intervals. In addition, as already said, based on the sign of the gradient, we infer whether $f(x)$ is increasing or decreasing. But we also check this after computing $x_{safe}$: if the derivative of $f(x)$ is positive (resp. negative) between $x'_i$ and $x_{safe}$, then $f(x)$ is increasing (resp. decreasing) between $x'_i$ and $x_{safe}$, which implies that there are no other roots of $f(x)$ between $x'_i$ and $x_{safe}$.

*Example 11*
Consider the following assertion for the classical `fibonacci` program:

```
:- check pred fib(N,F) : (nat(N), var(F))
                       + cost(ub, steps, exp(2, nat(N))-1000 ).
```

which expresses that for any call to `fib(N,F)` with the first argument bound to a natural number and the second one a free variable, an expected upper bound on the number of resolution `steps` performed by its whole computation is given by the function $\Phi^u_I(x) = 2^x - 1000$, where $x$ is the size of the first argument `N`. Since such argument is a natural number the size metric used for it is its value.

The lower bound inferred by the static analysis is $\Phi^l(x) = 1.45 \times 1.62^x - 1$. The intersection of $\Phi^l(x)$ and $\Phi^u_I(x)$ 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 $x \approx 10.18$.

As already said, and this example illustrates, usually cost functions depend on variables that range over natural numbers. For this reason, in this case, we will take the closest natural number to the left or right of the safely approximated root computed by the iterative algorithm described above, depending on the gradient, to obtain a safe value in the domain of the resource usage function. Thus, in this example, we will take the value 10 for $x$.

It turns out that the analysis also infers the same cost function as both a lower and upper bound (i.e., it infers the exact function). Thus, the upper bound cost function is given by $\Phi^u(x) = 1.45 \times 1.62^x - 1$.

Once the interval endpoints have been computed, we can reason as follows: to the left of the safe root $x = 10$, the cost upper bound declared in the specification given by the check assertion is less than the (safe) lower bound inferred by the analysis, therefore, the assertion is false in the interval $[0, 10]$. Since in this example we are dealing with exponential functions, we also have to verify every point in such interval, as already explained in Section 4.4. Moreover, to the right of the safe root $x = 10$, the cost upper bound declared in the specification is greater than the (safe) upper bound inferred by the analysis, and therefore the assertion is true in the interval $[11, \infty]$. Our algorithm from Section 4.4 also verifies that the functions never intersect in such interval, and thus we can ensure that the specification is met in it. Finally, the output of our assertion checking algorithm for the `fibonacci` program is:

```
:- false pred fib(N,F) : intervals(nat(N), [i(0,10)])
                       + cost(ub, steps, exp(2,nat(N))-1000 ).
:- checked pred fib(N,F) : intervals(nat(N), [i(11,+inf)])
                         + cost(ub, steps exp(2,nat(N))-1000 ).
```

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 $[11, \infty)$. Thus, the system infers a precondition, involving an interval of natural numbers, on which the assertion can be proved false, and another precondition, involving the rest of the range of the natural numbers, on which it can be proved true.

## 4.6 Comparing summation functions

Dealing with summation functions can be important in the analysis of recursive programs, and hence of imperative programs that contain loops. However, the function comparison operation is not straightforward when at least one of the operands contains a summation function, even in the case in which other operands are just simple arithmetic functions.

A summation cost function $C$ is an expression of the form $C(n) = \sum_{i=a}^{n} f(i)$, where $a, n \in \mathbb{N}$, and $f$ is a cost function. Our approach consists in transforming it into an equivalent closed form function $C^t$, i.e., an expression that does not contain any subexpressions built by using the $\sum$ operand. Instead, $C^t$ is built by using only elementary arithmetic functions, e.g., constants, addition, subtraction, multiplication, division, exponential, or even factorial functions. Such transformation is based on *finite calculus* Gleich (2005). The closed form function $C^t$ can be a polynomial, but also other non-polynomial function. Thus, the set of functions that can be represented as summation expressions is a superset of the functions that can be represented as polynomials. Finally, we replace the summation cost function $C$ by its closed form transformation $C^t$, and use the function comparison techniques explained in the previous sections.

Prior to explaining our algorithm for obtaining $C^t$, we provide some necessary background. We start by recalling the relation between infinite calculus and finite calculus, focusing on the concepts of *derivative* and *antiderivative* functions.

*Relating finite and infinite calculus.* In infinite calculus, the *derivative* of a function $f(x)$, denoted $\frac{\mathrm{d}}{\mathrm{d}x}f(x)$ or $f'(x)$, is defined as $\frac{\mathrm{d}}{\mathrm{d}x}f(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h}$. A similar concept is defined in finite calculus for a *discrete* function $f(x)$, the *discrete derivative*, denoted $\Delta f(x)$, by assuming discrete increments $h$ for variable $x$. Since the closest we can get to 0 is 1, in the limit, i.e., $h = 1$, we obtain the following definition.

*Definition 9*
The *discrete derivative* of function $f(x)$ is $\Delta f(x) = f(x+1) - f(x)$.

In infinite calculus, if $\frac{\mathrm{d}}{\mathrm{d}x}F(x) = f(x)$, then we say that $F(x)$ is an *antiderivative* function of $f(x)$. For any constant $c$, $F(x) + c$, is also an *antiderivative* of $f(x)$. Since the number of *antiderivatives* of $f(x)$ is infinite, we denote the class of such antiderivatives $F(x) + c$ as $\int f(x)\ \mathrm{d}x$, which is also called the *indefinite integral* of $f(x)$. Also, the *definite integral* of $f(x)$ over the interval $[a, b]$ is denoted as $\int_a^b f(x)\ \mathrm{d}x$. According to the fundamental theorem of calculus, if $f(x)$ is a real-valued continuous function on $[a, b]$ and $F(x)$ is an antiderivative of $f(x)$ in $[a, b]$, then $\int_a^b f(x)\ \mathrm{d}x = F(x)|_a^b = F(b) - F(a)$. Similarly, in finite calculus, if $\Delta F(x) = f(x)$, then $F(x)$ is a *discrete antiderivative* of $f(x)$, and $\sum f(x)\ \mathrm{d}x$ denotes the *discrete indefinite integral* of $f(x)$, i.e., $F(x) + c$, where $c$ is an arbitrary constant. The following definition allows extending the analogy.

*Definition 10*
The *discrete definite integral* of $f(x)$ over the discrete interval $[a, b]$, denoted as $\sum_a^b f(x)\ \mathrm{d}x$, is defined as:

$$\sum_a^b f(x)\ \mathrm{d}x = F(x)|_a^b = F(b) - F(a)$$

where $F(x)$ is a *discrete indefinite integral* of $f(x)$, i.e., $F(x) = \sum f(x)\ \mathrm{d}x$. Then, we get the following result, which makes it possible to transform a summation into a *definite integral*, and further into a closed form function.

*Theorem 3*
The fundamental theorem of finite calculus is:

$$\sum_{x=a}^{b} f(x) = \sum_{a}^{b+1} f(x)\ \mathrm{d}x$$

*Proof*
Let $F(x)$ be a *discrete indefinite integral* of $f(x)$, i.e., $\Delta F(x) = f(x)$. According to Definition 9, we have that $\Delta F(x) = F(x+1) - F(x) = f(x)$. Then:

$\sum_{x=a}^{b} f(x) = \sum_{x=a}^{b}(F(x+1) - F(x))$
$= F(a+1) - F(a) + F(a+2) - F(a+1) + \cdots + F(b) - F(b-1) + F(b+1) - F(b)$
$= F(b+1) - F(a)$
$= \sum_a^{b+1} f(x)\ \mathrm{d}x$ (according to Definition 10) □

The *falling power* in finite calculus is defined as:

$$\begin{array}{lcll} x^{\underline{0}} & = & 1 & \\ x^{\underline{m}} & = & (x-(m-1))\ x^{\underline{m-1}} & \text{if } m>0 \end{array}$$

Equivalently, if $m > 0$, then $x^{\underline{m}} = x\ (x-1)\ (x-2)\cdots(x-(m-1))$. For example: $x^{\underline{1}} = x$, $x^{\underline{2}} = x\ (x-1)$, $x^{\underline{3}} = x\ (x-1)\ (x-2)$, and so on.

The use of the *falling power* allows to define derivative and integration rules in finite calculus that are analogous to the corresponding ones in infinite calculus. For example, in infinite calculus, given the function $f(x) = x^m$, its derivative is given by $\frac{\mathrm{d}}{\mathrm{d}x}\ f(x) = m\ x^{m-1}$, and its *indefinite integral* is $\int f(x)\ \mathrm{d}x = \frac{1}{m+1}\ x^{m+1} + c$, where $c$ is an arbitrary constant. The rules for the *falling power* in finite calculus are analogous: given a discrete function $f(x) = x^{\underline{m}}$, its derivative is given by $\Delta f(x) = m\ x^{\underline{m-1}}$, and its *discrete indefinite integral* is $\sum f(x)\ \mathrm{d}x = \frac{1}{m+1}\ x^{\underline{m+1}} + c$.

Table 3 provides a set of rules for computing integrals and derivatives in finite calculus, including the ones already seen for the falling power.

