the underlying constraint system).

We plan to extend our tool in several ways. To improve the interface of the system we plan to construct a web interface. We plan to study both, how to carry out the implementation of the model-checking algorithm proposed in [6] for tcp programs, and how to adjust the Maude's model-checker to verify tcp programs. In this way we can establish a comparison which determines which approach is the most appropriate.

References


TOY: A System for Experimenting with Cooperation of Constraint Domains

S. Estévez-Martín1, A. J. Fernández2, and F. Sánchez-Pérez3

1 Dept. Sistemas Informáticos y Programación, Dpto Ingeniería del Software e Informática Añadida. Universidad Complutense de Madrid, Spain
2 Dept. Lenguajes y Sistemas de la Computación, Universidad de Málaga, Spain
3 esteves@dsi.uned.es, mdf@dsi.uned.es, fernandez@cs.uji.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 TOY, an implementation of a complete functional logic programming language. This implementation follows a cooperation model solving calculus based on lazy unification. It manages the invocation of solvers for each domain, and propagates updates for selecting constraints into more domains via communication channels. We implemented the cooperation among Hornlike, real arithmetic (R), finite domains (FD) and set (SD) domains. We provide two conditional constraints. The first one relates the integer domains FD and N, and the second one relates FD and S.

Keywords: Tools, Multiparadigm Programming, Constraint Functional Logic Programming, Prolog Cooperation

1 Introduction

TOY [1] is a multiparadigm programming language and system designed to support the main declarative programming styles and their combination. One of its characteristics is that it provides support for functional logic programming, and programs in 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 [2].

With the aim of increasing the efficiency of goal solving, TOY also provides capabilities for constraint programming, and programs can use constraints within the

1 First and third authors partially supported by projects TIN2008-05077, CYCLOPS, TIN2008-06027-C03-03, 546651/TIC/2007 and CIC-BSIC 039-04082. Second author partially supported by Spanish MICINN under contract TIN2008-05077 (NEMESIS project).

IX Jornadas sobre Programación y Lenguajes
I Taller de Programación Funcional
P. García, G. Moreno y R. Peña, Eds.
San Sebastián, 8-11 de septiembre de 2009
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 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 [9].

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

## 2 A Constraint Functional Logic Programming Scheme

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 sum of the domains to be coordinated, $D_1, \ldots, D_n$, along with a mediator 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 $CFLP(C)$ semantics [14].

## 3 Cooperation in TOY: Bridges and Projections

The current downloadable version of 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 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 in 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 example shows how communication between FID and K.

Example 3.1 In the cooperation finite domain-real domain, a bridge constraint (identified by the function \( \bar{f} \)) 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 \( y = a \cdot x + b \), and the corresponding \( FID \) program as follows, where the symbol \( => \) starts the conditional guard, \( \equiv \) represents the equality constraint, and \( \rightarrow \) stands for a substitution.

\[
\begin{align*}
\text{Program} & : \\
\text{metelesse } & M1 \; B1 \; M2 \; B2 \; \{X, Y\} \\
& <= \; X \equiv X_X, \; Y \equiv Y_Y, \; X \equiv M1 \cdot X1 + B1, \; Y \equiv M2 \cdot X + B2
\end{align*}
\]

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 [5,7,7] 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 \((3 \times 3)\)-size square discrete grid (defined in the discrete plane). A \( Toy \) goal that solves the problem, for any given even integer number \( S \), 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 \( \equiv \) and the function labeling\( /2 \)).
We have borrowed the idea of constraint projection from [11], adapting it to our \textit{CHIL} 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 \textit{TOY} System

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

![Figure 1: A Screenshot of Toy running into ACIDE](image)

This figure shows \textit{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 MIDI 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 \textit{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 \textit{TOY} commands.

