Proceedings of CICLOPS 2007

7^{th} International Colloquium on Implementation of Constraint and LOgic Programming Systems

Salvador Abreu and Vitor Santos Costa

September 8, 2007

Porto, Portugal

Preface

The last years have witnessed continuous progress in the technology available both for academic and commercial computing environments. Examples include more processor performance, increased memory capacity and bandwidth, faster networking technology, operating system support for cluster computing and the generalized use of mutiprocessor systems, including in particular multicore microprocessors. These improvements, combined with recent advances in compilation and implementation technologies, are causing high-level languages to be regarded as good candidates for programming complex, real world applications. Techniques aiming at achieving flexibility in the language design make powerful extensions easier to implement; on the other hand, implementations which reach good performance in terms of speed and memory consumption make declarative languages and systems amenable to develop non-trivial applications.

Logic Programming and Constraint Programming, in particular, seem to offer one of the best options, as they couple a high level of abstraction and a declarative nature with an extreme flexibility in the design of their implementations and extensions and of their execution model. This adaptability is key to, for example, the implicit exploitation of alternative execution strategies tailored for different applications (e.g., for domain-specific languages) without unnecessarily jeopardizing efficiency.

CICLOPS 2007 continues a tradition of successful workshops on Implementations of Logic Programming Systems, previously held with in Budapest (1993) and Ithaca (1994), the Compulog Net workshops on Parallelism and Implementation Technologies held in Madrid (1993 and 1994), Utrecht (1995) and Bonn (1996), the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages (ParImp) held in Port Jefferson (1997), Manchester (1998), Las Cruces (1999), and London (2000), and more recently the Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS) in Paphos (Cyprus, 2001), Copenhagen (2002), Mumbai (2003), Saint-Malo (France, 2004), Sitges (Spain, 2005), Seattle (USA, 2006), and the CoLogNet Workshops on Implementation Technology for Computational Logic Systems held in Madrid (2002), Pisa (2003) and Saint-Malo (France, 2004).

August 2007,

Salvador Abreu and Vitor Santos Costa
CICLOPS 2007 chairs

Organization

CICLOPS 2007 was organized by the Informatics Departments of Universidade do Porto and Universidade de Évora, in cooperation with the 23rd International Conference on Logic Programming (ICLP 2007) and the Association for Logic Programming (ALP).

Workshop Chairs

Salvador Abreu	Universidade de Évora, Portugal
Vitor Santos Costa	Universidade do Porto, Portugal

Program Commitee

Bart Demoen	K.U. Leuven, Belgium
David Scott Warren	State University of New York at Stony Brook, USA
Enrico Pontelli	New Mexico State University, USA
Gopal Gupta	University of Texas at Dallas, USA
Hai-Feng Guo	University of Nebraska at Omaha, USA
Inês Dutra	Universidade do Porto, Portugal
Kostis Sagonas	Uppsala University, Sweden
Manuel Carro	Universidad Politécnica de Madrid, Spain
Mats Carlsson	SICS, Sweden
Michael Leuschel	Heinrich Heine Universität Düsseldorf, Germany
Michel Ferreira	Universidade do Porto, Portugal
Neng-Fa Zhou	City University of New York, USA
Paul Tarau	University of North Texas, USA
Paulo Moura	Universidade da Beira Interior, Portugal
Ricardo Rocha	Universidade do Porto, Portugal
Tom Schrijvers	K.U. Leuven, Belgium

Table of Contents

Operational Approach to the Modified Reasoning, Based on the Concept of Repeated Proving and Logical Actors

Alexei A. Morozov

Institute of Radio Engineering and Electronics RAS
Mokhovaya 11, Moscow, Russia, 125009
morozov@cplire.ru
http://www.cplire.ru/Lab144/

Abstract. The message of this paper is the following: there is one more basic principle of operational semantics of logic programming (besides backtracking, recursion, etc.) that gives a solution of challenging problem of combining strict declarative semantics of logic languages with the dynamic behavior (that includes destructive assignment operations and interaction with dynamic environment). We have developed this principle, named repeated proving, in the Actor Prolog logic language. In this paper the repeated proving principle is explained with the help of an operational semantics (abstract machine) for sequential logic programs enhanced with logical actors. The problems of soundness and completeness of the control strategy are considered.

Introduction

We address the problem of ensuring strict declarative semantics of logic languages operating in dynamic environment [1,2,3,4]. Our approach reminds of so-called perturbation model of constraint-based languages. In the perturbation model, unlike the standard (refinement) one, at the beginning of execution cycle variables have specific associated values satisfying the constraints. The value of one or more variables is perturbed by some outside influence, such as an edit request from the user, and the task of the prover is to adjust the values of the variables in such a way as to satisfy the constraints again [5,6].

The problem is closely related to the problem of ensuring the declarative semantics of the destructive assignment operation in logic languages. One can consider the updates in the outer world as a kind of destructive assignment that violates the soundness of the logic program. In this article, this problem is solved using the principle of repeated proving of sub-goals.

In section 1, the ideas of repeated proving and logical actors are set forth. In section 2, a special notation is introduced along with the architecture of an abstract machine implementing a sequential control strategy of logic programs enhanced with logical actors. In section 3, transition diagrams of the abstract machine are defined. In section 4, the problems of soundness and completeness of the operational semantics are discussed.

1 The Idea of Repeated Proving and Logical Actors

Let us consider a logic program written in pure Prolog that has a classical model-theoretic semantics. The idea of repeated proving consists in dividing the AND-tree of the logic program into separate branches (sub-goals to be proved) called logical actors ($\alpha_1,\ldots,\alpha_n$ on the Fig. 1) that should have the following operational properties:

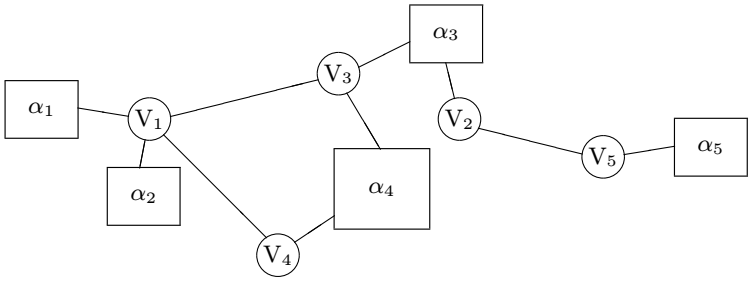


Fig. 1. The idea of repeated proving of sub-goals.

1. Common variables ($V_1,\ldots,V_m$) are the single channel of data exchange between the actors.
2. Proving of separate actors can be fulfilled independently in arbitrary order.
3. One can cancel the results of proving of some actors *without* logic program backtracking while keeping all other sub-goals of the program. After canceling results of proving of an actor, its proving is to be repeated.

Thus, one can implement a modification of reasoning. The results and consecution of reasoning itself can be partially modified in the process and after the logical inference. This makes possible to eliminate contradictions between the results of logical reasoning and new information income from outer world.

The best example of application of the idea is implementation of long-lived Web agents. Let us imagine a Web agent written in logic language. The purpose of the agent is to make a logical inference on the basis of several remote data sources and to check some assertions about the remote resources. Let us imagine also that the agent is long-lived, i.e., it operates during a period of time that is *longer* than the period of information update. Thus the agent should react on any modification of remote resources and inform the user about the current state of the assertions to be checked. The problem is that one cannot repeat execution of the logic program from the beginning with any change in the outer world — the repeat of the whole process of data collection performed during the long period of time is inefficient and, in some cases, technically impossible. Therefore one must change some branches of logic inference that depend on the modified data and keep all other branches unchanged. This is the case of modification of reasoning and the challenge is to provide soundness and (if possible) completeness of logical reasoning under the modification.

Another area that is recognized as a prospective application of the perturbation model of constrain-based languages is graphic user interface management [6]. We have successfully applied the logical actors approach for both the logical programming of Web agents [7,8] and visual user interface management [9]. An additional issue of our research is development of logic object-oriented model of asynchronous concurrent computations based on the logical actors approach [10].

In the following sections a conservative extension of standard control strategy of (sequential) Prolog is developed that implements the repeated proving of logical actors.

2 The Architecture of Abstract Machine

Let us consider an abstract machine that implements a sequential control strategy for logic programs enhanced with logical actors. The input language of this machine is the Horn subset of first order logic formulas enhanced with special means implementing logical actors.

The abstract machine implements the following general principles:

1. The standard control strategy (depth-first left-to-right search) is a part of the control strategy implemented by the abstract machine.
2. The AND-tree of logic program is to be divided into separate logical actors, i.e., any pending sub-goal of the program is a logical actor or a part of a logical actor.
3. Any logical actor obtains its own (local) substantiation (local values of common variables).
4. The results of proving of logical actor can be cancelled.
5. The logical actor can be proved once again after the canceling of results of its previous proving.
6. The states of logical actors are restored during the backtracking.

Thus, the abstract machine implements the standard control strategy exactly if there is only one logical actor in the program (i.e., all the branches of the AND-tree belong to the same actor).

Each logical actor A of the program has its own (local) values of variables. Actor A unifies its values with the values that belong to other actors in the following cases only:

1. The local values are compared in the course of successful termination of proving of actor A.
2. The local values are compared when actor A executes the *':='* built-in predicate (this predicate will be considered later).

During the comparison of values that belong to different actors, abstract machine can cancel results of proving of some actors to provide consistency between remaining actors of the program (to provide existence of the most general unifier for all the values of all the actors of the program that remain uncancelled). After that the abstract machine tries to prove the cancelled actors once again.

Let us name the operation of canceling of results of proving of the actor as *neutralization of actor*.

Thus, the proving of actor A includes the following main stages (see Fig. 2).

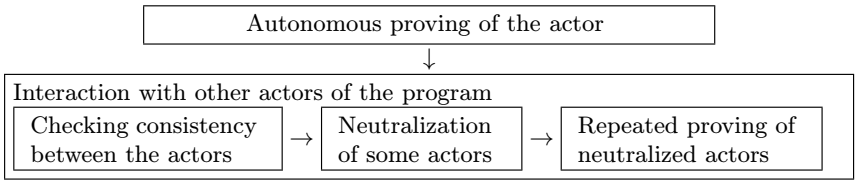


Fig. 2. The stages of execution of logical actor.

There are three possible states of the actor:

1. Let us name an actor *active* if the proving of this actor is performing at this moment and is not ended yet.
2. The actor that was successfully proved (and was not neutralized yet) is named *proven*.
3. The actor is named *neutral* if the proving of this actor was cancelled and the repeated proving of it was not started yet.

Neutralization of active actors is prohibited (see the formal rules of selecting actors for neutralization in section 3). Thus, sometimes the contradictions between the actors of the program cannot be eliminated with the help of actor neutralization. In this case standard backtracking occurs in the program, that returns actor A to the stage of autonomous proving.

In the case if the abstract machine successfully eliminates contradictions between the actors with the help of neutralization of some set NA of actors, repeated proving of all the neutral actors occurs. If proving of all neutral actors terminates with success (or set NA is empty), the proving of actor A terminates with success. In another case backtracking occurs in the logic program, that returns actor A to the stage of autonomous proving. Thus, a failure of the repeated proving of any actor of the NA set will backtrack the program.

Let us introduce some special notions to define the control strategy formally:

– *The state of abstract machine* is a set of actors:

$$\Gamma = \{A_1, A_2, \ldots, A_n\},$$

where A_i, $i = 1 \ldots n$, are the actors of the program.

– *Actor* A_i is a branch of AND-tree created as a result of execution of so-called *actor call* of a predicate $@m(t_1, \ldots, t_k)$:

$$A_i = \langle \alpha, m(t_1, \ldots, t_k), R \rangle,$$

where α is an (unique) name of actor; $m(t_1, \ldots, t_k)$ is an atomic formula that corresponds to given actor; R is a list named *the results of proving* of the actor.

– *The result of proving* of an actor is information obtained during the proving of the actor: instantiations of variables, backtrack points, etc.:

$$E = \langle \beta, F \rangle ,$$

where β is the name of actor that has invoked the proving under consideration; F is a stack of so-called *failure continuations* that is used for implementation of backtracking.

– *The failure continuation* is a stack containing sub-goals to be proved during investigation of one branch of OR-tree:

$$C = \langle G, \sigma, N, B \rangle ,$$

where G is *a list of sub-goals*; σ is an instantiation of variables used during investigation of the branch of the OR-tree under consideration; N is a list of actor names that were neutralized during investigation of given branch of the OR-tree; B is a list of actor names that were created during investigation of this branch.

– *The Subgoal* can be a usual predicate call $m(t_1, \ldots, t_k)$, an *actor* predicate call $@m(t_1, \ldots, t_k)$, compositions of sub-goals S_1 *and* S_2, S_1 *or* S_2, etc.

A special notation (@-language) necessary for definition of abstract machine states is given in tables 1, 2.

The semantics of formulas of kind $\Gamma.\alpha\,\{GL = S : G, Subst = \sigma\}$ is the following: there is an actor α in the Γ state of abstract machine, that has the following properties:

1. The GL cell situated on the top of the stack of failure continuations that is situated on the top of the stack of results of proving of the α actor has value $S : G$ (a list).
2. The $Subst$ cell situated on the top of the stack of failure continuations that is situated on the top of the stack of results of proving of the α actor has value σ.

In a similar manner, a formula of kind $\Gamma.\alpha = \langle \alpha, M, R \rangle$ has the following semantics: there is an actor α in state Γ of abstract machine. The value of this actor is equal to $\langle \alpha, M, R \rangle$. One can use given formulas in the following sense: "The Γ state, such that there is an actor α that has the following properties…".

We will use also formulas of the following kind in the transition diagrams: $\Gamma' = \Gamma : \alpha\,\{GL := G\}$. The semantics of these formulas is "The Γ' state of abstract machine differs from the Γ state in that a new value G was assigned to the GL cell that is situated on the top of the stack of failure continuations that is situated on the top of the stack of results of proving of the α actor."

Table 1. The table of basic symbols of the @-language.

Notion	**Symbol**	**Definition**	**Typical elements**
Constant	$Const$		a, b, c
Variable	Var		X, Y, Z
Functor	Fun		f
Term	$Term$	$Const$; Var; $f(t_1, \ldots, t_k),\ k \geq 1$	t, v, u
Atomic formula	$Atom$	$m(t_1, \ldots, t_k),\ k \geq 0$	M
Name of actor	$Name$	$\alpha, \beta, \gamma, \ldots; \tau; \xi$, where τ and ξ are special names	
Sub-goal	$Subgoal$	$true$; $fail$; M; $@M$; S_1 *and* S_2; S_1 *or* S_2; $del([\alpha_1, \ldots, \alpha_n])$; $back([\alpha_1, \ldots, \alpha_n])$; $wait(\gamma)$; $redo(\gamma)$; $neutralize(\{\alpha_1, \ldots, \alpha_n\})$; $restart(\{\alpha_1, \ldots, \alpha_n\})$	S
Procedure	$Procedure$	$M :- S$	P
Definition of procedures	$Procedures$	Function $Atom \rightarrow Subgoal$	D

Table 2. Definition of the @-language.

Notion	**Symbol**	**Definition**	**Typical elements**
State of abstract machine	$State$	$\{A_1, A_2, \ldots, A_n\},\ n \geq 1$	Γ
Actor	$Actor$	$\langle \alpha, M, R \rangle$	A
List of results of proving	RL	nil; $E : R$, is a list with head E and rest R.	R
Results of one proving	$Result$	$\langle \beta, F \rangle$; $neutral$ where $neutral$ is a special symbol	E
Stack of failure continuations	FL	nil; $C : F$	F
Failure continuation	$Cont$	$\langle G, \sigma, N, B \rangle$	C
List of sub-goals (named also success continuation)	GL	nil; $success$; $failure$; $S : G$	G
Substitution	$Subst$	$\sigma, \theta, \ldots; \varepsilon$ (ε is the empty substitution)	
List of names of neutral actors	$Neutr$	$[\alpha_1, \ldots, \alpha_n]$	N
List of names of created actors	$Built$	$[\alpha_1, \ldots, \alpha_n]$	B

A logic program is defined as a set D of procedures[1] (see designations of the @-language in table 1) and an initial state of the program:

$$\Gamma^0(\tau) = \left\langle \tau, m(t_1, \ldots, t_k), \left\langle \xi, \left\langle m(t_1, \ldots, t_k) : nil, \varepsilon, [\,], [\,] \right\rangle : nil \right\rangle : nil \right\rangle,$$

where τ is the name of an actor (the target actor hereafter) that is active in the Γ state, and ξ is dummy name of an actor situated in outer world (the external actor) that has invoked the program under consideration. All the actors except for the τ actor are *proven*[2] in the Γ^0 state of the abstract machine:

$$\forall \alpha : \Gamma^0.\alpha \{Name \neq \tau\} : is\_proven(\Gamma^0, \alpha)$$

The abstract machine can reach one of two final states:

1. The *success state*: $\Gamma^{SUCCESS}.\tau \{Cont = \langle success, \sigma, N, B \rangle\}$, where τ is the target actor introduced in the Γ^0 initial state.
2. The *failure state*: $\Gamma^{FAILURE}.\tau \{FL = \langle failure, \varepsilon, [\,], [\,] \rangle : nil\}$.

Note, that the *success* and the *failure* states are alternative in accordance with given definitions. Deadlocks never occur in the abstract machine.

3 Transition System

The transition system of abstract machine is defined with the help of set of transition schemas and set Λ of labels (let us denote the typical label by l).

Let us consider the main stages of the proving of logical actor (Fig. 2).

3.1 Autonomous Proving of Actor

Execution of logic program is performed in accordance with the standard control strategy (depth-first left-to-right search) on this stage of proving of the actor. This strategy is implemented with the help of the transition schemas: $True$, Rec, Loc_1, Seq, and Alt. Some auxiliary schemas implement creation, deletion, and modification of logical actors during the proving.

$True$ — elimination of the *true* sub-goal during the execution of actor α.

$$\Gamma.\alpha \{GL = true : G\} \xrightarrow{\langle True, \alpha \rangle} \Gamma.\alpha \{GL := G\}$$

The semantics of this transition schema is the following one: "If current state Γ of abstract machine is such that an actor α exists and current list of sub-goals of this actor $GL = true : G$, then state Γ can be transformed into new one. In new state of abstract machine current list of sub-goals of actor α is modified:

[1] Let us do not use different procedures with the same functor (name and arity) of heading M to simplify the presentation.

[2] The *is_proven* predicate is defined in section 3.2.

$GL := G$. All other attributes of actor α and all other actors of the abstract machine will not changed during the transformation."

Rec — a call of predicate m during the execution of actor α. The $rename : P \to P'$ function implements renaming of variables of given procedure in the standard manner. The $mgu : (M_1, M_2) \to \sigma$ function computes the most general unifier of terms M_1, M_2 (iff the unifier exists).

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = m(t_1,\ldots,t_k) : G, Subst = \sigma\}\\ \exists\, P \in D :\\ \quad rename(P) = (m(u'_1,\ldots,u'_k) :- S')\ \wedge\\ \quad \exists\,\theta = mgu(m(t_1,\ldots,t_k)\sigma, m(u'_1,\ldots,u'_k))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := S' : G, Subst := \sigma\theta\}\\ \Gamma \xrightarrow{\langle Rec,\alpha\rangle} \Gamma'\end{array}}$$

where $\langle Rec, \alpha\rangle$ is the label of transition scheme under consideration. The statements over the line determine the conditions when the *Rec* schema can be performed. The statements under the line explain what is the difference between old state Γ and new state Γ' that can be obtained with the help of the *Rec* transition schema.

Loc_1 — backtracking of given actor α. The '$-$' function designates the difference between lists: $L - L' = L''$, if $L'' = [\alpha_1,\ldots,\alpha_n]$ and $L = [\alpha_1,\ldots,\alpha_n|L']$. The '$+$' function designates concatenation of lists.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle S : G, \sigma, N, B\rangle : (\langle G', \sigma', N', B'\rangle : F')\}\,,\\ S = fail\ \vee\\ (S = m(t_1,\ldots,t_k)\ \wedge\ \neg\exists\, P \in D :\\ \quad rename(P) = (M' :- S')\ \wedge\\ \quad \exists\,\theta = mgu(m(t_1,\ldots,t_k)\sigma, M'))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{FL := \langle back(N'' + B'') : (del(B'') : G')\,, \sigma', N', B'\rangle : F'\}\,,\\ N'' = N - N',\ B'' = B - B'\\ \Gamma \xrightarrow{\langle Loc_1,\alpha\rangle} \Gamma'\end{array}}$$

Loc_2 — recognition of necessity to transmit backtracking from actor α to the actor that has invoked current proving of actor α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle S : G, \sigma, N, B\rangle : \underline{nil}\}\,,\\ S = fail\ \vee\\ (S = m(t_1,\ldots,t_k)\ \wedge\ \neg\exists\, P \in D :\\ \quad rename(P) = (M' :- S')\ \wedge\\ \quad \exists\,\theta = mgu(m(t_1,\ldots,t_k)\sigma, M'))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{FL := \langle back(N + B) : (del(B) : failure)\,, \varepsilon, [\,], [\,]\rangle : nil\}\\ \Gamma \xrightarrow{\langle Loc_2,\alpha\rangle} \Gamma'\end{array}}$$

Glo — transmission of backtracking from actor α to actor β.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Result = \langle\beta, \langle failure, \varepsilon, [\,], [\,]\rangle : nil\rangle\}\\ \Gamma.\beta\,\{Subgoal = wait(\alpha)\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \beta\,\{GL := fail : nil\}\\ \Gamma \xrightarrow{\langle Glo,\beta,\alpha\rangle} \Gamma'\end{array}}$$

$Back_0$ — termination of process of recovering the states of actors during backtracking of the program.

$$\Gamma.\alpha\,\{GL = back([\,]) : G\} \xrightarrow{\langle Back_0,\alpha\rangle} \Gamma.\alpha\,\{GL := G\}$$

$Back_1$ — recovery of the *active* or the *proven* state of actor γ during backtracking of actor α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = back([\gamma|BList]) : G_\alpha\}\\ \Gamma.\gamma\,\{RL = \langle\beta, \langle G_\gamma, \sigma_\gamma, N_\gamma, B_\gamma\rangle : F_\gamma\rangle : R_\gamma\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := back(N_\gamma + B_\gamma + BList) : (del(B_\gamma) : G_\alpha)\}\,,\\ \qquad\quad \gamma\,\{RL := R_\gamma\}\end{array}}$$
$$\Gamma \xrightarrow{\langle Back_1,\alpha,\gamma\rangle} \Gamma'$$

$Back_2$ — recovery of the *neutral* state of actor γ during backtracking of α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = back([\gamma|BList]) : G_\alpha\}\\ \Gamma.\gamma\,\{RL = neutral : R_\gamma\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := back(BList) : G_\alpha\}\,,\\ \qquad\quad \gamma\,\{RL := R_\gamma\}\end{array}}$$
$$\Gamma \xrightarrow{\langle Back_2,\alpha,\gamma\rangle} \Gamma'$$

Del_0 — termination of deletion of actors during backtracking of actor α.

$$\Gamma.\alpha\,\{GL = del([\,]) : G\} \xrightarrow{\langle Del_0,\alpha\rangle} \Gamma.\alpha\,\{GL := G\}$$

Del_1 — deletion of actor γ during backtracking of actor α. The '/' function designates deletion of actor: $\Gamma_1/\gamma = \Gamma_2$, such that $\{\langle\gamma, M_\gamma, R_\gamma\rangle\} \cup \Gamma_2 = \Gamma_1$, $\Gamma_1 \neq \Gamma_2$.

$$\frac{\Gamma.\alpha\,\{Subgoal = del([\gamma|DList])\}}{\Gamma' = (\Gamma : \alpha\,\{Subgoal := del(DList)\})\,/\gamma}$$
$$\Gamma \xrightarrow{\langle Del_1,\alpha,\gamma\rangle} \Gamma'$$

Seq — execution of conjunction of sub-goals of actor α.

$$\Gamma.\alpha\,\{GL = (S_1\ and\ S_2) : G\} \xrightarrow{\langle Seq,\alpha\rangle} \Gamma.\alpha\,\{GL := S_1 : (S_2 : G)\}$$

Alt — execution of disjunction of sub-goals of actor α.

$$\Gamma.\alpha\,\{FL = \langle(S_1\ or\ S_2) : G, \sigma, N, B\rangle : F\} \xrightarrow{\langle Alt,\alpha\rangle} \Gamma.\alpha\,\{FL := \langle S_1 : G, \sigma, N, B\rangle : (\langle S_2 : G, \sigma, N, B\rangle : F)\}$$

New_1 — execution of actor predicate call $@m$ during execution of actor α.