We can perform a translation from regular powers into falling powers, which is needed prior to applying some rules in Table 3, by using the following theorem:

$$x^m = \sum_{k=0}^{m} \left\{ {m \atop k} \right\} x^{\underline{k}} \tag{1}$$

where $\left\{ {m \atop k} \right\}$ is a Stirling number of the second kind, which represents the number of ways of partitioning $n$ distinct objects into $k$ non-empty sets Gleich (2005). For example:

$$\begin{array}{l} x^0 = x^{\underline{0}} \text{ since by definition } x^0 = 1 \text{ and } x^{\underline{0}} = 1\text{, but also:} \\ x^0 = \left\{ {0 \atop 0} \right\} x^{\underline{0}} = 1\ x^{\underline{0}} \\ x^1 = \left\{ {1 \atop 0} \right\} x^{\underline{0}} + \left\{ {1 \atop 1} \right\} x^{\underline{1}} = 0\ x^{\underline{0}} + 1\ x^{\underline{1}} = x^{\underline{1}} \\ x^2 = \left\{ {2 \atop 0} \right\} x^{\underline{0}} + \left\{ {2 \atop 1} \right\} x^{\underline{1}} + \left\{ {2 \atop 2} \right\} x^{\underline{2}} = x^{\underline{2}} + x^{\underline{1}} \\ x^3 = \left\{ {3 \atop 0} \right\} x^{\underline{0}} + \left\{ {3 \atop 1} \right\} x^{\underline{1}} + \left\{ {3 \atop 2} \right\} x^{\underline{2}} + \left\{ {3 \atop 3} \right\} x^{\underline{3}} = x^{\underline{3}} + 3x^{\underline{2}} + x^{\underline{1}} \end{array}$$

Thus, the $\Delta f(x)$ and $\sum f(x)\ \mathrm{d}x$ functions in finite calculus are analogous to the *derivative* ($\frac{\mathrm{d}}{\mathrm{d}x} f(x)$) and *antiderivative* ($\int f(x)\ \mathrm{d}x$) functions in infinite calculus, respectively. Note also, that the integer number 2 in finite calculus is analogous to Euler's number $e$ in infinite calculus, in the sense that $\Delta 2^x = 2^x$ and $\frac{\mathrm{d}}{\mathrm{d}x} e^x = e^x$, as well as $\sum 2^x\ \mathrm{d}x = 2^x + c$ and $\int e^x\ \mathrm{d}x = e^x + c$.

*Our algorithm for rewriting summations.* Based on Theorem 3 and Definition 10, given a summation of the form $\sum_{x=a}^{b} f(x)$, where $a, b \in \mathbb{N}$, we rewrite it as a *definite integral* in finite calculus:[6]

$$\sum_{x=a}^{b} f(x) = \sum_{a}^{b+1} f(x)\ \mathrm{d}x = F(b+1) - F(a) \tag{2}$$

[6] For simplicity of exposition we assume that $a, b \in \mathbb{N}$, but our algorithm can be also applied even when $a$ and $b$ are arithmetic expressions, i.e., functions $a, b : \mathbb{N} \to \mathbb{N}$.

Table 3. *A set of finite calculus rules used in our approach*

| #Rule | $f(x)$ | $\Delta f(x)$ | $\Sigma f(x)\mathrm{d}x$ |
|---|---|---|---|
| 1 | $x^{\underline{m}}$ | $m\ x^{\underline{m-1}}$ | $\frac{1}{m+1}\ x^{\underline{m+1}} + c$ |
| 2 | $2^x$ | $2^x$ | $2^x + c$ |
| 3 | $a^x$ | $(a-1)\ a^x$ | $\frac{1}{a-1}\ a^x + c$ |
| 4 | $a^{mx+n}$ | $(a^m - 1)\ a^{mx+n}$ | $\frac{1}{a^m-1}\ a^{mx+n} + c$ |
| 5 | $u(x) + v(x)$ | $\Delta u(x) + \Delta v(x)$ | $\Sigma u(x)\ \mathrm{d}x + \Sigma v(x)\ \mathrm{d}x + c$ |
| 6 | $k\ u(x)$ | $k\ \Delta u(x)$ | $k\ \Sigma u(x)\ \mathrm{d}x + c$ |
| 7 | $u(x)\ v(x)$ | $v(x+1)\ \Delta u(x) + u(x)\ \Delta v(x)$ | |
| 8 | $u(x)\ \Delta v(x)$ | | $u(x)\ v(x) - \Sigma v(x+1)\ \Delta u(x)\ \mathrm{d}x + c$ |

where $F(x)$ is the *indefinite integral* function of $f(x)$, i.e., $F(x) = \sum f(x)\ \mathrm{d}x$, and is obtained by using the integration rules provided in the fourth column of Table 3 for different classes of functions $f(x)$, specified in the second column of the table. The third column of the table shows some rules for obtaining the derivatives of the functions in the second column, which are needed for the application of the integration rule 8 provided in the fourth column, row 8.

The rules in Table 3 are applied to the resulting expression until it does not contain any integral nor summation. Note that $u - v$ and $\frac{u}{v}$ can rewritten as $u + (-v)$ and $u\ \frac{1}{v}$, respectively. However, we use the corresponding specialized rules for the subtraction and division.

For illustration purposes, we include here a simple and a more complex example of the application of such rules.

*Example 12*
In order to find a closed form of $\sum_{x=1}^{a} 2^x$, we proceed as follows:

1. Rewrite it as $\sum_1^{a+1} 2^x\ \mathrm{d}x$, according to Theorem 3.
2. Compute the corresponding discrete indefinite integral $\sum 2^x\ \mathrm{d}x$. This is done by using integration rule 2, so that $\sum 2^x\ \mathrm{d}x = 2^x$. Note that we omit the constant $c$ that appears in the rules of Table 3 since it is not relevant for the final result.
3. By using Definition 10 and the previous results, we have that:
   $\sum_{x=1}^{a} 2^x = \sum_1^{a+1} 2^x\ \mathrm{d}x$ (Theorem 3)
   $= 2^x|_1^{a+1}$ (Definition 10 and integration rule 2)
   $= 2^{a+1} - 2^1 = 2^{a+1} - 2$

*Example 13*
A closed form of $\sum_{x=1}^{a} x\ 2^{a-x}$ is obtained as follows:

1. Rewrite it as $\sum_1^{a+1} x\ 2^{a-x}\ \mathrm{d}x$ (Theorem 3).
2. Compute the corresponding discrete indefinite integral $\sum x\ 2^{a-x}\ \mathrm{d}x$ by using integration by parts rule 8, making $u(x) = x$ and $\Delta v(x) = 2^{a-x}\ \mathrm{d}x$. Thus, $\Delta u(x) = 1\ x^{\underline{0}}\ \mathrm{d}x = \mathrm{d}x$ (derivative rule 1), and $v(x) = \sum\ 2^{a-x}\ \mathrm{d}x = \frac{1}{2^{-1}-1}\ 2^{a-x} = -2\ 2^{a-x}$

(integration rule 4). Now, we have:

$$\begin{aligned}&\sum x\ 2^{a-x}\ \mathrm{d}x = x\ (-2\ 2^{a-x}) - \sum -2\ 2^{a-(x+1)}\ \mathrm{d}x\\ &= -x\ 2^{a+1-x} - \sum -2^{a-x}\mathrm{d}x = -x\ 2^{a+1-x} + \sum 2^{a-x}\mathrm{d}x\\ &= -x\ 2^{a+1-x} + (-2\ 2^{a-x})\ \text{(integration rule 4, as before)}\\ &= -x\ 2^{a+1-x} - 2^{a+1-x} = -2^{a+1-x}\ (x+1)\end{aligned}$$

3. By Definition 10 and the previous result, we have that:

$$\begin{aligned}&\sum_1^{a+1} x\ 2^{a-x}\ \mathrm{d}x = -2^{a+1-x}\ (x+1)|_1^{a+1}\\ &= -2^{a+1-(a+1)}\ ((a+1)+1) + 2^{a+1-1}\ (1+1) = -2^0\ (a+2) + 2^a\ 2\\ &= 2^{a+1} - a - 2\end{aligned}$$

4. Thus, $\sum_{x=1}^{a} x\ 2^{a-x} = \sum_1^{a+1} x\ 2^{a-x}\ \mathrm{d}x = 2^{a+1} - a - 2$.

*Termination of the algorithm.* The proof of termination of the recursive application of the rules of Table 3 is based on: (a) in any of the derivative rules (third column), the depth of the resulting expression, with respect to the derivative operator $\Delta$, is always 0 (rules 1–4) or decreases by 1 (rules 5–7); and (b) in any of the integration rules (fourth column), the depth of the resulting expression, with respect to the integral operator $\Sigma$ d$x$, is always 0 (rules 1–4) or decreases by 1 (rules 5, 6, and 8). In addition, in integration rule 8, we apply the derivative rules to the polynomial part, so that eventually, the depth of the resulting expression will shrink down to a constant.

Finally, as already said at the beginning of this section, our approach for comparing summation functions consists in transforming any summation cost function $C$ into an equivalent closed form cost function $C^t$ that does not contain any summation subexpressions, and then applying the comparison techniques explained in the previous sections to the resulting closed form functions. In general, such transformation is an *undecidable* problem. However, Table 3 provides a decidable fragment of summation expressions, which cover a large class of the functions that are produced by the analysis that we use. In addition, we detect functions that are not covered by our approach and report them to the user.