The last line in the console panel (\texttt{Toy>}) is the \textit{TOY} system prompt, which allows writing commands, executing goals, and computing expressions. The typical way of using the system is to write \textit{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 \textit{TOY} program:

\texttt{Toy> /run(Filename)}

Where \texttt{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:

\texttt{Toy> /run(bothIn.toy)}

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

5 Examples

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

5.1 Scheduling Tasks Problem via Cooperation between solver$^T$D and solver$^S$

![Figure 2: Precedence Graph](image)

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 $\text{CY}^a_{X, Y}$, 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 solver$^T$D and solver$^S$ with the program below. The constrain functions and operators that belong to the finite domain are: \texttt{sum}, \texttt{star}, and \texttt{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 $F$ is projected onto $S$ projects constraints involving the variable $S$ into constraints involving the variable $F$.

durationlist :: [int]
durationlist <- [1, 2, 4, 1]

% Auxiliary Functions

listFromTo :: int -> [int]
listFromTo X = take X (iterate (+1) 1)

% Main Function

scheduling :: (setOf int) -> [int] -> bool

scheduling TaskSet TaskFD = true False

TaskSet <- [T1S, T2S, T3S, T4S]
TaskFD <- [T1FD, T2FD, T3FD, T4FD]

% Bridges T1FD <- T1S ... T4FD <- T4S
oldS and true (zipWith (--) TaskFD TaskSet),
% The time of execution of all tasks is, at most,
% the sum of the durations of all the tasks,
% plus durationlist (--) Time,
% The time of execution of every task can be placed in the time
% interval defined from 1 to Time
domainsSet TaskSet (set [1]) (set (listFromTo 1))
% The duration of a task corresponds to the cardinal of its set
% map domainsSet TaskSet durationlist,
% Precedences

fd max T1FD <= fd min T1FD,
fd max T2FD <= fd min T2FD,
fd max T3FD <= fd min T3FD,
fd max T4FD <= fd min T4FD,
% Machine m1 can be assigned to a single task at a time
% intersectSet T1S T2S (set [1])
% Machine m2 can be assigned to a single task at a time
% intersectSet T3S T4S (set [1])

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+RP+Sp). In this prompt, FD+RP+Sp indicates that FD, RP, and Sp constraints libraries are loaded, and the projection (p) has been enabled, respectively. The next interactive session excerpt corresponds to the solution of the left-upper part of this figure. Here, T2S are the $S$ variables whilst T1FD are the $FD$ variables.

![Fig. 3: Some Solutions of the Scheduling Problem](image)

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(FD) and solve(RP)

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., 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 $R$ of $0.1 \, \text{MO}$ connected in parallel with a variable resistor $R_2$ of between $0.1 \, \text{MO}$ and $0.1 \, \text{MO}$, a capacitor $C$ connected in series with the two resistors. Also, capacitors of $1 \mu F$, $2 \mu F$, $3 \mu F$, $4 \mu F$, $5 \mu F$, and $6 \mu F$ 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 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.

```
writeout = circuit <= K1 <= R1 <= 10000,
          10000 <= R2 <= 40000,
          K1 <= 5
          K1 <= 5 ) = true
RP
```

6 Conclusions and Further Work

This paper demonstrates, via examples, the potential of the cooperation mechanism available in the 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. 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 TOY constitutes an appropriate setting to experimenting with solver collaboration.

As future work, we plan to optimize the set solver in 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


NiMoToons: a Totally Graphic Workbench for Program Tuning and Experimentation

Silvia Clerici a,1, Cristina Zoltan a,2 and Guillermo Prestiagio a,3

a Dept. Lenguajes y Sistemas Informáticos
Universidad Politécnica de Cataluña
Barcelona, Spain

Abstract

NiMo (Nets in Motion) is a Graphical Programming Language designed to visualize algorithms and their execution in an understandable way. Programs are processed networks that evolve showing the full state of each execution step. Processes are polychromic, highly ordered and have multiple outputs. The language has a set of primitives for process scheduling and supports open processes and interactive debugging. The new version of the environment NiMoToons includes: an object-oriented and incremental type inference system, multiple output processes, use of higher order parameters, symbolic execution, live evaluation results that can be globally observed for each process and dynamically changed, and facilities to inspect the used resources (parallelism level, number of steps, number of processes, etc.).

Keywords: graphical language, functional, data flow, streams programming, parallelism, visual type inference, symbolic computation

1 Introduction

NiMo (Nets in Motion) [3,9] is a graphical programming language inspired on the Dataflow representation of some LISP programs first proposed by Turner [14], where functions are viewed as processes and channels are (usually infinite) lists. Its main objective is to provide the user a full control over its application development, debugging and optimization. The fact of being totally graphical is the key point because all the execution internals can be visible. The net is the code but also the computation state. Edition and execution are interleaved, and any partially defined net is an open program that can be executed and modified step by step. All together allow incremental development even during execution, because the initial code can evolve and be stored at any step as a program that can be recovered later. On the other hand, execution steps can be undone, acting as an on line tracer and...

