Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 258 (2009) 79–91

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

[illegible]
Cooperation of Constraint Domains

S. Estévez-Martín[†], A. J. Fernández Leiva[∓], and F. Sáenz-Pérez[‡][1]

[†]*Dept. Sistemas Informáticos y Programación,* [‡]*Dept. Ingeniería del Software e Inteligencia Artificial*
Universidad Complutense de Madrid, Spain
[∓]*Dept. Lenguajes y Ciencias de la Computación*
Universidad de Málaga, Spain
`s.estevez@fdi.ucm.es, afdez@lcc.uma.es, fernan@sip.ucm.es`

Abstract

This paper presents, from a user point-of-view, the mechanism of cooperation between constraint domains that is currently part of the system $\mathcal{TOY}$, an implementation of a constraint functional logic programming scheme. This implementation follows a cooperative goal solving calculus based on lazy narrowing. It manages the invocation of solvers for each domain, and projection operations for converting constraints into mate domains via mediatorial constraints. We implemented the cooperation among Herbrand, real arithmetic ($\mathcal{R}$), finite domain (FD) and set ($\mathcal{S}$) domains. We provide two mediatorial constraints: The first one relates the numeric domains $\mathcal{FD}$ and $\mathcal{R}$, and the second one relates $\mathcal{FD}$ and $\mathcal{S}$.

Keywords: Tools, Multiparadigm Programming, Constraint Functional Logic Programming, Domain Cooperation.

1 Introduction

$\mathcal{TOY}$ [1] is a multiparadigm programming language and system designed to support the main declarative programming styles and their combination. One

[1] First and third authors partially supported by projects TIN2008-06622-C03-01, S-0505/TIC/0407 and UCM-BSCH-GR58/08-910502. Second author partially supported by Spanish MICINN under contract TIN2008-05941 (NEMESIS project).

of its characteristics is that it provides support for functional logic programming, and programs in $\mathcal{TOY}$ can include definitions of types, operators, lazy functions in Haskell style, as well as definitions of predicates in Prolog style. A predicate is viewed as a particular kind of function whose right-hand side is true. A function definition consists of an optional *type declaration* and one or more *defining rules*, which are possibly conditional rewrite rules. Both functions and predicates must be well-typed with respect to a polymorphic type system [4].

With the aim of increasing the efficiency of goal solving, $\mathcal{TOY}$ also provides capabilities for constraint programming, and programs can use constraints within the definitions of both predicates and functions. Constraints are integrated as functions to make them first-class citizens what means that they can be used in any place where a data can (e.g., as arguments of functions). This provides a powerful mechanism to define higher order constraints. The constraints that have been integrated and supported by the system in recent years include symbolic equations and disequations [2], linear and non-linear arithmetic constraints over the real numbers [12], and finite domain constraints [9]. Now, $\mathcal{TOY}$ also incorporates a solver to manage set constraints [6].

It is well-known that constraint solving defined on specific domains (e.g., reals, finite domain, sets, etc.) can help to speed up the goal resolution, and also opens up the range of problems that a system can efficiently attack. However, it also presents evident drawbacks as, in practice, constraints are often not specific to any given domain, and thus the formulation of real problems has to be artificially adapted to a domain that is supported by the system. Many problems are more naturally expressed using heterogeneous constraints, involving more than one domain. Precisely, this problem can be smoothed via solver cooperation and $\mathcal{TOY}$ has recently incorporated a mechanism to support it [5,7]. A detailed description of the mechanism involving three specific domains, namely *Herbrand*, *real*, and *finite* domains, is given in [8].

This document belongs to this collection of *papers-illustrating-$\mathcal{TOY}$-features*. However, the reader should note that, even though many features of $\mathcal{TOY}$ have already been reported in a number of papers, this paper presents original material and highlights some of the recent acquired capacities of $\mathcal{TOY}$ by means of examples. In particular, the paper focuses in the solver cooperation mechanism including the cooperation between the finite domain and set solvers that has recently been integrated into $\mathcal{TOY}$ [6]. Besides illustrating the new features of the $\mathcal{TOY}$ system, an additional goal of this paper is to show that this system constitutes an adequate framework on which experimentation with solver collaboration can be carried out.