### *4.7 Multiple variable cost function comparison*

Given two resource usage functions $\Psi_1(\bar{n})$ and $\Psi_2(\bar{n})$, where $\bar{n}$ is the abbreviation of $k$ variables $n_1 \ldots n_k$ representing input data sizes, we want to know which values of $\bar{n}$ meet the constraint $\Psi_1(\bar{n}) \leqslant \Psi_2(\bar{n})$, so that we can view this problem as a constraint satisfaction problem.

If the functions involved are *linear functions* the problem can be solved by using standard constraint programming techniques. In our implementation, we use the Parma Polyhedra Library to compute the solutions in this case. However, constraint programming cannot solve the problem for *polynomial functions* in general.

Unlike the case of single-variable cost functions, where we have numerically bounded intervals as (input data size) preconditions, in case of multiple-variable cost functions, we need to be able to express relations between variables as preconditions. For example, given a function $x + y - 10 \leqslant 0$, all combinations of values for $x$ and

Table 4. *Sufficient conditions checked by our general verification process for different scenarios depending on the available bounds*

| | | Specification | | |
|---|---|---|---|---|
| | | Upper bound ($S_{ub}$) | Lower bound ($S_{lb}$) | Upper and lower bound |
| Analysis | Upper bound ($A_{ub}$) | $c_1 \to T$, where $c_1 \equiv S_{ub} \geqslant A_{ub}$ | $c_3 \to F$, where $c_3 \equiv S_{lb} > A_{ub}$ | $c_3 \to F$ $\neg c_3 \to C$ |
| | Lower bound ($A_{lb}$) | $c_2 \to F$, where $c_2 \equiv S_{ub} < A_{lb}$ | $c_4 \to T$ $c_4 \equiv S_{lb} \leqslant A_{lb}$ | $c_2 \to F$ $\neg c_2 \to C$ |
| | Upper and lower bound | $c_1 \to T$ $c_2 \to F$ $\neg c_1 \wedge \neg c_2 \to C$ | $c_4 \to T$ $c_3 \to F$ $\neg c_3 \wedge \neg c_4 \to C$ | $c_1 \wedge c_4 \to T$ $c_2 \vee c_3 \to F$ $\neg(c_1 \wedge c_4) \wedge \neg(c_2 \vee c_3) \to C$ |

$y$ that satisfy the inequality cannot be concisely represented as intervals in the preconditions. Therefore, instead of using only intervals represented as pairs of numbers, we use arithmetic expressions that represent more general *size constraints*. Table 4 summarizes the sufficient conditions used by our general verification process, which can be applied to both multi- and single-variable cost functions, showing the size constraints that need to be checked for different cases, depending on whether the specification provides an *Upper bound* cost function (denoted as $S_{ub}$), a *Lower bound* cost function ($S_{lb}$), or both (columns 2–, respectively). A symbol representing the result of the verification process ($T$, $F$ or $C$), when such size constraints are true, is shown at the right-hand side of the implication symbol ($\to$), meaning that the specification has been verified ($T$), is false ($F$), or it cannot be proved whether the specification is true or false. Short names for the size constraints ($c_1$ to $c_4$) are also used in order to achieve a compact representation. The first column (Analysis) divides the table into three different scenarios, each one corresponding to a row, depending on whether the available analysis is able to infer upper-bound cost functions, lower bounds, or both. As already explained, in this work, we use the parametric resource analysis integrated in CiaoPP (see Navas *et al.*, 2007; Serrano *et al.*, 2014 and references therein), which infers both upper and lower bounds. Note that the conditions $c_1 \wedge c_4$ and $c_2 \vee c_3$ given in the last column and row of Table 4, correspond to sufficient conditions 1 and 2 of Corollary 1, respectively. Such conditions assume that both lower- and upper-bound cost functions are available for both analysis and specification. Either condition $c_1$ or $c_4$ in isolation is also equivalent to sufficient condition 1 of Corollary 1 if default, safe values for the corresponding missing bounds are assumed. The same applies to conditions $c_2$ and $c_3$, which are equivalent to sufficient condition 2 of Corollary 1.

*Example 14*

Consider the `inc_append/3` predicate in Figure 14, which is an extension of the classical `append/3`, also concatenating two lists of numbers, $A$ and $B$, but which also increments by 1 all the elements of the second list ($B$) beforehand. The user

```
:- check pred inc_append(A,B,C) + (cost(ub, steps, 2*length(A)-10)).

inc_append(A, B, C) :-
    inc_list(B, B1),
    append(A, B1, C).

inc_list([], []).
inc_list([E|R], [E1|T]) :-
    E1 is E + 1,
    inc_list(R, T).

append([],L,L).
append([A|R],S,[A|L]) :-
    append(R,S,L).
```

Fig. 14. Append with increment example.

assertion specifies that the upper bound on the cost of the program, in terms of the number of resolution steps, is $2 * length(A) - 10$, where $A$ is the first list to append. The analysis infers both an upper and a lower bound cost function, which in this case both bounds coincide, namely, $length(B) + length(A) + 3$. The output of the assertion checking is

```
:- false pred inc_append(A,B,C)
          : intervals([[lt(-13,-length(A)+length(B))]])
          + cost(ub,steps,2*length(A)-10).

:- checked pred inc_append(A,B,C)
            : intervals([[leq(13,length(A)-length(B))]])
            + cost(ub,steps,2*length(A)-10).
```

meaning that when $-13 < -length(A) + length(B)$ the assertion is false, and when $13 \leqslant length(A) - length(B)$ the assertion is correct.

## 5 Generic implementation and experimental results

In order to assess the accuracy and efficiency (as well as the scalability) of the resource usage verification techniques presented, we have implemented and integrated them in by extending the function comparison capabilities of the Ciao/CiaoPP framework.

Table 5 shows some experimental results obtained with our prototype implementation on an Intel Core i5 2.5 GHz with 2 cores, 10GB 1333 MHz DDR3 of RAM, running MacOS Sierra 10.12.6. 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. We will just mention that in the case of the core resource analysis, i.e., the one that processes the HC IR (to which other languages are translated into), and infers cost functions, its scalability follows generally from its compositional nature. Our study focuses on the scalability of the assertion comparison process. To this end, we have added a total number of 390 assertions to several programs that are then

Table 5. *Results of the interval-based static assertion checking integrated into CiaoPP*

| Program+ Analysis info + AvT | ID | Assertion | Verif. result | Time (ms) Tot | Time (ms) Avg |
|---|---|---|---|---|---|
| <u>*Fibonacci*</u><br>**lb,ub:** $1.45 * 1.62^x$<br>$+0.55 * -0.62^x - 1$ | *A1* | :- pred fib(N,R)<br>+cost(ub,steps,<br>exp(2,nat(N))-1000). | F in $[0,10]$<br>T in $[11,\infty]$ | 106.4 | 35.4 |
| x = nat(N)<br>**AvT=** $\frac{1402.6\ ms}{65\ a} = 21.5\frac{ms}{a}$<br><br>**AvT=** $\frac{\textbf{VTime}}{\textbf{\#Asser}}$ | *A2* | :- pred fib(N,R)<br>+ (cost(ub,steps,<br>exp(2,nat(N))-1000),<br>cost(lb,steps,<br>exp(2,nat(N))-10000)). | F in $[0,10] \cup [15,\infty]$<br>T in $[11,13]$<br>C in $[14,14]$ | | |
| | *A3* | :- pred fib(N,R)<br>:(intervals(nat(N),[i(1,12)]))<br>+ (cost(ub,steps,<br>exp(2,nat(N))-1000),<br>cost(lb,steps,<br>exp(2,nat(N))-10000)). | F in $[1,10]$<br>T in $[11,12]$ | | |
| <u>*Naive Reverse*</u><br>**lb,ub:** $0.5x^2 + 1.5x + 1$<br>x = length(A)<br>**AvT=** $\frac{1171.5\ ms}{54\ a} = 21.6\frac{ms}{a}$ | *B1* | :- pred nrev(A,B)<br>+ ( cost(lb,steps,length(A)),<br>cost(ub,steps,<br>exp(length(A),2))). | F in $[0,3]$<br>T in $[4,\infty]$ | 59.1 | 29.5 |
| | *B2* | :- pred nrev(A,_1)<br>+ (cost(lb, steps, length(A)),<br>cost(ub, steps, 10*length(A))). | F in $[0,0] \cup [17,\infty]$<br>T in $[1,16]$ | | |
| <u>*Quick Sort*</u><br>**lb:** $x + 5$<br>**ub:** $(\sum_{j=1}^{x} j2^{x-j}) + x2^{x-1}$ | *C1* | :- pred qsort(A,B)<br>+ cost(ub, steps,<br>exp(length(A),2)). | F in $[0,2]$<br>C in $[3,\infty]$ | 160.8 | 80.4 |
| $+2 * 2^x - 1$<br>x = length(A)<br>**AvT=** $\frac{1028.2\ ms}{56\ a} = 18.3\frac{ms}{a}$ | *C2* | :- pred qsort(A,B)<br>+ cost(ub, steps,<br>exp(length(A),3)). | C in $[0,\infty]$ | | |