The *code* auxiliary function is used for preparation of arguments of actor predicate call. This function (see Fig. 3) provides transfer of maximal quantity of information about the values of the arguments of predicate into the γ actor

to be created. The *code* function transfers the values of the instantiated variables and copies the variables that are unbound. The *copy* auxiliary function copies the values of variables. The *new_variable* function creates new variables. The *is_variable* function checks if the argument is an (unbound) variable. The $not\_exists(\Gamma, \gamma)$ expression means $\langle \gamma, M_\gamma, F_\gamma \rangle \notin \Gamma$.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle @m(t_1,\ldots,t_k) : G, \sigma, N, B\rangle : F\} \\ not\_exists(\Gamma,\gamma) \\ \exists\, P \in D : \\ \quad rename(P) = (m(u'_1,\ldots,u'_k) :\!-\, S')\ \wedge \\ \quad ([v_1,\ldots,v_k], \sigma') := code([t_1,\ldots,t_k], \sigma)\ \wedge \\ \quad \exists\, \theta = mgu(m(v_1,\ldots,v_k), m(u'_1,\ldots,u'_k))\end{array}}{\begin{array}{l}\Gamma' = \left(\Gamma : \alpha \left\{ \begin{array}{l} FL := \langle wait(\gamma) : G, \underline{\sigma'}, N, [\gamma|B]\rangle \\ : (\langle redo(\gamma) : G, \underline{\sigma'}, N, [\gamma|B]\rangle : F) \end{array} \right\}\right) \cup \\ \quad \{\langle \gamma, m(v_1,\ldots,v_k), \langle \alpha, \langle S' : nil, \underline{\theta}, [\,], [\,]\rangle : nil\rangle : nil\rangle\} \\ \Gamma \xrightarrow{\langle New_1, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

$code : [\{t_i\}, \sigma] \to [\{t'_i\}, \sigma'], i = 1 \ldots n$	$copy : [t, \sigma] \to [t', \sigma']$
$\sigma' := \sigma;$	$if\ \ t\sigma = f(\{u_j\}), j = 1 \ldots k$
$do\ \ i = 1 \ldots n$	$\quad \sigma' := \sigma;$
$\quad if\ \ t_i = f(\{u_j\}), j = 1 \ldots k$	$\quad do\ \ j = 1 \ldots k$
$\quad\quad [\{v_j\}, \sigma'] := code(\{u_j\}, \sigma);$	$\quad\quad if\ \ is\_variable(u_j)$
$\quad\quad t'_i := f(\{v_j\});\ \ \sigma := \sigma'$	$\quad\quad\quad u'_j := new\_variable();$
$\quad elsif\ \ is\_variable(t_i)$	$\quad\quad\quad \sigma' := \sigma \cup \{u'_j = u_j\}$
$\quad\quad if\ \ is\_variable(t_i\sigma)\ \ t'_i := t_i$	$\quad\quad else\ \ [u'_j, \sigma'] := copy(u_j, \sigma)$
$\quad\quad else\ \ [t'_i, \sigma'] := copy(t_i, \sigma);$	$\quad\quad fi;$
$\quad\quad\quad \sigma := \sigma'$	$\quad\quad \sigma := \sigma'$
$\quad\quad fi$	$\quad od;$
$\quad else\ \ t'_i := t_i$	$\quad t' := f(\{u_j\}), j = 1 \ldots k$
$\quad fi$	$else\ \ t' := t\sigma;\ \ \sigma' := \sigma$
od	fi

Fig. 3. Definitions of coding and copying functions.

New_2 — recognition of that an actor predicate $@m$ call cannot be performed during the execution of actor α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Subgoal = @m(t_1,\ldots,t_k), Subst = \sigma\} \\ \neg\exists\, P \in D : \\ \quad rename(P) = (m(u'_1,\ldots,u'_k) :\!-\, S')\ \wedge \\ \quad ([v_1,\ldots,v_k], \sigma') := code([t_1,\ldots,t_k], \sigma)\ \wedge \\ \quad \exists\, \theta = mgu(m(v_1,\ldots,v_k), m(u'_1,\ldots,u'_k))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := fail : nil\} \\ \Gamma \xrightarrow{\langle New_2, \alpha\rangle} \Gamma'\end{array}}$$

$Redo_1$ — backtracking of the γ actor during backtracking of actor α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle redo(\gamma) : G_\alpha, \sigma_\alpha, N_\alpha, B_\alpha\rangle : F_\alpha\} \\ \Gamma.\gamma\,\{RL = \langle \alpha, \langle G_\gamma, \sigma_\gamma, N_\gamma, B_\gamma\rangle : (C'_\gamma : F'_\gamma)\rangle : R_\gamma\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha \left\{\begin{array}{l} FL := \langle wait(\gamma) : G_\alpha, \sigma_\alpha, N_\alpha, B_\alpha\rangle \\ : (\langle redo(\gamma) : G_\alpha, \sigma_\alpha, N_\alpha, B_\alpha\rangle : F_\alpha)\end{array}\right\}, \\ \qquad\qquad \gamma\,\{RL := \langle \alpha, \langle fail : nil, \sigma_\gamma, N_\gamma, B_\gamma\rangle : (C'_\gamma : F'_\gamma)\rangle : R_\gamma\} \\ \Gamma \xrightarrow{\langle Redo_1, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

$Redo_2$ — recognition of that backtracking of actor γ cannot be performed during execution of actor α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Subgoal = redo(\gamma)\} \\ \Gamma.\gamma\,\{RL = \langle \alpha, C_\gamma : \underline{nil}\rangle : R_\gamma\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := fail : nil\} \\ \Gamma \xrightarrow{\langle Redo_2, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

3.2 Interaction of Logical Actors

The abstract machine implements the following operations on this stage:

1. The comparison of substitutions that correspond to various actors of the program.
2. Neutralization of some actors.
3. Repeated proving of neutral actors.

$Check_1$ — checking if the actors of the program are consistent (during termination of proving of actor α).

Let us introduce some additional notions:

- $is\_neutral(\Gamma, \gamma) \stackrel{def}{=} \Gamma.\gamma\,\{Result = neutral\}$;
- $is\_active(\Gamma, \gamma) \stackrel{def}{=} \neg is\_neutral(\Gamma, \gamma) \wedge \Gamma.\gamma\,\{GL \neq success\}$;
- $is\_proven(\Gamma, \gamma) \stackrel{def}{=} \neg is\_neutral(\Gamma, \gamma) \wedge \Gamma.\gamma\,\{GL = success\}$;
- $SUBST(\Gamma, \gamma)$ is substitution σ_γ, $\Gamma.\gamma\,\{Subst = \sigma_\gamma\}$, or empty substitution ε, if $is\_neutral(\Gamma, \gamma)$;
- $does\_exist(\Gamma, \gamma) \stackrel{def}{=} \langle \gamma, M_\gamma, R_\gamma\rangle \in \Gamma$;
- $\Sigma(\Gamma, \{\alpha_1, \ldots, \alpha_n\}) = \bigcup_{i=1}^{n} SUBST(\Gamma, \alpha_i)$ — is a set of substitution assignments corresponding to all the actors $\alpha_1, \ldots, \alpha_n$ in state Γ.

Definition 1. *Set S of substitution assignments is conflicting one, if there are two subsets σ_1 and σ_2 and a variable X such that:*

1. *σ_1 and σ_2 are substitutions.*
2. *These substitutions gives values V_1 and V_2 to the X variable, that have no most general unifier.*

$$inconsistent(S) \stackrel{def}{=} \exists\, \sigma_1 \subset S \ \wedge\ \exists\, \sigma_2 \subset S \ \wedge\ \exists\, X : \neg\exists\, mgu(X\sigma_1, X\sigma_2).$$

Definition 2. *$consistent(S) \stackrel{def}{=} \neg inconsistent(S)$ — is a consistent set of substitution assignments.*

Definition 3. *A set of names NA of actors to be neutralized and proved repeatedly $may\_be\_neutralized(\Gamma, NA)$:*

1. $\forall \beta \in NA : does\_exist(\Gamma, \beta) \ \wedge\ is\_proven(\Gamma, \beta)$;
2. $\forall \beta \in NA :$
   $\exists$ *set of actors* $\{\alpha_i\}, i = 1, \ldots, k : does\_exist(\Gamma, \alpha_i) :$
   $inconsistent(\Sigma(\{\alpha_1, \ldots, \alpha_k, \beta\})) \ \wedge\ consistent(\Sigma(\{\alpha_1, \ldots, \alpha_k\}))$;
3. *A set of substitution equations of actors of any subset of Γ that has no common elements with the NA set should be consistent one.*

The condition (2) excludes any unnecessary neutralization of actors that are irrelevant to the contradictions that should be eliminated.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = nil\} \\ \exists\, NA : may\_be\_neutralized(\Gamma, NA)\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := neutralize(NA) : (restart(NA) : success)\} \\ \Gamma \xrightarrow{\langle Check_1, \alpha\rangle} \Gamma'\end{array}}$$

$Check_2$ — recognition of impossibility to eliminate contradictions between the actors with the help of neutralization of some actors (during termination of proving of actor α).

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = nil\} \\ \neg\exists\, NA : may\_be\_neutralized(\Gamma, NA)\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := fail : nil\} \\ \Gamma \xrightarrow{\langle Check_2, \alpha\rangle} \Gamma'\end{array}}$$

$Neut_0$ — termination of neutralization of actors (during termination of proving of actor α).

$$\Gamma.\alpha\,\{GL = neutralize(\emptyset) : G\} \xrightarrow{\langle Neut_0, \alpha\rangle} \Gamma.\alpha\,\{GL := G\}$$

$Neut_1$ — neutralization of actor γ during execution of actor α:

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Cont = \langle neutralize(\{\gamma\} \cup NA') : G, \sigma, N, B\rangle\},\ \gamma \notin NA' \\ \Gamma.\gamma\,\{RL = R\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{Cont := \langle neutralize(NA') : G, \sigma, [\gamma|N], B\rangle\}, \\ \qquad \gamma\,\{RL := neutral : R\} \\ \Gamma \xrightarrow{\langle Neut_1, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

$Succ$ — termination of proving of actor α with success.

$$\Gamma.\alpha\,\{GL = restart(\emptyset) : G\} \xrightarrow{\langle Succ, \alpha\rangle} \Gamma.\alpha\,\{GL := G\}$$

Call — invocation of repeated proving of actor γ during execution of α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle restart(\{\gamma\} \cup RA') : G, \sigma, N, B\rangle : F\}\,,\ \gamma \notin RA' \\ \Gamma.\gamma = \langle \gamma, m(v_1, \ldots, v_k), R\rangle\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha \left\{\begin{array}{l} FL := \langle wait(\gamma) : (restart(RA') : G)\,, \sigma, [\gamma|N], B\rangle \\ : (\langle redo(\gamma) : (restart(RA') : G)\,, \sigma, [\gamma|N], B\rangle : F)\end{array}\right\}, \\ \qquad\qquad \gamma\,\{RL := \langle \alpha, \langle m(v_1, \ldots, v_k) : nil, \underline{\varepsilon}, [\,], [\,]\rangle : nil\rangle : R\} \\ \Gamma \xrightarrow{\langle Call, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

Note that the $Check_1$, the $Neut_1$, and the $Call$ schemas make abstract machine nondeterministic one.

Con — resumption of proving of actor β after termination of proving of actor α that was invoked by actor β.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = success\} \\ \Gamma.\beta\,\{GL = wait(\alpha) : G\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \beta\,\{GL := G\} \\ \Gamma \xrightarrow{\langle Con, \beta, \alpha\rangle} \Gamma'\end{array}}$$

Note that defined abstract machine provides a possibility for modeling destructive assignment of variables with the help of logical actors. For instance, the $X := Y$ build-in predicate is implemented in the Actor Prolog language, that invokes the interaction between the actors of the program. The operational semantics of the *':='* predicate is straightforward one:

1. The predicate tries to unify the X and the Y terms.
2. If the most general unifier exists, the interaction of actors of the program is performed in accordance with the rules described above.
3. If neutralization and repeated proving of actors provides consistency between the actors of the program, the execution of the *':='* predicate terminates with success. In another case backtracking occurs in the program.

The model-theoretic semantics of this predicate is exactly the same as the semantics of the usual equality *'='* in pure Prolog and the operational semantics of the *'='* predicate is a special case of the *':='* predicate operational semantics.

4 Operational Semantics

The operational semantics of sequential logic program enhanced with logical actors is a map $\mathcal{O}$ that projects definition of procedures D and an initial state of program Γ^0, $\Gamma^0.\tau = \langle \tau, m(t_1, \ldots, t_k), R_\tau\rangle$, into the set of finite and infinite chains of states obtained with the help of transition schemas defined above.

Definition 4. *Operational semantics $\mathcal{O}$:*

$$\mathcal{O}[D, \Gamma^0] \stackrel{def}{=} \begin{array}{l}\left\{\Gamma^0 \xrightarrow{l_1} \Gamma_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} \Gamma_n^{SUCCESS}\right\} \cup \\ \left\{\Gamma^0 \xrightarrow{l_1} \Gamma_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} \Gamma_n^{FAILURE}\right\} \cup \\ \left\{\Gamma^0 \xrightarrow{l_1} \Gamma_1 \xrightarrow{l_2} \ldots\right\}.\end{array}$$

Note that the model-theoretic semantics of defined @-language strictly corresponds to the model-theoretic semantics of pure Prolog without negation.

Definition 5. *An initial set of actor constraints is a set of logical statements that corresponds to all the proven actors of initial state* Γ^0*:*

$$Init \stackrel{def}{=} \bigwedge_i M_i \text{ for all } \langle \alpha_i, M_i, R_i \rangle \in \Gamma^0, \text{such that } is\_proven(\Gamma^0, \alpha_i).$$

Proposition 1 (on soundness of the operational semantics). *The operational semantics* $\mathcal{O}$ *is sound, i.e., the success final state of the program can be obtained only if union of procedure definitions* D *with the negation of conjunction of initial set* $Init$ *and goal statement* $m(t_1, \ldots, t_k)$ *is unsatisfiable:*

$$\left(\Gamma^0 \stackrel{\star}{\longrightarrow} \Gamma^{SUCCESS}\right) \Rightarrow \left(D \cup \{\neg\,(Init \ \wedge \ m(t_1, \ldots, t_k))\} \models \bot\right).$$

Proposition 2 (on completeness of the operational semantics). *The success final state of the program will be obtained if a substitution* θ *exists, such that*

$$D \models (Init \ \wedge \ m(t_1, \ldots, t_k))\,\theta,$$

and no infinite computations arise: $\Gamma^0 \stackrel{\star}{\longrightarrow} \Gamma^{SUCCESS}$.

Thus, the program can fall into an infinite computation even if a success branch is present in the AND-OR tree, like the standard sequential Prolog.

Nevertheless the additional operation of neutralization of actors cannot provoke looping of the program, because the neutralization of active actors is prohibited in schema $Check_1$.

The practical use of the control strategy under consideration requires that the abstract machine stops after the obtaining of the first success final state despite the fact that the abstract machine can implement the exhaustive search until all existed answers are computed or an infinite computation occurs. This restriction corresponds to the perturbation model of constraint-based languages, i.e., the problem to be solved by the abstract machine is to fit given system of constraints to new information income from outer world only. After that, the abstract machine will wait for a new outside influence.

Conclusion

The logical actors concept gives an alternative to the nonmonotonic approach in logic programming. It forms a basis for solving the problem of ensuring soundness and completeness of the destructive assignment operation as well as strict classical model-theoretic semantics of logic programs operating in dynamic environment (such as graphical user interface and Internet).

The repeated proving of sub-goals allows to modify the logical reasoning during the execution of a logic program. Following the principle of modifiable

reasoning, we have developed concurrent object-oriented logic language Actor Prolog that ensures soundness of logic programs operating under conditions of permanent altering and updating of input information [11,8,12]. The ideas stated in this paper are approved by practical experiments with visual logic programming and Web agent logic programming [7].

The author is grateful to Prof. Yu.V. Obukhov, Dr. A.F. Polupanov, Dr. A.N. Kruglov, and Dr. S.V. Remizov (IRE RAS) for help and support in implementing the project, to Acad. Yu.I. Zhuravlev and Prof. V.A. Zakharov (Moscow State University) for fruitful discussions of the problem.

This work was supported by RFBR, project no. 06-07-89302.

References

1. Chesnevar, C.I., Maguitman, A.G., Loui, R.P.: Logical models of argument. ACM Computing Surveys **32**(4) (2000) 337–383
2. Alferes, J., Pereira, L.: Logic programming updating — a guided approach. In Kakas, A., Sadri, F., eds.: Computational Logic: From Logic Programming into the Future — Essays in honour of Robert Kowalski. Volume 2. Springer (2002) 382–412
3. Dix, J., Furbach, U., Niemelae, I.: Nonmonotonic reasoning: Towards efficient calculi and implementations. In Voronkov, Robinson, eds.: Handbook of Automated Reasoning. Volume 2. Elsevier (2001) 1121–1234
4. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: Using methods of declarative logic programming for intelligent information agents. Theory and Practice of Logic Programming **2**(6) (2002) 645–709
5. Rossi, F.: Constraint (Logic) Programming: A Survey on Research and Applications. In: New Trends in Constraints: Joint ERCIM/Compulog Net Workshop, Paphos, Cyprus, October 1999. Selected Papers. Volume 1865 of LNAI., Springer (1999) 40–74
6. Sannella, M., Maloney, J., Freeman-Benson, B., Borning, A.: Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. Software — Practice and Experience **23**(5) (1993) 529–566
7. Morozov, A.: Development and application of logical actors mathematical apparatus for logic programming of Web agents. In Palamidessi, C., ed.: ICLP 2003. LNCS 2916, Springer (2003) 494–495
8. Morozov, A., Obukhov, Y.: An approach to logic programming of intelligent agents for searching and recognizing information on the Internet. Pattern Recognition and Image Analysis **11**(3) (2001) 570–582 Available at: http://www.cplire.ru/Lab144/pria570m.pdf.
9. Morozov, A.: Visual logic programming based on the SADT diagrams. In Dahl, V., Niemela, I., eds.: ICLP 2007. LNCS 4670, Springer (2007) 436–437
10. Morozov, A.: Logic object-oriented model of asynchronous concurrent computations. Pattern Recognition and Image Analysis **13**(4) (2003) 640–649 Available at: http://www.cplire.ru/Lab144/pria640.pdf.
11. Morozov, A.: Actor Prolog: an object-oriented language with the classical declarative semantics. In Sagonas, K., Tarau, P., eds.: Proc. of the IDL'99 Int. Workshop, Paris, France (1999) 39–53 Available at: http://www.cplire.ru/Lab144/paris.pdf.
12. Morozov, A.: Getting Started in Actor Prolog. IRE RAS (2002) Available at: http://www.cplire.ru/Lab144/start/.

Rule based classification of documents using Logic Programming

Gergely Lukácsy and Péter Szeredi

Budapest University of Technology and Economics
Department of Computer Science and Information Theory
1117 Budapest, Magyar tudósok körútja 2., Hungary
Phone: +36 1 463-2585 Fax: +36 1 463-3157
{lukacsy,szeredi}@cs.bme.hu
Keywords: rules, document classification, meta-data, logic programming

Abstract. This paper presents the results of an ongoing Hungarian research project aiming at the development of the SREngine framework. This software system manages a pool of generic objects together with their properties and supports reasoning on these. The main idea of the system is to infer new properties about the objects, using their existing properties and a set of user defined rules. These rules are given in a special logic programming language introduced in the paper, providing intuitive syntax and adequate expressive power.
In a typical scenario SREngine is used to classify documents, i.e. to attach type information to them. For this we usually need to analyse the textual content of the documents which is done by dedicated information extractor modules.
SREngine realises a bottom-up reasoner fully implemented in Prolog. The system is intended to be used in real-life applications, providing robust implementation and web-service based interfaces for standardised information exchange.

1 Introduction and goals

The Sense/Net Rule Engine (SREngine) is a generic rule based inference system implemented in Prolog. SREngine is primarily designed to cooperate with host systems managing documents together with their properties (so called *document stores*). These include Web portals, content management systems and certain business applications dealing with documents.

The basic motivation of this research and development work is to add reasoning capabilities to the Sense/Net Portal Engine, a commercial enterprise portal management system providing content management, application integration and collaboration facilities [7].[1] Our high-level goal is to use logic based techniques to achieve more intelligent behaviour of queries involving documents. This is done by providing a knowledge representation formalism for storing background knowledge, and using it to infer new properties within the document store, i.e. provide new pieces of meta-information about documents. These can then be utilised during query execution, a task delegated to the host system.

[1] We will refer to the Sense/Net Portal Engine as the *host system* in the sequel.

As an additional benefit, user interfaces provided by the host system can also become more intelligent, for similar reasons. For example, SREngine can be used (1) to infer that some documents are related to each other and (2) to represent this knowledge in the Document Store. Now, whenever the user navigates to one of these documents, the related documents can also be presented to her creating a much richer user experience.

The paper is structured as follows. In Section 2 we give a general introduction to the SREngine system describing the main components and their interactions. In the next section we discuss our rule language called SRLang, introducing the syntactic constructs and discussing the modelling decisions we have applied. In Section 4 we present the implementation details of the system. Section 5 discusses the preliminary test results. In Section 6 we examine related work. Finally, we conclude with a summary of our results.

2 Overview of the system

In this section we give a general overview of the SREngine system. First we introduce the notion of *document store* by describing its properties and its content. Next, we discuss the various usage scenarios of SREngine. Finally, we introduce the main components of SREngine: we discuss the general architecture of the system and describe the communication interfaces of SREngine.

2.1 The document store

At an abstract level, a document store is considered to be a graph, where the nodes represent *document-like entities* or *values*. A document-like entity is an object representing a document, the actual content of a document, a directory, a document category, or any other object related to documents. If this does not cause confusion, we will simply refer to document-like entities as *documents*.

The edges in a document store, called *properties*, run between documents or between documents and values. In the first case, properties represent relations between documents. For example, we can describe that document `A` is the draft version of document `B`. In the second case properties describe the *attributes* of the documents, such as the title, the authors, etc. We note that this model actually corresponds to existing knowledge representation formalisms, such as RDF [2].

Some properties have special roles. The property `has_binary` is used to connect a document with its content. We use this terminology as very often the content is given in a non textual, binary representation (e.g. in case of Microsoft Word documents). Property `has_child` is used to describe the hierarchical inclusion relation between the nodes, i.e. to express the fact that a document is located in a directory or that a document category is the subcategory of another. If we traverse the nodes along this relation we get a tree containing all the nodes of the document store, providing the basis for *path expressions* (see Section 3).

2.2 Usage scenarios

SREngine works with rules, written in a custom designed logic language, which describes how to infer new properties from existing ones, using information

retrieved from the actual content of the documents. For obtaining the latter we use *Information extractor* modules. These software components are specialised to extract certain kinds of information from the documents.

In the simplest scenario, SREngine is used to execute rules that are mostly based on the Information Extractor modules. In this scenario the system actually fills in those pieces of meta-information that the users of the host system have not supplied for some reason (e.g. because of lack of time, migration of the already existing document structure, etc.). Typically, this means that SREngine is used to fill in some document properties, such as the title, the authors, etc. The most important of these are the properties specifying the document classification, e.g. whether it is a contract, a technical note, etc.

In another scenario, rules are mostly used to infer new, complex relations between the nodes (or values of specific properties) of the document store using the available meta-information. These relations may easily represent information that one cannot expect the users to specify. For example, one can create a rule that counts how many other documents refer to the given document. This can be useful in certain situations, for example, when one would like to rank the documents based on their importance.

Practically, of course, SREngine is used in the mixture of the modes described above. The main point is that because the system derives new information it helps us to achieve our overall goals, i.e. to answer user queries more intelligently and to provide more user friendly interfaces towards users.

2.3 Architecture

Figure 1 shows SREngine as a black box focusing on its I/O behaviour. The system has two inputs: (1) the content of the document store and (2) a set of rules written in the SRLang language (see Section 3). Using these, SREngine produces instructions on how to change the content of the document store.

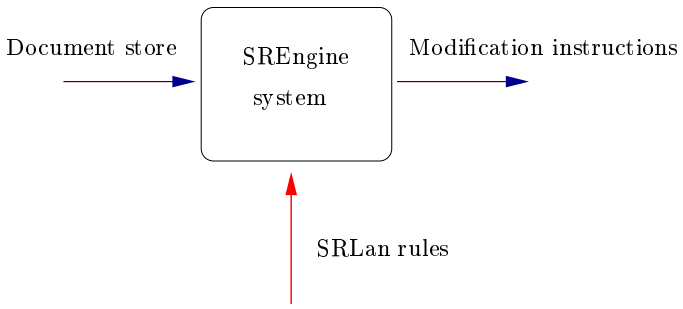


Fig. 1. Inputs and outputs of SREngine

The execution is batch-like, i.e. SREngine is supposed to be executed from time to time, preferably at a time when the host system is offline, i.e. its document store does not change during the reasoning process (see Section 7).