1 Email: silvia@it.uc3m.es
2 Email: czoltan@upc.edu
3 Email: gprestiagio@upc.es

IX Jornadas sobre Programación y Lenguajes
1 Taller de Programación Funcional
P. Lucio, G. Moreno i R. Peña, Eds.
San Sebastian, 8-11 de septiembre de 2009
Programación y Lenguajes

IX Jornadas sobre Programación y Lenguajes, PROLE'09
I Taller de Programación Funcional, TPF'09

San Sebastián, España

del 8 al 11 de Septiembre de 2009

Editores:

PAQUI LUCIO, GINES MORENO Y RICARDO PENA
Comité de Programa de PROLE-2009

Presidente Comité: Ginés Moreno (U. de Castilla-La Mancha)
Elvira Albert (U. Complutense de Madrid)
Jesús Almendros (U. de Almería)
María Alpuente (U. Politécnica de Valencia)
Gilles Barthe (IMDEA)
Miquel Bertran (U. Ramón Llull)
Santiago Escobar (U. Politécnica de Valencia)
Antonio Fernández (U. de Málaga)
Victor Guías (U. de A Coruña)
Pauqui Lucio (U. del País Vasco)
Narciso Martí-Oliet (U. Complutense de Madrid)
Susana Muñoz (U. Politécnica de Madrid)
Marisa Navarro (U. del País Vasco)
Manuel Niñez (U. Complutense de Madrid)
Fernando Orellana (U. Politécnica de Catalunya)
Yolanda Ortega (U. Complutense de Madrid)
Francisco Ortín (U. de Oviedo)
Ernesto Pimentel (U. de Málaga)
German Puebla (U. Politécnica de Madrid)
Enric Rodríguez (U. Politécnica de Catalunya)
Jaime Sánchez (U. Complutense de Madrid)
German Vidal (U. Politécnica de Valencia)

Comité de Programa de TPF-2009

Presidente Comité: Ricardo Peña (U. Complutense de Madrid)
Víctor Guías (U. de A Coruña)
Francisco Gutiérrez (U. de Málaga)
Salvador Lucas (U. Politécnica de Valencia)
Pablo Nogueira (U. Politécnica de Madrid)
Julio Rubio (U. de la Rioja)
Alberto Verdejo (U. Complutense de Madrid)
Mateu Villarell (U. de Girona)
Comité Organizador de JISBD/PROLE-2009

Presidenta Comité: Gouria Sagardui (U. de Mondragón)
Vicepresidenta Comité: Paqui Lucio (U. del País Vasco)
Gontzane Aldekoa (U. de Mondragón)
Ana Altuna (U. de Mondragón)
Javier Alvez (U. del País Vasco)
Lorea Belategui (U. de Mondragón)
Leire Exeberría (U. de Mondragón)
Jose Gaintzatxain (U. del País Vasco)
Montserrat Hernu (U. del País Vasco)
Marisa Navarro (U. del País Vasco)
Xabier Sagaria (U. de Mondragón)

Comité Ejecutivo de PROLE-2009

Presidente Comité: Fernando Orejas (U. Politécnica de Cataluña)
Jesús Almendros (U. de Almería)
María Alpuente (U. Politécnica de Valencia)
Manuel Hermetegildo (U. Politécnica de Madrid)
Paqui Lucio (U. del País Vasco)
Juan José Moreno (U. Politécnica de Madrid)
Gines Moreno (U. de Castilla-La Mancha)
Ricardo Peña (U. Complutense de Madrid)
Ernesto Pimentel (U. de Málaga)