1.1 $\mathcal{TOY}$ Distribution

From `http://toy.sourceforge.net` the preferred distribution for $\mathcal{TOY}$ can be downloaded. There are some possibilities: Choose either a binary distribution (a portable application that does not need installation) or a source-code distribution (which requires SICStus Prolog previously installed). Therefore, almost any platform can run $\mathcal{TOY}$ (e.g., the system can be started as a Windows application or in a Linux console). It features a command interpreter for submitting goals and system commands. In addition, it has been connected to ACIDE [15], a graphical and configurable integrated development environment.

1.2 An Overview of $\mathcal{TOY}$

$\mathcal{TOY}$ computations solve goals and display computed answers. $\mathcal{TOY}$ solves goals by means of a demand driven lazy narrowing strategy [13] combined with constraint solving. Answer constraints can represent bindings for logic variables, as in answers computed by a Prolog system. Some features of $\mathcal{TOY}$ are:

(i) *Curried Style.* This allows that partial applications of curried functions can be used to express functional values as partial patterns.

(ii) *Non-deterministic Functions.* These are introduced either by means of defining rules with overlapping left-hand sides or using extra variables in the right-hand side that do not occur in the left-hand side.

(iii) *Sharing* for values of all variables which occur in the left-hand sides of defining rules and have multiple occurrences in the right-hand side and/or the conditions. Sharing implements the so-called *call-time choice* semantics of non-deterministic functions.

(iv) *Higher-Order Functions* in the style of Haskell, except that lambda abstractions are not allowed. In $\mathcal{TOY}$, higher-order can be naturally combined with non-determinism.

(v) *Dynamic Cut.* Optimization that detects deterministic functions at compile time, and the generated code includes a test for detecting at run-time the computations that can actually be pruned [3].

(vi) *Finite Failure.* The primitive Boolean function *fails* is a direct counterpart to finite failure in Prolog.

2 A Constraint Functional Logic Programming Scheme

$\mathcal{TOY}$ implements a Constraint Functional Logic Programming scheme $CFLP(D)$ over a parametrically given constraint domain D, proposed in [14]. $CFLP(D)$

is a logical and semantic framework for lazy Constraint Functional Logic Programming over D, which provides a clean and rigorous declarative semantics for $CFLP$ languages.

In particular, D is the *coordination domain* C introduced in [7] as the amalgamated sums of the domains to be coordinated, $D_1, \ldots, D_n$, along with a *mediatorial domain* M which supplies special communication constraints, called *bridges*, used to impose the equivalence between values of different base types.

The Cooperative Constrained Lazy Narrowing Calculus $CCLNC(C)$ presented in [7] provides a fully sound formal framework for functional logic programming with cooperating solvers over various constraint domains. $CCLNC(C)$ has been proved fully sound w.r.t. the $CRWL(C)$ semantics [14].

3 Cooperation in $\mathcal{TOY}$: Bridges and Projections

The current downloadable version of $\mathcal{TOY}$ (see Section 1.1) comes equipped with solvers corresponding to three constraint domains:

(i) *Herbrand*, with equality and disequality constraints.

(ii) *Real Arithmetic*, with arithmetic constraints over real numbers.

(iii) *Finite domain*, with constraints over integer numbers.

The Herbrand Solver is always available, and the real and finite domain solvers can be optionally loaded. A beta version of $\mathcal{TOY}$ (available soon) now also includes a solver to handle set constraints that allows constraint solving on intervals of sets of integers. The set constraint domain has been implemented in the beta version which has not been yet released.

With the aim of extending the system applicability, a mechanism for solver cooperation on these domains has been recently incorporated. This mechanism has two main pillars: *bridges*, necessary for solver communication, and *projection*, that improves the efficiency of some programs.