Figure 2 shows the detailed architecture of SREngine together with an outline of the Document Store. The boundary of the SREngine is indicated by a dashed line. Accordingly, SREngine consists of two main parts: the *Knowledge Base* and the *Reasoner*.

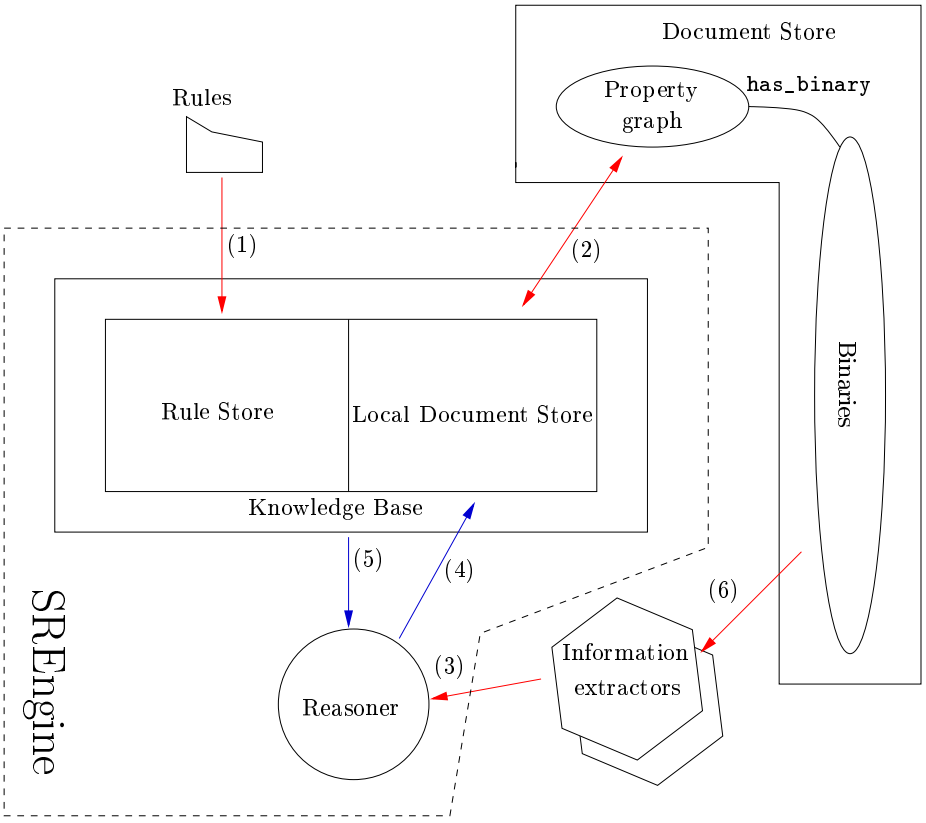


Fig. 2. The detailed architecture of SREngine

The Knowledge Base stores the rules and the content of (the relevant part of) the Document Store. This part excludes the binaries, i.e. SREngine only works with meta-information of documents and not with document themselves. The two parts of the Knowledge base are called (local) Rule Store and Local Document Store, respectively. The Reasoner works on the information available in the Knowledge Base and on the data the Information Extractors provide. The results are written back to the Knowledge Base (note arrow (4) from the Reasoner to the Knowledge Base).

Finally, the role of the Information Extractor components is to extract useful information, such as the name of the author, from the binaries of the documents in the Document Store. This usually involves using some kind of data extraction

technique. The extracted information is used by the Reasoner. Although the Information Extractors are also developed for the SREngine system, Figure 2 indicates that they belong to the host system. The reason for this is that the Information Extractors must communicate with the host system in a very efficient way as they work with huge amounts of data corresponding to the binaries (cf. arrow (6) in Figure 2). Thus, this component is expected to be part of the host system.

Once the reasoning process is finished, the content of the Local Document Store is synchronised with that of the Document Store maintained by the host system, as indicated by arrow (2) between these components in Figure 2.

2.4 Interfaces

SREngine uses several kinds of interfaces to communicate with the host system. In Figure 2 we have three such interfaces (1)-(3). Interface (1) makes it possible to populate the content of the Rule Store, while (2) does the same for the Local Document Store. Interface (3) defines the communication between the Reasoner and the Information Extractor component.[2] In the following we discuss these interfaces in more detail.

Rule Store interface SREngine supports two formats for reading SRLang rules: we are capable of processing rules given as text files as well as reading rules in XML format (see Section 3). Technically, the rules are read either from a file or via Web Service calls. In the latter case, the host system is expected to provide the appropriate Web Service which SREngine can invoke.

The interface specifies the details of such a Web Service, e.g. it must have a WSDL operation called `GetRules` responsible for returning the rules.

Document Store interface The Document Store interface serves two purposes. It allows us to import the content of the Document Store as well as to export the modifications back to the Document Store. This interface is implemented by a Web Service of the host system.

Because of the size of the Document Store, the interface specifies that the Web Service must be capable of returning the relevant content of the Document Store in parts. Namely, the Web Service must realise two operations: (1) `GetNumberOfNodes` which returns the number of nodes in the Document Store and (2) `GetNodesByID` returning nodes within a given interval.

Exporting the results is done by invoking the `UpdateProperties` operation of the Web Service. The input of this operation is a set of modifications that should be performed on the Document Store. This operation can be called multiple times, to support the option of performing the export operation in parts, whenever the reasoning process has too many results.

[2] As the Information Extractor is part of the host system its communication interface (6) is not covered here.

Information Extractor interface The Information Extractor is a fairly autonomous component in a sense that it is expected to be developed independently from the other parts of the system. As this component becomes more and more intelligent, i.e. new data extraction algorithms are built in, the users of SREngine are automatically able to use these when formulating rules.

To support this behaviour, we have created a registration mechanism which lets the Reasoner know about the *external calls*, i.e. calls that are executed not by the Reasoner, but the Information Extractor (see Section 3.3).

The registration is contained in the configuration part of the rules (see Section 3), where the user specifies what Web Service should the system invoke in order to execute a specific external call. Such a Web Service must implement two operations called `ex_property` and `ex_calculate`. The first is used for external calls where binaries of the documents are also involved, the latter is used in other cases. Examples for external calls are given in Section 3.

The reply of the Information Extractor always contains the status field which can have three different values: `success`, `failure` and `exception`. In case of success the Information Extractor may provide some return values.

3 The SRLan Rule language

In this section we introduce the rule language of SREngine. First we give some basic examples, then we explain the quantified operations available within the language. Next, we discuss the external calls and the first order logic semantics of the rules. The examples in this section describe rules that classify documents based on some criteria: the profession of their authors, their content, etc.

Throughout this section we use the text format of the rules as opposed to the much more verbose XML format. The text format is actually based on the Prolog syntax. Namely, the rules are Prolog expressions using appropriate operator declarations as connectives.

3.1 Basics of SRLan

The Local Rule Store consists of configuration elements and rules. Using configuration elements one can specify the *active properties*, i.e. new properties that have to be written back to the Document Store. An example can be seen below.

```
deducible(document_type).
deducible(has_author).
```

This describes that the results of the rules producing `document_type` and `has_author` properties will be exported to the Document Store. Other rules can be used during deduction, but their results are not exported directly (however, they can contribute indirectly to some exported results).

The other use of the configuration elements is to specify which Web Service to invoke in case of external calls (see Section 3.3). Below we can see two examples.

```
ex_calculate_location("contains", "http://152.66.71.114:1520/util").
ex_property_location("bulleted_list", "http://152.66.89.101/Tools").
```

Here we describe that in case of calls `contains` and `bulleted_list` the web services at the given locations should be invoked.

Now we turn our attention to the notion of SRLan rules. Similarly to Prolog, an SRLan rule consists of a *head* and a *body* separated by the `<==` character sequence. Below we show a simple example.

```
document_type(Document, 'category:/documents/scientific') <==
     has_author(Document, Author) and                          (1)
     has_profession(Author, 'jobs:/education/teacher').
```

This rule states that a document should be classified as a scientific document if it has an author who is a teacher. Conditions `has_author`, `has_profession` and `document_type` correspond to properties in the Document Store. The latter has a special importance as this is used to describe the category of a document, i.e. rules with heads like this are classification rules.

Identifiers given as Prolog atoms, e.g. `'jobs:/education/teacher'`, are called *path expressions*. These expressions identify nodes within the Document Store by navigating along the containment relation. In theory, the user could write node identifiers here as well, but practically, path expressions are more readable. For compactness, if this does not cause confusion, we will use a simplified version of path expressions in this paper: for example, we will simply write `teacher` instead of `'jobs:/education/teacher'`.

In our next example we classify a document object as belonging to the `contract` category if it is a Microsoft Word document, its binary (i.e. the document itself) is a bulleted list and it contains the word "contract":

```
document_type(Document, contract) <==
    has_binary(Document, Binary) and
    has_type(Binary, "word") and
    ex_property("contains", ["contract",Binary], _) and
    ex_property("bulleted_list", [Binary], _).
```

This rule is a good example for illustrating that we can use information extracted from the content of the documents when creating rules. The truth values of the `ex_property` expressions above are given by the Information Extractor component (cf. Figure 2). As we have seen above, the technical details on how to invoke such an operation can be described in the configuration part of the Rule Store. The content of the input parameters of an external call varies from call to call, so the expert should know with what parameters should she invoke the given operation. We will discuss external calls in more detail in Section 3.3.

SRLang also supports the use of negation as failure, i.e. the closed world variant of the classical negation. In the example below we classify a document as a `singleton` if it has a name and it does not have a known sibling having binaries.

```
document_type(Document, singleton) <==
        has_name(Document, Name) and
        not (
          has_sibling(Document, Sibling) and
          has_binary(Sibling, _)
        ).
```

3.2 Quantified operations

In SRLang we can express that a certain property holds for *at least one* element or *for all* the elements of a given solution set. The former corresponds to the existential (∃), the latter to the universal (∀) quantifier. For example, we can formulate a condition that *every* author of a document must have a specific property:

```
document_type(Document, art) <==
       forall Author in has_author(Document,Author)::          (2)
       has_profession(Author, artist).
```

Here we say that if all the authors of a document are artists, then we classify it as an art document.

The `forall` construct can also be used to express more complicated conditions. For example, we can formulate a rule with a nested `forall` condition requiring that all degrees of all authors of a document are of certain kind:

```
document_type(Document, engineering) <==
        forall Author in has_author(Document,Author)::
          forall Degree in has_degree(Author, Degree)::
          (
             has_type(Degree, "engineer") or
             has_type(Degree, "mathematician")
          ) and
          has_binary(Document, Binary) and
          ...
```

Generally a `forall` expression has the following form, where variables $\texttt{X}_1$, ..., $\texttt{X}_j$ must be present in expressions $\texttt{F}_1$, ..., $\texttt{F}_n$:

$$\texttt{forall X}_1\texttt{, ..., X}_j\texttt{ in F}_1\texttt{ and ... and F}_n\texttt{ :: T}_1\texttt{ and ... and T}_k \qquad (3)$$

This requires that we collect all solutions of $\texttt{F}_1$ `and ...` `and` $\texttt{F}_n$ in variables $\texttt{X}_1$, ..., $\texttt{X}_j$, and for each such solution we check that $\texttt{T}_1$ `and ...` `and` $\texttt{T}_k$ holds.

We have already seen examples with implicit existential quantification. For example, rule (1) classifies a document to belong to a given type if *at least one of the authors* has the specified profession.

For the sake of symmetry we also allow explicit existential quantification, listing the quantified variables. Accordingly, rule (1) can also be given in the following way:

```
document_type(Document, scientific) <==
       exist Author::
       has_author(Document, Author) and                    (4)
       has_profession(Author, teacher).
```

Using the `exist` construct also helps SREngine to warn the user in certain situations. Namely, if one of the variables in a condition is existentially quantified, but at runtime it is not instantiated, SREngine gives a warning to the user (cf. *floundering* in Prolog [9]).

The general form of the `exist` construct is analogous to that of the `forall` expression:

```
exist X1, ..., Xj in F1 and ... and Fn :: T1 and ... and Tk    (5)
```

3.3 External calls

SRLang gives us the possibility to access functions implemented in external libraries while evaluating the rules.

The `ex_property` calls are used to execute different kinds of data extraction algorithms on one or more binaries. An `ex_property` has three parameters. The first is a string describing the operation we would like to invoke. The second parameter is a list which contains references to binaries and possibly to other objects we need for the evaluation. The third parameter is also a list. This list will contain the results after a successful execution. In the following we show some examples for the usage of `ex_property`:

```
ex_property("contains", ["(pr(5),treaty or contract)",Binary], [])
ex_property("extract_author", [Binary], [Author])
ex_property("basic_metadata", [Binary], [Date, Format, Language])
```

In the first example we ask whether the given document contains the phrase `treaty` or `contract` in the paragraph 5. We can see that here we have two inputs and no output (this is denoted by the empty list). Note that the input arguments can be fairly complex. Handling these is the task of the program code responsible for the execution of the operation in question.

In the second example we extract the author of the document and put it in the variable `Author`. In this example we have one input and one output.

In the third example we extract several pieces of data from the document (the creation date, the format and the language of the binary) and put them in the variables of the output list.

The `ex_calculate` calls have the same syntax as `ex_property` calls: the first argument specifies the operation, the second and third arguments describe the

input and output parameters. The difference between the two kinds of external calls is that during the execution of an `ex_calculate` call binaries are not involved. This implies that while `ex_property` calls must be handled by an external component (as it has to efficiently access the binaries), `ex_calculate` calls can also be implemented within the SREngine itself. In fact, there are several predefined `ex_calculate` operations, including string manipulations, arithmetic, etc. Some examples are shown below.

```
ex_calculate("multiply", [A,B], [C])
ex_calculate("append", [FirstName, LastName], [Name])
ex_calculate("contains", ["bill", Name], [])
```

In the first example we multiply numbers `A` and `B` and expect the result to be assigned to variable `C`. The second example concatenates two names, while in the third we examine whether the string `Name` contains `bill` as a substring.

3.4 Semantics

The evaluation of a set of SRLan rules with respect to a given document store results in a set of new document properties. This set is expected to contain exactly those properties that are *entailed* by the first order logic equivalent of the SRLan rules in question.

We now discuss how to transform SRLan constructs to first order logic formulas. SRLang rules which do not contain negation and `forall` constructs correspond to simple Horn clauses. The variables in the rules correspond to logic variables, the literals, numbers, and path expressions (after evaluation) correspond to logic constants. The `<==` connective denotes implication, while the `and` construct corresponds to conjunction ($\wedge$). Accordingly, the example rule (1) corresponds to the following logic formula:

$$\begin{aligned}(\forall \text{Document}, \text{Author})(\text{document\_type}(\text{Document}, \text{scientific}) \;\leftarrow\; \\ \text{has\_author}(\text{Document}, \text{Author}) \wedge \\ \text{has\_profession}(\text{Author}, \text{teacher}))\end{aligned}$$

The `forall` construct corresponds to a special kind of universal quantification. For example, the logic form of rule (2) is shown below.

$$\begin{aligned}&\forall \text{Document}[\text{document\_type}(\text{Document}, \text{art}) \;\longleftarrow \\ &\forall \text{Author}(\text{has\_author}(\text{Document}, \text{Author}) \;\rightarrow\; \text{has\_profession}(\text{Author}, \text{artist}))]\end{aligned}$$

The general form of the `forall` expression shown in (3) corresponds to the following logic formula:

$$(\forall X_1 \ldots X_j)(T_1 \wedge \ldots \wedge T_k \leftarrow F_1 \wedge \ldots \wedge F_n)$$

The logical equivalent of the `exist` construct is the existential quantification. For example, rule (4) corresponds to the logic formula shown below.

$\forall$Document(document_type(Document, scientific) $\leftarrow$
$\exists$Author(has_author(Document, Author) $\wedge$ has_profession(Author, teacher)))

The general form of the `exist` construct introduced in (5) is equivalent to the following logic formula:

$$(\exists X_1 \ldots X_j)(F_1 \wedge \ldots \wedge F_n \wedge T_1 \wedge \ldots \wedge T_k)$$

4 Implementation

In this section we discuss implementation details of the SREngine system. First we give a high-level overview of the execution process. Next, we discuss the properties of the bottom-up reasoner SREngine uses: we describe how we stratify the rules to ensure a sound reasoning process.

4.1 Overview of the SREngine execution process

The operation of the SREngine is performed in the following 6 steps:

1. process the configuration files
2. fetch the content of the Document Store
3. fetch the rules
4. build layers
5. perform bottom-up reasoning
6. export the results

First, the system reads the configuration files where the general parameters are stored. These include important details on how to access the Document Store and from where to fetch the rules. In the next two steps we load these two sources, respectively. In step 4, we group the rules into *layers* (see more below). Next, we perform a bottom-up reasoning process and finally we export the results.

4.2 Bottom-up reasoning

In the SREngine system we have implemented a *bottom-up* reasoner. This choice, contrasting with the top-down execution mechanism of Prolog or any other resolution based logic programming system such as XSB, is justified by the fact that our task here is to produce *all* document properties that can be deduced from the document store and the rules (while in the top-down approach we look for the solutions of a single goal).

A further advantage of bottom-up reasoning is that it is much more robust than the top-down approach, e.g. in terms of ensuring termination. As no attention needs to be paid to this aspect, rules can be formulated more freely and do not require Prolog expertise.

A disadvantage is that using bottom-up reasoning in the presence of negation as failure or the `forall` construct raises some problems. For example, even

though at a given point it seems that a property holds for *every* author of a specific document (and thus the corresponding rule can be applied) this cannot be taken for granted. This is because a bottom-up reasoner can infer *later* that the document has another author who may not have the desired property.

These problems are actually avoided as we only allow set of rules that can be stratified [1] with respect to negation and `forall`. In this case the rules can be divided into layers in such a way that if a rule calls another through negation or `forall`, then this other rule has to be placed in a lower layer than the calling rule. If such a stratification can be found, then bottom-up reasoning can be safely applied to the layers one-by-one, starting from the lowest.

In SREngine we apply a slight generalisation of this technique. Namely, when building the layers, we consider every kind of dependency between the rules, not only the calls via negation and `forall`. This strategy is discussed below.[3]

4.3 Layers

In SREngine, the layers are created in the following way. First we build the *dependency graph* G_d from the actual set of rules. Each node in G_d corresponds to a rule. An edge from node A to node B represents the fact that the execution of B may depend on the execution of A, i.e. A can possibly produce new facts which can trigger the rule corresponding to node B. As an example, the dependency graph of the rules shown in Figure 3 is presented in Figure 4. Here the three rules producing `has_author` properties are denoted by `has_author1`, `has_author2` and `has_author3` respectively.

```
has_author(Document, Creator) <==
        has_creator(Document, Creator).
has_author(Document, hesse) <==
        has_binary(Document, Binary) and
        has_title(Document, Title) and
        ex_calculate("contains", ["demian", Title], []).
has_author(Document, Contributor) <==
        has_contributor(Document, Contributor).

has_contributor(Document, Contributor) <==
        has_author(Document, Contributor) or
        is_related(Document, Contributor).

document_type(Document, scientific) <==
        forall Contributor in has_contributor(Document,Contributor)::
        has_profession(Contributor, teacher).
```

Fig. 3. A sample set of rules

[3] We realise that excluding non-stratified sets of rules is a strong restriction. However, we argue that (1) in the context of document management stratified rule sets seem to be sufficient and (2) this simplification makes the reasoning more scalable.

Having constructed G_d, our next step is to partition this graph according to its strongly connected components. A strongly connected component (SCC) is such a subgraph of G_d in which for every pair of vertices A and B there is a path from A to B and also a path from B to A. In Figure 4 the strongly connected components are marked by dash-lined boxes.

Based on the SCCs we build a reduced graph G_r. The vertices of this graph are the strongly connected components of G_d. There is an edge in G_r from A to B if and only if there is an edge in G_d from one of the vertices in the SCC corresponding to A to one of the vertices in the SCC corresponding to B.

As G_r is acyclic by definition, a *topological ordering* of G_r can be constructed, i.e. a full ordering on the nodes such that if there is an edge from A to B, then A precedes B in the ordering.

The graph G_r and a topological ordering of the nodes of G_r is used for stratifying the initial set of rules. The set of rules corresponding to a node of G_r will form a stratum. The ordering of the strata is determined by the topological ordering of G_r, so that the lowest value forms the bottommost layer, and so on. For example, a valid topological ordering of the graph in Figure 4 is indicated by the numbers in the figure: layer 1 is the bottommost, layer 4 is the topmost.

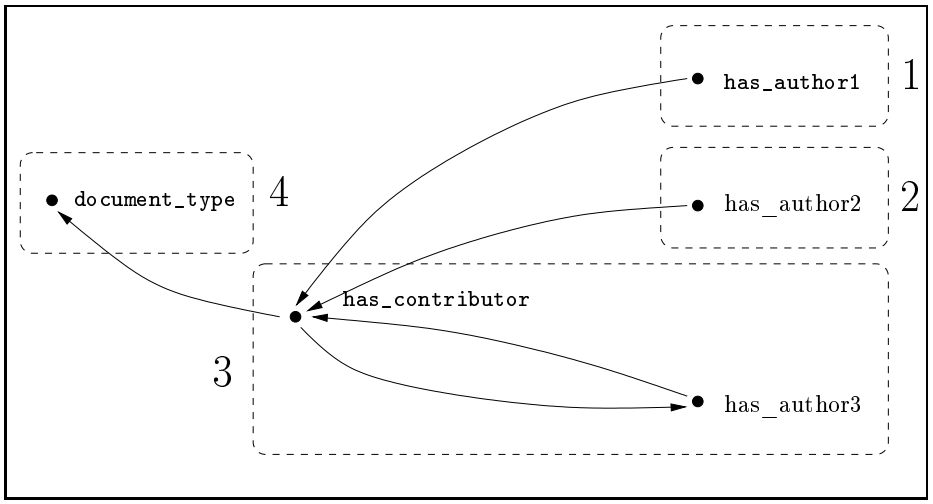


Fig. 4. The dependency graph of rules in Figure 3

Given a stratification we execute the rules in the bottommost layer until they stop deducing new properties. We then repeat this process for all layers upwards.

5 Performance Evaluation

We have carried out an initial performance evaluation with a relatively small number of rules, but huge amounts of meta-information in the document store. This is a typical setup as in real applications the content of the document store is several magnitudes larger than the number of rules.

For our tests, the document stores were randomly generated. To each of these we applied a fixed set of rules. This sample rule set, containing 10 rules, was chosen to include the most important SRLang constructs, such as recursive rules, external functions, `forall`, etc.

The tests were executed in the following hardware and software environment: Pentium-M 1.7GHz, 512MByte memory, Ubuntu Linux, SICStus Prolog 3.12.5.

The left hand side of Table 1 shows the properties of the document stores we used for testing. The columns show the size of the document store, the number of properties it contains and the average branching factor. This factor describes how many outgoing edges a node in the property graph contains on average.

Size	# of prop	Branch	New prop	Load time	Exec time
1.7KB	33	5	8	0.01sec	0.01sec
3.3KB	81	5	16	0.02sec	0.01sec
19KB	541	6	80	0.06sec	0.05sec
348KB	10194	7	1315	0.75sec	1.06sec
697KB	20443	7	2641	1.40sec	0.95sec
1.2MB	35837	4	6506	2.50sec	2.30sec
2.4MB	69287	6	9889	3.60sec	3.50sec
3.9MB	112528	8	13299	5.10sec	4.60sec
12.6MB	359833	4	65491	13.60sec	23.70sec
25.6MB	720177	4	130683	25.50sec	50.50sec

Table 1. Results of the performance analysis

The right hand side of Table 1 shows the results of the test. Here, the first column indicates the number of new properties deduced by the SREngine during the execution. The second column contains the time it took to read the content of the document store. Finally, the last column contains the execution time, i.e. the time needed for the deduction.

The results clearly indicate that for a relatively small set of rules SREngine has a fairly good performance even for big document stores.

6 Related work

Document classification in general is a huge field in computer science. Lots of approaches have been applied in the past few decades involving machine learning, neural networks, pattern recognition, natural language processing and other AI

related fields [8,6]. There are also numerous approaches which rely on some form of rules [3,4]. The common properties of these systems is that they aim to cluster a given collection of documents into groups that have similar contents.