Revisores adicionales de PROLE/TPF-2009


Índice

Prólogo .................................................. IX
Charla Invitada ......................................... 1
K. RUSTAN M. LEINO
Understanding program verification ......................... 3
Taller de Programación Funcional .......................... 5
FRANCISCO-JESÚS MARTÍN-MATEOS, JOSÉ-LUIS RUIZ-REINA,
JULIO HUBO, LAUREANO LAMBOAL
Verificación y eficiencia en programación usando el cálculo simbólico: estudio de un caso ............................ 7
MARÍA ALPUENTE, MARCO A. FEHÉ, CHRISTOPHE JOUBERT,
ALICIA VILLANUEVA
Implementing Datalog in Maude .......................... 15
JOSÉ IBORRA
Explicitly Typed Exceptions for Haskell .................. 23
MANUEL MONTENEGRO, RICARDO PEÑA, CLARA SEGURA
Experiences in developing a compiler for Safe using Haskell .................................................. 34
HENRIQUE FERREIRO, DAVID CASTRO, VÍCTOR M. GULLÁS,
AYZE DUKRUSTA
Implementing memory reusing in the UHC Haskell compiler .................................................. 39
Tipos, Estructuras de Datos y Gestión de Memoria .......................... 47
FRANCISCO ORTÍN, DANIEL ZAPICO
Hacia un sistema de tipos estático y dinámico .......................... 49
JAVIER DE DIOS, RICARDO PEÑA, MANUEL MONTENEGRO
Certified Absence of Dangling Pointers in a Language with Explicit Deallocation ........................... 65
ELVIRA ALBERT, SAMIR GENAIM, MIGUEL GÓMEZ-ZAMALLOA
Live Heap Space Analysis for Languages with Garbage Collection ................................................ 75
JESÚS MANUEL ALMENDROS JIMÉNEZ
A Rule-based Implementation of XQuery ................................................ 77
Herramientas y Sistemas Software ....................... 87

MARISA LLORENS, JAVIER OLIVER, JOSEP SILVA, SALVADOR TAMARIT
An implementation of the MEB and CEB analyses for CSP ............ 89

PASCUAL JULIÁN-IRANZO, CLEMENTE RUBIO-MANZANO
UNICORN: A Programming Environment for Bonsi-Prolog .......... 99

ALEXEI LESCUYEL, ALICIA VILLANUEVA
The trcp Interpreter .................................. 109

SONIA ESTÉVEZ-MARTÍN, ANTONIO FERNÁNDEZ,
FERNANDO SÁENZ-PÉREZ
TOY: A System for Experimenting with Cooperation of Constraint
Domains ............................................. 119

SHAILA CERISE, CRISTINA ZOLOT, GUILLERMO PRESTIGIACOMO
NáMoIcons: a Totally Graphic Workbench for Program Tuning and
Experimentation ........................................ 129

ELVIRA ALBERTI, PURI ARENAS, SAMIR GENAIM, GERMÁN PEÑALZA,
DAMIÁN ZANARDINI, DIANA VANESSA RAMÍREZ DEANTES, MIGUEL
GÓMEZ-ZAMALOA, GUILLERMO ROMÁN-DÍEZ
Termination and Cost Analysis with COSTA and its User Interfaces .. 139

Razonamiento, Lógica y Semánticas.............................. 149

MIGUEL ACEA, JAVIER ÁLVAREZ, MONTSEHARRI HERMO,
EGOTZ LÁPARA
A New Proposal for Using First-Order Theorem Provers to Reason with
OWL-DL Ontologies .................................. 151

GABRIEL ARANDA LÓPEZ, SUSANA NIEVA, FERNANDO SÁENZ-PÉREZ,
JAVIE SÁNCHEZ-HERNÁNDEZ
Implementación de una semántica de punto fijo para un sistema de bases
de datos deductivas con restricciones ................................ 161

FRANCISCO JAVIER LÓPEZ-FRAGUAS, JUAN RODRÍGUEZ-HORTALÁ, JAIME
SÁNCHEZ-HERNÁNDEZ
A Fully Abstract Semantics for Constructor Systems .................. 177

JOHN LEVY, MATÉU VILLARET
Nominal Logic from a Higher-Order Perspective ......................... 179

José-Luis Ruiz-Reina, David A. Greve, Matt Kaufmann, Panagiotis
Manoliou, J. Moore, Sandip Ray, Rob Sumners, Damon
Vroon, Matthew Wilding
Efficient execution in an automated reasoning environment ............ 181

Programación Lógico Funcional y con Restricciones ............ 183

SONIA SANTIAGO, CAROLYN L. TALCOTT, SANTIAGO ESCOBAR,
CATHERINE MEADOWS, JOSÉ MESQUIDA
A Graphical User Interface for Manle-NPA ................................... 185