A bridge is a special kind of 'hybrid' constraint which allows the communication between two constraint domains and instantiates a variable occurring at one end of a bridge whenever the other end becomes ground. The next examples show this communication between $\mathcal{FD}$ and $\mathcal{R}$.

Example 3.1 In the cooperation *finite domain-real domain*, a bridge constraint (identified by the function #== /2; see the code below) can be used to impose an integral constraint over its right (real) argument. As an example, suppose we want to know whether two different lines can meet at one integer point. A line can be described algebraically by the linear equation `y = m * x + b`, and the corresponding $\mathcal{TOY}$ program and goal are as follows, where the symbol `<==` starts the conditional guard, `==` represents the equality constraint, and `->` stands for a substitution.

Program
`meetLines M1 B1 M2 B2`
`== (X,Y)`
`<== X #== RX,`
`Y #== RY,`
`RY == M1*RX + B1,`
`RY == M2*RX + B2`

Goals	Solutions
`meetLines 2 4 1 2 == L`	`L->(-2, 0)`
`meetLines 1 1 1 2 == L`	`no` (parallel lines)
`meetLines 1 1 3 2 == L`	`no` (real point)

□

Projection takes place during goal solving whenever a constraint is submitted to its solver. At that moment, projection builds a mate constraint which is submitted to the mate solver (think, for instance, of a finite domain solver as the mate of a real solver, and vice versa). Projection rules described in [5,7] relying on the available bridges are used for building mate constraints between the finite and real domains. The next example shows how projection builds and posts new mate constraints.

Example 3.2 Suppose we want to calculate the intersection of a triangular region (defined in the continuous plane) with an (N×N)-size square discrete grid (defined in the discrete plane). A $\mathcal{TOY}$ goal that solves the problem, for any given even integer number `N`, is shown below; the triangular region is described by the inequalities in the real domain whereas the square grid is described by the finite domain constraints (i.e., those labeled with `#` and the function `labeling/2`).

Goals		Mate Constraints	Solutions
`X#==RX, Y#==RY`			`X->2000, RX->2000`
`RY >= (4000/2)-0.5,`	⇒	`Y #>= ⌈4000/2-0.5⌉,`	`Y->2000, RY->2000`
`RY-RX <= 0.5,`	⇒	`Y #- X #<= ⌊0.5⌋,`	
`RY+RX <= 4000+0.5,`	⇒	`Y #+ X#<=⌊4000+0.5⌋,`	
`domain [X,Y] 0 4000,`	⇒	`0<=RX, RX<=4000,`	
`labeling [] [X,Y]`		`0<=RY, RY<=4000`	
`X#==RX, Y#==RY`			`X->1999, RX->1999`
`RY >= (4000/2)-1,`	⇒	`Y #>= ⌈4000/2-1⌉,`	`Y->1999, RY->1999`
`RY-RX <= 0.5,`	⇒	`Y #- X #<= ⌊0.5⌋,`	`X->2000, RX->2000`
`RY+RX <= 4000+0.5,`	⇒	`Y #+ X#<=⌊4000+0.5⌋,`	`Y->1999, RY->1999`
`domain [X,Y] 0 4000,`	⇒	`0<=RX, RX<=4000,`	`X->2000, RX->2000`
`labeling [] [X,Y]`		`0<=RY, RY<=4000`	`Y->2000, RY->2000`
			`X->2001, RX->2001`
			`Y->1999, RY->1999`

In this example, mate constraints generated during goal solving, allow the finite domain solver to drastically prune the domains of `X` and `Y`. Therefore, if we have a huge grid and a tiny triangle and the projection is enabled, then the computation time is notably reduced. Note that not all the constraints are projected; for example, the labeling constraint. □

The new bridge constraint for the cooperation *finite domain-set domain* is provided via the infix function: `#--`/2, where its left argument is an integer and the right one is a set, which follows the datatype declaration `data setOfInt = set [int]`. The next example shows its behaviour when projection has been both disabled and enabled.