Table 5. *Continued*

| | | | | Time (ms) | |
|---|---|---|---|---|---|
| Program+ Analysis info + AvT | ID | Assertion | Verif. result | Tot | Avg |
| *Client*<br>**ub:** $8x$<br>x = length(I) | *D1* | :- pred main(Op, I, B)<br>+ cost(ub, bits_received,<br>exp(length(I),2)). | C in $[1, 7]$<br>T in $[0, 0] \cup [8, \infty]$ | 31.8 | 10.6 |
| **AvT=** $\frac{1682.7\ ms}{60\ a} = 28.04 \frac{ms}{a}$ | *D2* | :- pred main(Op, I, B)<br>+ cost(ub, bits_received,<br>10*length(I)). | T in $[0, \infty]$ | | |
| | *D3* | :- pred main(Op, I, B)<br>: intervals(length(I),<br>[i(1,10),i(100,inf)])<br>+ cost(ub, bits_received,<br>10*length(I)). | T in $[1, 10] \cup [100, \infty]$ | | |
| *Reverse*<br>**lb,ub:** $x + 2$<br>x = length(A)<br>**AvT=** $\frac{760.9\ ms}{60\ a} = 12.6 \frac{ms}{a}$ | *E1* | :- pred reverse(A, B)<br>+ (cost(ub, steps,<br>500 * length(A))). | F in $[0, 0]$<br>T in $[1, \infty]$ | 30.0 | 30.0 |
| *Palindrome*<br>**lb,ub:** $x2^{x-1} + 2 * 2^x - 1$<br>x=length(X) | *F1* | :- pred palindrome(X,Y)<br>+ cost(ub,output_elements,<br>exp(length(X),2)). | F in $[0, \infty]$ | 31.5 | 15.7 |
| **AvT=** $\frac{1187.1\ ms}{52\ a} = 22.8 \frac{ms}{a}$ | *F2* | :- pred palindrome(X,Y)<br>+ cost(ub,output_elements,<br>exp(length(X),3)). | F in $[0, 2] \cup [5, \infty]$<br>T in $[3, 4]$ | | |
| *Powerset*<br>**ub:** $0.5 * 2^{x+1}$<br>x = length(A)<br>**AvT=** $\frac{880.9\ ms}{49\ a} = 17.9 \frac{ms}{a}$ | *G1* | :- pred powset(A,B)<br>+ cost(ub,output_elements,<br>exp(length(A),4)). | C in $[0, 1] \cup [17, \infty]$<br>T in $[2, 16]$ | 35.5 | 35.5 |
| *Hanoi*<br>**lb,ub:** $2^{x+1} - 2$<br>x = nat(A)<br>**AvT=** $\frac{1114.6\ ms}{64\ a} = 17.41 \frac{ms}{a}$ | *H1* | :- pred hanoi(A,B,C,D)<br>+ costb(steps, exp(2,nat(A)-3) + 2,<br>exp(2,nat(A)-3) + 30). | F in $[0, 1] \cup [5, \infty]$<br>T in $[2, 4]$ | 121.2 | 121.2 |

Table 6. *Comparison of assertion checking times for two methods*

| | | Intervals | | | |
|---|---|---|---|---|---|
| ID | Method | [1,12] | [1,100] | [1,1000] | [1,10000] |
| *A3* | Root | 58.1 | 64.6 | 71.7 | 66.5 |
| | Eval | 257 | 256.2 | 261.1 | 262.9 |
| *D3* | Root | 11.2 | 9 | 8.2 | 9.3 |
| | Eval | 39.7 | 41.5 | 38.8 | 55.2 |

statically checked. Column *Program* shows an expression $\mathbf{AvT} = \frac{\mathbf{VTime}}{\#\mathbf{Asser}}$ for each program giving the total time **VTime** in milliseconds spent by the verification of the number assertions given by the denominator **#Asser**, and the resulting average time per assertion (**AvT**). A few of those assertions are shown as examples in column *Assertion*, where 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. In all the experiments, in Table 5, the comparison of resource usage functions was precise, in the sense that the input data size intervals for which one function is greater, equal or smaller than another were exact, i.e., coincided with the actual intervals.

Note that, as mentioned before, the system can deal with different types of resource usage functions: polynomial functions (e.g., *Naive Reverse*), exponential functions (e.g., *Fibonacci*), and summation 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 of two reasons: (a) the analysis inferred an imprecise upper-bound cost function, exponential,[7] and (b) our approach to finding the data size intervals based on a transformation for removing summations and an approximation by polynomials did not cover such function. In some cases, either reason (a) or (b) in isolation can be the cause for our approach to fail to prove a given assertion. Even in the case when the cost bound function inferred by the analysis is precise, if

[7] This is due to the particular configuration of the analysis used in the experiments. Note, however, that the CiaoPP system also includes techniques that allow obtaining tighter bounds for divide-and-conquer programs, and in particular for this benchmark (Debray *et al.*, 1997).

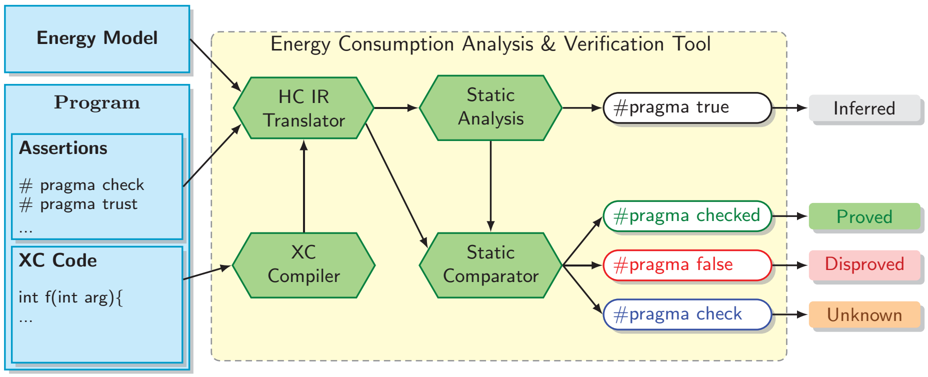


Fig. 15. Specialization of CiaoPP for energy consumption verification in XC programs.

it is too complex, our approach may still fail to find roots and data size intervals, and hence to prove the assertion.

Table 6 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 5. We can see that checking time grows quite slowly compared to the length of the interval, which grows exponentially.

*Root* is expected to be slower than *Eval* in the comparison of non-polynomial functions (*A3*), because *Root* must look for the functions intersections, and then must check every value in the intervals to ensure the absence of other roots. This behavior is not exhibited in this experiment because the intervals encountered by *Root* are narrow, and therefore the cost of checking every value in them is negligible. On the other hand, in the last interval, which grows wider as we increase the input data size interval, *Eval* is penalized by the task of checking every value in the interval, but *Root* is not penalized because it uses syntactic comparison.

## 6 Application to energy verification of imperative/embedded programs

As an application of the techniques presented, in this section, we provide an overview of a prototype tool that we have developed for performing *static energy consumption verification* of XC programs running on the XMOS XS1-L architecture. The tool has been implemented by specializing the CiaoPP general verification framework to process XC source, LLVM IR Lattner and Adve (2004), and ISA code. Figure 15 shows an overview diagram of the architecture of the tool. Hexagons represent different tool components and arrows indicate the communication paths among them. The tool takes as input an XC source program (left part of Fig. 15) that can optionally contain assertions in a C-style syntax. As explained in Section 1, such assertions are translated into the Ciao assertion language.

In our tool, the user can choose between performing the analysis at the ISA or LLVM IR levels (or both). We refer the reader to (Liqat *et al.*, 2016) for an experimental study that sheds light on the trade-offs implied by performing the analysis at each of these two levels, which can help the user to choose the level that fits the problem best.[8]

The associated ISA and/or LLVM IR representations of the XC program are generated using the xcc compiler. Such representations include useful metadata. The *HC IR translator* component (which will be described in Section 6.1) produces the internal representation used by the tool, HC IR, which includes the program and possibly specifications and/or trusted information (expressed in the Ciao assertion language). The HC IR translator performs several tasks:

1. Transforming the ISA and/or LLVM IR into HC IR.
2. Transforming specifications (and trusted information) written as C-like assertions (as described in Section 6.2) into the Ciao assertion language.
3. Transforming the energy model at the ISA level Kerrison and Eder (2015), expressed in JSON format, into the Ciao assertion language. In this specialization, such assertions express the energy consumed by individual ISA instruction representations, information which is required by the analyzer in order to propagate it during the static analysis of a program through code segments, conditionals, loops, recursions, etc., in order to infer analysis information (energy consumption functions) for higher-level entities such as procedures, functions, or loops in the program, as mentioned in Example 1. Figure 4 shows the transformed energy model in the Ciao assertion language. Each trust assertion provides information for one machine instruction. The model of the figure is simple, providing just constant upper and a lower bounds (and which are the same in most cases), but the bounds given (model for the instruction) can be functions of input data to the instruction (such as operand sizes) or context variables (such as voltage or clock speed, previous instruction, pipeline state, cache state, etc.).
4. In the case that the analysis is performed at the LLVM IR level, the *HC IR translator* component produces a set of Ciao assertions expressing the energy consumption corresponding to LLVM IR block representations in HC IR. Such information is produced from a mapping of LLVM IR instructions with sequences of ISA instructions and the ISA-level energy model. The mapping information is produced by the *mapping tool* that was first outlined in López-García (2014) (Section 2 and Attachments D3.2.4 and D3.2.5) and is described in detail in Georgiou *et al.* (2017).