IGNACIO CASTÍNEAS, FERNANDO SÁENZ
Integración de ILOG CP en TOY ................................ 201

SONIA ESTÉVEZ-MARTÍN, ANTONIO FERNÁNDEZ,
FERNANDO SÁENZ-PÉREZ
Cooperación del Módulo de Sets y Solvers en TOY ..................... 217

FRANCISCO JAVIER LÓPEZ-FRAGUAS, ENRIQUE MARTÍN-MARTÍN,
JUAN RODRÍGUEZ-HORTALÁ
Advances in Type Systems for Functional Logic Programming .......... 227

Análisis de Terminación ........................................ 237

SALVADOR LUCAS
Automatic proofs of termination with elementary interpretations ... 239

BEATRIZ ALARCÓN, SALVADOR LUCAS, RAFAEL NAVARRO-MARSET
Using Matrix Interpretations over the Reals in Proofs of Termination . 255

RAÚL GUTIÉRREZ, SALVADOR LUCAS
Mechanizing Proofs of Termination in the Context-Sensitive Dependency
Pairs Framework ....................................... 265

MICHAEL LEUSCHEL, SALVADOR TAMARIT, GERMÁN VIDAL
A Fast Procedure for the Strong Termination Analysis of Logic
Programs .............................................. 275

CSP, Concurrencia y Pérez ..................................... 285

MARISA LLORENS, JAVIER OLIVER, JOSEP SILVA, SALVADOR TAMARIT
A Semantics for Tracing CSP .................................. 287

MIQUEL BOFILL, MIQUEL PALÀFÍ, MATÉU VILLARET
A system for CSP solving through Satisfiability Modulo Theories ...... 303
Las herramientas sobre Programación y Lenguajes, PROLE, se visieron como un marco propio de revisión, discusión, divulgación para las herramientas y aplicaciones relacionadas con la programación y los lenguajes. En el marco de este avance, se realizaron diferentes estudios de caso en muchas ciudades participantes en el proyecto PROLE, los cuales incluyen: los casos de estudio de Programación, Lenguajes, Programación, Lenguajes, Programación, Lenguajes y Programación, Lenguajes. En este contexto, se identifican algunos retos y desafíos para la implementación de estas herramientas.

En este contexto, se identifican algunos retos y desafíos para la implementación de estas herramientas. En el caso de las herramientas de programación, se evidencia la necesidad de una mayor investigación y divulgación para que sean conocidas y utilizadas por una mayor cantidad de personas. En esta línea, se recoge la importancia de realizar investigaciones sobre las herramientas utilizadas y su impacto en la programación y el desarrollo de software.

Además, se plantea la necesidad de realizar un análisis más profundo sobre las herramientas utilizadas y su impacto en la programación y el desarrollo de software. En este sentido, es fundamental seguir realizando investigaciones y divulgando sobre las herramientas utilizadas para que sean conocidas y utilizadas por una mayor cantidad de personas. En este sentido, se evidencia la necesidad de realizar investigaciones sobre las herramientas utilizadas y su impacto en la programación y el desarrollo de software.
con este agradecimiento, es justo felicitar a los autores por la calidad de sus trabajos y su contribución a que esta edición sea la de mayor participación en la evolución histórica de PROLE, lo que garantiza la buena salud del evento.

Por otro lado, además de las tres conferencias invitadas que compartimos con la planificación de JISBD’09, el programa de PROLE’09 cuenta este año con una excelente conferencia específica (un resumen de la misma se incluye en este volumen) que, bajo el título de “Understanding program verification”, será impartida por K. Rustan M. Leino, de Microsoft Research, USA, a quien agradecemos el haber aceptado tan amablemente nuestra invitación.

Finalmente, queremos agradecer la confianza que han depositado en nosotros, para conducir la presente edición de estas jornadas, a todos los miembros del comité ejecutivo de PROLE, y esperamos no haberles defraudado. En el desempeño de esta tarea, ha sido determinante la ayuda y experiencia prestada por Jesús Almudres, quien presidió la anterior edición de PROLE en Gijón, y a quien aprovechamos para darle gracias desde aquí.

Septiembre de 2009

Pepa Lucio
Ginés Morvino
Ricardo Peña

PATROCINADORES