Example 3.3 The following table shows a goal with projection disabled, where no mate constraints have been created. Here, `FDVar in Min..Max` stands for a domain constraint stating that the value of `FDVar` must be in the integer closed range `[Min,Max]`. Also, `SVar in Min..Max` is a domain constraint stating that `SVar` must contain, at least, the elements in the set `Min`, and, at most, the elements in the set `Max`.

Goal	Solutions
`F1 #--S1, F2 #--S2,`	`S1 in (set [2,4])..(set [2,3,4,5,6]),`
`domain [F1,F2] 1 1000,`	`S2 in (set [2,4])..(set [2,3,4,5]),`
`domainSet [S1] (set [2])`	`F1 in 1..1000,`
`(set [2,3,4,5,6]),`	`F2 in 1..1000`
`domainSet [S2] (set [2,4])`	
`(set [1,2,3,4,5,7,9]),`	
`subSet S2 S1`	

Next, enabling projection, we get a more constrained answer because of the projection due to the constraints `domainSet` and `subSet`. Here, it can be seen a finite domain interval constraint which is expressed by means of integer closed intervals and unions of integers (`1..5` represents the set containing integers from `1` to `5`, and ∨ is set union).

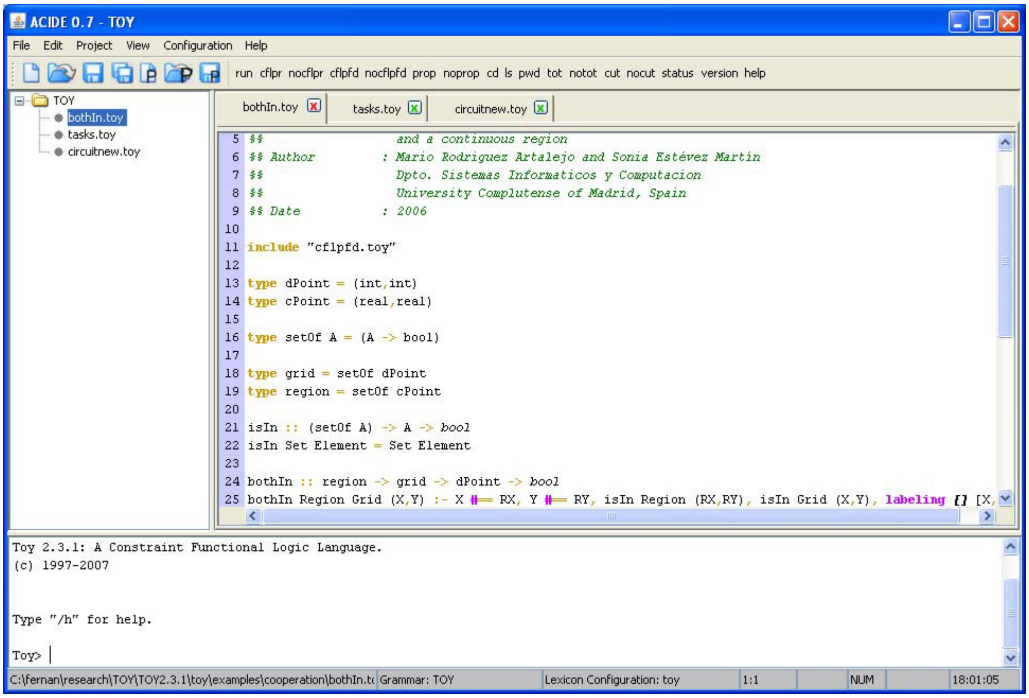
Goal		Mate Const.	Solutions
`F1 #--S1, F2 #--S2,`			`S1 in (set [2,4])`
`domain [F1,F2] 1 1000,`			`..(set [2,3,4,5,6]),`
`domainSet [S1] (set [2])`	⇒	`F1 in 2..6`	`S2 in (set [2,4])`
`(set [2,3,4,5,6]),`			`..(set [2,3,4,5]),`
`domainSet [S2] (set [2,4])`	⇒	`F2 in (1..5)`	`F1 in 2..6,`
`(set [1,2,3,4,5,7,9]),`		`∨{7}∨{9}`	`F2 in 2..5`
`subSet S2 S1`	⇒	`subset F2 F1`	

□

We have borrowed the idea of constraint projection from [11], adapting it to our *CFLP* scheme and adding bridge constraints as a novel technique which makes projections more flexible and compatible with the type discipline.