Then, the CiaoPP parametric static resource usage analyzer (Navas *et al.*, 2007, 2008; Serrano *et al.*, 2014) takes the HC IR, together with the assertions, which express the energy consumed by LLVM IR blocks and/or individual ISA

[8] As a brief summary of the conclusions of Liqat *et al.* (2016), the ISA level allows somewhat tighter bounds when the analyzer can generate precise functions, but the LLVM IR level allows the analyzer to produce precise functions more often, because more structural information is preserved at that level. Overall, the LLVM IR level emerges as a good compromise.

instructions, and possibly some additional (trusted) information, and processes them, producing the analysis results, which are expressed also using Ciao assertions. Such results include energy usage functions (which depend on input data sizes) for each block in the HC IR (i.e., for the whole program and for all the procedures and functions in it.). The procedural interpretation of the HC IR programs, coupled with the resource-related information contained in the (Ciao) assertions, together allow the resource analysis to infer static bounds on the energy consumption of the HC IR programs that are applicable to the original LLVM IR and, hence, to their corresponding XC programs.

The verification of energy specifications is performed by the general component already described (see Section 1 and Fig. 15), which compares the energy specifications with the (safe) approximated information inferred by the static resource analysis, and produces the possible verification outcomes for different input-data size intervals.

## 6.1 ISA/LLVM IR to HC IR transformation

In this section, we briefly describe the transformations into the HC IR representation described in Section 2.2 that we developed in order to achieve the verification tool presented in Section 1 and depicted in Figure 15. The transformation of ISA code into HC IR was described in Liqat *et al.* (2014). We provide herein an overview of the LLVM IR to HC IR transformation.

LLVM IR programs are expressed using typed assembly-like instructions. Each function is in SSA form, represented as a sequence of basic blocks. Each basic block is a sequence of LLVM IR instructions that are guaranteed to be executed in the same order. Each block ends in either a branching or a return instruction. In order to represent each of the basic blocks of the LLVM IR in the HC IR, we follow a similar approach as in the ISA-level transformation (Liqat *et al.*, 2014). However, the LLVM IR includes an additional type transformation as well as better memory modelling. It is explained in detail in Liqat *et al.* (2016). The main aspects of this process, are the following:

1. Infer input/output parameters to each block.
2. Transform LLVM IR types into HC IR types.
3. Represent each LLVM IR block as an HC IR block and each instruction in the LLVM IR block as a literal ($S_i$).
4. Resolve branching to multiple blocks by creating clauses with the same signature (i.e., the same name and arguments in the head), where each clause denotes one of the blocks the branch may jump to.

The translator component is also in charge of translating the XC assertions to Ciao assertions and back. Assuming the Ciao type of the input and output of the function is known, the translation of assertions from Ciao to XC (and back) is relatively straightforward. Assuming the schema for `pred` assertions described in Section 2.2, the *Pred* field of the Ciao assertion is obtained from the scope of the XC assertion to which an extra argument is added representing the output of

```
⟨assertion⟩      ::=  '#pragma' ⟨status⟩ ⟨scope⟩ ':' ⟨body⟩
⟨status⟩         ::=  'check' | 'trust' | 'true' | 'checked' | 'false'
⟨scope⟩          ::=  ⟨identifier⟩ '(' ')' | ⟨identifier⟩ '(' ⟨arguments⟩ ')'
⟨arguments⟩      ::=  ⟨identifier⟩ | ⟨arguments⟩ ',' ⟨identifier⟩
⟨body⟩           ::=  ⟨precond⟩ '==>' ⟨cost_bounds⟩ | ⟨cost_bounds⟩
⟨precond⟩        ::=  ⟨upper_cond⟩ | ⟨lower_cond⟩ | ⟨lower_cond⟩ '&&' ⟨upper_cond⟩
⟨lower_cond⟩     ::=  ⟨ground_expr⟩ '<=' ⟨identifier⟩
⟨upper_cond⟩     ::=  ⟨identifier⟩ '<=' ⟨ground_expr⟩
⟨cost_bounds⟩    ::=  ⟨lower_bound⟩ | ⟨upper_bound⟩ | ⟨lower_bound⟩ '&&' ⟨upper_bound⟩
⟨lower_bound⟩    ::=  ⟨expr⟩ '<=' 'energy_nJ'
⟨upper_bound⟩    ::=  'energy_nJ' '<=' ⟨expr⟩
⟨expr⟩           ::=  ⟨expr⟩ '+' ⟨mult_expr⟩ | ⟨expr⟩ '-' ⟨mult_expr⟩
⟨mult_expr⟩      ::=  ⟨mult_expr⟩ '*' ⟨unary_expr⟩ | ⟨mult_expr⟩ '/' ⟨unary_expr⟩
⟨unary_expr⟩     ::=  ⟨identifier⟩
                    | ⟨integer⟩
                    | 'sum' '(' ⟨identifier⟩ ',' ⟨expr⟩ ',' ⟨expr⟩ ',' ⟨expr⟩ ')'
                    | 'prod' '(' ⟨identifier⟩ ',' ⟨expr⟩ ',' ⟨expr⟩',' ⟨expr⟩ ')'
                    | 'power' '(' ⟨expr⟩ ',' ⟨expr⟩ ')'
                    | 'log' '(' ⟨expr⟩ ',' ⟨expr⟩ ')'
                    | '(' ⟨expr⟩ ')'
                    | '+' ⟨unary_expr⟩
                    | '-' ⟨unary_expr⟩
                    | 'min' '(' ⟨identifier⟩ ')'
                    | 'max' '(' ⟨identifier⟩ ')'
```

Fig. 16. Syntax of the XC assertion language.

the function. The *Precond* fields are produced directly from the type of the input arguments: for each input variable, its regular type and its regular type size are added to the precondition, while the added output argument is declared as a free variable. Finally, the *Comp-Props* field is set to the usage of the resource `energy` by using the `costb` property, which also includes the lower and upper bounds from the XC energy consumption specification.

### *6.2 The XC assertion language*

The assertions within XC files are essentially equivalent to those of the Ciao assertion language, but written using a syntax that is closer to standard C notation and friendlier for C developers. These assertions are transparently translated into Ciao assertions (Puebla *et al.*, 2000a; Hermenegildo *et al.*, 2012) when XC files are loaded into the tool. The Ciao assertions output by the analysis are also translated back into XC assertions and added inline to a copy of the original XC file.

More specifically, the syntax of the XC assertions accepted by our tool is given by the grammar in Figure 16, where the non-terminal ⟨*identifier*⟩ stands for a standard C identifier, ⟨*integer*⟩ stands for a standard C integer, and the non-terminal ⟨*ground_expr*⟩ for a ground expression, i.e., an expression of type ⟨*expr*⟩ that does not contain any C identifiers that appear in the assertion scope (the non-terminal ⟨*scope*⟩).

XC assertions are directives starting with the token `#pragma` followed by the assertion *status*, the assertion *scope*, and the assertion *body*. The assertion *status* can take several values, including `check`, `checked`, `false`, `trust` or `true`, with the same meaning as in the Ciao assertions.

The assertion scope identifies the function the assertion is referring to, and provides the local names for the arguments of the function to be used in the body of the assertion. For instance, the scope `biquadCascade(state, xn, N)` refers to the function `biquadCascade` and binds the arguments within the body of the assertion to the respective identifiers `state`, `xn`, `N`. While the arguments do not need to be named in a consistent way w.r.t. the function definition, it is highly recommended for the sake of clarity. The *body* of the assertion expresses bounds on the energy consumed by the function and optionally contains preconditions (the left-hand side of the `==>` arrow) that constrain the argument sizes.

