Towards Resource Usage Analysis
of MiniZinc Models

F. Bueno[1], Maria Garcia de la Banda[2], M. V. Hermenegildo[1,3],
P. Lopez-Garcia[3,7], E. Mera[4], and P. J. Stuckey[5,6]

[1]Universidad Politécnica de Madrid (UPM), Spain
[2]Faculty of Information Technology, Monash University, Australia
[3]IMDEA Software Institute, Spain
[4]Universidad Complutense de Madrid (UCM), Spain
[5]National ICT Australia, Victoria Laboratory
[6]Dept. of CS & SE, University of Melbourne, Australia
[7]Spanish Research Council (CSIC), Spain
bueno@fi.upm.es — Maria.GarciadelaBanda@infotech.monash.edu.au
{manuel.hermenegildo,pedro.lopez}@imdea.org
edison@fdi.ucm.es — pjs@cs.mu.oz.au

Abstract. We present a method for the static resource usage analysis
of MiniZinc models. The analysis can infer upper bounds on the usage
that a MiniZinc model will make of some resources such as the number
of constraints of a given type (equality, disequality, global constraints,
etc.), the number of variables (search variables or temporary variables),
or the size of the expressions before calling the solver. These bounds
are obtained from the models independently of the concrete input data
(the instance data) and are in general functions of sizes of such data. In
our approach, MiniZinc models are translated into Ciao programs which
are then analysed by the CiaoPP system. CiaoPP includes a parametric
analysis framework for resource usage in which the user can define re-
sources and express the resource usage of library procedures (and certain
program constructs) by means of a language of assertions. We present
the approach and report on a preliminary implementation, which shows
the feasibility of the approach, and provides encouraging results.

Key words: Resource Usage Analysis, Constraint Modeling Languages,
Constraint Programming, Complexity Analysis.

1 Introduction and Motivation

Inferring information about the resource usage of computations can be useful
for a variety of applications, including resource usage verification and debug-
ging, resource control in parallel/distributed computing, and resource-oriented
specialisation for the selection among different design implementations. Analysis
of a wide range of resources, from execution time and memory to energy con-
sumption, has been studied for several programming paradigms including logic
languages [2, 7], functional languages [1, 4, 9], and imperative languages [10, 3].

In this paper we present a method for the static resource usage analysis of
MiniZinc models [8]. The analysis can infer upper bounds on the usage that a

MiniZinc model will make of resources such as the number of constraints of a given type (equality, disequality, global constraints, etc.), the number of variables (search variables or temporary variables), or the size of the expressions before calling the solver. This is, to our knowledge, the first proposal for resource usage analysis in constraint programming.

An interesting feature of our approach is that the bounds on resource usages are obtained from the models (which describe the structure of a class of problems) independently of the (yet unknown) concrete input data (the data which specifies a particular problem within such class). In general, the inferred bounds are functions on the size of the input data. These functions, which will be referred to as *resource usage functions*, allow us to apply our method to, for example, select among different solvers in order to reduce the usage of some resource, or to verify and debug a given resource usage property.

Our approach leverages on the significant amount of work and tools developed for resource analysis in the context of logic programming. To this end we propose a transformation from the MiniZinc model into an operational version encoded as a program in the logic programming subset of the Ciao [6] language. This allows us to analyse this transformed program with Ciao's preprocessor, CiaoPP [5], which includes a parametric analysis framework for resource usage that must be instantiated to infer the resources of interest. This instantiation is done by means of a language of assertions which allows the user to define resources and express the resource usage of library procedures (and certain program constructs). Based on this information, the analyser can infer bounds on the resource usage of the whole program. This paper presents the CiaoPP instantiation that we have developed and applied to the transformed MiniZinc programs, obtaining the desired bounds. The final step is to map the information inferred by the analysis of the Ciao program back to the original MiniZinc model. Note that there is no disadvantage in translating a restricted language like MiniZinc into a richer language and then performing the analysis. For example, the "where" MiniZinc clauses (to be described later) add considerable complexity to the resulting Ciao programs. Moreover, the CiaoPP approach efficiently deals with program schemas resulting from the translation of simple MiniZinc loops and has certain accuracy guarantees.

2 The MiniZinc Modeling Language

A MiniZinc [8] problem specification has two parts: (a) the *model*, which describes the structure of a class of problems; and (b) the input *data*, which provides values to the parameters in the model and, thus, specifies a particular problem within this class. The pairing of a model with a particular data set is referred to as a *model instance* (or simply *instance*).

The model and the data often appear in separate files, with data files simply containing assignments to the parameters declared in the model. Users can specify the particular data files to be used by a model through the command line, rather than naming them in the model file, thus ensuring the model is not tied to any particular data file. This separation of data and model is important for the purposes of this paper, since we wish to analyse the model independently

```
1  % (square) job shop scheduling in MiniZinc
2  int: size;                                    % size of problem
3  array [1..size,1..size] of int: d;            % task durations
4  int: total = sum(i,j in 1..size) (d[i,j]);    % total duration
5  array [1..size,1..size] of var 0..total: s;   % start times
6  var 0..total: end;                            % total end time
7
8  predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
9      s1 + d1 <= s2 \/ s2 + d2 <= s1;
10
11 constraint
12     forall(i in 1..size) (
13         forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\
14         s[i,size] + d[i,size] <= end /\
15         forall(j,k in 1..size where j < k) (
16             no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
17         )
18     );
19
20 solve minimize end;
21
22 output [ show(s[i,j]) ++ if j == size then "\n" else " " endif |
23          i,j in 1..size];
```

Fig. 1. MiniZinc model (jobshop.mzn) for the job shop problem.

of the data with the aim of learning generic formulas about resource usage that are dependent on the as yet unknown data of a given instance.