4 Getting Started with the $\mathcal{TOY}$ System

Whichever method you use to start $\mathcal{TOY}$ as described in the manual [1], you get a banner and a system prompt as displayed in the bottom panel of Figure 1.


ACIDE 0.7 - TOY
File Edit Project View Configuration Help
run cflpr nocflpr cflpfd nocflpfd prop noprop cd ls pwd tot notot cut nocut status version help
TOY
bothIn.toy
tasks.toy
circuitnew.toy
bothIn.toy
tasks.toy
circuitnew.toy

```
$$                   and a continuous region
$$ Author          : Mario Rodriguez Artalejo and Sonia Estévez Martín
$$                   Dpto. Sistemas Informaticos y Computacion
$$                   University Complutense of Madrid, Spain
$$ Date            : 2006

include "cflpfd.toy"

type dPoint = (int,int)
type cPoint = (real,real)

type setOf A = (A -> bool)

type grid = setOf dPoint
type region = setOf cPoint

isIn :: (setOf A) -> A -> bool
isIn Set Element = Set Element

bothIn :: region -> grid -> dPoint -> bool
bothIn Region Grid (X,Y) :- X #== RX, Y #== RY, isIn Region (RX,RY), isIn Grid (X,Y), labeling [] [X,
```

```
Toy 2.3.1: A Constraint Functional Logic Language.
(c) 1997-2007


Type "/h" for help.

Toy> 
```

C:\fernan\research\TOY\TOY2.3.1\toy\examples\cooperation\bothIn.t
Grammar: TOY
Lexicon Configuration: toy
1:1
NUM
18:01:05

Fig. 1. A Screenshot of Toy running into ACIDE.

This figure shows $\mathcal{TOY}$ running into ACIDE [15], a configurable IDE (Integrated Development Environment) consisting of three main panels. The left panel shows the organization of the current project, the MDI windows to the right are the opened files, which may belong to the project (files can be opened without assigning them to the project). Below, the $\mathcal{TOY}$ console panel is shown, which allows the user to interact by means of typed commands and expressions. Both shell and project panels can be hidden and, moreover, it is not mandatory to work with projects if they are not needed. The menu bar includes some common entries about files, edition, projects, views, configuration and help. In addition, there is a fixed toolbar which includes common buttons for file and project-related basic operations: New, Open, Save, and Save All (this last one only for files). Next to the fixed toolbar, there is the configurable toolbar, which in this case includes the most usual $\mathcal{TOY}$ commands.

The last line in the console panel (`Toy>`) is the $\mathcal{TOY}$ system prompt, which allows writing commands, executing goals, and computing expressions. The typical way of using the system is to write $\mathcal{TOY}$ program files (with default extension `.toy`) and consulting them before submitting goals. Following this, you write the program in a text file, and then you use the following command in order to compile and load the $\mathcal{TOY}$ program:

```
Toy> /run(Filename)
```

Where *Filename* is the name of the file, as `bothIn.toy` (the default extension `.toy` can be omitted). If the file is located in the distribution directory, you can also type:

```
Toy> /run(bothIn.toy)
```

Otherwise, when the file is located at another path, you can firstly change to the new path using the command `/cd(Path)`, where *Path* is the new directory (relative or absolute). However, things are much easier from the ACIDE environment since you can simply push the button run and get the file compiled and loaded. In addition, solvers can be activated by pushing the buttons cflpr, cflpfd, and cflpset.

5 Examples

The main part of the demonstration will be devoted to display examples of $\mathcal{TOY}$ programs to solve cooperation problems, as those described in the following.