In contrast with this, in SREngine the classification rules are not necessarily used to classify documents because of similar content. For example, a category could be "Books of Hermann Hesse", where an SRLang rule may classify a document to belong to this category just using the author meta-information. It is obvious that the books of Hermann Hesse do not necessarily have similar content from the traditional document classification point of view.

More closely related to our approach are the rule-based languages working on some kind of graph structure. This is, because we think that document stores can easily be represented in an appropriate graph description language, such as the RDF framework [2] used by the Semantic Web community. For RDF, several rule languages have been proposed such as the TRIPLE [11], the SWRL [5] or the RuleML [10], all of which are based on Horn logic, similarly to SREngine. These languages are expressive enough to capture most of the rules an expert would like to formulate in the context of document classification. However, we argue that SREngine provides a viable alternative to other systems because of the intuitive syntax, the modelling constructs (like `forall`), the bottom-up execution mechanism and the document extraction capabilities the system has.

7 Future Work

Up to now, SREngine supports only "positive" consequences, i.e. the result of the reasoning process is the list of new edges that should be added to the document store. As an extension, we plan to provide the user the ability to create rules that lead to "negative" consequences, i.e. they can *delete* metadata from the document store by specifying appropriate rules.

So far we have considered all the properties multi-valued, i.e. a given document object can have several edges attached to it with the same property. However, some properties are typically single valued, such as the `has_binary`. Handling these and the related consistency problems is future work.

Finally, we need to find practical applications where SREngine is used in real corporate environments providing us relevant feedback about the system usability, performance and further extension possibilities.

8 Conclusion

In this paper we presented the SREngine system, which is a generic framework that can be used to infer new document properties in a document store, using a set of rules.

We have presented the main components of SREngine: the Knowledge Base which stores the rules and the relevant parts of the document store, the Reasoner, which performs a bottom-up reasoning process to infer new properties and the Information Extractor, which provides data extraction functionality, to be used when formulating rules. Here, we have also introduced the interfaces SREngine uses for communicating with the host system managing the document store.

We described the SRLang language, which is a Prolog based language for describing user friendly rules involving documents. We have introduced the main constructs, the quantified operations and the external calls. We have also presented the semantics of SRLang rules.

Next, we have discussed the bottom-up reasoning process we use to execute SRLan rules. Finally, we have evaluated the implementation of SREngine by presenting some basic performance results.

As a conclusion we believe that SREngine provides a robust, extensible framework for deducing new meta-information about documents. SREngine does not require Prolog expertise and can be used together with various host systems.

Acknowledgements

We acknowledge the support of the Sense/Net Ltd. (http://www.sensenet.hu) and of the Hungarian research programme GVOP 2005/3.3.3. We also would like to thank all the people participating in this project, especially Gábor Baráth.

References

1. Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.
2. Dave Beckett. RDF/XML syntax specification (revised). W3C recommendation, February 2004.
3. Yixin Chen and James Z Wang. Support vector learning for fuzzy rule-based classification systems. *IEEE Transactions on Fuzzy Systems*, 11(6):716–728, 2003.
4. Hui Han, Eren Manavoglu, Hongyuan Zha, Kostas Tsioutsiouliklis, C. Lee Giles, and Xiangmin Zhang. Rule-based word clustering for document metadata extraction. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1049–1053, New York, NY, USA, 2005. ACM Press.
5. Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining owl and ruleml. Technical report, W3C Member submission 21 May 2004.
6. Thorsten Joachims. Text categorization with support vector machines: learning with many relevant features. In *Proc. of the 10th European Conference on Machine Learning*, number 1398, pages 137–142, Chemnitz, DE, 1998. Springer.
7. Sense/Net ltd. Sense/Net Portal Engine. http://english.sensenet.hu/, 2007.
8. Kamal Nigam, Andrew K. McCallum, Sebastian Thrun, and Tom M. Mitchell. Text classification from labeled and unlabeled documents using EM. *Machine Learning*, 39(2/3):103–134, 2000.
9. Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pages 442–453, 1994.
10. Michael Schroeder and Gerd Wagner, editors. *Rules and Rule Markup Languages for the Semantic Web, Second International Workshop, RuleML 2003, Sanibel Island, FL, USA, October 20, 2003, Proceedings*, volume 2876 of *Lecture Notes in Computer Science*. Springer, 2003.
11. Michael Sintek and Stefan Decker. Triple - a query, inference, and transformation language for the semantic web. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 364–378, London, UK, 2002. Springer-Verlag.

Design for a Parallel and Distributed Hybrid Constraint Programming Library

Luís Almas[1], Rui Machado[2], and Salvador Abreu[3]

[1] Deimos Engenharia S.A., Portugal
lalmas@deimos.com.pt
[2] IBM Deutschland Entwicklung GmbH, Germany
machador@de.ibm.com
[3] Universidade de Évora and CENTRIA FCT/UNL, Portugal
spa@di.uevora.pt

Abstract. Constraint programming libraries are useful when building applications developed mostly in mainstream programming languages: they do not require the developers to acquire skills for a new language, providing instead declarative programming tools for use within conventional systems. Some approaches to constraint programming favour completeness, such as propagation-based systems. Others are more interested in getting to a good solution fast, regardless of whether all solutions may be found; this approach is used in local search systems.
Parallel architectures are becoming more commonplace, partly due to the large-scale availability of individual systems but also because of the trend towards generalizing the use of multicore microprocessors.
We propose an architecture for mixed constraint solvers, relying both on propagation and local search, which is designed to function effectively in a mixed shared and distributed-memory multiprocessor system.

1 Introduction

Constraint Programming is a useful declarative methodology which has been applied in several ways:

1. As an extension to existing programming languages, such as Prolog, taking advantage of the complementarity provided by the two approaches (backtracking vs. propagation). This is the case for most CLP implementations.
2. As a library in which constraints become data structures of the host language, which are operated on by the library procedures. This is the case, for instance, for ILOG Solver [9] and AJACS [5].
3. As a special-purpose language, appropriate for solving problems formulated as constraints over variables. This is the case with, among others, Oz [11], OPL [6] or Comet [7].

The declarative nature of constraint satisfaction problems strongly suggests that one try to parallelize the computational methods used to perform the tasks related to solving CSPs, namely propagation. Indeed, this has been explicitly incorporated into most languages mentioned in point 3, which provide mechanisms to promote distributed execution of various aspects of the process.

In the work we present herein, we chose to follow approach number 2: to provide a library for constraint programming for an existing language. AJACS/C is inspired by the scheme used in AJACS [5] and extends it to include both propagation and local search techniques. AJACS/C relies on a purely functional approach to representing search-space state stores, and is designed to ensure that parallelization is viable on distributed-memory multiprocessors. These requirements translate to the following architectural options, which are carried over onto the present work:

- Should there be the need for data structure sharing, it will be done for read-only access: the values are viewed as single-assignment.
- While exploring a search space, there is no proper backtracking to speak of: a store derived from another one is actually a partially modified copy of its ancestor.

In terms of its memory model, AJACS relies on a distributed shared memory (DSM) system, operating under a special JVM implementation, Hyperion [2], which compiles to C. The target code then uses the PM2 multithreading library, over which a DSM implementation has been constructed [1] and is used to share memory ranges (in the form of Java objects), under an appropriate consistency model.

For AJACS/C, the most significant departure from this model – other than the fact that the host language is C, rather than Java – lies in the approach to data structure sharing: in AJACS we assumed a shared address space, whereas in AJACS/C this requirement will be relaxed. To recap, in AJACS data produced by one thread is directly available to others, at the same address. This is made possible by the DSM system, which creates the illusion of a common adressing space for multiple processes running on separate, network connected, machines. In the case of PM2-DSM, the network layer could be one of several modalities, including TCP/IP and VIA.

The work on AJACS and its experimental assessment led to the conclusion that Hyperion and PM2-DSM were not a good match for a distributed constraint propagation library: the performance was too frequently disappointing. The reasons for the lackluster results could be attributed to several reasons, the most relevant one being that it is very difficult, if not impossible, to control *where* a given piece of data (e.g. a CSP store) is located, both in terms of address space and host processor.

In AJACS/C we decided to "take matters into our own hands," as all DSM aspects were confined and gradually removed from AJACS and the system is now completely rewritten to respond to the following requirements:

- The implementation (in C instead of Java) relies on a much simpler runtime organization.
- Initially we rely on a memory organization that reproduces the AJACS approach, i.e. one in which shared data structures are replicated using the DSM system,

– Instead of relying exclusively on DSM and trying to adapt to its best behaving memory usage patters, we will gradually resort to simple message-passing as provided by the communication layers, for instance Madeleine [3].

The remainder of this position article is organized as follows: in section 2 we present a revised implementation of AJACS/C, which provides the basis for a distributed constraint solver library. We then describe in section 3 an evolution of the previously presented system, overhauled to perform well in a particular architecture: the Cell/B.E.processor. We then outline some of the present results and describe the lines along which work is progressing.

2 A Distributed Constraint Solver Library

AJACS-C presents an adaptation of the AJACS [5] system to the C programming language as an adequacy study of this approach to constraint solving in a distributed environment, using the PM2 distributed shared memory capabilities.

The goals underlying the development of the AJACS-C system are:

– To develop a Constraint Solving System in the C language (the native language of the PM2 experimented Parallel library).
– To adapt and experiment this Constraint Solving System to a distributed environment using PM2-DSM;
– To evaluate the PM2-DSM adequacy for distributed constraint programming by testing different distributed approaches and at the same time try to obtain run-time performance speedups.

2.1 Brief Introduction to PM2-DSM

PM2 (Parallel Multithreaded Machine) [8] is a multithreaded environment for distributed architectures. It provides a POSIXlike interface to create, manipulate and synchronize lightweight threads in user space, in a distributed environment. Its basic mechanism for internode interaction is the Remote Procedure Call (RPC).

PM2 includes two main components. For multithreading, it uses Marcel, an efficient, userlevel, POSIXlike thread package. To ensure network portability, PM2 uses an efficient communication library called Madeleine [3], which was ported across a wide range of communication interfaces, including highperformance ones such as BIP, SISCI, VIA, as well as more traditional ones such as TCP, and MPI.

PM2-DSM provides the illusion of a common address space shared by all PM2 threads irrespective of their location and thus implements the concept of *Distributed Shared Memory*, page based, on top of the distributed architecture of PM2. But PM2-DSM is not simply a DSM layer for PM2: its goal is to provide a portable implementation platform for multithreaded DSM consistency protocols. Given that all DSM communication primitives have been implemented using PM2's RPC mechanism based on Madeleine, PM2-DSM inherits PM2's wide network portability.

2.2 An Overview of AJACS- C

In its organization AJACS-C is aimed (similarly to AJACS itself) at producing independent states as result of state expansion, independent in the sense that each store (plus the constraint problem containing the constraints themselves) carries all the information necessary to be considered a possible solution for a given problem. By way of this state independence, AJACS-C does not need to use backtracking.

The state independence is the basis for a distributed concept to take shape since in theory it shall be possible to parallelize constraint problem solving by spreading each produced state(s) among several processing units without too much foreseen interaction. This way all processing nodes should be able to 'walk' through the problem space with minimal knowledge or awareness of each other.

The minimal information each processing node requires, for its state iteration and propagation, is to know:

- where to look at for new states to search;
- where to store the expanded new stores, i.e. the states that resulted from a successful propagation;
- where and how to signal the eventually found solutions to the problem master controller.

2.3 Experimenting with AJACS-C over PM2-DSM

Before we go about discussing the performance of AJACS-C implementations and distributed execution strategies, it is important to mention that we followed two distinct approaches, both relying on DSM, but with definitely different sharing patterns: one which we called *centralized* and another which we designated as *local*. The two sections that follow discuss these approaches and their differences.

Centralized Distributed Pattern This Pattern designates one cluster node as the **master node** and the remaining as the **worker nodes** – different nodes mean different machines in the configured cluster. The master and worker profiles are triggered/enrolled at run-time execution where the first configured cluster node will assume the master profile and the remaining the worker profile.

The idea behind the master profile is for it to maintain a central DSM data structure (therefore the name for the pattern) that holds all the current, still to be investigated, problem states. All nodes have write and read access to this central structure. Every worker will be allowed to get stores (new jobs) and put stores (the resulting state produts of the last iteration and propagation).

For a more complete and efficient approach the designed pattern will make sure that, after the initialization phase comprehending the central data structure creation and remote execution of all the threads on all nodes, the node that took the master profile will spawn an additional local thread to endorse the worker profile, this way all machines/nodes will behave as workers after the initial initialization step has been performed.

This way the centralized data-structure will be the communication link between all the nodes, the PM2-DSM coherence protocol will abstract the user to the synchronization overhead management of the shared structure and assure the correct access and behaviour from all threads in all nodes in a safe and coherent manner.

```
init_states = search_initial_states(sInit);
dsm_list.put(init_states);
FOR i = 1 TO n
      Worker[i] = new RemoteWorker(dsm_list);
Worker[0] = new LocalWorker(dsm_list);
WHILE <not_all_finished()> DO
      wait();
printSolutions();
```

Fig. 1. The master thread in the Centralized Distributed Pattern

```
WHILE <dsm_list not empty> DO
      j = dsm_list.get();
      L = search_solutions(j);
      FOREACH l in L DO
              IF <l is solution>
              THEN print_solution(l);
              ELSE dsm_list.put(j);
```

Fig. 2. Worker threads in the Centralized Distributed Pattern

Figures 1 and 2 display simplified versions of the algorithms associated with these processes.

Local Distributed Pattern In this distribution model all workers have a dedicated DSM local data structure to manage the expansion of the new states, so every worker gets and puts jobs directly from/into its local data stucture. On execution start the initial state is spawned into its child states and those are distributed among all the workers on a round-robin fashion. After this point all workers start their search independently.

Figures 3 and 4 display simplified versions of the algorithms associated with these processes in the local distributed pattern.

```
init_states = search_initial_states(sInit);
FOREACH k in initial_states DO
        i=0 to n
        Worker[w].local_dsm_list.put();
        w = (w + 1) % number_nodes
```

Fig. 3. The master thread in the Local Distributed Pattern

```
WHILE <local_dsm_list is not empty>
      nextS = local_dsm_list.get();
      L = search_solutions();
      FOREACH k in L DO
            IF < k is solution >
               THEN printSolution(k);
            ELSE local_dsm_list.put(k);
END
```

Fig. 4. The worker threads in the Local Distributed Pattern

2.4 A Simple Example - N Queens

Specification The N-queens puzzle is the problem of putting `N` chess queens on an `N` by `N` chessboard such that none of them is able to capture any other using the standard chess queen's moves. The colour of the queens is meaningless in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

Problem Description On this problem one "queen" position on the chess board maps directly to one **Variable**, so the **Store** structure will hold as many variables as the number of queens specified.

As for **Constraints**, and as for specification, the relation between the queens (variables) will be such that on a given solution no queen can "attack" any other queen. The constraint "noattack" is then designed as relation between two queens. For the `N`-queens problem there will be `N*(N-1)/2` "noattack" constraints.

Search, in AJACS-C PM2-DSM and for the Queens example, will be the successive handling of available stores (spawning from sInit) using a Breadh-first search / FIFO approach. After the split of some store, the child stores are appended to the tail of the stores list. Note: FIFO was the chosen approach but LIFO / Depth-First search could also be used. FIFO was prefered due to the fact that we are doing full tree traversal, i.e. looking for all possible solutions for a problem. LIFO could be better suited if we where insterested in just searching for a first solution, this happens not to be the case.

In each search step the adopted **Strategy** is to look for the first non-ground variable available on the current store (using a top-down look-up). That variable

is chosen to be the next to be iterated upon and reduced, that is, all singleton values of that variable are successively tested, triggering propagation on the rest of the store. If propagation is deemed successful, the resulting store will be added to the search list for further search inspection.

The search & strategy steps are repeated until there are no more stores available to iterate on.

In the case of the Queens example, the **Problem** solutions will be the collection of stores that contain only ground (singleton) values. The remaining stores are the ones that survived all the iterations and propagations without getting to a contradiction (a variable without any allowable values: an empty domain) and that satisfy the "noattack" constraints, no queen can attack another.

3 Hybrid Multicore Solver Library

Multi-core microprocessors have become the direction in the industry for microprocessor design. One such microprocessors is the Cell Broadband Engine [10].

Multi-core advantages include a better ratio of performance to power usage, less heat dissipation, and a smaller physical footprint. But such architectures pose new challenges to the software level which must be written with multiple cores in mind - a time-consuming and difficult task known as parallel programming.

Our work tries to match today's architectures tendency to parallelism and AJACS's characteristics in order to get a declarative approach to sofware development in a parallel environment while extracting good performance from such architectures in constrained problem solving.

3.1 The Cell Broadband Engine

The Cell Broadband Engine (Cell/B.E.) is a multiprocessor core design with nine processor cores. The architecture is heterogeneous, consisting of two different core types. The principal core, the 64-bit Power Processor Element (PPE), is a PowerPC processor assuming a supervisory role and more keen to deliver system-wide services. The eight Synergistic Processor Element (SPE) cores are the computational workhorses. SPEs are accelerator cores implementing a novel, pervasively data-parallel computing architecture based on SIMD RISC computing and explicit data transfer management. A SPE has two components: the Synergistic Processing Unit and the Synergistic Memory Flow Controller (MFC), both responsible for independent computation and data transfers, respectively.

The PPE accesses main storage (the effective-address space) with load and store instructions that move data between main storage and a private register file, the contents of which may be cached. The SPEs, in contrast, access main storage with Direct Memory Access (DMA) commands that move data and instructions between main storage and a private local memory, called a local store (LS). A SPE's instruction-fetches and load and store instructions access its private LS rather than shared main storage, and the LS has no associated cache.

The nine cores, memory and I/O controllers are connected by the Element Interconnect Bus (EIB). This high bandwidth bus consists of four 16-byte-wide data rings through which the processor elements can drive and receive data simultaneously.

Although regarded as a very powerful processor, the Cell/B.E. presents great challenges to programmers. One architectural aspect is the small high-speed local store at each SPE. The local store has a limited size of the range of L2-cache sizes (256 KB for the first generation Cell/B.E. processor) and must be software managed. Another aspect is the existence of two instruction sets, one for each processor type. Presently this means having to work with two different toolchains.

3.2 AJACS/C(ell)

The similarity between the Cell/B.E. architecture and the AJACS model has some striking aspects to it. The same terms are used to name the different entities: controller and worker. In Cell/B.E., the PPE can be seen as the controller processor while the SPEs are the workers. In AJACS, there is also a controller agent for the problem and several workers who try to find a solution. Therefore it is a natural choice to make the PPE responsible for the master role and the provide the SPEs with the worker role in the AJACS model.

In our approach, the PPE (assuming the controller role) sets up the working environment. This means carrying out the following steps:

1. **Do a first expansion of the search tree**. This is done according to the AJACS model. Each worker will have a different branch of the tree to work on.
2. **Create the SPE contexts**. This is the typical scenario when writing Cell/B.E. applications. Create the SPE contexts that represent a logical SPE.
3. **Setup the information to be passed to the SPEs**. The PPE (controller) and the SPEs (workers) must share some data. This data is stored in the main memory to be easily and quickly accessed by all processors. When setting up all the environment, the PPE must provide the location of the common data to the SPEs for these to be able to fetch it via a DMA transfer. The supply is done through a control block holding all the information.
4. **Create pthreads that manage the contexts**. In order to have concurrent SPE contexts, POSIX threads (pthreads) are used. Basically, each pthread runs a SPE context or in other words, a worker. Thus there are as many pthreads as workers and contexts.
5. **Wait for all to finish**. For now, the controller waits for all workers to finish, does some cleanup and exits. This wait simply means to wait for the created pthreads to terminate.

The SPEs assume the role of workers. The steps performed by the each worker can be summarized as follows:

1. **Get the information block from main memory**. The worker needs the data in order to carry out its process. Hence, the very first step done by the worker needs to be getting the control block with all the information, the one which was setup by the controller/PPE on steps 2 and 3.
2. **Get the Problem**. Once it knows the location of what it needs, the first thing the worker fetches is the Problem data structure itself.
3. **Look for solutions**. After all the setup needed has been performed, the worker enters a loop:

```
while (there's work to be done)
  dma transfer another store to work on;
  invoke the search on the transfered store;
  decrement the amount of work to be done;
```

   in which it performs the actual steps in solving the constraint problem.

It is noteworthy that any fetching of data by the SPEs implies doing a DMA transfer. In Cell/B.E., the SPEs can only fetch information from main memory via asynchronous DMA transfers. There is a lot of tuning to be done concerning to this and some space for optimization.

One of the informations in the control block is the number of stores to be worked on. As long as the number of (yet unsolved) stores in the work queue is greater than zero, the worker loops.

The search step is the most important. The current *Strategy* takes one `Store` to process and partitions it in two complementary ones in which one `Store` holds a ground value in the `Variable` chosen by the *Strategy* while the complementary `Store` has all the remaining possible values for that Variable, i.e. all excluding the chosen one. One should note that this Strategy is only a simple possibility since the AJACS model allows for the definition of new Strategies when choosing a variable.

After choosing a variable, the propagation is executed on the Store which has the ground variable. Here three things can happen: we have a solution, the propagation has succeeded or the propagation failed. In case a solution was found or we came up with failed propagation we simply continue with the complementary Store; if the propagation succeeded, we continue to work on the same Store and put the complementary one in the work queue.

This "*always forward and down in the tree*" approach saves a lot of space which is a scarce resource in the SPE's Local Store.

So far, our search is exhaustive which guarantees us completeness. But sometimes this is not so important and local search methods provide a very fast way to get a solution. Adaptive search is a heuristic method in which the key idea of the approach is to take into account the structure of the problem given by the description, and to use in particular variable-based information to design general meta-heuristics.

We extended our propagation-based search with a local search component. At a certain state in the complete search – up to which we can guarantee completeness – we switch to adaptive search and its heuristic method, by taking the

the so far grounded variables as constants and a random value from the domain of the non-ground variables as starting points for the search procedure.

4 Initial Assessment and Progress Report

The PM2-DSM based implementation of AJACS-C has been in use and is fairly well tested. It provides a reasonable performance increase over AJACS, particularly in the local distributed pattern variant. The code was used as a basis for the Cell/B.E. implementation, to which we added support for hybrid constraint satisfaction solution by integrating a local search component.

We are presently scaling up the test cases for AJACS-C on the cluster system. The recent availability of a larger test system (a 12-node dual Opteron cluster) will improve our ability to evaluate how the DSM-based approaches behave.

The Cell/B.E. implementation is still at an early stage and we cannot really comment on performance or usability yet, although the initial results seem promising. The issues which differentiate the Cell/B.E. library design from AJACS-C and which we expect to bring about more interesting results have to do both with its expected performance increase due to switching to local search from a certain point onwards but also the avoidance of DSM, which has turned out to be complex to efficiently master.

Issues to be dealt with, in particular for the Cell/B.E. version, include:

1. Single-source: the Cell/B.E. requires two toolchains to be used, one for the PPE and another for the SPEs. This is particularly relevant for programming propagators and other constraint procedures, which must be usable in both kinds of context.
2. Differentiated use for the SPEs: at present, they simply perform all the tasks an AJACS worker has to carry out. It is in our plans to experiment with different roles for each SPE, possibly creating pipelines among these to exploit the Cell/B.E.'s inner bus, which has a very high bandwidth. A possible approach is to split SPE functionality into selector and propagators, for instance.
3. Another concern which we are looking forward to have an effective answer to is the class of problems which can be modelled using the Cell/B.E.'s limited SPE memory, i.e. whether the problems which we can fit into SPEs have a sufficiently complex processing associated with them to result in a significant performance gain for the overall constraint solving goal.
4. Mixing the single-Cell/B.E. solver with other instances thereof is another line which we are following: there are dual-Cell/B.E. blade systems which provide shared memory (albeit NUMA). These provide one first level of distribution outside a single Cell/B.E. processor and represent a shared memory layer similar to the original AJACS organization: stores (or problems, as per AJACS terminology) may be shared among different processors. A further distribution layer can be obtained when we consider a network of such blades, falling back onto the AJACS-C model introduced earlier in this article.

In short, the initial port of AJACS to C, based on PM2-DSM and its message-passing counterpart, be it the one based on the Cell/B.E. processor or otherwise, is undergoing active development and we expect to be able to have more significant experimental results soon.

5 Acknowledgements

The authors would like to thank Daniel Diaz for helping us with the port of the adaptive search library [4] to the Cell/B.E. processor. Dr. Werner Kriechbaum and IBM Deutschland Entwicklung GmbH are acknowledged for supporting Rui Machado and providing the necessary resources for the work described herein pertaining to the Cell/B.E. system to have been carried out.