A MiniZinc Example Each MiniZinc model is a sequence of *items*, which may appear in any order. Let us introduce the main kinds of items that can appear in a MiniZinc model and their characteristics by means of the example model shown in Figure 1 for a restricted kind of job shop scheduling problem.

Line 1 shows a comment, since it is introduced by the '%' character.

Lines 2–6 show different *variable declaration items*. In particular, lines 2-4 declare three *parameters*, i.e., variables that are fixed in the model and will be assigned a value by means of an *assignment item* either in the data file (as is the case for integer parameter size and 2D array of integers d), or in the model (as it is the case for total). In contrast, line 5 declares s to be a 2D array of *decision variables*, and line 6 declares end to be an integer decision variable with a restricted range. Note that variables are known to be decision variables if and only if they are declared with the var prefix, and parameters otherwise.

Lines 8–9 show a user-defined *predicate item*, no_overlap, which uses disjunction and implication over the start times and durations of two tasks to ensure they do not overlap in time. Lines 11–18 show a *constraint item*. It uses the built-in forall to loop over each job ensuring that: (line 13) the tasks are in order; (line 14) they finish no later than end; and (lines 15–17) no two tasks in the same column overlap in time. Multiple constraint items can appear in a model and, if so, they are implicitly conjoined.

Every model must include exactly one *solve item*. In our example, this item appears in line 20 indicating we want a solution that minimises the end time. We

```
24  size = 2;
25  d = [  2,5,
26         3,4  ];
```

Fig. 2. MiniZinc data (`jobshop2x2.data`) for the job shop problem.

can also maximise a variable or just look for any solution ("`solve satisfy`").
Models can also have *output items*, such as the one given in lines 22–23, specifying
how to display the results of the optimisation.

Figure 2 shows a possible data file for our model, containing 2 assignment
items in lines 24–26 assigning values to the `size` and `d` parameters.

3 The Ciao Programming Language

Ciao [6] is a general-purpose programming language that supports a number
of programming paradigms including functional, logic, and constraint program-
ming. In this paper we use Ciao's logic programming subset (i.e., the Prolog
syntax and operational semantics) and its programming environment, which
provides a powerful *assertion language*. In particular, as we will show later,
we translate MiniZinc programs into Ciao programs composed of one or more
modules containing Horn clauses and assertions. The procedural interpretation
of these Ciao programs, coupled with resource-related information contained in
the assertions, will allow the resource analysis capabilities of its preprocessor
(CiaoPP [5]) to infer static bounds on the resource consumption of the Ciao
programs that are applicable to the original MiniZinc model. We also take ad-
vantage of Ciao's novel module system [6] that allows writing language extensions
(*packages*) by grouping together syntactic definitions, compilation options, and
plugins to the compiler.

We use a subset of the Ciao assertion language which allows expressing global
"computational" properties and, in particular, resource usage. While a detailed
introduction to the assertion language can be found in [5], for brevity, we only
introduce here the class of **pred assertions**, which describes a particular pred-
icate and, in general, follows the schema:

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

where *Pred* is a predicate symbol applied to distinct free variables while *Precond*
and *Postcond* are logic formulae about execution states. An execution state is de-
fined by variable/value bindings in 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* also holds in the success state. Finally, the
Comp-Props field is used to describe properties of the whole computation for
calls to predicate *Pred* that meet *Precond*. In our application *Comp-Props* are
precisely the resource usage properties.

For example, the following assertion for a typical `append/3` predicate:

```
:- trust pred append(A,B,C): (list(A),list(B),var(C)) =>
       (list(A),list(B),list(C)) + cost(ub,steps,length(A)+1).
```

states that for any call to predicate `append/3` with the first and second arguments
bound to lists and the third one unbound, if the call succeeds, then the third
argument is also bound to a list. It also states that an upper bound on the

```
forall1_j(0,_,_,_,_,_,_).
forall1_j(Ind_j,Uj,Index_i,
          Size,S,D,End) :-
  Ind_j > 0,
  Index_j is Uj - Ind_j + 1,
  I is Index_i - 1 + 1,
  J is Index_j - 1 + 1,
  element2d_dint(I,J,S,Sij),
  element2d_int(I,J,D,Dij),
  J1 is J + 1,
  element2d_dint(I,J1,S,Sij1),
  plus(Sij, Dij, A),
  leq(A, Sij1),
  NInd_j is Ind_j - 1,
  forall1_j(NInd_j,Uj,Index_i,
            Size,S,D,End).

forall2_j(0,_,_,_,_,_).
forall2_j(Ind_j,Uj,Index_i,
          Size,S,D) :-
  Ind_j > 0,
  Index_j is Uj - Ind_j + 1,
  Ind_k is Size - 1 + 1,
  forall2_jk(Ind_k,Size,Index_i,
             Index_j,S,D),
  NInd_j is Ind_j - 1,
  forall2_j(NInd_j,Uj,
            Index_i,Size,S,D).
```

```
forall2_jk(0,_,_,_,_,_).
forall2_jk(Ind_k,Uk,Index_i,Index_j,
           S,D) :-
  Ind_k > 0,
  Index_k is Uk - Ind_k + 1,
  forall2_jk_where(Index_i,Index_j,
                   Index_k,S,D),
  NInd_k is Ind_k - 1,
  forall2_jk(NInd_k,Uk,Index_i,
             Index_j,S,D).

forall2_jk_where(Index_i,Index_j,
                 Index_k,S,D) :-
  Index_j < Index_k,
  !,
  element2d_dint(Index_j,Index_i,S,Sji),
  element2d_int(Index_j,Index_i,D,Dji),
  element2d_dint(Index_k,Index_i,Ski),
  element2d_int(Index_k,Index_i,D,Dki),
  no_overlap(Sji,Dji,Ski,Dki).
forall2_jk_where(_,_,_,_,_).

no_overlap(S1,D1,S2,D2) :-
  plus(S1,D1,A),
  reif_leq(A,S2,B1),
  plus(S2,D2,B),
  reif_leq(B,S1,B2),
  or(B1,B2,1).
```