5.1 *Scheduling Tasks Problem via Cooperation between solve$^{\mathcal{FD}}$ and solve$^{\mathcal{S}}$*

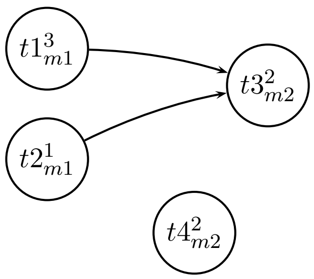


Fig. 2. Precedence Graph.

The tasks scheduling problem requires resources to complete, and consists of fulfilling precedence constraints. Figure 2 shows a precedence graph for four tasks which are labeled as tX^Y_{mZ}, where X stands for the identifier of a task t,

Y for its time to complete (duration), and Z for the identifier of a machine m (a resource needed to perform task tX). In this case, this problem is solved using the cooperation of $solve^{\mathcal{FD}}$ and $solve^{\mathcal{S}}$ with the program below. The constraint functions and operators that belong to the finite domain are: `sum`, `#=`, and `#<`, and set constraint functions are: `domainSet`, `cardinalSet`, and `intersectSet`. Bridges between finite domain variables and set variables are established by the function `#--/2` in such a way that a goal `F #-- S` projects constraints involving the variable `S` into constraints involving the variable `F`.

```
durationList :: [int]
durationList = [3,1,2,2]

% Auxiliary Functions
listFrom1To :: int -> [int]
listFrom1To X = take X (iterate (+ 1) 1)

% Main Function
scheduling :: [setOf int] -> [int] -> bool
scheduling TasksSet TasksFD = true <==
   TasksSet == [T1S, T2S, T3S, T4S],
   TasksFD == [T1FD, T2FD, T3FD, T4FD],

   % Bridges T1FD #-- T1S ... T4FD #-- T4S
   foldl and true (zipWith (#--) TasksFD TasksSet),

   % The time of execution of all tasks is, at most,
   % the sum of the durations of all the tasks,
   sum durationList (#=) Time,

   % The time of execution of every task can be placed in the time
   % interval defined from 1 to Time
   domainSet TasksSet (set []) (set (listFrom1To Time)),

   % The duration of a task corresponds to the cardinal of its set
   map cardinalSet TasksSet == durationList,

   % Precedences
   fd_max T1FD #< fd_min T3FD,
   fd_max T2FD #< fd_min T3FD,

   % Machine m1 can be assigned to a single task at a time
```

```
intersectSet T1S T2S (set []),

% Machine m2 can be assigned to a single task at a time
intersectSet T3S T4S (set [])
```

Some solutions to a goal for this problem are represented in Fig. 3, which corresponds to selected answers given at the system prompt `Toy(FD+R+S+p)`. In this prompt, `FD+R+S+p` indicates that $\mathcal{FD}$, $\mathcal{R}$, and $\mathcal{S}$ constraints libraries are loaded, and projection (`p`) has been enabled, respectively. The next interactive session excerpt corresponds to the solution of the left-upper part of this figure. Here, `T`i`S` are the $\mathcal{S}$ variables whilst `T`i`FD` are the $\mathcal{FD}$ variables.