Disclaimer

References

1. Gabriel Antoniu and Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, volume 2026 of *Lect. Notes in Comp. Science*, pages 55–70, San Francisco, April 2001. Held in conjunction with IPDPS 2001. IEEE TCPP, Springer-Verlag.
2. Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27:1279–1297, October 2001.
3. Olivier Aumage. Heterogeneous multi-cluster networking with the Madeleine III communication library. In *Proc. 16th Intl. Parallel and Distributed Processing Symposium, 11th Heterogeneous Computing Workshop (HCW 2002)*, Fort Lauderdale, April 2002. Held in conjunction with IPDPS 2002. 12 pages. Extended proceedings in electronic form only.
4. Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In Kathleen Steinhöfel, editor, *SAGA*, volume 2264 of *Lecture Notes in Computer Science*, pages 73–90. Springer, 2001.
5. Lígia Ferreira and Salvador Abreu. Design for AJACS, yet another Java Constraint Programming framework. *Elsevier Electronic Notes in Theoretical Computer Science*, 48, 2001.
6. Pascal Van Hentenryck, Laurent Perron, and Jean-Franois Puget. Search and strategies in opl. *TOCL*, 1(2), October 2000.
7. Laurent Michel, Andrew See, and Pascal Van Hentenryck. Distributed constraint-based local search. In Frédéric Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 2006.

8. Raymond Namyst and Jean-Franois Mhaut. PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.
9. Jean-Francois Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In *ILPS*, pages 513–527, 1995.
10. IBM Systems and Technology Group. *Cell Broadband Engine Programming Tutorial*. IBM Corporation, June 2006.
11. Peter Van Roy and Seif Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, November 1999. Part of International Conference on Logic Programming (ICLP 99).

A portable and efficient implementation of global constraints: the `tree` constraint case

Guillaume Richaud, Xavier Lorca, and Narendra Jussien

École des Mines de Nantes, LINA FRE CNRS 2729, FR – 44307 Nantes Cedex 3
{guillaume.richaud,xavier.lorca,narendra.jussien}@emn.fr

Abstract. Global constraints represent invaluable modeling tools for Constraint Programming (CP). Efficiently solving recurrent subproblems is a key point for CP successes. However, global constraints mainly remain strongly attached to a given constraint solver. Indeed, they heavily rely on internal mechanisms in order to be as efficient as possible. In this paper, we emphasize the interest of decoupling global constraint implementations from the underlying solver. We show, on a `tree` constraint, that even more decoupling it by providing fully dynamic algorithms enhances efficiency and, which is much more important, allow an efficient portability of the constraint. We illustrate this for the *Choco* and *Gecode* solvers.

1 Introduction

Constraint Programming (CP) is an ever evolving field whose aim it is to solve combinatorial problems in a declarative and flexible paradigm. At the heart of a constraint program is a constraint satisfaction problem (CSP) which is defined by a set $\mathcal{V} = \{v_1, \ldots, v_n\}$ of variables (in the mathematical sense), a set $\mathcal{D} = \{dom(v_1), \ldots, dom(v_n)\}$ of domains which represent the set of possible values that each variable can take, and a set $\mathcal{C}$ of constraints (relations) upon subsets of variables. A solution for a CSP is a variable assignment (a value for each variable) that simultaneously satisfies the constraints of the problem. A constraint solver is meant to look for such a solution. End-users of constraint programming only need to enunciate the variables and the constraints of their problem.

In this context, *global constraints* represent invaluable modeling tools for the CP field. Indeed, global constraints represent compact solutions and solving algorithms for recurrent subproblems in CSP. They are used as classical constraints and usually encompass a set of constraints defined upon a large set of variables. For example, the well-known `alldifferent` constraint [11] is used to replace a clique of difference constraints. Global constraints offer a more precise and more efficient view of the subproblem they are defined upon. They actively use the underlying structure to provide efficient filtering algorithms. Indeed, the explicit knowledge of this structure leads to an improved propagation-search technique by avoiding repeatedly discovering the same inconsistencies (this phenomenon is called *thrashing*). As expected, global constraints usually imply a higher worst-case time complexity for the filtering algorithm w.r.t. the original set of constraints. However, this overhead is most often largely compensated by the filtering power achieved by the global constraint. Actually, there exists a trade-off between efficiency (*i.e.* running time) and effectiveness (*i.e.* filtering power).

Powerful global constraints are generally attached to one solver: `cycle`, `diffn`, `cumulative` with *chip* [1], `tree` [2] with *choco*, `standard deviation` [12] with *Ilog*. This is probably due to the fact that implementing a global constraint can be highly solver-dependent. Indeed, global constraint implementations usually involve both internal data structure and solver-related structures. The latter are most often backtrackable structures offered by the solver to be used by the constraint to ease the implementation. Not all solvers provide the same backtrackable structure leading to solver-dependent constraint implementations. This can lead to less efficient constraints from one solver to another. This is for example the case with the `alldifferent` constraint (one of the rare ones that made it through several solvers). The efficiency of the constraints is not the same w.r.t. the services provided by the host solver.

In this paper, we would like to investigate these aspects of global constraint implementations. More precisely, we are interested in pointing out that once a global constraint has been defined it needs quite a fine tuning to take advantage of the underlying constraint solver. Hence, we would like to address the point of being able to provide both flexible (in the sense of easing addition or removal of filtering algorithms) and portable (in the sense of being able to adapt the constraint to another solver) implementations of global constraints.

Our test case throughout the paper will be a graph partitioning constraint introduced in [2], the `tree` constraint. We will show that implementing fully dynamic filtering algorithms (*i.e.* not relying upon backtracking and managing their own data structure) transforms the constraint as a plugin for the solver (see Figure 1).

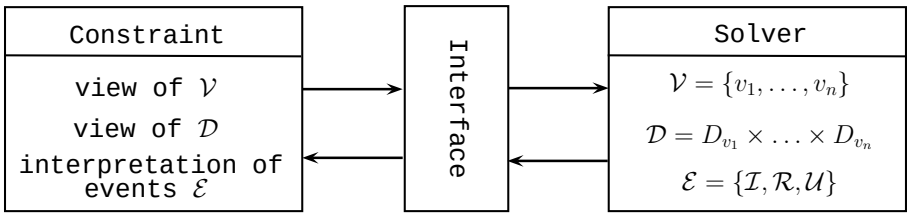


Fig. 1: A *pluggable constraint* defined according to three kinds of data: a view of the variables $\mathcal{V}$ that define the problem, a view of the domain of each variable $\mathcal{D}$ and, an interpretation of the events that can occur on the variables. These events could be the *instantiation* ($\mathcal{I}$) of a variable to a value, the *removals* ($\mathcal{R}$) of a value in the domain of a variable and, the *update* ($\mathcal{U}$) of the lower bound or the upper bound of a variable's domain.

The paper is organized as follows: we first provide a quick description of the `tree` constraint and the current implementation provided within the `choco` constraint solver (`http://choco-solver.net`). Next, we show how the filtering rules used in this constraint can be implemented with fully dynamic algorithms, leading to a pluggable `tree` constraint. We show that this constraint is pluggable by integrating it into `gecode` (`http://gecode.org/`). Moreover, we show that the pluggable version is more efficient with `choco` than the fully integrated version and show that the

gecode version can be made quite as efficient despite the event management system of gecode's java interface.

2 Description of the tree constraint

The tree constraint partitions a given directed graph (digraph for short) into a forest of node-disjoint trees. More precisely, the digraph is partitioned into a set of node-disjoint anti-arborescences[1]. tree is a useful constraint that can be used for modeling various graph-related problems like, for example, surpertree phylogenetic problems [3, 6], ordered disjoint path problems [3, 10], or mission planning problems [7].

The constraint has the form tree(NTREE, VER), where NTREE is a domain variable[2] specifying the number of trees in the forest (MINTREE and MAXTREE respectively denote the minimum and maximum values of $dom(\texttt{NTREE})$) and, VER is the collection of n nodes $\texttt{VER}[1], \ldots, \texttt{VER}[n]$ of the given digraph. Each node $v_i = \texttt{VER}[i]$ has the following attributes, which complete the description of the digraph:

- L is a unique integer in $[1, n]$. It can be interpreted as the *label* of v_i.
- F is a domain variable whose domain consists of elements (node labels) of $[1, n]$. It can be interpreted as the *unique successor* (or *father*) of v_i.

When speaking of global constraints, it is often convenient to reason about a digraph that models the constraint rather than directly about the constraint. We model the extended tree constraint by the digraph $\mathcal{G}$ in which the nodes represent the elements of VER and the arcs represent the successor relation between them. Formally, $\mathcal{G}$ is defined as follows:

Definition 1 (Associated digraph to a tree constraint). *The* associated digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *of a* tree(NTREE, VER) *constraint is defined by* $\mathcal{V} = \{v_i \mid i \in [1, n]\}$ *and* $\mathcal{E} = \{(v_i, v_j) \mid j \in dom(\texttt{VER}[i].\texttt{F})\}$.

A tree(NTREE, VER) constraint specifies that its associated digraph $\mathcal{G}$ should be a forest of NTREE trees, formally:

Definition 2 (Solution of a tree constraint). *A ground instance of a* tree(NTREE, VER) *constraint is said to be a* solution *if and only if:*

- $\forall i \in [1, n] : \texttt{VER}[i].\texttt{L} = i$.
- *The associated digraph $\mathcal{G}$ consists of* NTREE *connected components.*
- *Each connected component of $\mathcal{G}$ has no circuit involving more than one node (notice that each component contains exactly one node that has a self-loop and that corresponds to the root of the tree).*

We recall some definitions and notations regarding the digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ associated with a tree constraint, as well as a lower and upper bound on the number of trees needed for partitioning $\mathcal{G}$. These notions are introduced in the original version of [2]:

[1] A digraph $\mathcal{A}$ is an *anti-arborescence* with *anti-root* r iff for each node v in $\mathcal{A}$ there is a path from v to r and the underlying undirected graph of $\mathcal{A}$ is a tree.

[2] A *domain variable* V is a variable ranging over a finite set of integers denoted by $dom(V)$; $\min(V)$ and $\max(V)$, respectively, denote the minimum and maximum values of $dom(V)$.

Definition 3 (Reduced graph). *To each instance of a* `tree`$(\texttt{NTREE}, \texttt{VER})$ *constraint we associate the* reduced digraph $\mathcal{G}_r$ *derived from* $\mathcal{G}$ *in the following way: to each strongly connected component of* $\mathcal{G}$ *we associate a vertex of* $\mathcal{G}_r$*; to each arc of* $\mathcal{G}$ *that connects different strongly connected components corresponds an arc in* $\mathcal{G}_r$.

Notations 1 (Sink component) *A strongly connected component of* $\mathcal{G}$ *that corresponds to a sink of* $\mathcal{G}_r$ *is called a* sink component.

Notations 2 (Potential root & loop) *A node* v *of* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *such that* $(v, v) \in \mathcal{E}$ *is called a* potential root. *The arc* (v, v) *is called a* loop.

Notations 3 (Door) *A node* u *of* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *is a* door *of the strongly connected component associated with* u *iff there exists* $(u, v) \in \mathcal{E}$ *such that* u *and* v *do not belong to the same strongly connected component of* $\mathcal{G}$.

Definition 4 (Dominator [9]). *Given a digraph* $\mathcal{G}$ *and two distinct nodes* i, j *of* $\mathcal{G}$ *such that there is at least one path from* i *to* j*, a node* d *is a* dominator *of* j *with respect to* i *iff there is no path from* i *to* j *in* $\mathcal{G} \setminus \{d\}$. *The set of dominators of* j *with respect to* i *is denoted by* $DOM_{\langle \mathcal{G}, i \rangle}(j)$.

We are now in position to detail how the initial version [2] of the `tree` constraint is effectively implemented in Choco. Next, we propose a fully dynamic version of this constraint allowing a new design of its implementation that exploits the incrementality in order to avoid using backtrackable data structures dedicated to the solver. A nice property of such an implementation is that the `tree` constraint becomes a plugin for any solver.

3 Implementation of `tree` constraint

3.1 An *ad-hoc* version for *choco*

Several constraint solvers are available but not two of them provide the exact same set of global constraints. When facing a constraint satisfaction problem, the choice of the constraint solver therefore highly depends on the desired constraints and their efficiency. Moreover, quite often only some part of the problem is efficiently handled and one need to either develop or simulate a global constraint in order to solve the problem. Thus, when implementing a global constraint two situations arise: in the first one, the constraint solver has already been chosen and therefore the provided data structures must be used; in the second one, one can choose the most promising solver for the constraint, i.e. a solver which proposes the most adapted data structures to the filtering algorithms to be implemented. In the `tree` constraint case, the *choco* constraint solver was selected. Indeed, it allows a fine-grained management of events [3] occurring on variables. Each variable associated to a problem knows the constraints that involve

[3] *Choco* distinguishes three main kinds of events: instantiation of a variable, removal of a value in the domain of a variable, update of the lower bound of the domain of a variable and, update of the upper bound of the domain of a variable.

it, and symmetrically, each constraint knows the subset of variables that it is posted upon. Then, for a given constraint, distinct treatments can be done for each kind of event occurring on variables involved in the constraint. This leads to the fact that a fix point can easily be reached because the constraint may specify if it has to be awoken by an event produced by itself. The main benefit of such a property is that for a given n-ary constraint, the implementation of its filtering algorithm can be triggered for each kind of event that occurs on the variable domains involved in the constraint.

We now provide the skeleton of the `tree` constraint such as it is implemented in the *choco* solver. The details of each kind of propagation can be seen in [2]. Notice that, w.l.o.g., the notion of *strong articulation points* is generalized to the notion of *dominators* [9]. Obviously, this generalization does not change any filtering algorithm of the initial paper except the computation of dominators in the digraph $\mathcal{G}$ associated with the `tree` constraint.

Initial awake of the `tree` constraint:

- Compute `MINTREE` and `MAXTREE`.
- If there is at least one solution satisfying the constraint then, do propagation related to the constraint:
  1. Update `NTREE` according to `MINTREE` and `MAXTREE`.
  2. Propagate according to the dominator nodes of $\mathcal{G}$.
  3. Propagate according to the doors and the potential roots of $\mathcal{G}$.
  4. Propagate according to the values of $\max(\texttt{NTREE})$ and $\min(\texttt{NTREE})$.

Each time an event occurs on a domain variable involved in the `tree` constraint do:

- If this event occurs on a domain variable modeling `NTREE` then:
  1. Update `NTREE` according to `MINTREE` and `MAXTREE`.
  2. Propagate according to $\max(\texttt{NTREE})$ and $\min(\texttt{NTREE})$.
- If this event occurs on a domain variable modeling a node of $\mathcal{G}$ do:
  1. Update `NTREE` according to `MINTREE` and `MAXTREE`.
  2. Propagate according to new dominators of $\mathcal{G}$.
  3. Propagate according to the doors and the potential roots of $\mathcal{G}$.

The *choco* constraint solver is based on a *trailing*[4] [13] approach to record a decision (e.g., instantiation of variables, removals of values in the domains, etc) and its effects on the data structures involved in the constraint. In our purpose, these effects consist in modifications of the graph structure, for example, if an arc (i, j) is removed from $\mathcal{G}$ (i.e., $j \notin dom(\texttt{VER}[i].\texttt{F})$) then, this removal may:

1. decrease the number of potential roots (see Notation 2), if $i = j$. This leads to an update of `MAXTREE`.
2. increase the number of sink components (see Definition 1). This leads to an update of `MINTREE`.

[4] A *trailing* approach records for each events modifying a data structure, the necessary information to undo its effect.

3. increase the number of strongly connected components (scc) in $\mathcal{G}$. This leads first to increase the number of doors (see Notation 3) contained in $\mathcal{G}$, and second, to change the reduced digraph $\mathcal{G}_r$ associated with $\mathcal{G}$ (see Definition 3).
4. create new dominator nodes in $\mathcal{G}$ (see Definition 4).

Thus, in order to record the necessary information, the Choco solver proposes some *backtrackable* data structures using storable integers, storable booleans and storable bitsets. The original `tree` constraint uses these backtrackable data structures in order to dynamically record and restore some graph properties like strongly connected components and dominator nodes of the digraph $\mathcal{G}$.

However, state-of-the-art graph algorithms propose several fully dynamic algorithms [5] maintaining the graph properties involved in the `tree` constraint like strongly connected components and transitive closure. Then, two straightforward issues are: is it really necessary to use backtrackable data structures when fully dynamic algorithms exist? If no backtrackable data structures are finally used (i.e. it is not necessary to trail the variation of the data structures involved in the constraint) what are the exact relations between the constraint and the solver?

One interesting point here is that when being able to provide *solver-independent* global constraints, other perspectives are open: such a constraint could be plugged in other problem solvers. For example, a good candidate could be COMET, a local search development environment [8], another interesting one would be PaLM [4], the explanation-based extension of the Choco solver.

3.2 A pluggable `tree` constraint

On the one hand, one can summarize the bottleneck of the complexity of a `tree` constraint to repeatedly maintaining several graph properties (strongly connected components (scc), transitive closure, dominator nodes) related to the digraph $\mathcal{G}$ associated with $\mathcal{G}$ between two search steps. For each property, several filtering algorithms are proposed in order to remove inconsistent arcs during the propagation step. On the other hand, the propagation-search techniques only consist in selecting and removing values in variable domains, or restoring values in variables domains. In the context of a `tree` constraint, this technique modifies the `NTREE` variable as well as the set of arcs involved in the digraph $\mathcal{G}$ associated with the constraint.

Basically, a new approach implementing such a constraint can be decomposed in the following way (Figure 2):

1. The **Graph module** is based on a generic fully incremental data structure modeling a digraph $\mathcal{G}$ and its associated properties (i.e., scc's and transitive closure). This module contains primitives updating the data structure according to arc removals and restorations. These primitives basically compute each property on a necessary partial graph[5] of the original digraph $\mathcal{G}$.
2. The **Constraint module** proposes the filtering algorithms (based on the properties maintained by the graph module) that remove arcs of $\mathcal{G}$ inconsistent with the `tree` constraint.

[5] Given a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a *partial graph* $\mathcal{G}'$ of $\mathcal{G}$ is defined by $(\mathcal{V}' \subseteq \mathcal{V}, \{(i,j) \in \mathcal{E} \mid i,j \in \mathcal{V}'\})$.

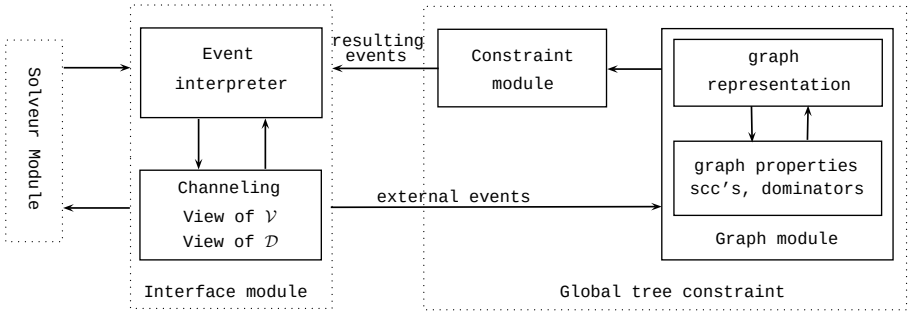


Fig. 2: General implementation schema of a pluggable `tree` constraint.

3. The **Interface module** performs a bijective relation between events occurring on domain variables (i.e., removals/restorations of values in the domains) and events occurring on the digraph $\mathcal{G}$ (i.e., removals/additions of arcs).

Practically, in the current implementation on the pluggable `tree` constraint, each time an event occurs on a domain variable, first this event is interpreted by the interface module, next the graph module updates its data structures, next the filtering algorithms dedicated to the `tree` constraint are applied, and finally the resulting events provided by the constraint module are interpreted in term of domain variable updates.

We are now in position to discuss the interface module. From the domain variables involved in a CSP, basically four kinds of events are distinguished: removal of values in the domains, instantiation of variables, update of lower bounds and update of upper bounds. However, instantiation and bound updates can be easily reduced to a set of removals. Thus, for each kind of event received by the interface from the solver, an event translation to the corresponding set of removals in the graph module is performed. However, all removals are not handled in the same way by the interface.

Indeed, the event at the origin of the considered set of removals is considered to improve the efficiency of the graph module updates. For example, a set of removals related to an instantiation event occurring on a variable leads to a local modification in the neighborhood of the corresponding node in the digraph $\mathcal{G}$ associated with the constraint. This information can be taken into account in order to perform these modifications quite efficiently.

In other words, this new approach for implementing global constraints leads to a reorganization of the code of the constraints. Concerns are clearly separated: data structure management upon domain modifications (events), propagation-related algorithms at the heart of the constraint, and solver/constraint communications. It can be seen as a kind of rationale for global constraints. We think that such a clear separation of concerns is an important point in terms of software engineering.

4 Evaluation

We now report on several experiments we have conducted to evaluate the pluggable `tree` constraint. First, in Section 4.1, we discuss our experiments on the comparison

between the current *ad-hoc* version of the constraint with the new version proposed in this paper. Then, in Section 4.2, we report on the performance of the pluggable `tree` constraint on two distinct constraint solvers: Gecode and Choco.

All experiments were performed with the Choco constraint solver (version 1.2.04) and Gecode constraint solver (version 1.3.1) on an Intel Xeon CPU with 2.4GHz and a 1GB RAM, but only 128MB were allocated to the Java Virtual Machine.

4.1 Original constraint versus pluggable constraint

Graph Order	Density	Average time (*ms*)	
		Ad-hoc	Pluggable
25	≤ 0.5	55	45
	> 0.5	90	38
50	≤ 0.5	610	310
	> 0.5	1532	307
75	≤ 0.5	3856	1174
	> 0.5	8896	1064
100	≤ 0.5	13040	3156
	> 0.5	32568	2682
150	≤ 0.5	69220	11543
	> 0.5	219174	9645
200	≤ 0.5	204497	33763
	> 0.5	> 300000	26315

Table 1: Evaluation of the pluggable `tree` constraint with the existing *ad-hoc* implementation.

The aim of these experiments is to show that the pluggable `tree` constraint outperforms the original version implemented within the Choco constraint solver. Moreover, we point out that the dynamic approach proposed throughout this new constraint is, on average, much less sensitive to the variation of the input digraph density which was originally the bottleneck of the previous constraint version [3].

This set of experiments points out two main features of the pluggable `tree` constraint. For each order of graph in $\{25, 50, 75, 100, 150, 200\}$, and the densities in $[0.05; 1]$ with steps of 0.05, we generate 30 instances (i.e. globally, 3600 digraphs). Notice that we add a timeout fixed to 300000ms, and the solver search uses a random variable-value selector.

First, Table 1 highlights a global improvement of the original (ad-hoc) version of the `tree` constraint; indeed, the pluggable version is 3.8 times more efficient in the case of a density less or equal to 0.5, and 10 times more efficient in the case of a density greater than 0.5. Second, Figures 3 and 4 show that the pluggable constraint is significantly more efficient in the case of dense digraphs. Indeed, Figure 3 depicts the behaviors of the pluggable and ad-hoc constraints for a given graph order fixed to 100

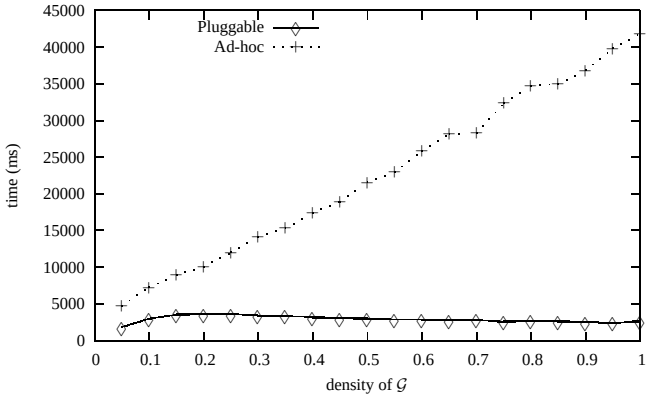


Fig. 3: For a given digraph of order 100, the dotted curve depicts results of the *ad-hoc* implementation of the `tree` constraint while the plain curve depicts the pluggable implementation of the constraint.