Fig. 3. Ciao code from translation of (parts of) the MiniZinc model in Figure 1.

number of resolution steps required to execute any of such calls is $length(A)+1$, a function on the length of list A.

The global non-functional property **cost/3** (appearing in the "+" field), is used for expressing resource usages and follows the schema:

$$\mathbf{cost}(\textit{Approx}, \textit{Res\_Name}, \textit{Arith\_Expr})$$

where *Res_Name* is a user-provided identifier for the resource the assertion refers to, *Arith_Expr* is an arithmetic function that maps input data sizes to resource usage, and *Approx* indicates, for example, whether *Arith_Expr* provides an upper bound (**ub**) or a lower bound (**lb**).

Each assertion can be in a particular *status*, marked with the keyword prefixes **check** (indicating the assertion needs to be checked), **checked** (it 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** (it provides information coming from the programmer and needs to be trusted), or **true** (it is the result of static analysis and thus correct, i.e., safely approximated). The default status is **check**.

4 Translating MiniZinc Models into Ciao Programs

As mentioned before, our approach is based on translating MiniZinc programs into Ciao programs. We now present how this translation is performed.

Transformation Rules Before we formalise the transformation rules used during our translation, let us introduce some notation. We use $[\![t]\!]_V^T$ to denote the goal + program resulting from converting the MiniZinc term t into a Prolog term assigned to Prolog variable T in the context of variable declaration set V. A set of variable declarations V is stored as a list of pairs of the form (v, T) where

v is a unique ground variable name for Prolog variable T. Let $\{T_1, \ldots, T_n\} = \{T \mid (v, T) \in V\}$. Then, $pv(V)$ represents the sequence $T_1, \ldots, T_n$, made up of all the Prolog variables in V separated by commas. In practice, variables that are not used are removed from $pv(V)$. For variable declarations $\llbracket type : var \rrbracket_V^T$ returns in T a list of variable declarations. We use $\llbracket t \rrbracket_V^1$ on a Boolean MiniZinc expression t to denote that instead of generating a variable with the value of the expression, we simply force the Boolean expression to be true.

The transformation rules are as follows:

$\llbracket v \rrbracket_V^T \implies \epsilon$, such that $(v, T) \in V$ (ϵ denotes an empty sequence of characters).
$\llbracket t_1 + t_2 \rrbracket_V^T \implies \llbracket t_1 \rrbracket_V^{T1}$, $\llbracket t_2 \rrbracket_V^{T2}$, `plus(T1,T2,T)`.

$\llbracket t_1 \wedge t_2 \rrbracket_V^B \implies \llbracket t_1 \rrbracket_V^{B1}$, $\llbracket t_2 \rrbracket_V^{B2}$, `and(B1,B2,B)`.
$\llbracket t_1 \wedge t_2 \rrbracket_V^1 \implies \llbracket t_1 \rrbracket_V^1$, $\llbracket t_2 \rrbracket_V^1$.

$\llbracket$ if b then t_1 else t_2 endif $\rrbracket_V^T \implies$ `if_then_elseXX`$(pv(V),$`T)`
where
        `if_then_elseXX`$(pv(V),$`T)` `:-` $\llbracket b \rrbracket_V^1$, `!`, $\llbracket t_1 \rrbracket_V^T$.
        `if_then_elseXX`$(pv(V),$`T)` `:-` $\llbracket t_2 \rrbracket_V^T$.

$\llbracket forall(i$ in $l..u)(\mathsf{T}\{i\}) \rrbracket_V^B \implies$
        $\llbracket l \rrbracket_V^{Li}$, $\llbracket u \rrbracket_V^{Ui}$, `Ind_i is Ui-Li+1`, `forallXX(Ind_i,Ui,`$pv(V)$`,B)`.
where
        `forallXX(0,_Ui,_,1)`.
        `forallXX(Ind_i,Ui,`$pv(V)$`,B)` `:-`
          `Ind_i > 0,`
          `Index_i is Ui - Ind_i + 1,`
          $\llbracket \mathsf{T}\{i\} \rrbracket_{[(i,Index\_i)|V]}^{B1}$
          `NInd_i is Ind_i - 1,`
          `forallXX(NInd_i,Ui,`$pv(V)$`,B2),`
          `and(B1,B2,B)`.