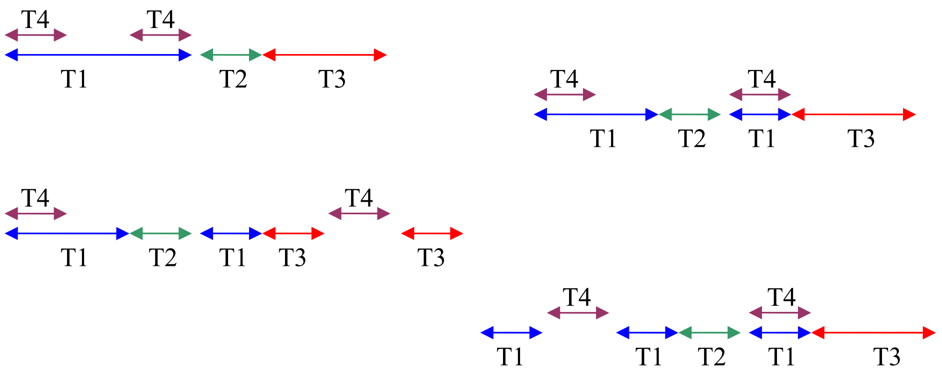


Fig. 3. Some Solutions of the Scheduling Problem.

```
Toy(FD+R+Set+p)> scheduling [T1S,T2S,T3S,T4S] [T1FD,T2FD,T3FD,T4FD]
     { T2S -> [ 4 ],
       T2FD -> 4 }
     { T1FD #-- T1S,
       T3FD #-- T3S,
       T4FD #-- T4S,
       T1S in (set [])..(set [1,2,3,4,5,6,7,8]),
       T3S in (set [])..(set [1,2,3,4,5,6,7,8]),
       T4S in (set [])..(set [1,2,3,4,5,6,7,8]),
       cardinalSet T1S 3,
       cardinalSet T3S 2,
       cardinalSet T4S 2,
       T3S in ([],close):*:(min T4S)..top,
       T4S in ([],close):*:(min T3S)..top,
       T1FD in 1..3,
       T3FD in 5..6,
       T4FD in {1}\/{3} }
```

This problem can be solved using only finite domain constraints [1], but solver cooperation leads to a more natural formulation.

5.2 Electrical Circuit Problem requiring the Cooperation between $solve^{\mathcal{FD}}$ and $solve^{\mathcal{R}}$

Consider also a problem taken from [10], in which one has an electric circuit with some connected resistors (i.e., real variables) and a set of capacitors (i.e., $\mathcal{FD}$ variables). The goal consists of knowing which capacitor has to be used so that the voltage reaches the 99% of the final voltage within a given time range. Particularly, we consider an instance of the problem (see Figure 4) with a resistor R1 of 0.1 $M\Omega$ connected in parallel with a variable resistor R2 of between 0.1 $M\Omega$ and 0.4 $M\Omega$, a capacitor K connected in series with the two resistors. Also, capacitors of $1\mu F$, $2.5\mu F$, $5\mu F$, $10\mu F$, $20\mu F$, and $50\mu F$ are available. The considered range time is [0.5,1], i.e., the duration until the capacitor is loaded is between 0.5 seconds and 1 second. Below we show a very simple $\mathcal{TOY}$ program (and a goal solved at the command line level) to solve this instance using distinct numerical solvers. Note that this problem cannot be solved by a unique solver and thus requires solver cooperation.

```
ecircuit :: int
ecircuit = KI <== R1 == 10000,
 10000 <= R2, R2 <= 40000,            % R Constraints
 R == R1*R2/(R1+R2),                  %
 50000.0 <= R, R <= 80000.0,          %
 T == -(ln 0.01)*R*K/10000000.0,      %
 0.5 <= T, T <= 1.0,                  %
 KI #== K,                            % FD-R Bridge
 belongs KI [10,25,50,100,200,500],   % FD constraints
 labeling [ ] [KI]                    %

Toy(R+FD)> ecircuit == L              % Goal solving
     { L -> 25 }
```

6 Conclusions and Further Work

This paper demonstrates, via examples, the potential of the cooperation mechanism available in the $\mathcal{TOY}$ system, a functional logic language that provides four constraint computation domains (i.e., Herbrand domain, real numbers, integers - the finite domain -, and sets of integers), and one domain (i.e., the mediatorial constraint domain), for communicating the computation domains.

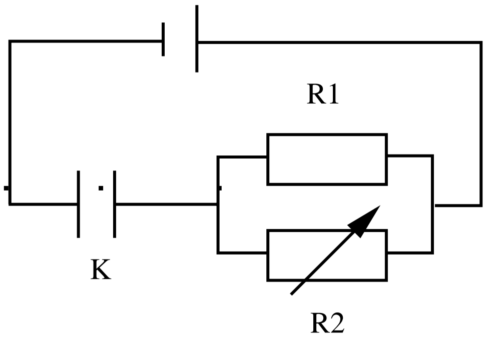


Fig. 4. Electrical Circuit Problem.