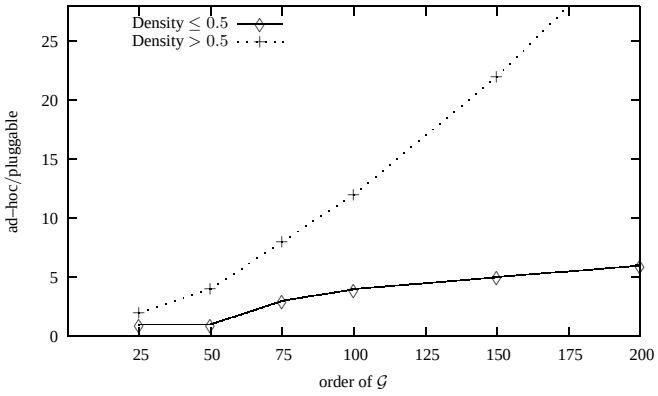


Fig. 4: The dotted curve depicts the ratio between the *ad-hoc* constraint and the pluggable constraint running times, in the case of a density greater than 0.5. The plain curve depicts the same ratio in the case of a density less or equal to 0.5.

nodes. Figure 4 points out the ratio between the pluggable constraint and the ad-hoc constraint running times. In both cases, we directly notice that the pluggable version outperforms the original one in the case of dense graphs.

We are now in position to discuss why the original `tree` constraint is outperformed by the new one. First, we detail the feasibility implementation of the constraint. Next, we show how the filtering part was improved. In the following, we denote by n the order of $\mathcal{G}$ and by m the number of arcs in $\mathcal{G}$.

In [2], a necessary and sufficient condition for the `tree` constraint was introduced: a `tree` constraint has at least one solution iff all sink components of $\mathcal{G}$ contain at least one potential root and $dom(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] \neq \emptyset$ where, `MINTREE` is the

number of sink components in $\mathcal{G}$ and `MAXTREE` is the number of potential roots in $\mathcal{G}$. At each waking up of the contraint, the ad-hoc version of the `tree` constraint computes the strongly connected components (scc) associated with the digraph $\mathcal{G}$, by a suitable depth first search procedure, introduced by Tarjan [14], running in $O(n + m)$ time. However, computing scc at each waking up of the constraint is useless. Indeed, during the propagation/search steps, removing or adding arcs in $\mathcal{G}$ is a local modification of the digraph then, we can reduce this cost by recomputing scc on a necessary partial graph of $\mathcal{G}$ induced by the scc previously computed. In practice, during search the size of the scc decreases (and the number of scc increases) to reach 1. Thus, dynamically maintaining the scc is part of the improvements provided by the pluggable `tree` constraint.

In the original filtering algorithm proposed in [2], the constraint detected *the strong articulation points*. But in practice, we use a generalization of strong articulation points called dominator nodes [9]. For each dominator nodes, the filtering algorithm detects and removes the outgoing arcs that do not allow to reach at least one potential root. Thus, for each dominator, we have to compute a depth first search tree to detect if a potential root can be reached. Thus, for a given digraph $\mathcal{G}$, this can be done in $O(nm)$ time. In the pluggable `tree` constraint a new approach is proposed. We associate to the digraph $\mathcal{G}$, its transitive closure [6]. Explicitly computing the transitive closure of $\mathcal{G}$ is only done during the initial waking up of the constraint in $O(nm)$. Next, a dynamic handling of the transitive closure is performed by updating the current transitive closure according to a necessary partial graph of $\mathcal{G}$ induced by the events modifying $\mathcal{G}$. In practice, this dynamic maintain of the transitive closure is very efficient. Finally, the knowledge of the transitive closure provides the reachability conditions that allow us to dynamically filter the dominator nodes when they are identified by the algorithm.

4.2 Towards portability of global constraints

The aim of these experiments is to illustrate that the pluggable `tree` constraint can easily be plugged into several constraint solvers. The choice of the Gecode and Choco constraint solvers leads to show that there is a slight degradation in the case of the Gecode solver due to the interface module.

Gecode is a "propagator-centered" constraint solver. Thus, it cannot dynamically record events occurring on the variables during search. Then, in order to translate the modifications of variable domains in term of arc additions/removals in the graph module, the interface module has to compute events occurring on the domains from the previous awake of the constraint.

The interface module is a global constraint watching domains modifications. At set up, the constraint stores a copy of the domain of each variables. Then, upon each awake, the constraint compares the current variables' domain with their copy. If a domain has been modified since the last awake, the constraint updates the domain's copy and sends an arc modification event to the graph module.

[6] The *transitive closure* of directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is the directed graph $(\mathcal{V}, \mathcal{E}')$ such that for all v, w in $\mathcal{V}$ there is an arc (v, w) in $\mathcal{E}'$ iff there is a non-empty path from v to w in $\mathcal{G}$.

This computation leads to an overhead of the interface with the Gecode constraint solver. Notice that this is not the case with the Choco interface because Choco is a "variable-centered" constraint solver.

So, the interface module knows what kind of modifications happen on variables domain without having to scan them. Consequently, when the interface module constraint is awaken, it only translates variable events to graph events and send them to the graph module.

For each order of graph in $\{25, 50, 75, 100, 150, 200\}$, and densities in $\{0.05, 0.2, 0.4, 0.5, 0.6, 0.8, 0.95\}$, we generate 50 instances (i.e. globally, 2100 digraphs). Table 2 provides the running time details of each part composing the pluggable `tree` constraint (Figure 2), both for the Choco and Gecode constraint solvers. Notice that both solver searches use the same random variable-value selector.

The column "Interface Module" of Table 2 perfectly points out the overhead due to the interface in Gecode. Moreover, we notice that the running times related to the constraint itself are equivalent in both solvers: the columns "Constraint Module" and "Graph Module" illustrate this result.

Graph Order	Running time *Choco* (ms)			Running time *Gecode* (ms)		
	Interface module	Constraint module	Graph module	Interface module	Constraint module	Graph module
25	5	10	27	6	10	27
50	5	57	229	24	56	228
75	11	190	863	58	186	879
100	18	440	2287	120	439	2310
150	50	1466	9524	368	1329	9596
200	82	3214	27551	812	3009	27097

Table 2: Running time comparison of the pluggable `tree` for two distinct constraint solvers (Choco and Gecode) according to the input digraph order. For each one running time of each part Interface, Constraint and Graph is detailed.

4.3 Implementation requirements

A final interesting point about designing pluggable global constraints is related to development times. The original `tree` constraint [2] took something like three months (including the theoretical study of the constraint) to be fully implemented and debugged. The fully dynamic pluggable version took three weeks to be developed (we obviously used our expertise obtained during the first implementation). But, what is more interesting is the time required to port this *choco* specific constraint to the *Gecode* constraint solver: three days!

5 Conclusion and future works

In this paper, we emphasized the interest in decoupling global constraint implementations from the underlying solver. We showed, on a `tree` constraint, that decoupling leads to efficient (compared to the original version of the constraint) and portable global constraint implementations.

As soon as fully incremental algorithms are available for a given global constraint, one should seriously consider developing a solver-independent constraint so as to improve efficiency but also to give the opportunity to the constraint to be widely used. This can be either with other combinatorial problem solving techniques or other constraint solver. Notice that there exist many fully incremental algorithms with low complexity for maintaining individual graph properties. However, combining several of them usually leads to a compromise in choosing the most interesting data structure. In this paper, we made a quite basic choice that certainly would need to be improved.

One key lesson learnt from our paper is that investing in such algorithms is well worth it: efficiency is improved and portability is a reality (it took us three days to port the `tree` constraint to *Gecode* without loss of efficiency).

We are currently working on a generic way to interface global constraints for constraint solvers in order to be able to define a kind of domain specific language for global constraint implementations.

References

1. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Math. Comput. Modelling*, 20(12):97–123, 1994.
2. N. Beldiceanu, P. Flener, and X. Lorca. The *tree* constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'05)*, volume 3524 of *LNCS*, pages 64–78. Springer-Verlag, 2005.
3. N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, incomparability, and degree constraints, with an application to phylogenetic and ordered-path problems. Technical Report 2006-020, Department of Information Technology, Uppsala University, Sweden, April 2006.
4. H. Cambazard and N. Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11(4):295–313, 2006.
5. D. Eppstein, Z. Galil, and G. Italiano. *Dynamic graph algorithms*. CRC Press, 1997.
6. I. P. Gent, P. Prosser, B. M. Smith, and W. Wei. Supertree construction using constraint programming. In F. Rossi, editor, *Principles and Practice of Constraint Programming CP'03*, volume 2833 of *LNCS*, pages 837–841. Springer-Verlag, 2003.
7. C. Guettier. Solving Planning and Scheduling Problems in Network based Operations. In *Principles and Practice of Constraint Programming CP'07*. LNCS, Springer Verlag, 2007.
8. P. V. Hentenryck and L. Michel. Growing COMET. In F. Benhamou, N. Jussien, and B. O'Sullivan, editors, *Trends in Constraint Programming*, chapter 17, pages 291–297. ISTE, London, UK, May 2007.
9. T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans Program. Lang. Syst.*, 1(1):121–141, 1979.
10. L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *PADL'06*, volume 3819 of *LNCS*, pages 73–87, 2006.

11. J.-C. Régin. A filtering algorithm for constraints of difference in CSP. In *AAAI'94*, pages 362–367, 1994.
12. P. Schaus, Y. Deville, P. Dupont, and J.-C. Régin. The Deviation Constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'07)*, volume 4510 of *LNCS*, 2007.
13. C. Schulte. Comparing Trailing and Copying for Constraint Programming. In D. D. Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, 1999. The MIT Press.
14. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

About Implementing a Constraint Functional Logic Programming System with Solver Cooperation

Sonia Estévez-Martín[1], Antonio. J. Fernández[2] and Fernando Sáenz-Pérez[3]

[1] Univ. Complutense de Madrid, Dpto. de Sistemas Informáticos y Comp, Spain
[2] Univ. de Málaga. Dpto. de Lenguajes y Ciencias de la Computación, Spain
[3] Univ. Complutense de Madrid, Dpto. de Inteligencia Artificial e Ing. SW, Spain

Abstract. This paper provides unreported implementation details of a programming system which implements a seamless integration of constraint, functional, and logic paradigms, and that recently has incorporated a mechanism for solver cooperation on several domains: Herbrand (with equality and disequality constraints), finite domains (for constraint programming constraints over integers), and real numbers (for linear and non-lineal constraints). In particular, the cooperation of constraint solvers over numerical domains is specially interesting because of their practical use for developing many heterogeneous applications relating variables in both domains. This paper gives information about the compilation scheme of the system, its specific libraries, and focuses particularly on how solver cooperation has been integrated into the system. Regarding this last issue, the paper also provides preliminary performance results that supports the suitability of this cooperation mechanism.

1 Introduction

The addition of constraint solving technology to declarative programming systems has caused that these systems are now being considered suitable options (i.e., alternatives to traditional programming systems such as the imperative programming-based systems) for programming complex and real problems. Existing declarative constraint languages are high level programming tools that ease the task of programming (wrt. the formulation of the problem and/or program analysis) and provide a reasonable balance between program formulation and solving efficiency. Moreover, declarative constraint systems combine a high level of abstraction and a declarative nature with an extreme flexibility in the design of their implementations (e.g., wrt. their execution model). This means that they can be used not only as development tools for implementing non-trivial applications but also as platforms where research on key concepts of the implementation of programming languages (including concurrent/parallel models and memory management) can be done.

In this context, the design, implementation, and optimization of declarative constraint programming systems can be considered one of the major issues

treated in recent years in the constraint programming and logic programming areas. In this paper, we consider specifically the constraint functional logic programming (CFLP) language $\mathcal{TOY}$ [1, 2]. This language combines functional and relational notation, curried expressions, higher-order functions, patterns, partial applications, non-determinism, lazy evaluation, logical variables, types, domain variables, and constraint composition. It also provides technology for finite domain ($\mathcal{FD}$) constraint solving (including a wide set of $\mathcal{FD}$ constraints comparable to existing CLP($\mathcal{FD}$) systems and which is competitive with them as shown by performance results [3]), support for managing arithmetic linear and non-linear constraints defined on the real domain $\mathcal{R}$ [4], and provision of strict equality and disequality constraints [5] defined in the Herbrand domain $\mathcal{H}$. Each domain-specific constraint is solved in the associated domain-specific solver (i.e., $solve^{\mathcal{FD}}$, $solve^{\mathcal{R}}$, $solve^{\mathcal{H}}$, respectively) that are connected to the system via an adequate interface.

The set of constraint solvers of $\mathcal{TOY}$ provides support for solving a wide set of practical problems that require constraint solving over each single domain. However, there exist many practical problems that are better expressed using heterogeneous constraints (i.e., involving more than one domain) and, as a consequence, the formulation of these practical problems has to be artificially adapted to one of the domains supported by the connected solvers. With the aim of extending the applicability of the system, $\mathcal{TOY}$ has incorporated recently new features such as solver cooperation. The implementation of this feature in $\mathcal{TOY}$ is based on the theoretical framework described in [6].

This paper focuses specifically on implementation issues, not reported so far, of the $\mathcal{TOY}$ system. Among these issues, the paper briefly describes the compilation procedure and, more particularly, how solver cooperation, as described in [6], has been implemented. Thus, this paper can help other implementors of declarative constraint systems to understand the implementation fundamentals of $\mathcal{TOY}$, and can provide them further ideas to incorporate in their systems. In addition, some performance results are given to show the effectiveness of the solver cooperation mechanism implemented in $\mathcal{TOY}$.

2 Compiling Programs

$\mathcal{TOY}$ programs consist of *datatypes*, *type alias*, *infix operator* definitions, and rules for defining *functions*. The syntax is mostly borrowed from Haskell with the remarkable exception that variables and type variables begin with uppercase letters, whereas constructor symbols and type symbols begin with lowercase (see Example in Section 4.3). In particular, functions are *curried* and the usual conventions about associativity of application hold. As usual in functional programming, types are inferred, checked and, optionally, can be declared in the program.

Instead of using an abstract machine for running byte-code or intermediate code from compiled programs, the $\mathcal{TOY}$ system relies on an efficient Prolog system (i.e., SICStus Prolog [7]) for running $\mathcal{TOY}$ programs compiled to Pro-

log, as in other related systems [8]. The compilation follows a demand driven computation strategy for lazy narrowing [9], and its data-flow is described next.

A $\mathcal{TOY}$ program `program.toy` is compiled as follows (see Figure 1): First, functions, types and constructors defined in this user program are joined with predefined ones (coded in the file `basic.toy`). Next, lexical, syntactical, and semantical analysis are done, and the result contains type declarations, functional dependencies, and definitional trees [10]. Finally, from the last intermediate file, non-declared types are inferred and user-declared types are checked, generating the compiled Prolog code in `program.pl`. In addition, this result contains, among others, code for dynamic cut [11], totality constraints [12], type declarations for predefined functions and constructors, head normal form (hnf) computations, definitions for partial applications and declarations of precedence and associativity for infix operators. This Prolog code is compiled (from Prolog to the underlying SICStus abstract machine) and loaded into the SICStus system, in order to be able to evaluate expressions (i.e., solve goals) typed at the system prompt.

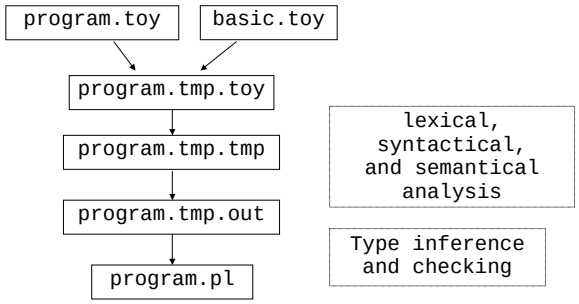


Fig. 1. Compilation Data-Flow

3 Libraries

$\mathcal{TOY}$ provides a number of independent libraries that contain specific definitions for types, data constructors and functions that enable an adequate handling of files, graphics, constraints over real numbers, and constraints over finite domains for integers (Herbrand constraints are always available). When these libraries are loaded, via the appropriate commands typed at the system prompt, these definitions are added to the basic ones. More specifically:

- **Files:** This library provides functions to handle text files, and includes operations to read from and write to files.
- **Graphics:** This library contains functions for building GUIs (Graphical User Interfaces) based on the Tcl/Tk library.

– **Constraints over Reals:** This library enables the constraint domain $\mathcal{R}$ with arithmetical constraints over real numbers, both linear and non-linear. However, these last ones are suspended until they become linear via instantiations. Note that loading this library implies that arithmetical (in)equations require no groundness on their variables.
– **Constraints over Finite Domains:** This library enables the constraint domain $\mathcal{FD}$ with finite domain constraints over integer numbers. For a detailed explanation of this kind of constrains see [1].

4 Implementing Solver Cooperation

This section describes the implementation fundamentals of the cooperation mechanism. Initially, an outline about the architectural components involved in the implementation of the cooperation mechanism is given, and a global overview of the two main pillars of this mechanism (i.e., bridges and projections) is provided. Further, an example of cooperation using both bridges and projections is shown. Later, the implementation of bridges and projections is described in detail. The section ends by giving some relevant comments about how constraint information is exchanged among solvers supported in $\mathcal{TOY}$, and by discussing related work.

4.1 Architectural Components of the Cooperation Schema

Figure 2 shows the Herbrand domain $\mathcal{H}$ for equality and disequality constraints dealing with constructed terms, $\mathcal{R}$ for (linear and non-linear) arithmetical constraints over real numbers, $\mathcal{FD}$ for finite domain constraints over integers, and the mediatorial constraint domain $\mathcal{M}$ for communication constraints among solvers, allowing their cooperation by means of *bridges* and *projections*. This last domain is a hybrid domain that supplies *bridge constraints* (`X #== Y`) for the communication among $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$ domains (cf. Section 4.2).

Each constraint domain ($\mathcal{H}$, $\mathcal{R}$, $\mathcal{FD}$, and $\mathcal{M}$) has an attached constraint store (*H*, *R*, *FD*, and *M*, resp.) and solver ($solve^{\mathcal{H}}$, $solve^{\mathcal{R}}$, $solve^{\mathcal{FD}}$, and $solve^{\mathcal{M}}$, resp.). We take advantage of the SICStus Prolog constraint stores for storing $\mathcal{R}$ and $\mathcal{FD}$ primitive constraints.

$\mathcal{TOY}$ provides lazy narrowing dealing with constraints and takes care of decomposing constraints by introducing new local (produced) variables [13]. Eventually, primitive constraints for $\mathcal{R}$ and $\mathcal{FD}$ arise, which must be submitted to their respective solvers i.e., $solve^{\mathcal{R}}$ and $solve^{\mathcal{FD}}$, resp., and stored in their corresponding stores. $\mathcal{TOY}$ uses the $\mathcal{FD}$ and $\mathcal{R}$ solvers provided by SICStus along with Prolog glue code for interfacing them with $solve^{\mathcal{FD}}$ and $solve^{\mathcal{R}}$, respectively, code for implementing $solve^{\mathcal{H}}$, and code for implementing lazy narrowing dealing with constraints. $solve^{\mathcal{M}}$ follows [6, 14] and the implementation of its bridge constraint is described next in Section 4.4.

Equality and disequality constraints for $\mathcal{H}$ are implemented as already reported in [5]. Also, disequality constraints may affect a variable whose type is

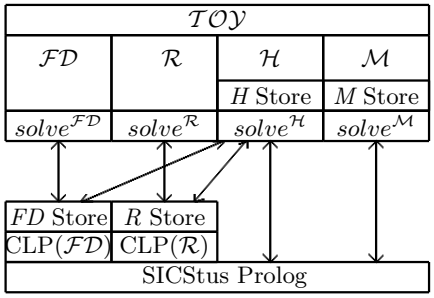


Fig. 2. Architectural Components of the Cooperation Schema in TOY

unknown [15]. These constraints are assumed to involve constructed terms and, therefore handled by $solve^{\mathcal{H}}$. But, along computation, these variables may be instantiated to a number, so that the corresponding disequality constraint is moved to $solve^{\mathcal{R}}$, or $solve^{\mathcal{FD}}$, depending on whether this number is a real or an integer, resp.

4.2 Bridges and Projections: Cooperation Fundamentals

Basically, the cooperation mechanism implemented in the system allows the communication among $\mathcal{H}$, $\mathcal{FD}$, and $\mathcal{R}$, using special communication constraints called *bridges*. Bridges implement binding, equivalence between numbers in those constraint domains, as well as disequalities (antibridges) between variables in those constraint stores. A bridge `X #== Y` constrains `X` $\in \mathbb{Z}$ and `Y` $\in \mathbb{R}$ to take the same integer value. Bridges are kept in a special store and they are used for two purposes, namely *binding* and *projection*. Binding simply instantiates a variable occurring at one end of a bridge whenever the other end becomes a numeric value. Projection is a more complex operation which takes place whenever a pure constraint is submitted to $solve^{\mathcal{FD}}$ or $solve^{\mathcal{R}}$. At that moment, projection rules relying on the available bridges are used for building a mate constraint [14, 6] which is submitted to the mate solver (think of $solve^{\mathcal{FD}}$ as the mate of $solve^{\mathcal{R}}$, and vice versa). Thus, projection enables each of the two solvers to take advantage of the constraints sent to the mate solver. In order to maximize the opportunities for projection, the goal solving procedure has been enhanced with operations to create bridges whenever possible, according to certain rules. Obviously, independent computing of solvers remains possible.

The goal solving rules in [14] describe the process of solver cooperation by means of the creation of new bridge constraints stored in M with the aim of enabling projections of mate constraints via bridges. Solver cooperation can be enabled only for bridges and also for both bridges and projections, which allows

to analyze the trade-off between communication flow and performance gain, so that the user can decide the best option for a given program.

4.3 Example

We show a problem (taken from [16]) which requires the cooperation of an $\mathcal{FD}$ solver and a continuous domain solver. In this example there exists an electric circuit with some connected resistors (modelled with real variables) and there is a set of capacitors (modelled with $\mathcal{FD}$ variables). The goal consists of knowing which capacitor has to be used so that its voltage reaches the 99% of the final voltage given a time range. The $\mathcal{TOY}$ program formulating the problem is shown below.

```
include "cflpfd.toy"

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

In this program, some relational constraint operators have been used (`==` for strict equality, and `<=` for "less or equal than"). Further, a finite domain constraint `belongs` is used, which prunes the domain of `KI` to take values in the given list of capacitor values. An $\mathcal{FD}$ enumeration procedure is applied with `labeling`, which selects the predefined enumeration strategy over the single variable `KI`. Finally, a bridge is used to connect the $\mathcal{FD}$ variable `KI` and the real variable `K` (that represents the continuous value of the capacitor). Note that, due to the imprecision of the real solver, a coupled variable `ET` is added. Real variables `C` and `T` represent, respectively, an input tolerance parameter and the time.

The following goal computes which capacitor has to be used if we consider the time interval [0.5,1] (measured in seconds).

```
Toy(R+FD)> ecircuit 0.001 == K
      { K -> 25 }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

4.4 Bridges

Our Prolog implementation of bridges codes the application of the transformation rules for $solve^{\mathcal{M}}$ in Tables 3 and 4 of [6]. Each defined function in $\mathcal{TOY}$ is implemented as a Prolog predicate with the following arguments: its function arguments (as many as its arity), the function result, and two arguments representing the input constraint store and the output (i.e., modified) constraint store. The code excerpt below shows the basic implementation of the constraint `bridge` (i.e., `#==`):

```
(1) #==(L, R, Out, Cin, Cout):-
(2)    hnf(L, HL, Cin, Cout1),
(3)    hnf(R, HR, Cout1, Cout2),
(4)    tolerance(Epsilon),
(5)    ( (Out=true,
(6)          Cout3 = ['#=='(HL,HR)|Cout2],
(7)          freeze(HL, {HL - Epsilon =< HR, HR =< HL + Epsilon} ),
(8)          freeze(HR, (HL is integer(round(HR)))));
(9)      (Out=false,
(10)         Cout3 = ['#/=='(HL,HR)|Cout2],
(11)         freeze(HL, (F is float(HL), {HR =\= F})),
(12)         freeze(HR, (0.0 is float_fractional_part(HR) ->
(13)                  (I is integer(HR), HL #\= I); true)))),
(14)   cleanBridgeStore(Cout3,Cout).
```

As the $\mathcal{TOY}$ constraint *bridge* has arity 2, its Prolog implementation has two first arguments: `L` and `R` for the left (integer) and right (real) arguments of `#==`, respectively. `Out` is the argument for the result of its evaluation. `Cin` and `Cout` are the arguments for the incoming and outcoming constraint store. This store implements the stores H and M, including constraints from both domains $\mathcal{H}$ and $\mathcal{M}$, i.e., disequality constraints for constructed terms, and bridges and antibridges, resp. Notice that there is no need of explicit accounting for equality constraints on $\mathcal{H}$ since they are handled by Prolog unification.

Lines `(2)` and `(3)` flattens both `L` and `R` by calculating their head normal forms (hnfs), which always delivers either a variable or a number, therefore ensuring that no suspensions will occur from line `(4)` on. So, this implements the demandness of these arguments: They are required to be a variable or a number for a bridge constraint relating them to be posted. In addition, note that a hnf computation may develop new $\mathcal{H}$ disequality constraints during narrowing that have to be added to its constraint store.