$\llbracket forall(i$ in $l..u$ where $\mathsf{B}\{i\})(\mathsf{T}\{i\}) \rrbracket_V^1 \implies$
        $\llbracket l \rrbracket_V^{Li}, \llbracket u \rrbracket_V^{Ui}$, `Ind_i is Ui-Li+1`, `forallXX(Ind_i,Ui,`$pv(V)$`)`.
where
        `forallXX(0,_Ui,_)`.
        `forallXX(Ind_i,Ui,`$pv(V)$`)` `:-`   `forallXX_where(`$pv(V)$`)` `:-`
          `Ind_i > 0,` $\llbracket \mathsf{B}\{i\} \rrbracket_{[(i,Index\_i)|V]}^1$, `!`,
          `Index_i is Ui - Ind_i + 1,`
          `forallXX_where(`$pv(V)$`),` $\llbracket \mathsf{T}\{i\} \rrbracket_{[(i,Index\_i)|V]}^1$.
          `NInd_i is Ind_i - 1,`   `forallXX_where(_)`.
          `forallXX(NInd_i,Ui,`$pv(V)$`)`.
Note that the generalisation of the following rule to n indices instead of 2 is straightforward (we chose 2 just for clarity of exposition):
$\llbracket forall(i$ in $li..ui, j$ in $lj..uj)(\mathsf{T}\{i, j\}) \rrbracket_V^1 \implies$
        $\llbracket li \rrbracket_V^{Li}$, $\llbracket ui \rrbracket_V^{Ui}$, `Ind_i is Ui-Li+1`, `forallXX(Ind_i,Ui,`$pv(V)$`)`.
where

```
forallXX(0,_Ui,_).
forallXX(Ind_i,Ui,pv(V)) :-
    Ind_i > 0,
    Index_i is Ui - Ind_i + 1,
    [[forall(j in lj..uj)(T{i,j})]]¹...
    NInd_i is Ind_i - 1,
    forallXX(NInd_i,Ui,pv(V)).
```

$[\![type : var]\!]_V^D \Longrightarrow [\![type]\!]_V^T$, `D = [(var,T)]`

$[\![type : var = val]\!]_V^D \Longrightarrow [\![type]\!]_V^T$, $[\![val]\!]_V^A$, `T = A`, `D = [(var,T)]`

$[\![\texttt{int}]\!]_V^T \Longrightarrow$ `true`

$[\![\texttt{var } l..u]\!]_V^T \Longrightarrow [\![l]\!]_V^L$, $[\![u]\!]_V^U$, `T in L..U`

$[\![\texttt{ array}[l..u] \texttt{ of var } dl..du]\!]_V^A \Longrightarrow$
$\quad [\![l]\!]_V^L$, $[\![u]\!]_V^U$, $[\![dl]\!]_V^{DL}$, $[\![du]\!]_V^{DU}$, `initXX(L,U,DL,DU,A).`

$[\![a[t]]\!]_V^{AT} \Longrightarrow [\![t]\!]_V^T$, $[\![a]\!]_V^A$, `index(A,T,AT).`

$[\![\texttt{let } \{d,ds\} \texttt{ in e }]\!]_V^B \Longrightarrow [\![d]\!]_V^U$, `append(V,U,VU)`, $[\![\texttt{ let } \{ds\} \texttt{ in e }]\!]_{VU}^B$.

$[\![\texttt{let } \{\} \texttt{ in e }]\!]_V^B \Longrightarrow [\![e]\!]_V^B$.

$[\![\texttt{constraint } c_1 \wedge c_2]\!]_V \Longrightarrow [\![c_1]\!]_V^1$, $[\![\texttt{ constraint } c_2]\!]_V$.

$[\![\texttt{constraint }]\!]_V \Longrightarrow \emptyset$

$[\![t_1 \le t_2]\!]_V^1 \Longrightarrow [\![t_1]\!]_V^{T1}$, $[\![t_2]\!]_V^{T2}$, `leq(T1,T2).`

$[\![t_1 \le t_2]\!]_V^B \Longrightarrow [\![t_1]\!]_V^{T1}$, $[\![t_2]\!]_V^{T2}$, `reif_leq(T1,T2,B).`

$[\![\texttt{predicate p(\_:a1,...,\_:aN) = b }]\!]_V^1 \Longrightarrow$ `p(A1,...,AN)` `=` $[\![b]\!]_{[(a1,A1),...,(aN,AN)]}^1$.

The rules for translating arrays deserve special mention. The translation rule
for the indexing construct "$a[t]$" is implicitly recursive. A multidimensional
indexing of the form "v[i,cj,k]" will be transformed using that rule as if it was
"v[i,j][k]". Thus, indices i and j will be translated first recursively as part of
"a", and then index k as "t" in the rule. However, in a real implementation of
the translation, specialised versions are used for 2-dimensional, 3-dimensional,
... indexing (as illustrated in Figure 3). This leads to more compact code and
simplifies program analysis.

Also, the Prolog predicates `initXX/5` and `index/3` used in the translations of
MiniZinc constructs "$array$" and "$a[t]$", respectively, represent the particular
implementation of arrays used in Prolog. This can be typically done using lists
of lists or, alternatively, nested plain (non-recursive) structures. In the latter
case the predicates would look as follows:

```
index(array(L,U,Arr),T,AT) :-          initXX_1(0,_,_,_).
   Index is T − L + 1,                  initXX_1(Ind,DL,DU,Arr) :-
   arg(Index,Arr,AT).                      Ind > 0,
                                           arg(Ind,Arr,Ai),
initXX(L,U,DL,DU,array(L,U,Arr)):-        Ai in DL..DU,
   Size is U − L + 1,                     NInd is Ind − 1,
   functor(Arr,array,Size),               initXX_1(NInd,DL,DU,Arr).
   initXX_1(Size,DL,DU,Arr).
```

Note that we could have translated the MiniZinc if-then-else into the Prolog if-then-else. However, we have used the transformation of the later which is performed by CiaoPP prior to analysis, i.e., the one actually analysed, to easy the interpretation of analysis results.