Within the body, expressions of type $\langle expr \rangle$ are built from standard integer arithmetic functions (i.e., `+`, `-`, `*`, `/`) plus the following extra functions:

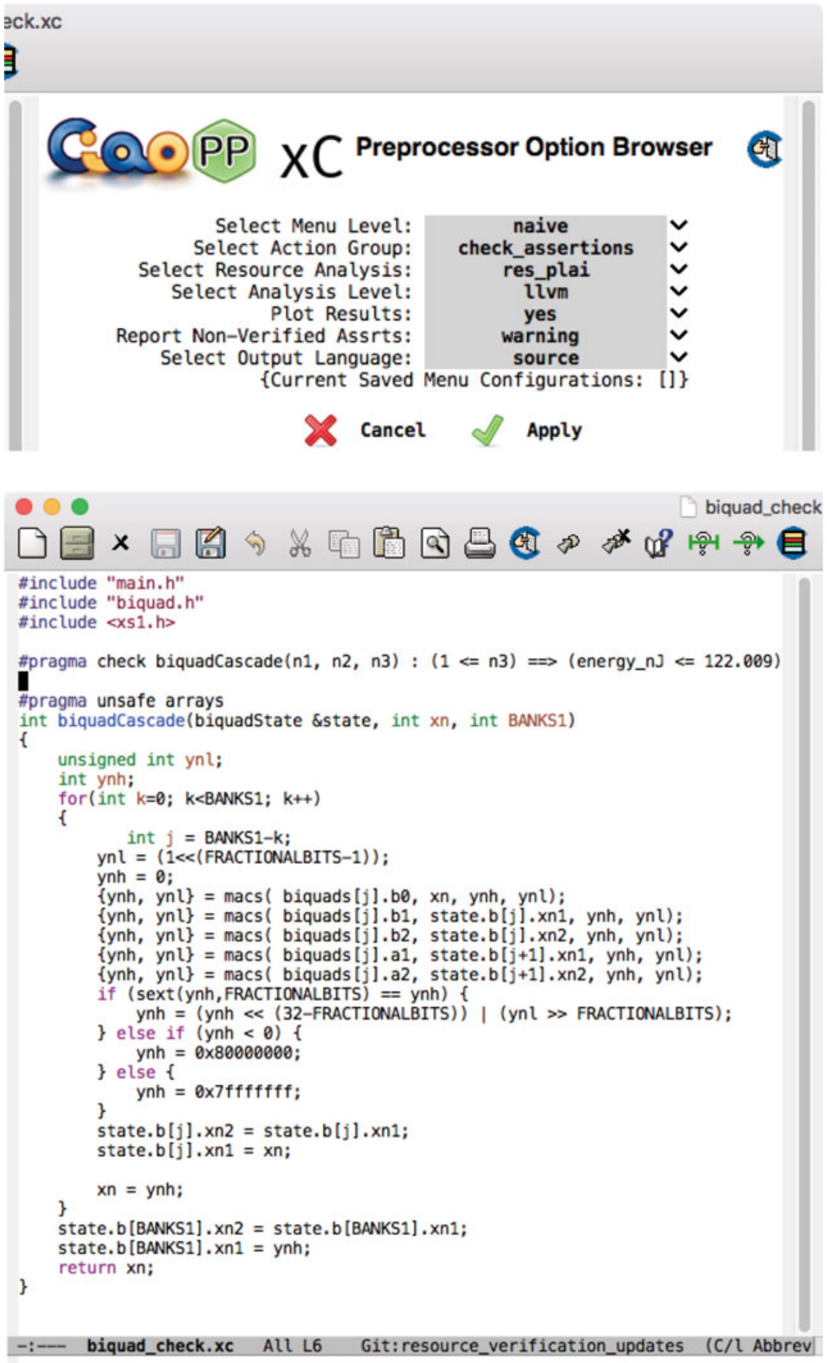
- `power(base, exp)` is the exponentiation of `base` by `exp`;
- `log(base, expr)` is the logarithm of `expr` in base `base`;
- `sum(id, lower, upper, expr)` is the summation of the sequence of the values of `expr` for `id` ranging from `lower` to `upper`;
- `prod(id, lower, upper, expr)` is the product of the sequence of the values of `expr` for `id` ranging from `lower` to `upper`;
- `min(arr)` is the minimal value of the array `arr`;
- `max(arr)` is the maximal value of the array `arr`.

Note that the argument of `min` and `max` must be an identifier appearing in the assertion scope that corresponds to an array of integers (of arbitrary dimension).

### *6.3 Using the tool for energy verification: Example*

In this section, we illustrate the use of the tool described above for the energy verification application, in a scenario where an embedded software developer has to decide values for program parameters that meet an energy budget. In particular, we consider the development of an equalizer (XC) program using a biquad filter. In Figure 17, we can see what the graphical user interface of our prototype looks like, with the code of this biquad example ready to be verified. The purpose of an equalizer is to take a signal, and to attenuate/amplify different frequency bands. For example, in the case of an audio signal, this can be used to correct for a speaker or microphone frequency response. The energy consumed by such a program directly depends on several parameters, such as the sample rate of the signal, and the number of banks, typically between 3 and 30 for an audio equalizer. A higher number of banks enables the designer to create more precise frequency response curves.

Assume that the developer has to decide how many banks to use in order to meet an energy budget while maximizing the precision of the frequency response curves at the same time. In this example, the developer writes an XC program, where the number of banks is a variable, say `N`. Assume also that the energy constraint to be



```
Select Menu Level:  naive
Select Action Group:  check_assertions
Select Resource Analysis:  res_plai
Select Analysis Level:  llvm
Plot Results:  yes
Report Non-Verified Assrts:  warning
Select Output Language:  source
{Current Saved Menu Configurations: []}
```

```
#include "main.h"
#include "biquad.h"
#include <xs1.h>

#pragma check biquadCascade(n1, n2, n3) : (1 <= n3) ==> (energy_nJ <= 122.009)

#pragma unsafe arrays
int biquadCascade(biquadState &state, int xn, int BANKS1)
{
    unsigned int ynl;
    int ynh;
    for(int k=0; k<BANKS1; k++)
    {
           int j = BANKS1-k;
        ynl = (1<<(FRACTIONALBITS-1));
        ynh = 0;
        {ynh, ynl} = macs( biquads[j].b0, xn, ynh, ynl);
        {ynh, ynl} = macs( biquads[j].b1, state.b[j].xn1, ynh, ynl);
        {ynh, ynl} = macs( biquads[j].b2, state.b[j].xn2, ynh, ynl);
        {ynh, ynl} = macs( biquads[j].a1, state.b[j+1].xn1, ynh, ynl);
        {ynh, ynl} = macs( biquads[j].a2, state.b[j+1].xn2, ynh, ynl);
        if (sext(ynh,FRACTIONALBITS) == ynh) {
            ynh = (ynh << (32-FRACTIONALBITS)) | (ynl >> FRACTIONALBITS);
        } else if (ynh < 0) {
            ynh = 0x80000000;
        } else {
            ynh = 0x7fffffff;
        }
        state.b[j].xn2 = state.b[j].xn1;
        state.b[j].xn1 = xn;

        xn = ynh;
    }
    state.b[BANKS1].xn2 = state.b[BANKS1].xn1;
    state.b[BANKS1].xn1 = ynh;
    return xn;
}
```

```
-:---  biquad_check.xc    All L6    Git:resource_verification_updates  (C/l Abbrev
```

Fig. 17. Graphical user interface of the prototype with the XC biquad program.

met is that an application of the biquad program should consume less or equal than 122 nJ (nanojoules). This constraint is expressed by the following check assertion:

```
#pragma check biquadCascade(state,xn,N) :
        (1 <= N) ==> (energy_nJ <= 122)
```

where the precondition `1 <= N` in the assertion (left-hand side of `==>`) expresses that the constraint should hold when the number of banks is greater than 1.

Then, the developer makes use of the tool by selecting the following menu options, as shown in the right-hand side of Figure 17: `check_assertions`, for `Action Group`; `res_plai`, for `Resource Analysis`; `llvm`, for `Analysis Level` (which will tell the analysis to take the LLVM IR option by compiling the source code into LLVM IR and transforming it into HC IR for analysis); `source`, for `Output Language` (the language in which the analysis/verification results are shown, in this case, the original XC source); and finally, `yes` for `Plot results` (in order to obtain a graphical representation of the results). After clicking on the `Apply` button below the menu options, the analysis is performed, which infers a lower and an upper bound function for the consumption of the program. Specifically, those bounds are represented by the following assertion, which is included in the output of the tool:

```
#pragma true biquadCascade(A,B,C) :
        (16.502*C+5.445 <= energy_nJ && energy_nJ <= 16.652*C+5.445)
```

Then, the verification of the specification, i.e., check assertion, is performed by comparing the energy bound functions above with the upper bound expressed in the specification, i.e., 122 nJ, a constant value in this case, as illustrated in Figure 18. Such figure has been automatically generated by our tool and includes the plots of both the specification and the analysis results, which contributes to a better understanding of the results. The $x$-axis represents the input data size, in this case, the number of banks given by `N`, on which the cost function depends, and the $y$-axis represents the energy consumption. The flat (blue) region corresponds to the specification, whereas the sloping green region that lies between two red lines represents the area bounded by the cost functions automatically inferred by the analyzer.

As a result of the comparison, the following two assertions are produced and included in the output file of the tool:

```
#pragma checked biquadCascade(state,xn,N) :
         (1 <= N && N <= 7) ==> (energy_nJ <= 122)
#pragma false biquadCascade(state,xn,N):(8 <= N)==>(energy_nJ <= 122)
```

The first one expresses that the original assertion holds subject to a precondition on the parameter `N`, i.e., in order to meet the energy budget of 122 nJ, the number of banks `N` should be a natural number in the interval [1, 7] (precondition `1 <= N && N <= 7`). The second one expresses that the original specification is not met (status `false`) if the number of banks is greater or equal to 8.