Note that a bridge constraint `X #== Y` accepts reification. This means that if the value for `Out` is `true`, then the constraint `X #== Y` is posted to the store M (line `(6)`), whereas if the value is `false`, then the complementary constraint (the antibridge `X #/== Y`) is otherwise posted (line `(10)`). Also, note that this constraint can be used to impose an integral constraint over its right argument.

Implementing both `X #== Y` and `X #/== Y` is accomplished by using the concurrent predicate `freeze` available in SICStus Prolog. This predicate suspends the evaluation of its second argument until the first one becomes ground.

For the first case (`#==`), we need to reflect in this constraint the equality of its two arguments (variables or constants), which are of different type, i.e., real and integer, so that type casting is needed. `HR` is assigned to the float version of `HL` (line `(7)`) and `HL` is is assigned to the integer version of `HR` (line `(8)`). But, due to the imprecision nature of real solvers, occasionally, `HR` must take an approximation to an integer value. So, in order to avoid failures due to requiring exact integer values, it is necessary to introduce a tolerance value via the user-defined parameter `Epsilon` (line `(4)`), which is zero by default. Casting from floats to integers is performed by the Prolog operators `round` and `integer` (line `(8)`).

For the second case (`#/==`), we have to state in $solve^{\mathcal{M}}$ that both arguments are not equal, which cannot be directly handled, as before. So, whenever an argument becomes (or is) ground in a domain $\mathcal{FD}$ or $\mathcal{R}$, then a disequality constraint between the casted ground variable and its mate variable can be posted to the underlying solver (lines `11-13`).

Finally, the store is cleaned of ground bridges (i.e., variable-free) in line `(14)`. As well, variables occurring at the same end of two bridges are unified whenever the variables occurring at the other end become unified.

4.5 Projection: $\mathcal{FD}$ to $\mathcal{R}$

Projecting a constraint in $\mathcal{FD}$ to $\mathcal{R}$ is possible if the user has enabled projection and the constraint is allowed to be projected (see Table 1 in [14] or Table 3 in [6]). The projection amounts to, first, create bridges for the rest of variables in the $\mathcal{FD}$ constraint that are not involved in bridges, therefore creating new $\mathcal{R}$ variables with integral values which may be further related in other $\mathcal{R}$ constraints, and, second, send a mate constraint from $\mathcal{FD}$ to $\mathcal{R}$. The code excerpt below shows its basic implementation for the concrete constraint `#>` (i.e., greater than):

```
(1) #>(L, R, Out, Cin, Cout):-
(2)   hnf(L, HL, Cin, Cout1),
(3)   hnf(R, HR, Cout1, Cout2),
(4)   ((Out=true,  HL #>  HR);
(5)    (Out=false, HL #=< HR)),
(6)   toSolverFD(HL,Cout2,Cout3),
(7)   toSolverFD(HR,Cout3,Cout4),
(8)   (proj_active -> (
(9)      searchVarsR(HL,Cout4,Cout5,HLR),
(10)     searchVarsR(HR,Cout5,Cout,HRR),
(11)     ((Out==true, { HLR > HRR });
(12)      (Out==false, { HLR =< HRR }))));
(13)   Cout=Cout4).
```

This code implements the application of the rules for the cooperation (see Table 4 of [6]) among solvers. It follows the same prototype (line `(1)`) as `#==`, since it is a binary function. Its two input arguments (`L` and `R`) are demanded to be in hnf (lines `(2-3)`), and a primitive constraint is posted to the underlying

$\mathcal{FD}$ solver, depending on the Boolean result of the function (lines `(4)-(5)`). Moreover, these variables can be involved in (Herbrand) disequality constraints because they were not identified as $\mathcal{FD}$ variables already. If so, $\mathcal{FD}$ disequality constraints are also posted to $solve^{\mathcal{FD}}$ (lines `(6-7)`). This scenario appears because the type of a variable is not always known since types are checked and inferred at compile-time, but this information is not present at run-time.

If projection is active (indicated by the dynamic predicate `proj_active` in line `(8)`), then bridges relating the arguments of `#>` are looked for in the mediatorial store in order to find mate variables in $\mathcal{R}$ (lines `(9-10)`). This search, if unsuccessful, will otherwise create bridges relating new mate variables in $\mathcal{R}$. Finally, a mate constraint is sent to the underlying $\mathcal{R}$ solver (lines `(11-12)`).

4.6 Projection: $\mathcal{R}$ to $\mathcal{FD}$

Projecting a constraint from $\mathcal{R}$ to $\mathcal{FD}$ is possible in the same conditions stated in the previous section. The projection amounts to send mate constraints as before, but bridges for the rest of variables in an $\mathcal{R}$ constraint are not created since their integral nature is not for sure. The code excerpt below shows its basic implementation (without considering obvious optimizations) for a concrete constraint `>` (i.e., greater than):

```
(1) >(L, R, Out, Cin, Cout):-
(2)  hnf(L, HL, Cin, Cout1),
(3)  hnf(R,HR, Cout1, Cout2),
(4)  (Out = true, {HR > HL} ;
(5)   Out = false, {HL =< HR}),
(6)  toSolver(HL, Cout2, Cout3),
(7)  toSolver(HR, Cout3, Cout4),
(8)  toSolver(Out, Cout4, Cout),
(9)  (proj_active ->
(10)    (searchVarsFD(HL, Cout, BL, FDHL),
(11)     searchVarsFD(HR, Cout, BR, FDHR),
(12)     ((BL == true,  BR == true,  Out == true,  FDHL #> FDHR);
(13)      (BL == true,  BR == true,  Out == false, FDHL #=< FDHR);
(14)      (BL == true,  BR == false, Out == true,  FDHL #> FDHR);
(15)      (BL == true,  BR == false, Out == false, FDHL #=< FDHR);
(16)      (BL == false, BR == true,  Out == true,  FDHL #>= FDHR);
(17)      (BL == false, BR == true,  Out == false, FDHL #< FDHR);
(18)      true);
(19) true).
```

After analogous steps to the previous subsection, lines `(6)-(8)` deal with the explicit interaction between $solve^{\mathcal{H}}$ and $solve^{\mathcal{R}}$ [17], checking whether the disequality affects a real variable; if so, the constraint is sent to the underlying solver for reals and removed from the Herbrand store.

Next, if projection is active, a similar procedure to the one performed for the projection in the other direction follows. However, notice that there are more possibilities for sending a mate constraint to $solve^{\mathcal{FD}}$ (see Table 4 in [6]),

depending on values of BL, BR, and Out. For BL and BR, a true value means that a bridge relating this variable has been found; a false value means that HL (resp. HR) is a real value with non-zero fractional part. Following the lines (10)-(11), FDHL (resp. FDHR) is the greatest integral value less or equal to HL (resp. HR).

Lines (12)-(13) correspond to true values for BL and BR therefore the mate constraint #> is sent to $solve^{\mathcal{FD}}$, or its counterpart #=<, depending on the Boolean result of the function.

Lines (14)-(17) determine the correct mate constraint which is sent to $solve^{\mathcal{FD}}$, which is selected in terms of the values for BL, BR, and Out, as specified in [6]. For example, line (14) is selected for solving the right argument of the conjunctive goal X #== RX, RX > 4.3. Here, BL is true as the real variable RX has a mate finite variable X, (FDHL is X), BR is false because 4.3 has a non-zero fractional part (so, FDHR is 4). Finally, the mate constraint X #> 4 is posted to $solve^{\mathcal{FD}}$.

4.7 Handling of Numerical Types

$\mathcal{TOY}$ is a typed programming language, based essentially on the Hindley-Milner-Damas polymorphic type system [18]. Programs are tested for well-typedness at compile time. In particular, each occurrence of an expression in a $\mathcal{TOY}$ program has a type that can be determined at compile time. Syntactically, types are built from *type variables* $tvar(\tau)$ and *type constructors* TC. Any identifier starting with an uppercase letter can be used as a type variable, while identifiers for type constructors must start with a lowercase letter. Type constructors are introduced in *datatype declarations*, along with data constructors. Primitive types (such as bool, int, and real) can be viewed as type constructors of arity 0.

Function symbols are required to come along with a so-called *principal type declaration*, which indicates its most general type. For example, the types of the function "greater than" are >::real -> real -> bool and #>::int -> int -> bool, distinguishing the $\mathcal{FD}$ version from the real number one by the prefix symbol #; the exception is the equality and disequality constraints (== and /== respectively) that are overload in order to work in both domains.

Type of variables is not always known at run-time because type inference information is not kept. Thus, a disequality constraint between variables is assumed to range over $\mathcal{H}$, so that it is sent to the Herbrand store. During the constraint solving process, $\mathcal{FD}$ and real constraints are continuously involved in a projection process that gives rise to the update of different $\mathcal{FD}$ and real variables; as a consequence, the disequality constraints stored may be affected by the updates on any finite domain or real variable; if so, each of these disequality constraints is sent to the underlying solver for $\mathcal{FD}$ or real domain in order to look for inconsistences.

Equality constraints are treated differently since these are handled by unification. Narrowing process reduces both arguments to hnf, but $\mathcal{TOY}$ uses a sophisticated process that analyzes the structure of both arguments in order to cut the search space.

In our system, some problems arose with types in different scenarios. For example, we can solve the goals `X+2>4` and `X+2/=4`, but the goal `X+2==4` throws an exception because it tries to unify a real value (i.e., the result of the narrowing of the expression `X+2`) with the integer value `4`. This can be avoided in $\mathcal{TOY}$ by identifying correctly the nature of the result; in the example, the exception is not thrown if we type `X+2==4.0` (note that the left argument of the equality provides always a real value as function `+` is defined as `+::real -> real -> real` and thus the value 2 is interpreted as a real value).

4.8 Related Work

In general, solver cooperation has been widely analyzed in the literature and there are a number of declarative constraint systems that provide support for the interaction among solvers. For example: CLP(BNR) [19], Prolog III [20] and Prolog IV [21] allow solver cooperation, mainly limited to Booleans, reals and naturals. Also, the language NCL [22] provides an integrated constraint framework that strongly combines Boolean logic, integer constraints, and set reasoning. The integration of new constraint domains such as the reals is described as future work in [22]. In general, all those systems provide a limited form of cooperation that is very specific to the predefined computation domains existing in the system. Solver cooperation as integrated in $\mathcal{TOY}$ is quite different from all those systems as its implementation follows the theoretical principles recently described in [6]. Particularly, solver cooperation in $\mathcal{TOY}$ follows an *interoperative approach*, which means that the system has the ability to communicate and use independently-written software components, thus allowing independent systems to cooperate. In the literature, one can find different proposals catalogued in this approach. For instance, [23] proposes a C++ constraint solving library called aLiX for communicating different solvers, possibly written in different languages. One of the main shortcomings of the current aLiX version is that a component for solving continuous constraints is not integrated into the system yet (this is claimed to be one of their main priorities for future development work).

Also, [24] describes a client/server architecture to enable communication among the component solvers. This consists of both managers of the system and the solvers that must be defined on the same computational domain (e.g., real numbers) but with different classes of admissible constraints (e.g., linear and non-linear constraints). The CLP system *CoSAc* is an implementation of their system. This system is very different from our proposal as the exchange of information is managed by means of pipes and the exchanged data is a character string. Also, in his thesis [25], Monfroy constructed the system **BALI** (Binding Architecture for Solver Integration) that facilitates the integration of heterogeneous solvers, as well as the specification of solver cooperation via a number of cooperations primitives. There are may differences with our implementation but one of the most significant is that Monfroy's approach assumes that all the solvers work over a common store, while our present proposal requires communication among different stores.

Perhaps, regarding solver cooperation, the most similar system to $\mathcal{TOY}$ is the system `Meta-S` [26], a meta-solver framework that implements the ideas proposed in [27] for the dynamic integration of external stand-alone solvers to enable the collaborative processing of constraints. The similarities between $\mathcal{TOY}$ and `Meta-S` are because solver cooperation in `Meta-S` also relies on two main constructs, namely constraint propagation (that enables to submit a constraint belonging to some domain $\mathcal{D}$ to its constraint store, say $S_{\mathcal{D}}$) and projection of constraint stores (that consults the contents of a given store $S_{\mathcal{D}}$ and deduces constraints for another domain). Our projection differs from `Meta-S` projection in the creation and use of bridges; `Meta-S` propagation corresponds to our goal solving rules for placing constraints in stores and invoking constraint solvers. An important difference is the lack of bridges in `Meta-S` approach that corresponds to the lack of mediatorial domains within the combined domains that can be constructed in this system. From the implementation point of view, there are additional structural differences between $\mathcal{TOY}$ and `Meta-S`. So, `Meta-S` does not provide facilities for constraint optimization (as $\mathcal{TOY}$). Also, `Meta-S` is implemented in Common Lisp whereas $\mathcal{TOY}$ is implemented in Prolog.

5 Performance

In this section, we briefly show empirically that the projection mechanism of $\mathcal{TOY}$ helps to accelerate the cooperative constraint solving. To do so, we have considered a number of benchmarks[4]: a *non-linear crypto-arithmetic (nl-csp)* problem (9 $\mathcal{FD}$ variables with non-linear equations), two problems for solving systems of 10 (*eq10*) and 20 (*eq20*) linear equations with 7 $\mathcal{FD}$ variables, an *electrical circuit* problem, a *knapsack* optimization problem, and a set of cryptoarithmethic problems i.e., *send+more=money (smm)* problem (8 $\mathcal{FD}$ variables, 1 linear equation, 2 disequations, and 1 *all different* constraint), the *Wrong+Wrong=Wright* (wwr) problem (8 $\mathcal{FD}$ variables, 1 linear equation, 1 *all different* constraint), the *alpha* problem (26 $\mathcal{FD}$ variables, 20 linear equations, and 1 *all different* constraint), and the *donald* problem (10 $\mathcal{FD}$ variables, 1 linear equations, and 1 *all different* constraint). All the benchmarks were coded to require $\mathcal{FD}$ constraint solving as well as solving of (non-)linear equations in $solve^{\mathcal{R}}$.

All the benchmarks were executed on the same Linux machine, operating system Suse Linux 9.3, with an Intel(R) Pentium(R) M processor running at 1.70GHz and with a RAM memory of 1 GB. For the sake of brevity, in Table 1 we only provide the results for first solution search. The first column displays the configuration employed: (1) $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$, which corresponds to $\mathcal{TOY}$ with both numerical solvers activated, and (2) $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$-proj which corresponds to $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$ with the mechanism for constraint projections activated. In addition, two labeling strategies were considered: *naïve*, in which variables are labelled in a prefix order, and *first fail (ff)*, in which the variable with the smallest domain is chosen first for enumerating. The label ($\mathcal{FD} \sim \mathcal{R}$)

[4] All the benchmarks are available in `http://www.lcc.uma.es/~afdez/cflpfdr`.

means that labelling of $\mathcal{FD}$ variables and global constraints are executed in $solve^{\mathcal{FD}}$ whereas (non-)linear-equations are sent to $solve^{\mathcal{R}}$. The numbers for the different versions of $\mathcal{TOY}$ represent the average of ten runs. Note that, in general, activating the projection mechanism provides a significant performance improvement. Further experiments with more benchmarks have also been executed leading to the same conclusion.

Configuration	knapsack	donald	smm	nl-csp	wwr	eq10	eq20	alpha	circuit
$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$									
naïve $\mathcal{FD} \sim R$	16	304970	22528	411	411	266	402	314	14
ff $\mathcal{FD} \sim R$	15	288700	22627	383	420	271	408	272	**13**
$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$-proj									
naïve $\mathcal{FD} \sim R$	11	8305	41	44	54	290	433	291	14
ff $\mathcal{FD} \sim R$	16	601	40	87	58	269	397	283	20
Speed-Up									
naïve $\mathcal{FD} \sim R$	1.45	36.72	549.43	9.34	7.61	0.91	0.92	1.07	1
ff $\mathcal{FD} \sim R$	0.93	480.36	565.67	4.4	7.24	1	1.02	0.96	0.65

Table 1. Running time (milliseconds) for first solution search

6 Conclusions and Future Work

In this paper, we have dealt with implementation issues of the constraint functional logic programming system $\mathcal{TOY}$ unreported up to now. Among these implementation issues, we have described the data-flow program compilation process, available libraries, and the integration of constraint solving technology in the system. With special emphasis, we have explained how solver cooperation has been recently incorporated in $\mathcal{TOY}$. This is a very important issue as the interaction among solvers makes it easier to express compound problems, and good communication can help the efficiency of the systems [28].

More specifically, we have described the internal communication among $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ via bridges and projections. We have sketched their implementation, and shown that bridges manage the communication between two variables that belong to different computation domains, whereas propagation generates, from a primitive constraint defined on one source computation domain, new (semantically-equivalent) constraints that are propagated to another computation domain which demands cooperation with the source domain. This solver cooperation can lead to drastic reductions in the search space of the problem, and can be translated into a reduction of the solving time as it was shown in [6].

Acknowledgements

First and third authors were partially supported by the Spanish National Projects MERIT-FORMS (TIN2005-09027-C03-03) and PROMESAS-CAM(S-0505/TIC/0407). Second author was partially supported by Spanish MCyT projects under contracts TIN2004-7943-C04-01 and TIN2005-08818-C04-01.

References

1. Arenas, P., Fernández, A., Gil, A., López-Fraguas, F., Rodríguez-Artalejo, M., Sáenz-Pérez, F.: $\mathcal{TOY}$. A Multiparadigm Declarative Language. Version 2.3.0 (July 2007) R. Caballero and J. Sánchez (Eds.), Available at `http://toy.sourceforge.net`.
2. González-Moreno, J., Hortalá-González, M., López-Fraguas, F., Rodríguez-Artalejo, M.: An Approach to Declarative Programming Based on a Rewriting Logic. The Journal of Logic Programming **40**(1) (July 1999) 47–87
3. Fernández, A.J., Hortalá-González, T., Sáenz-Pérez, F., del Vado-Vírseda, R.: Constraint Functional Logic Programming over Finite Domains. Theory and Practice of Logic Programming (2007) In Press.
4. Hortalá-González, T., López-Fraguas, F., Sánchez-Hernández, J., Ullán-Hernández, E.: Declarative Programming with Real Constraints (1997) Technical Report SIP 5997, Univ. Complutense Madrid, 1997.
5. Arenas, P., Gil, A., López-Fraguas, F.: Combining Lazy Narrowing with Disequality Constraints. In: PLILP'94, Springer LNCS 844 (1994) 385–399
6. Estévez-Martín, S., Fernández, A., Hortalá-González, T., Rodríguez-Artalejo, M., Sáenz-Pérez, F., del Vado-Vírseda, R.: A Proposal for the Cooperation of Solvers in Constraint Functional Logic Programming. ENTCS **188** (2007) 37–51
7. SICStus Prolog: (2006) `http://www.sics.se/isl/sicstus`.
8. Antoy, S., Hanus, M.: Compiling Multi-Paradigm Declarative Programs into Prolog. In: Frontiers of Combining Systems. (2000) 171–185
9. Loogen, R., López-Fraguas, F., Rodríguez-Artalejo, M.: A Demand Driven Computation Strategy for Lazy Narrowing. In: PLILP. (1993) 184–200
10. Antoy, S.: Definitional Trees. In: 3rd International Conference on Algebraic and Logic Programming (ALP'92). Number 632 in LNCS, Volterra, Italy, Springer-Verlag (1992) 143–157
11. Caballero, R., López-Fraguas, F.J.: Dynamic-Cut with Definitional Trees. In: FLOPS '02: Proceedings of the 6th International Symposium on Functional and Logic Programming, London, UK, Springer-Verlag (2002) 245–258
12. Caballero, R.: A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In: WCFLP '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, ACM Press (2005) 8–13
13. López-Fraguas, F., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: A Lazy Narrowing Calculus for Declarative Constraint Programming. In: Proc. of PPDP'04, ACM Press (August 2004) 43–54
14. Estévez-Martín, S., Fernández, A.J., Hortalá-González, M.T., Rodríguez-Artalejo, M., del Vado Vírseda, R.: A fully sound goal solving calculus for the cooperation of solvers in the cflp scheme. ENTCS **177** (2007) 235–252

15. López-Fraguas, F., Sánchez-Hernández, J.: $\mathcal{TOY}$: A Multiparadigm Declarative System. In Narendran, P., Rusinowitch, M., eds.: RTA. Number 1631 in LNCS, Springer (1999) 244–247
16. Hofstedt, P.: Cooperation and Coordination of Constraint Solvers. Phd thesis, Technischen Universität Dresden, Fakultät Informatik (2001)
17. Sánchez-Hernández, J.: $\mathcal{TOY}$: Un Lenguaje Lógico funcional con restricciones (1998) Available (in Spanish) at `http://babel.dacya.ucm.es/jaime/publications.html`.
18. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL'82, ACM Press (1982) 207–212
19. Benhamou, F., Older, W.: Applying interval arithmetic to real, integer and Boolean constraints. The Journal of Logic Programming **32**(1) (July 1997) 1–24
20. Colmerauer, A.: An introduction to PROLOG III. Communications of the ACM (CACM) **33**(7) (July 1990) 69–90
21. N'Dong, S.: Prolog IV ou la programmation par contraintes selon PrologIA. In: Sixièmes Journées Francophones de Programmation Logique et Programmation par Contraintes (JFPLC'97), Orléans, France, Edition HERMES (1997) 235–238
22. Zhou, J.: Introduction to the constraint language NCL. The Journal of Logic Programming **45**(1-3) (2000) 71–103
23. Goualard, F.: Component programming and interoperatibility in constraint solver design. In K.Apt, Barták, R., Monfroy, E., Rossi, F., eds.: ERCIM Workshop on Constraints, Prague, Czech Republic, Charles University/Faculty of Mathematics and Physics (2001)
24. Monfroy, E., Rusinowitch, M., Schott, R.: Implementing non-linear constraints with cooperative solvers. Research Report 2747, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine (December 1995)
25. Monfroy, E.: Solver collaboration for constraint logic programming. PhD thesis, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine (November 1996)
26. Frank, S., Hofstedt, P., Reckman, D.: Meta-s - combining solver cooperation and programming languages. In Wolf, A., Frühwirth, T.W., Meister, M., eds.: Proc. W(C)LP 2005, Ulmer Informatik-Berichte 2005-01 (2005) 159–162
27. Hofstedt, P., Pepper, P.: Integration of Declarative and Constraint Programming. Theory and Practice of Logic Programming (2006) In Press.
28. Granvilliers, L., Monfroy, E., Benhamou, F.: Cooperative solvers in constraint programming: a short introduction. ALP Newsletter **14**(2) (May 2001)

Distributed Multi-Threading in GNU Prolog

Nuno Morgadinho and Salvador Abreu

Departamento de Informática, Universidade de Évora
Portugal
{nm,spa}@di.uevora.pt

Abstract. We connect the GNU Prolog compiler with the programming environment PM2, obtaining a system that allows the development of distributed multi-threaded Prolog applications. This is especially useful for computationally intensive problems, where performance is an important factor. The system API offers thread management primitives, as well as explicit communication between threads via message-passing. In the initial evaluation the system shows an almost linear speedup for three benchmark programs and also a good performance for a real-world application, when compared to a sequential version.

Keywords: Distributed, Multi-Threading, Prolog, Logic Programming, Parallel Computing

1 Introduction

In this paper we focus the attention upon a system that allows for distributed multi-threading in GNU Prolog [8].

Parallelism in Prolog programs has been fairly researched by implementing parallel Prolog systems and by combining existing Prolog systems with parallel programming environments, as can be witnessed by Aurora [12], Muse [2], &-Prolog [11], DDAS [19], Andorra-I [7], Parlog [5], YapOr [18] and PVM-Prolog [13], to name a few.

PM2 [15], a programming environment similar to PVM, allows distributed multi-threading C applications to be developed and adheres to the *Single Program Multiple Data* (SPMD) programming model. The user writes a program and copies of it are launched, by a specific load command, on each machine. It is based primarily on two distinct libraries: one for thread management (Marcel [16]) and another for communication (Madeleine [3]). By making use of these libraries, the program is able to determine its identity and run different actions accordingly. Since PM2 programs are to be executed on remote machines, they must not depend on third party applications or libraries that may not exist.

GNU Prolog, which is based on the WAM (Warren Abstract Machine), uses in the compilation process a mini-assembly language as an intermediate level between the WAM code and the assembly language of the target machine, and for this reason, is capable of producing stand alone executables that can be loaded with PM2. Further more, the size of these executables is relatively small since GNU Prolog avoids linking the code of most unused built-in predicates.