The definitions of the predicates into which the MiniZinc basic constraints are translated into are provided in a Ciao module. This allows to supply different implementations (each of them in a module using a different package defining a particular solver) that can be selected at will. For example, there are definitions for the following predicates in a module that uses the Ciao "fd" package (as expressed by the assertion in the first line):

```
:- use_package(fd).

plus(X, Y, Z):- Z .=. X + Y.

leq(X, Y):- X .=<. Y.

in(E,DL,DU):- E in DL..DU.
```

Translation Example Each MiniZinc item constructed with the built-in `forall` is translated into one or more recursive predicates that simulate the iteration. For example, lines 15–17 in Figure 1 are translated into the Ciao predicate `forall2_j/6` (and its auxiliary predicates `forall2_jk/6` and `forall2_jk_where/5`) as shown in Figure 3.

A Ciao predicate is created for each index in the `forall` MiniZinc item. In our example, the `forall2_j/6` predicate simulates the iteration over index `j`, whose associated Ciao variable is `Index_j`. Note that this variable is updated by using the variables `Uj` (which stands for "upper limit" of `j`) and `Ind_j`, and increases in each iteration (since `Ind_j` decreases by 1 in each recursive call). The reason for creating a new index `Ind_j` is to allow the resource usage analysis to set up cost expressions for predicates that are actually difference equations (whose solutions are closed form resource usage functions). This will be clarified in Section 5.2.

The first argument of `forall2_j/6` (`Ind_j`) is initialised in the first call to `forall2_j/6` to the number of values in the range (`1..size`) of index `j` (i.e., to the value of `size`, the parameter declared in line 2 in Figure 1, which is also the upper limit of indices `j` and `k`). The second argument (`Uj`) is also initialised in the first call to `forall2_j/6` to the value of `size`, but remains unchanged through the recursion. The third argument (`Index_i`) corresponds to the `i` index in the `forall` item in line 12, Figure 1. The remaining arguments of the predicate (`Size`, `S` and `D`), correspond to the `size`, `s` and `d` parameters in Figure 1, respectively.

Similarly, predicate `forall2_jk/6` simulates the iteration over index `k`. It is called from predicate `forall2_j/6`, which initialises its parameters accordingly. In particular, variables `Ind_k` (the number of values in the range of index `k`) and `Uk` (the upper limit of index `k`) are both initialised to the value of `size`.

Predicate `forall2_jk_where/5` in Figure 3 corresponds to the "where" condition (line 15) and the body (line 16) of the `forall` item in lines 15–17, Figure 1. Line 16 is translated into calls to the predicates `element2d_int/4` and `element2d_dint/4` to access the elements of the 2D arrays `d` and `s` (which are integers and *decision variables*, respectively), and a call to predicate `no_overlap/4`. The latter predicate is the Ciao translation of the MiniZinc predicate with the same name/arity defined in lines 8–9 of Figure 1. Note that since the types of the variables are taken from the *variable declaration items*, it is possible to use specialised versions of Ciao predicates that access arrays (or other data structures) to help the static analysis (as we have done in Figure 3).

Finally, predicate `forall1_j/7` in Figure 3 corresponds to the `forall` item in line 13, Figure 1.

5 Resource Usage Analysis

In this section we introduce the CiaoPP general resource usage analysis framework and discuss how to instantiate it for the analysis of the Ciao programs resulting from the translation of MiniZinc Models. As mentioned before, CiaoPP includes a global static analyser which is parametric with respect to resources and type of approximation (lower- and upper-bounds) [7]. The user can define the parameters of the analysis for a particular resource by means of assertions that associate basic cost functions with elementary operations of programs, thus expressing how they affect the usage of a particular resource. The global static analysis can then infer bounds on the resource usage of all the procedures in the program, providing such usage bounds as functions of input data sizes. Examples of resources that can be analysed by instantiating the CiaoPP general framework are execution time, execution steps, memory, number of accesses to a database, etc.

5.1 Instantiating the General Framework

A *resource* in our approach is a quite general, user-defined (and possibly application-dependent) notion that associates a basic cost function with elementary operations in the base language and/or to some procedures in libraries.

Assume for example that we are interested in estimating upper bounds on the amount of the following three resources used by the MiniZinc model in Figure 1: (a) number of basic constraints set up, (b) number of decision variables created, and (c) number of calls to the `all_different` global constraint primitive.

Defining Resources We start by defining three identifiers ("counters") associated to each of the mentioned resources, through the following Ciao declarations:

```
:- resource constraints.
:- resource numvars.
:- resource alldiff.
```

These declarations are included in a Ciao *package* so that they can be used by other programs.

Expressing Resource Usages of Library (and External) Predicates

The resource usage of Ciao library predicates resulting from the translation of basic constraints of MiniZinc models is expressed using "trust" assertions (see Section 3). For example, the assertions for the Ciao `plus/3` predicate are:

```
:- trust pred plus(X,Y,Z): (dint(X),dint(Y),dint(Z))
                        => (dint(X),dint(Y),dint(Z))
                         + (cost(ub,constraints,1),
                            cost(ub,numvars,0)).

:- trust pred plus(X,Y,Z): (dint(X),int(Y),var(Z))
                        => (dint(X),int(Y),dint(Z))
                         + (cost(ub,constraints,1),
                            cost(ub,numvars,1)).
```

The precondition of the second assertion (`(dint(X),dint(Y),dint(Z))` expresses that `X` is a *decision variable*, `Y` is bound to an integer, and `Z` is unbound. Note that the resource usage of the call `plus(S1,D1,A)` in the `no_overlap/4` predicate in Figure 3 is obtained from such "trust" assertions. Thus, if `S1` is a solver variable, `D1` is an integer constant, and `A` a new (Prolog) variable (corresponding to the mode of usage given by the second assertion above), the number of constraints set up by this call is 1 (as well as the number of decision variables created).