As a novelty, the paper has also illustrated the collaboration between the finite and set domains. Moreover, it should be clear from our exposition that $\mathcal{TOY}$ constitutes an appropriate setting to experimenting with solver collaboration.

As future work, we plan to optimize the set solver in $\mathcal{TOY}$ as well as formalize the cooperation between the Herbrand, finite domain and set domains following the same approach described in [8] for the Herbrand, real and finite domains.

References

[1] Arenas, P., A. Fernández, A. Gil, F. López, M. Rodríguez and F. Sáenz, *$\mathcal{TOY}$. A Multiparadigm Declarative Language. Version 2.3.0* (2007), R. Caballero and J. Sánchez (Eds.), Available at http://toy.sourceforge.net.

[2] Arenas, P., A. Gil and F. López, *Combining Lazy Narrowing with Disequality Constraints*, in: *PLILP'94*, LNCS **844** (1994), pp. 385–399.

[3] Caballero, R. and Y. García-Ruiz, *Implementing Dynamic Cut in Toy*, ENTCS **177** (2007), pp. 153–168.

[4] Damas, L. and R. Milner, *Principal Type-Schemes for Functional Programs*, in: *POPL'82* (1982), pp. 207–212.

[5] Estévez, S., A. Fernández, T. Hortalá, M. Rodríguez, F. Sáenz and R. del Vado, *A Proposal for the Cooperation of Solvers in Constraint Functional Logic Programming*, ENTCS **188** (2007), pp. 37–51.

[6] Estévez, S., A. Fernández and F. Sáenz, *Cooperation of the Finite Domain and Set Solvers in TOY*, in: P. Lucio, G. Moreno and R. Peña, editors, *IX Jornadas sobre Programación y Lenguajes (Prole'09)*, San Sebastían, Spain, 2009, pp. 217–226.

[7] Estévez, S., A. J. Fernández, M. T. Hortalá, M. Rodríguez and R. del Vado, *A Fully Sound Goal Solving Calculus for the Cooperation of Solvers in the CFLP Scheme*, ENTCS **177** (2007), pp. 235–252.

[8] Estévez-Martín, S., T. Hortalá-González, Rodríguez-Artalejo, R. del Vado-Vírseda, F. Sáenz-Pérez, and A. J. Fernández, *On the Cooperation of the Constraint Domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ in CFLP*, Theory and Practice of Logic Programming **9** (2009), pp. 415–527.

[9] Fernández, A. J., T. Hortalá, F. Sáenz and R. del Vado, *Constraint Functional Logic Programming over Finite Domains*, Theory and Practice of Logic Programming **7** (2007), pp. 537–582.

[10] Hofstedt, P., *Better Communication for Tighter Cooperation*, in: *CL'2000*, LNCS **1861** (2000), pp. 342–357.

[11] Hofstedt, P. and P. Pepper, *Integration of Declarative and Constraint Programming*, Theory and Practice of Logic Programming **7** (2007), pp. 93–121.

[12] Hortalá, T., F. López, J. Sánchez and E. Ullán, *Declarative Programming with Real Constraints*, Research Report SIP 5997, U.C.M. (1997).

[13] Loogen, R., F. López-Fraguas and M. Rodríguez-Artalejo, *A Demand Driven Computation Strategy for Lazy Narrowing*, in: *Proc. PLILP'93*, LNCS **714** (1993), pp. 184–200.

[14] López, F., M. Rodríguez and R. del Vado, *A New Generic Scheme for Functional Logic Programming with Constraints*, Higher-Order and Symbolic Computation **20** (2007), pp. 73–122.

[15] Sáenz-Pérez, F., *ACIDE: An Integrated Development Environment Configurable for LaTeX*, The PracTeX Journal **2007** (2007).