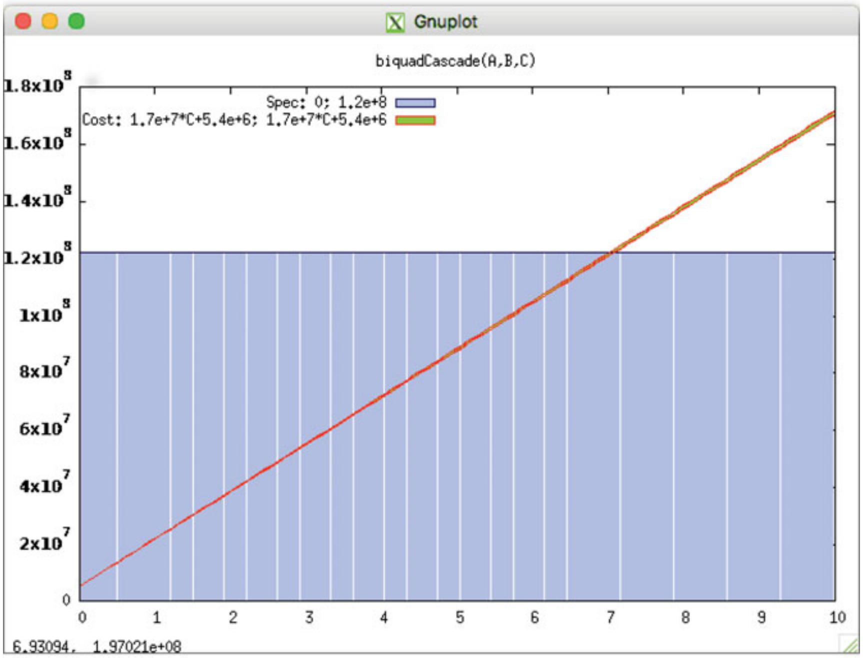


Fig. 18. Visualization of analysis results and specifications in the tool.

Since the goal is to maximize the precision of the frequency response curves and to meet the energy budget at the same time, the number of banks should be set to 7. The developer could also be interested in meeting an energy budget but this time ensuring a lower bound on the precision of the frequency response curves. For example, by ensuring that $\mathtt{N} \geqslant 3$, the acceptable values for N would be in the range $[3,\ 7]$.

In the more general case, where the energy function inferred by the tool depends on more than one parameter, the determination of the values for such parameters is reduced to a constraint solving problem. The advantage of this approach is that the parameters can be determined analytically at the program development phase, without the need of determining them experimentally by measuring the energy of expensive program runs with different input parameters, which in any case cannot provide hard guarantees.

Our tool produces sound results, provided of course that the energy model expresses correct information. Also, the accuracy of the bounds obtained depends on the accuracy of the energy model. Note that, if the objective is to choose parameters that guarantee completely that the specifications are met, even not very tight bounds will be better than testing/profiling, which, as mentioned before, cannot provide hard guarantees. On the other hand, having tight bounds is always desirable, in order to get more efficient values.

In order to illustrate this, assume that the user uses a slightly different energy model for the verification, which considers a 10% error in its energy measurements,

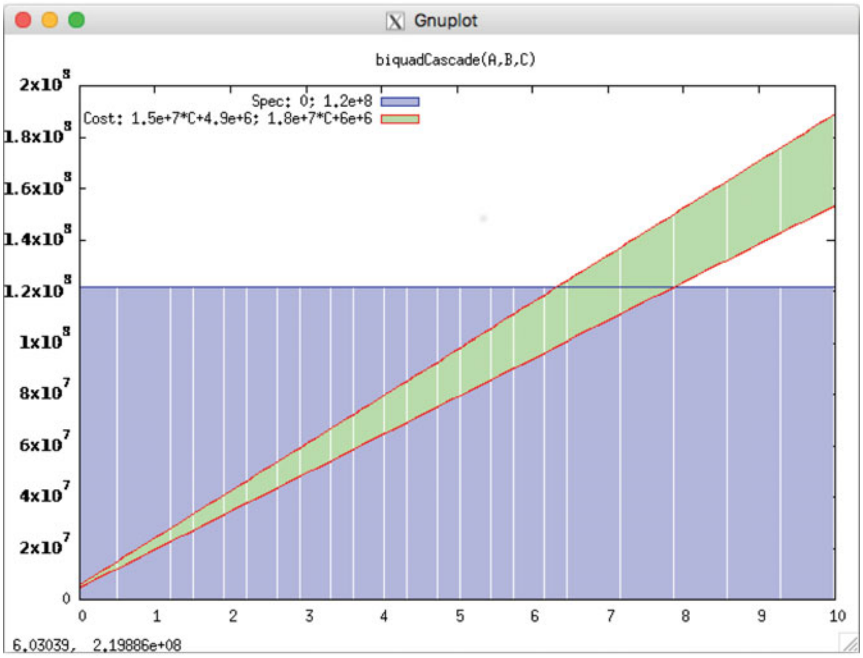


Fig. 19. Visualization of analysis results and specifications, using a different energy model.

and assume that this model, expressed again as a set of *trust* assertions in the Ciao assertion language, as in Figure 4, is contained in file energy_llvm_10. In this case, the user needs to provide this information to the tool as follows:

```
#pragma model <energy_llvm_10>
```

Following the same procedure as before, after running the tool the following results are obtained:

```
#pragma true biquadCascade(A,B,C) :
        (14.851*C+4.9 <= energy_nJ && energy_nJ <= 18.317*C+5.989)

#pragma checked biquadCascade(state,xn,N) :
         (1 <= N && N <= 6) ==> (energy_nJ <= 122)
#pragma check biquadCascade(state,xn,N) :
         (7 <= N && N <= 7) ==> (energy_nJ <= 122)
#pragma false biquadCascade(state,xn,N) :
         (8 <= N) ==> (energy_nJ <= 122)
```

As we can see, the area delimited by the lower and upper bound functions inferred is wider, and the verification results include an additional check assertion for $N = 7$. The assertion with status *check* indicates that for the value of the argument $N = 7$, the verification cannot conclude if the energy budget will be met or not. This fact is represented in Figure 19, where the sloping/green analysis region intersects the flat/blue specification region but is not completely included in it.

## 7 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 (Albert *et al.*, 2010, 2015). 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 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, (Albert *et al.*, 2010) 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. Note also that, although we have presented our work applied to Horn clause programs and XC programs, the CiaoPP system can also deal with Java bytecode (Navas *et al.*, 2009; Méndez-Lojo *et al.*, 2007).

In a more general context, using abstract interpretation in verification, debugging, and related tasks has now become well established. To cite some early work, abstractions were used in the context of algorithmic debugging in Lichtenstein and Shapiro (1988). Abstract interpretation has been applied by Bourdoncle (1993) to debugging of imperative programs and by Comini *et al.* (1995) to the algorithmic debugging of logic programs (making use of partial specifications in Comini *et al.*, 1999), and by Cousot (2003) to verification, among others. The CiaoPP framework (Bueno *et al.*, 1997; Hermenegildo *et al.*, 1999, 2005) was pioneering, offering an integrated approach combining abstraction-based verification, debugging, and run-time checking with an assertion language. This approach has recently also been applied in a number of contract-based systems (Fähndrich and Logozzo, 2011; Tobin-Hochstadt and Van Horn, 2012; Nguyen and Horn, 2015).

Horn clauses are used in many different applications nowadays as compilation targets or intermediate representations in analysis and verification tools (Navas *et al.*, 2009; Méndez-Lojo *et al.*, 2007; Grebenshchikov *et al.*, 2012; Hojjat *et al.*, 2012; de Moura and Bjørner, 2008; Bjørner *et al.*, 2014; Kafle *et al.*, 2016).

## 8 Conclusions

Taking as starting point our configurable framework for static resource usage verification where specifications can include both lower and upper bound, data size-dependent resource usage functions, we have reviewed how this framework supports different programming languages (both declarative and imperative) as well as different compiler representations. This is achieved by a translation of the corresponding input language to an internal representation based on Horn clauses (HC IR). The framework is architecture independent, since we use low-level resource usage models that are specific for each architecture, describing the resource usage of basic elements and operations.

We have also generalized the assertions supported to include preconditions expressing intervals within which the input data size of a program is supposed to lie (i.e., intervals for which each assertion can be tested). These extended assertions can be used both in specifications and in the output of the analyzers. In addition, we have provided a formalization of how the traditional framework is extended for the data size interval-dependent verification of resource usage properties.

Our framework can deal with the different types of resource usage functions (e.g., polynomial, exponential, summation, or logarithmic functions), in the sense that the analysis can infer them, and the specifications can involve them.

A key aspect of the framework is to be able to compare these mathematical functions. We have proposed methods for function comparison that are safe/sound, in the sense that the results of verification either give a valid answer (true or false) or return "unknown." 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. 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.

We have also reported on a prototype implementation of the proposed general framework for resource usage verification and provided experimental results, which are encouraging, suggesting that our techniques are feasible and accurate in practice. We have also specialized such implementation for verifying energy consumption specifications of imperative/embedded programs. Finally, we have shown through an example, and using the prototype implementation for the XC language and XS1-L architecture, how our verification system can prove whether energy consumption specifications are met or not, or infer particular conditions under which the specifications hold. We have illustrated through this example how embedded software developers can use this tool, in particular, for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

## References

Albert, E., Arenas, P., Genaim, S., Herraiz, I. and Puebla, G. 2010. Comparing cost functions in resource analysis. In *Proc. of 1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*. Lecture Notes in Computer Science, vol. 6234. Springer, 1–17.