Assertions are also used to express information that is instrumental in the resource usage analysis, such as determinism. For example, assertion:

```
:- trust comp plus/3 + is_det.
```

indicates that the `plus/3` predicate is deterministic, i.e., it does not create choices. There are similar assertions for other Ciao library predicates, such as:

```
:- trust pred reif_leq(X, Y, R): (dint(X),int(Y),var(R))
                             => (dint(X),int(Y),dint(R))
                              + (cost(ub,constraints,1),
                                 cost(ub,numvars,1)).

:- trust pred and(X, Y, R): (dint(X),dint(Y),dint(R))
                        => (dint(X),dint(Y),dint(R))
                         +  cost(ub,constraints,1).
```

Expressing Default Resource Usages

There are also assertions to express default values to be taken by the analysis for the resource usage of predicates. For example, the assertion:

```
:- trust_default + cost(ub,constraints,0).
```

states that when there is no "trust" assertion available expressing an upper bound (`ub`) on the usage of the resource named `constraints` for any (library or external) predicate, then the (default) value taken by the analysis should be 0. In our implementation, there are similar assertions for the resources `numvars` and `alldiff`.

Expressing Resource Usages of Basic Program Constructs

The `head_cost`(*Approx*, *Res_name*, Δ^H) declarations are used to describe how predicates in general *update* the value for those resources that are applicable

to predicate heads (such as counting the number of arguments passed or total resolution steps). While *Approx* and *Res_name* are as before, $\Delta^H : cl\_head \rightarrow arith\_expr$ is a function that takes a clause head and returns an arithmetic resource usage expression. This function is provided by means of a user-defined (or imported) predicate, written in the source language, which will be called by the analyser when the clause head is analysed. This code gets loaded into the compiler in a similar way to, e.g., macro expansion code.

The `literal_cost`(*Approx*, *Res_name*, Δ^L) declarations describe how predicate bodies *update* the value of resources that are applicable to body literals (such as the number of unifications). In this case, $\Delta^L : body\_lit \rightarrow arith\_expr$ is also user- (or library-)provided code which will be executed when the body literals of different predicates are analysed. We use the notation $\delta(Approx, Res\_name)$ and $\beta(Approx, Res\_name)$ for referring to the functions Δ^H and Δ^L, respectively. In our implementation, both Δ^L and Δ^H are constant functions that return zero for the three resources under consideration. This is expressed, for example, with the following assertions for the resource **constraints**:

```
:- head_cost(ub, constraints ,0).
:- literal_cost(ub, constraints ,0).
```

5.2 Performing the Analysis

Once the parameters of the general resource analysis framework have been defined and assertions for library predicates have been provided, the CiaoPP global static analysis can infer the resource usage of all the procedures in the program (as functions of input data sizes). A full description of how this is done can be found in [7]. In the following we sketch the main steps of the approach assuming, for simplicity, that we are only interested in upper bound estimations.

Mode Analysis determines for each argument in each predicate to be analysed whether the argument acts as an input or an output argument. This is done using the (very accurate) information from the CiaoPP "sharing + freeness" (*shfr*) abstract domain. Accuracy is critical here to set up meaningful and solvable recurrence expressions for resource usages (as we will see later).

Size Measure Analysis: CiaoPP currently uses some predefined measures for the size of an input, such as the actual value of an integer (*int*), the number of constant and function symbols of a term (*size*), the length of a list (*length*), or the depth of the tree representation of a term (*depth*). These are automatically assigned to each argument of the predicates involved in the analysis according to their type (in turn inferred using the *eterms* abstract domain). A new, experimental version of the size analysers is in development to deal with user-defined size metrics (i.e., predicates) and to synthesise automatically size measures.

Size Analysis determines the relative sizes of variable bindings at different program points. A directed acyclic graph for each clause representing data dependencies between argument positions (*argument dependency graph*) is built and used for this purpose. The size analysis (as well as the resource usage analysis) is performed for each strongly-connected component of the call graph of the program in reverse topological order. For each clause, size relations are propagated to transform each size relation corresponding to an input position in a

body literal into a function over the size of the input arguments of the clause head. However, for recursive clauses, we first need to solve the symbolic expression due to recursive literals into an explicit function. Thus, in general the size analysis sets up difference equations representing the size of each output argument as a function of the input argument sizes, and computes bounds to the solutions of these equations. The result is a closed form function for each output argument which provides (an upper bound on) its size as a function of the sizes of the input arguments.

Resource Usage Analysis uses the size information inferred by the size analysis to set up difference equations representing the resource usage of predicates, and computes bounds to their solutions. As a result, we obtain a closed form function for each predicate giving (an upper bound on) its resource usage as a function of its input argument sizes.

Let $\text{RU}(p, ap, r, \overline{n})$ denote the units of resource r used during the computation of predicate p for a vector of input argument sizes $\overline{n}$ with approximation ap. Let us illustrate the analysis by means of an example. Assume that the (upper-bound) resource usage analysis has been performed for predicates `no_overlap/4` and `forall2_jk_where/5` in Figure 3, inferring that:

$$\text{RU}(\text{no\_overlap}, \text{ub}, \text{constraints}, \overline{n}) = 5 \text{ and}$$
$$\text{RU}(\text{forall2\_jk\_where}, \text{ub}, \text{constraints}, \overline{n}) = 5.$$

Such information is expressed via the following assertions, which are part of the analysis output:

```
:- true pred no_overlap(S1,D1,S2,D2)
   : (dint(S1),int(D1),dint(S2),int(D2))
   + (cost(ub,alldiff,0), cost(ub,constraints,5),
      cost(ub,numvars,4)).

:- true pred forall2_jk_where(Index_i,Index_j,Index_k,S,D)
   : (int(Index_i),int(Index_j),int(Index_k),
      arracy_2d_int(S),array_2d_int(D))
   + (cost(ub,alldiff,0), cost(ub,constraints,5),
      cost(ub,numvars,4)).
```

Consider now the predicate `forall2_jk/6` in Figure 3. For simplicity we will focus only on its first argument, since the others are not relevant for the resource usage estimation. The first argument has been inferred to be input and of integer type by previous analyses and, thus, is has been assigned the `int` (integer value) size measure.

The resource usage analysis of a clause uses the size relations (previously inferred) that express the size of input positions in body literals as a function of the size of the input arguments of the clause head. Consider the recursive clause of predicate `forall2_jk/6`. Let n represent the size of the first argument (`Ind_k`) in the head (under the size measure `int`). The size analysis infers that the size of the first (input) argument to the recursive (i.e., last) call in the body clause equals the size of the first head argument minus one: $size_{int}(\text{NInd\_k}) = size_{int}(\text{Ind\_k}) - 1 = n - 1$. Using this relation, an upper bound on the resource usage of that clause is expressed as a function of the size

of the first head argument, in terms of the resource usage of its body literals:

$\mathrm{RU}(\texttt{forall2\_jk}, \texttt{ub}, \texttt{constraints}, n) =$
$\delta(ub, \texttt{constraints})(\texttt{forall2\_jk}) + \beta(\texttt{ub}, \texttt{constraints})(\texttt{>}) + \mathrm{RU}(\texttt{>}, \texttt{ub}, \texttt{constraints}, \_) +$
$\beta(\texttt{ub}, \texttt{constraints})(\texttt{is}) + \mathrm{RU}(\texttt{is}, \texttt{ub}, \texttt{constraints}, \_) +$
$\beta(\texttt{ub}, \texttt{constraints})(\texttt{forall2\_jk\_where}) + \mathrm{RU}(\texttt{forall2\_jk\_where}, \texttt{ub}, \texttt{constraints}, \_) +$
$\beta(\texttt{ub}, \texttt{constraints})(\texttt{is}) + \mathrm{RU}(\texttt{is}, \texttt{ub}, \texttt{constraints}, \_) +$
$\beta(\texttt{ub}, \texttt{constraints})(\texttt{forall2\_jk}) + \mathrm{RU}(\texttt{forall2\_jk}, \texttt{ub}, \texttt{constraints}, n - 1)$

In Section 5.1, both functions $\delta(ub, \texttt{constraints})$ and $\beta(\texttt{ub}, \texttt{constraints})$ have been defined as zero constant functions using assertions of type "head_cost" and "literal_cost", respectively. Therefore, we have that:

$\mathrm{RU}(\texttt{forall2\_jk}, \texttt{ub}, \texttt{constraints}, n) =$
$0 + 0 + 0 + 0 + 0 + 0 + 5 + 0 + 0 + 0 + \mathrm{RU}(\texttt{forall2\_jk}, \texttt{ub}, \texttt{constraints}, n - 1) =$
$5 + \mathrm{RU}(\texttt{forall2\_jk}, \texttt{ub}, \texttt{constraints}, n - 1)$

For the non-recursive clause of **forall2_jk/6** the analysis infers:

$$\mathrm{RU}(\texttt{forall2\_jk}, \texttt{ub}, \texttt{constraints}, 0) = 0$$

which can be used as boundary condition for solving the previous difference equation, yielding the following closed form resource usage function:

$$\mathrm{RU}(\texttt{forall2\_jk}, \texttt{ub}, \texttt{constraints}, n) = 5\,n$$

The analysis proceeds for each strongly-connected component of the call graph of the program in reverse topological order. For predicate **forall2_j/6** (which corresponds to lines 15–17 in Figure 1) the analysis output includes the following assertion:

```
:- true pred forall2_j(Ind_j,Uj,Index_i,Size,S,D)
        : (int(Ind_j), int(Uj), int(Index_i), int(Size),
           array_2d_int(S), array_2d_dint(D))
        + (cost(ub,alldiff,0),
           cost(ub,constraints,5*(int(Size)*int(Ind_j))),
           cost(ub,numvars,4*(int(Size)*int(Ind_j)))).
```

Since both **Ind_j** and **Size** are given the value of the MiniZinc parameter **size** at the first call to predicate **forall2_j/6**, an upper bound on the resource usage (given in number of constraints set up) associated with lines 15–17 in Figure 1 is $5 \times \texttt{size}^2$ (and the number of decision variables created $4 \times \texttt{size}^2$).

Similarly, the resource usages associated with the **forall** item in line 13, Figure 1 are $2 \times \texttt{size}$ and **size** for **constraints** and **numvars**, respectively. For the whole constraint defined in lines 11–18, Figure 1 the resource usages are $5 \times \texttt{size}^3 + 2 \times \texttt{size}^2$ (for **constraints**), and $4 \times \texttt{size}^3 + \texttt{size}^2$ (for **numvars**).

Similarly, the resource usages inferred for the **forall1_j/7** predicate in Figure 3 in terms of **constraints** and **numvars** are $2 \times \texttt{int(Ind\_j)}$ and $\texttt{int(Ind\_j)}$, respectively. Since this predicate corresponds to the **forall** item in line 13, Figure 1, and its parameter **Ind_j** is given the value $\texttt{size} - 1$ (where **size** is a parameter of the MiniZinc model) at the first call to it in the Ciao program, we have that the resource usages corresponding to the **forall** item in line 13, Figure 1 are $2 \times \texttt{size} - 2$ and $\texttt{size} - 1$, respectively.