Albert, E., Arenas, P., Genaim, S. and Puebla, G. 2015. A Practical Comparator of Cost Functions and its Applications. *Science of Computer Programming 111*, 483–504. Special Issue on Foundational and Practical Aspects of Resource Analysis (FOPARA 2009).

Bjørner, N., Fioravanti, F., Rybalchenko, A. and Senni, V., Eds. 2014. In *Proc. of Workshop on Horn Clauses for Verification and Synthesis*. Electronic Proceedings in Theoretical Computer Science.

Bourdoncle, F. 1993. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation*. R. Cartwright, Ed. ACM, 46–55.

Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M. V., Maluszynski, J. and Puebla, G. 1997. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Proc. of the 3rd International Workshop on Automated Debugging (AADEBUG'97)*. University of Linköping Press, Linköping, Sweden, 155–170.

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

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

Cousot, P. 2003. Automatic verification by abstract interpretation, invited tutorial. In *Proc of 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'03)*. Number 2575 in Lecture Notes in Computer Science. Springer, 20–24.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of ACM Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, 238–252.

Dart, P. and Zobel, J. 1992. A regular type language for logic programs. In *Types in Logic Programming*. MIT Press, 157–187.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Proc of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, C. R. Ramakrishnan and J. Rehof, Eds. Lecture Notes in Computer Science, vol. 4963. Springer, 337–340.

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

Debray, S. K., Lin, N.-W. and Hermenegildo, M. V. 1990. Task granularity analysis in logic programs. In *Proc. of ACM Conference on Programming Language Design and Implementation (PLDI'90)*. ACM Press, 174–188.

Debray, S. K., López-García, P., Hermenegildo, M. V. and Lin, N.-W. 1997. Lower bound cost estimation for logic programs. In *Proc. of International Logic Programming Symposium.* MIT Press, Cambridge, MA, 291–305.

Fähndrich, M. and Logozzo, F. 2011. Static contract checking with abstract interpretation. In *Proc. of International Conference on Formal Verification of Object-oriented Software (FoVeOOS'10)*. Lecture Notes in Computer Science, vol. 6528. Springer, 10–30.

Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M. and Rossi, F. 2009. *GNU Scientific Library Reference Manual - 3rd ed. (v1.12)*. Network Theory Ltd. Available at http://www.gnu.org/software/gsl/.

Georgiou, K., Kerrison, S., Chamski, Z. and Eder, K. 2017. Energy transparency for deeply embedded programs. *ACM Transactions on Architecture and Code Optimization 14,* 1 (March), 8:1–8:26.

Gleich, D. F. 2005. *Finite Calculus: A Tutorial for Solving Nasty Sums.* Combinatorics, Stanford University.

Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C. and Rybalchenko, A. 2012. HSF(C): A software verifier based on Horn clauses — (competition contribution). In *TACAS*, C. Flanagan and B. König, Eds. Lecture Notes in Computer Science, vol. 7214. Springer, 549–551.

Hermenegildo, M. V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J. and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1–2 (January), 219–252. http://arxiv.org/abs/1102.5497.

Hermenegildo, M. V., Puebla, G. and Bueno, F. 1999. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging.

In *The Logic Programming Paradigm: A 25–Year Perspective*, K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, Eds. Springer-Verlag, 161–192.

Hermenegildo, M. V., Puebla, G., Bueno, F. and Lopez-Garcia, P. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming 58,* 1–2 (October), 115–140.

Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V. and Rümmer, P. 2012. A verification toolkit for numerical transition systems—Tool paper. In *Proc. of Formal Methods (FM'12).* Lecture Notes in Computer Science, vol. 7436. Springer, 247–251.

Kafle, B., Gallagher, J. P. and Morales, J. F. 2016. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *Proc. of 28th International Conference on Computer Aided Verification Part I (CAV'16)*, Toronto, ON, Canada, July 17–23, S. Chaudhuri and A. Farzan, Eds. Lecture Notes in Computer Science, vol. 9779. Springer, 261–268.

Kerrison, S. and Eder, K. 2015. Energy modeling of software for a hardware multithreaded embedded microprocessor. *ACM Transactions on Embedded Computing Systems 14,* 3 (April), 1–25.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the International Symposium on Code Generation and Optimization (CGO'04).* IEEE Computer Society, 75–88.

Lichtenstein, Y. and Shapiro, E. Y. 1988. Abstract algorithmic debugging. In *Proc. of 5th International Conference and Symposium on Logic Programming*, R. A. Kowalski and K. A. Bowen, Eds. MIT, Seattle, Washington, 512–531.

Liqat, U., Georgiou, K., Kerrison, S., Lopez-Garcia, P., Hermenegildo, M. V., Gallagher, J. P. and Eder, K. 2016. Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. In *Proc. of 4th International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'15),* London, UK, April 11, 2015. *Revised Selected Papers*, M. V. Eekelen and U. D. Lago, Eds. Lecture Notes in Computer Science, vol. 9964. Springer, 81–100.

Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M. V. and Eder, K. 2014. Energy consumption analysis of programs based on XMOS ISA-level models. In *Proc. of 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13), Revised Selected Papers*, G. Gupta and R. Peña, Eds. Lecture Notes in Computer Science, vol. 8901. Springer, 72–90.

López-García, P., Ed. 2014. *Initial Energy Consumption Analysis.* ENTRA Project: Whole-Systems Energy Transparency (FET project 318337). Deliverable 3.2, http://entraproject.eu.

López-García, P., Darmawan, L. and Bueno, F. 2010. A framework for verification and debugging of resource usage properties. In *Proc. of Technical Communications of the 26th International Conference on Logic Programming (ICLP'10)*, M. V. Hermenegildo and T. Schaub, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 7. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 104–113.

Lopez-Garcia, P., Darmawan, L., Bueno, F. and Hermenegildo, M. V. 2012. Interval-based resource usage verification: Formalization and prototype. In *Proc. of 2nd International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'11), Revised Selected Papers*, R. P. na, M. Eekelen, and O. Shkaravska, Eds. Lecture Notes in Computer Science, vol. 7177. Springer-Verlag, 54–71.

Lopez-Garcia, P., Haemmerlé, R., Klemen, M., Liqat, U. and Hermenegildo, M. V. 2015. Towards energy consumption verification via static analysis. In *Proc. of Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES'15)*, arXiv:1501.03064. arXiv:1512.09369.

Méndez-Lojo, M., Navas, J. and Hermenegildo, M. 2007. A flexible (C)LP-based approach to the analysis of object-oriented programs. In *Proc. of 17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*. Number 4915 in Lecture Notes in Computer Science. Springer-Verlag, 154–168.

Mera, E., López-García, P. and Hermenegildo, M. V. 2009. Integrating software testing and run-time checking in an assertion verification framework. In *Proc. of 25th International Conference on Logic Programming (ICLP'09)*. Lecture Notes in Computer Science, vol. 5649. Springer-Verlag, 281–295.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming 13,* 2/3 (July), 315–347.

Navas, J., Méndez-Lojo, M. and Hermenegildo, M. 2008. Safe upper-bounds inference of energy consumption for Java Bytecode applications. In *Proc of The 6th NASA Langley Formal Methods Workshop (LFM'08)*. 29–32. Extended Abstract.

Navas, J., Méndez-Lojo, M. and Hermenegildo, M. V. 2009. User-definable resource usage bounds analysis for Java Bytecode. In *Proc. of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*. Electronic Notes in Theoretical Computer Science, vol. 253. Elsevier–North Holland, 65–82.

Navas, J., Mera, E., López-García, P. and Hermenegildo, M. 2007. User-definable resource bounds analysis for logic programs. In *Proc. of 23rd International Conference on Logic Programming (ICLP'07)*. Lecture Notes in Computer Science, vol. 4670. Springer.

Nguyen, P. and Horn, D. V. 2015. Relatively complete counterexamples for higher-order programs. In *Proc. of Programming Language Design and Implementation (PLDI'15)*. ACM, 446–456.

Puebla, G., Bueno, F. and Hermenegildo, M. V. 2000a. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming*, P. Deransart, M. V. Hermenegildo, and J. Maluszynski, Eds. Number 1870 in Lecture Notes in Computer Science. Springer-Verlag, 23–61.

Puebla, G., Bueno, F. and Hermenegildo, M. V. 2000b. Combined static and dynamic assertion-based debugging of constraint logic programs. In *Proc. of Logic-based Program Synthesis and Transformation (LOPSTR'99)*. Number 1817 in Lecture Notes in Computer Science. Springer-Verlag, 273–292.

Serrano, A., Lopez-Garcia, P. and Hermenegildo, M. V. 2014. Resource usage analysis of logic programs via abstract interpretation using sized types. In *Theory and Practice of Logic Programming, 30th International. Conference on Logic Programming (ICLP'14) Special Issue 14,* 4–5, 739–754.

Tobin-Hochstadt, S. and Van Horn, D. 2012. Higher-order symbolic execution via contracts. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12*). ACM, 537–554.

Vaucheret, C. and Bueno, F. 2002. More precise yet efficient type inference for logic programs. In *Proc. of 9th International Static Analysis Symposium (SAS'02)*. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, 102–116.

Watt, D. 2009. *Programming XC on XMOS Devices*. XMOS Limited.