6 Experimental Results

In order to show the feasibility of our approach we have manually translated some MiniZinc models into Ciao programs, and analysed these programs with

Program	Size Measure	Resource Usage Function				Time (ms)
		`Ap.`	`constraints`	`numvars`	`alldiff`	
`langford`	int,int	E	$\lambda x,y.3x^2y^2 + xy - x$	$\lambda x,y.3x^2y^2 + 3xy - x$	2	1,440
`knapsack`	int,int	E	$\lambda x,y.xy$	$\lambda x,y.xy + x$	0	760
`photo`	int,int	E	$\lambda x,y.5y + 1$	$\lambda x,y.x + 6y$	1	840
`queen`	int	E	$\lambda x.3.5x(x-1)$	$\lambda x.2x^2 - x$	0	380
`jobshop`	int	U	$\lambda x.5x^3 + 2x^2$	$\lambda x.4x^3 + 2x^2$	0	1,280
		L	$\lambda x.2x^2$	$\lambda x.2x^2$	0	1,190
		E	$\lambda x.2.5x^3 - 0.5x^2$	$\lambda x.2x^3$	0	—
`oss`	int,int	U	$\lambda x,y.5xy(x+y) + 2xy$	$\lambda x,y.3xy(x+y) + xy + 1$	0	1,690
		L	$\lambda x,y.2xy$	$\lambda x,y.xy + 1$	0	1,470
		E	$\lambda x,y.2.5xy(x+y) - 3xy$	$\lambda x,y.1.5xy(x+y) - 2xy + 1$	0	—
`sudoku`	int	U	$\lambda x.2x^6 - x^4$	$\lambda x.x^4$	0	1,450
		L	$\lambda x.x^6 - x^4$	$\lambda x.x^4$	0	1,340
		E	$\lambda x.2x^6 - 2x^5$	$\lambda x.x^4$	0	—

Table 1. Accuracy and efficiency in milliseconds of the analysis.

Program	constraints		numvars	
	U	L	U	L
jobshop	$\lambda x.2 + 4/(5x-1) \to 2$	$\lambda x.4/(5x-1) \to 0$	$\lambda x.2 + 1/x \to 2$	$\lambda x.1/x \to 0$
oss	$\lambda xy.2 + \frac{16}{5x+5y-6} \to 2$	$\lambda xy.\frac{4}{5x+5y-6} \to 0$	$\lambda xy.2 + \frac{10-2/xy}{3x+3y-4+\frac{2}{xy}} \to 2$	$\lambda xy.\frac{2+2/xy}{3x+3y-4+\frac{2}{xy}} \to 0$
sudoku	$\lambda x.1 + \frac{1}{2x-2} + \frac{1}{2x} \to 1$	$\lambda x.\frac{1}{2} + \frac{1}{2x} \to \frac{1}{2}$	1	1

Table 2. Estimated bound/exact cost ratio and limit when data sizes tend to ∞.

CiaoPP. The programs include a "package" (in Ciao terminology) that we have implemented and which contains all the assertions needed to instantiate the CiaoPP general resource analysis framework for our purposes (See Section 5.1).

The first column of Table 1 shows the programs analysed. The column **Resource Usage Function** shows resource usage functions, given as lambda terms, for the resources `constraints`, `numvars` and `alldiff` (as described in Section 5.1), according to the approximation expressed by column **Ap.**. Such functions depends on the size of (some of) the parameters of the MiniZinc model, or, equivalently, on the size of (some of) the input arguments to the Ciao program that the model is translated into. The column **Size Measure** shows the size measure used for such input arguments (the relevant arguments). The column **Ap.** is used to express the upper- and lower-bounds inferred by the analysis (U and L respectively), and the actual (exact) resource usage functions (E). For example, the upper-bound resource usage functions for the `jobshop` model (the one used as a running example in this paper) depend on the size (integer value) of the parameter `size`. Note that only the value E is shown in the column **Ap.** for the first four programs. This is because the upper- and lower-bounds inferred by the analysis for such programs are the same, and thus both bounds are equal to the actual (exact) resource usage function. Finally, the column labeled **Time** shows the resource analysis times in milliseconds, taken when run on an **Intel Core i7**, 4 cores x 2.67GHz (2 threads per core), 12GB of RAM, running Ubuntu Linux 10.10 (kernel 2.6.35).

Table 2 shows the deviation of the resource usage bound functions estimated by the analysis w.r.t. the actual functions. Such deviation is given as the estimated bound/exact function ratio and its limit when data sizes tend to infinity.

The preliminary results are encouraging, showing that the analysis efficiently infers the actual resource usage functions for more than half of the programs and reasonable upper- and lower-bounds for the rest.

References

1. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theoretical Computer Science*, 318(1-2), 2004.
2. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
3. J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic Identification of Timing Anomalies for Cycle-Accurate Worst-Case Execution Time Analysis. In *Proc. of DDECS'06*, pages 15–20. IEEE Computer Society, 2006.
4. G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 75–88. ACM Press, 2002.
5. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, 2005.
6. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 2011. http://arxiv.org/abs/1102.5497.
7. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.
8. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.
9. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.
10. Reinhard Wilhelm. Timing Analysis and Timing Predictability. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium (FMCO)*, volume 3657 of *LNCS, Revised Lectures*, pages 317–323. Springer, 2004.