Most Prolog implementations, such as SWI-Prolog, SICSTUS Prolog and the Yap Prolog system, which provide machine-independent saved-states aren't as straighforward to use with PM2.

This makes GNU Prolog ideal to combine with PM2. We developed a system that explores such combination, which we call **PM2-Prolog**, and that we think is specially suitable when dealing with:

- Applications that potentially have some degree of parallelization whose performance needs to be improved;
- Applications comprised of intelligent agents, hosting one or more agents per machine;
- Scientific and business problems, such as simulation applications, that produce large amounts of data that need to be processed for visualization, data mining or machine learning. For many cases, faster results can be potentially achieved, subdividing the problem and processing each sub-task in a different processor;
- Applications where some loss of accuracy in result can be traded for faster execution times.

Supporting distributed multi-threading in GNU Prolog by connecting it to PM2 also favors portability and permits to take advantage of functionalities already provided with PM2, such as fault-tolerance.

Our system is based on distributed memory and message-passing. What is to be executed on the memory of other nodes is sent over the network.

Since PM2 can sustain more than one entity of the application on each node, PM2-Prolog offers the possibility of choosing how many entities run locally on each machine besides how many run distributed. This is especially attractive to exploit mutiprocessor systems, namely multicore microprocessors.

In summary, our contribution introduces a system that allows the exploitation of explicit parallelism and multi-threading in logic programming, based on a well known ISO-compliant Prolog, GNU Prolog and on PM2, a distributed multi-threading programming environment widely used in academia.

The implemented architecture allows new abstraction layers to be easily defined on top of it, providing a framework to develop parallel and distributed Prolog applications, or as a layer for the support of the execution of other applications that require heavy computational resources or to which applying some degree of parallellization may be beneficial.

The remainder of this paper is organized as follows: **Section 2** provides insight into parallel and multi-threaded prolog systems.

Section 3 covers the execution model, architecture and implementation of PM2-Prolog. The API it provides is listed and an explanation of how to use it is also provided.

In **Section 4** the system is experimentally evaluated and the obtained performance results discussed.

Finally, **Section 5** draws conclusions and outlines possible proposals for future work.

2 Parallel and Multi-Threaded Prolog

Prolog systems, such as YapOr [18], automatically exploit the (potential) parallelism in programs without any input from the programmer. Others, like Delta Prolog [17] and CS-Prolog [10] offer parallel primitives that allow the definition of how the parallel execution is to be carried out and to control related aspects like process synchronization, communication or task partitioning. The terms *implicit* and *explicit parallelism* describe these approaches.

Although implicit parallelism is appealing because the sequential programming model is retained, providing the ability to parallelize legacy code, there is a limit on how much parallelism can be exploited automatically, and current techniques only achieve maximum parallel potential on certain kinds of programs.

Explicit parallelism gives more work to the programmer because it requires specific instructions to be called in order to execute a program but it also allows for very efficient code to be written.

2.1 Message Passing Prolog Systems

One of the basic methods of explicit parallelism is the use of message passing libraries. These libraries manage transfer of data between instances of a parallel program running (usually) on multiple processors in a parallel computing architecture. The Message Passing Interface (MPI) is the de facto standard[1] for this type of communication.

Both Delta Prolog [17] and CS-Prolog [10] present a system where multiple Prolog engines are mapped to processes that are running in parallel and communicate with each other via explicit message passing. These implementations were the first systems that exploited explicit parallelism based on message passing for Prolog, but meanwhile have ceased to exist or aren't being actively developed anymore.

PVM-Prolog [13] introduced a programming interface to the PVM system where multiple distributed Prolog processes cooperate using a message passing model. One of its limitations is that PVM itself isn't multi-threaded.

2.2 Multi-threading

With respect to multi-threading, Prolog systems commonly offer implementations based on the POSIX threads (pthread) API. This is exemplified by Qu-Prolog [6], SICStus MT [9], IC-Prolog II [4], PMS-Prolog [21] and more recently by SWI-Prolog [20].

In these implementations each Prolog thread is normally a POSIX thread running a Prolog engine and threads communicate among each other either by using FIFO message queues or a blackboard system (an area of shared memory).

[1] The MPI standard is comprised of 2 documents: MPI-1 (published in 1994) and MPI-2 (published in 1996). MPI-2 is, for the most part, additions and extensions to the original MPI-1 specification. The MPI-1 and MPI-2 documents can be downloaded from the official MPI Forum web site: http://www.mpi-forum.org/.

Although when dealing with computationally intensive problems, multi-threading can be used to improve performance and responsiveness, there is always a limit because such programs can only run on a single-machine.

3 PM2-Prolog

Our approach to introduce distributed multi-threading in GNU Prolog is based on explicit parallelism with calls to message-passing primitives.

Since PM2 programs are developed in C and compiled with gcc and GNU Prolog compiles Prolog code, it was necessary to estabilish a model for connecting the two programming environments.

The approach used doesn't involve modifications to GNU Prolog neither modifications to PM2. Instead, it relies on:

- a new program (Tabard[2]), written in C with the PM2 libraries, that manages distributed instances of gprolog engines, and that is transparent for the end-user.
- a new Prolog library (pm2prolog-lib), implemented partly in C and partly in Prolog, which allows the development of distributed multithreaded Prolog applications.

When using a PM2-Prolog program we first generate the binary by compiling our Prolog program and link Tabard, the libraries of PM2 and the libraries of GNU Prolog (Figure 1).

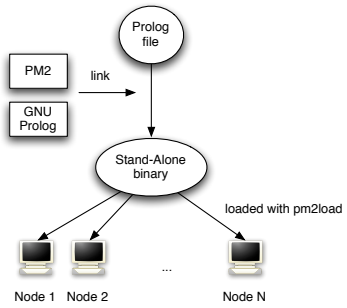


Fig. 1. PM2-Prolog architecture as seen by the end-user.

Before compiling there is a configuration that specifies the list of machines on which the application is going to run. That configurations maps each machine

[2] Literally, a tabard is a short coat, that in the late middle ages was worn by knights over their armour. This fits nicely as a name because Tabard will allow "wearing" a Prolog program over PM2 (the armour).

to one or more *processing nodes* or *virtual processors* (VPs). While it may seem common sense practice to use one virtual processor per physical node, nothing in PM2-Prolog requires such association, as illustrated by figure 2.

When a PM2-Prolog program is launched it starts the following execution model:

1. The binary is copied to all the machines. The `main()` function of Tabard is called on every VP.
2. In VP 0 (master) a gprolog engine is created calling `Start_Prolog()`, that will start executing the linked Prolog code.
3. In the other VPs (workers) a pthread in C is created and stands awaiting messages. This is done via a call to a blocking read function.
4. In the master, now in the Prolog thread, a predicate is called to send a message to every worker, ordering the starting of a Prolog thread by calling `Start_Prolog()`.
5. The workers receive that message, initiate a gprolog engine and the new Prolog thread stands awaiting more messages to come by calling a blocking read pm2prolog-lib predicate. At this time, there are two threads awaiting messages, one in C and another in Prolog, for each worker.
6. In the master, work is distributed throughout the workers through message-passing.
7. The workers receive tasks which they execute locally. As soon as they finish, they send their results back to the master and return to their prior state, awaiting for messages.
8. The master assembles the work results by reading as many messages as the number of previously sent messages.
9. The master redistributes work again (5.) or orders the workers to finish their execution.
10. The workers terminate.
11. The master reiniciates the workers (4.) or terminates itself.

In PM2-Prolog each machine in the configuration will have a C thread (*listener*) and a Prolog thread, for each VP. The purpose of the C thread is to control the associated GNU Prolog engine that runs in the Prolog thread, in terms of creation, termination, monitoring, etc.

Since GNU Prolog doesn't support multi-threading, PM2-Prolog novelty is that it allows to control more than one GNU Prolog thread in the same machine without introducing changes in GNU Prolog itself.

Also, with this approach, it supports all predicates of the GNU Prolog libraries without the need of modifications.

In summary, it achieves a multi-threading that is very appellative from a technical point of view because of its simplicity but that for each Prolog thread has an attached C thread (Figure 2). However, once GNU Prolog introduces support for multi-threading it is trivial to change to an architecture where there is only one C thread by machine that controls N GNU Prolog threads.

Branching is crucial in PM2-Prolog since it allows to differentiate between the different threads and execute different things in each one to our benefit. The

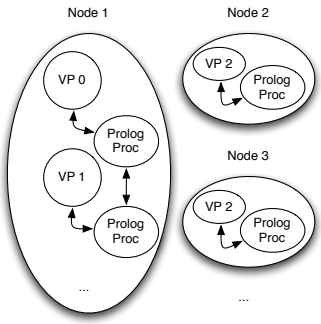


Fig. 2. Architecture where a node has more than one VP.

case described earlier, in which we have a *master* thread, that distributes tasks to be done by the workers is known as *task farming* and is a commonly used way of parallelizing applications.

This rank is an `unsigned int` between 0 and `pm2_max_rank/1`, the configuration size. A VP can learn about its own rank by calling `pm2_self/1`.

Processing nodes are able to execute code and simultaneously check if any messages arrives. The listener thread receives commands or orders in form of messages that can result in different actions being carried out on a specific VP, such as creating another thread or execute specific code.

Two important messages were specified and implemented: 1) create a Prolog thread and 2) terminate the listener thread. All other messages that arrive at a VP will be interpreted not as commands, but as common messages that must be delivered to the running Prolog thread inside that VP. A mechanism of quoting to enable passing messages equivalent to these command is not yet implemented, but is being thought of.

As can be observed on figure 3, the listener thread and the Prolog thread communicate using a shared data structure. The listener delivers messages by writing them to a *message queue* and the Prolog thread accesses them by reading in *First In First Out* (FIFO) order from that structure. The message queues provide a means for threads to wait for data without using the CPU. Other means to do this, like checking via a polling loop, would cause *busy-waiting*, that generally should be avoided.

The listener thread receives messages from a socket and the Prolog thread sends messages via its listener "support" thread.

The Prolog thread can also write to its message queue in the special case where the destination VP is the same as the sender.

In terms of communication, the Madeleine layer provides an API that is similar to POSIX socket. Around this API we've implemented primitives for sending and receiving Prolog terms over the network. As with Madeleine, it is also not possible in PM2-Prolog to know the source address when a message

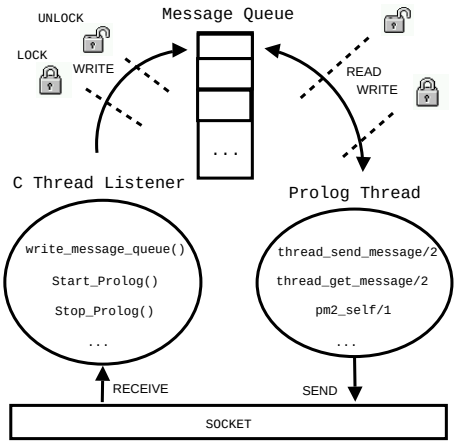


Fig. 3. Inside a processing node or virtual processor.

is received. Complementary, we observe that most applications don't have this requirement. That doesn't mean we will only communicate between master and workers, because we can configure any host which is network accessible.

To implement the routines we needed to be able to convert a Prolog term into a C string and also be capable of doing the opposite process. This is required since the Madeleine routines receive a (char *) buffer as an argument.

The approach used consists in transforming the term to a character code list by using the built-in GNU Prolog predicate `writeq_to_codes/2`. This is similar to `write/2` except that characters are not written into a text-stream but are collected as a character code list which is then unified with the first argument.

By using the foreign type `term` a C string will be ready to be sent. On the reception side, once the string is read we use `Mk_Codes()` to convert it again to a character code list and `read_term_from_codes/3` to transform it back into the original Prolog term.

Threads living on distinct nodes may not directly interact together unless by message-passing. When this happens the listener thread receives the message and is responsible for delivering it to the correct local thread.

Prolog threads are created by calling `marcel_create()`, that is part of the Marcel library . Once Marcel threads are created they basically behave by default as native pthreads. The reason to use it rather than pthreads is that Marcel is designed to work together with Madeleine.

The thread starts by executing the function given as argument to `marcel_create()`. In this case, this is `Start_Prolog()`, which initializes and starts the Prolog engine.

The mutexes provided by the Marcel API are used to make the operations on the message queue thread-safe. Their behaviour is also similar to the ones offered by the pthread API.

One thing that also needs to be guaranteed in the message queues is that no initial `read` is made before `write`. For that purpose another mutex has been used, as follows:

1. The mutex is initialized and a lock is made.
2. Some VP send a message. The listener thread receives it, writes it to the message queue and an unlock is made.
3. Since an unlock has been made, the Prolog thread will now acquire the lock and retrieve the message from the queue. Finally, a lock is made and we go back to step 1, starting over again when a new message arrives.

A computation in Prolog is always a process of production of bindings, known as unification. This process consists in binding a variable to a value, the scope of which is local to the Prolog process and not visible or accessible to the outside. In PM2-Prolog, this principle remains the same, meaning that a variable that had a certain value on the sender side will not have the same value on the receiver side, except if the user deliberatly via message-passing produces such binding.

3.1 API

The PM2-Prolog interface is designed to be simple and as close as possible of the draft technical recommendation (DTR) for Prolog multi-threading support [14].

Extending PM2-Prolog is trivial since new Prolog predicates or new C functions can be added using the foreign interface. The current PM2-Prolog API is the following:

PM2 Facilities

`pm2_self(-Rank)`

Unifies with the rank number of the processing node, a unique integer number assigned to each machine.

`pm2_is_master/0`

Will succeed when the rank where it is being called is zero, usually the thread that distributed work.

`pm2_max_rank(-Rank)`

Unifies with the highest rank number of the configuration.

`finish_listeners/0`

Terminates the listeners threads in each VP. Called upon termination.

Creating and destroying Prolog threads

`start_prolog_workers(+HighestRank)`

Start the Prolog thread in each VP.

`stop_prolog_workers(+HighestRank)`

Stop the Prolog thread in each VP.

Thread Communication

`thread_send_message(+ThreadId, +Term)`

Place Term in the queue of the indicated thread (which can even be the message queue of itself). Any term can be placed in a message queue, but note that the term is copied to the receiving thread and variable-bindings are thus lost. This call returns immediately.

Since each thread has by default its own message queue the other threads will be unaffected by this call.

ThreadId - ThreadId is a compound term of the form vid(Rank, ThreadId). Rank is given by `pm2_self/1` and ThreadId is an integer number assigned sequentially to each thread inside a specific machine.

`thread_get_message(-Term)`

Examines the thread message queue and blocks execution until a term arrives in the queue. After a term from the queue has been unified to Term, the term is deleted from the queue and this predicate returns.

`read_results(-Number)`

Calls `thread_get_message/1` a *Number* of times.

4 Experimental Evaluation

To assess the suitability of PM2-Prolog for a particular purpose, many users will consider it's performance as the most important and indeed critical feature.

The environment on which PM2-Prolog can be used can widely vary. It can be a cluster of networked workstations or a set of workstations wide-spread throughout the Internet. As a matter of fact, these workstations need only to be running Linux and have ssh/rsh access.

Our study focuses on a cluster of SMP systems and the speedup that can be obtained from problems that consist of a large task and that can be split into subtasks distributed over a pool of threads.

What interests us here is the elapsed real (*wall-clock*) time used by the process. It represents the total time needed to complete a task, including disk accesses, I/O activity, operating system overhead - everything. We obtain the wall-clock time with the Unix `time(1)` command (not the shell built-in `time` but the one normally found in /usr/bin/time) and with the format set to `elapsed time` only, specified with the parameter `-f %E`, e.g.:

```
/usr/bin/time -f%E ls
```

This time will be composed by the initialization time of the program (T_0) plus the time the program will spend actually doing some processing (T).

The initialization time is not constant. It grows along with the number of workers, while the processing time will decrease until it reaches a point where it is less than the initialization time and by then it no longer compensates to have more workers on the problem.

In order to compare the real processing time between configurations with different number of workers, the initialization time must be calculated and then subtracted from the obtained elapsed time.

The initialization time (T_0) is given by the elapsed time obtained for what is called the *empty problem*, which consists of no more than a program that initializes the system and exits.

Having these measurements we calculate the speedup (S) using the formula:

$$S = \frac{T_1 - T_1(0)}{T - T(0)}$$

where T_1 is the elapsed time obtained with M + 1 worker, $T_1(0)$ the elapsed time obtained with M + 1 worker for the "empty" problem, T the elapsed time obtained with M + N workers and $T(0)$ the elapsed time obtained with M + N workers for the "empty" problem.

The hardware environment used consisted of 7 units of the machine shown below:

Table 1. Hardware Environment (x7)

CPU	Intel(R) Pentium(R) 4 CPU 2.80GHz each
Hyper-Threading	Enabled
Cache size per CPU	512 Kb
FPU	Yes, integrated
Memory	512 Mg
Filesystem	IDE disk shared via NFS
Filesystem type	Ext2
Network	TCP/IP over Ethernet
Network interfaces	RealTek RTL8139 Fast Ethernet
Background load average	Minimum or none

We use a suite of three classic literature problems plus a real-world application to measure the speedup that can be obtained in each one by using PM2-Prolog.

The first one is a matrix multiply program, where the system obtained a speedup of 2.92 times with 6 CPUs and of 4.71 times with 12 CPUs, comparing to the same program running in a single processor.

Table 2. Obtained times for 64x64 matrix (integer) multiplication executed fifty times

Workers	CPUs	Elapsed Time	Speedup
1	2	01:09.7s	1.00
3	6	00:23.9s	2.92
6	12	00:14.8s	4.71

The second one is the N-Queens program. In this program, a job consists in a valid placement of queens up until a certain column. A result consists in the number of solutions found for that particular job. A worker must find all the solutions for that board prefix, then send the number back to the master. A speedup of 2.98 times with 6 CPUs and 4.78 times with 12 CPUs was obtained.

Table 3. Obtained elapsed time for the parallel nqueens problem

Workers	CPUs	Elapsed Time	Speedup
1	2	03:23.2s	1.00
3	6	01:08.1s	2.98
6	12	00:42.5s	4.78

The third one is a parallel array search program. The program will find all occurrences of a certain integer by dividing an input list for processing by the available workers and doing a local search. Then, on the master, the number of occurrences found is assembled.

The conducted tests refer to a parallel search on a list of 10000 elements executed one hundred times in each worker, and a speedup of 3 times with 6 CPUs and 5.20 with 12 CPUs was obtained.

Table 4. Obtained elapsed time for the parallel number of occurrences problem

Workers	CPUs	Elapsed Time	Speedup
1	2	03:23.2s	1.00
3	6	01:08.1s	3.00
6	12	00:42.5s	5.20

So far, we have only considered synthetic benchmarks. That means we have only considered artificial programs that try to match the characteristics of large programs. Although they are fine for testing, real benchmarks can only be obtained from testing real-world applications. We also tested PM2-Prolog with OpenArp [1], and obtained a speedup of 2.24 times with 6 CPUs and 2.50 times with 12 CPUs.

Table 5. Obtained elapsed time for parallel anaphora resolution

Workers	CPUs	Elapsed Time	Speedup
1	2	00:05.0s	1.00
3	6	00:02.2s	2.24
6	12	00:02.0s	2.50

Quantifying the obtained speedup is important but very dependent on the problem we are dealing with and on its degree of parallelization. Also of note, the elapsed time shown already excludes the time for the empty problem.

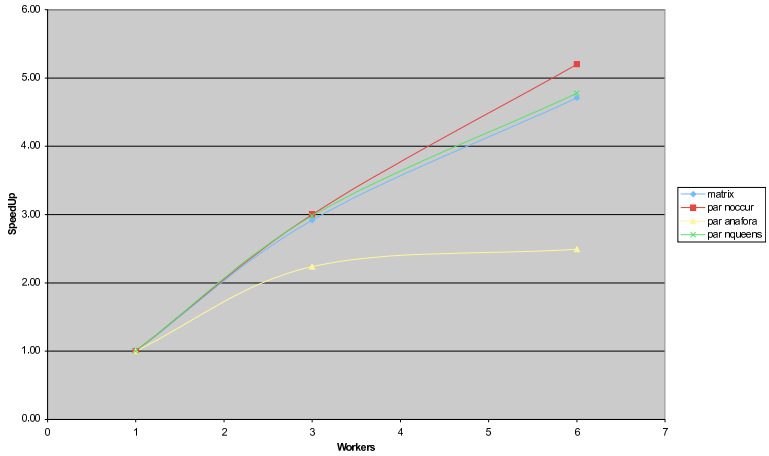


Fig. 4. Speedup with an increasing number of workers defined as elapsed time using one worker divided by elapsed time using N workers.

Looking at figure 4 we observe that the speedup is almost linear for the first three examples. That shows that the system scales without problems, at least until the considered number of workers.

Since the cluster used in our benchmarking exercise is relatively small (7 nodes) we can't always observe the point of speedup convergence for all the tested programs, but in the case of the anaphora resolution system we can observe that the speedup is unlikely to reach more than 3 times, independent of the number of workers used.

Another issue that should be taken into account is the correctness of the results. What happened while distributing OpenArp, in which we chose to modify the search-space of the algorithm in order to distribute the problem, affecting herewith the correctness of the results, will probably happen with other real-world applications. If this separation affects somehow the algorithm of the

application, by e.g. modifying the search-space, the accuracy of the system will also be affected. Being so, the accuracy of the results must be assessed in order to verify if the obtained speedup compensates the loss.

In summary, the results show the model is valid and can obtain good performance gains, even when the number of distributed machines is low.

5 Conclusions and Future Work

A system that allows the development of distributed multi-threaded applications in GNU Prolog on top of the PM2 programming environment was developed.

It is at a prototype state and will need further development and testing. The conducted tests to evaluate the preliminary performance of our system used three classic literature problems plus a real-world application and measured the obtained speedup in each one using one worker (sequential version), three workers and six workers.

In summary, the results show the model is valid and can obtain good performance gains, even when the number of distributed machines is low.

The conducted tests obtained an almost linear speedup on the first three problems. On a more real-world application, OpenArp, we obtained speedup but relatively less comparing with the other tested programs. Our results also showed how the accuracy of an application might be affected by distributing it's algorithm. In OpenArp this happened because we modified the search-space of the algorithm instead of using a parallel algorithm for anaphora resolution, if such algorithm is even possible. The case has been made to show that if a parallel algorithm isn't possible, then accuracy might be sacrified in favor of a speedup in execution time.

We noticed that concurrent Prolog programs perform very good and we feel encouraged to test bigger configurations. Performance, however, degrades quickly when using predicates that require synchronization or that make intensive use of the network. A solution for this issue might reside in the duplication of what is going to be passed over the network and send only a reference over the network. This might not be possible to execute in several scenarios, such as problems where the messages are created at runtime.

Other issues that are associated with the current implementation include:

- Many situations, if not handled carefully, can lead to deadlock, e.g. a thread not receiving the terminate message, due to an error, will cause the main thread to deadlock;
- Theads have to be terminated explicitly. It would increase performance if the threads terminated as soon as no more jobs are available. This would release CPU for other threads.

While working towards improving our proposal, solving these issues, we also want to pursuit several traits. These are:

- Extend the API with introspection and monitoring predicates. That will permit programmers to control better the running distributed program;

– Test the system with bigger configurations, namely with GRID, and more powerful applications;
– Use distributed multi-threading to build and control intelligent agents.

In a non-distributed multi-threaded environment, powerful applications are limited by the number of threads that can effectively run concurrently.

In a distributed multi-threaded environment resources are pooled and a scheduler sets the rules for routing the jobs to help optimize resources automatically, for accelerated results and help reducing processing time.

Prolog can play a fundamental part in the next generation of applications that will exploit multi-core architectures and bring concurrency to the masses. It will permit on many cases programs to have a declarative, logic interpretation and will allow for the programmer to omit most control, helping the expression of complex applications and algorithms. The developed system is a tool to help in this process.

References

1. Ana Aires, Jorge Coelho, Sandra Collovini, Paulo Quaresma, and Renata Vieira. *Avaliação de Centering em Resolução Pronominal da Língua Portuguesa*. 5th International Workshop on Linguistically Interpreted Corpora of the Iberamia'2004, 2004.
2. Khayri A. M. Ali and Roland Karlsson. *The Muse Or-parallel Prolog model and its performance*. MIT Press, Cambridge, MA, USA, 1990.
3. Olivier Aumage. *Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes*. Thèse de doctorat, spécialité informatique, École normale supérieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07, France, September 2002. 154 pages.
4. D. Chu. *I.C. Prolog II: A Multi-Threaded Prolog System*. Kluwer, Dordrecht, 1994.
5. Keith Clark and Steve Gregory. *PARLOG: parallel programming in logic*, volume 8. ACM Press, New York, NY, USA, 1986.
6. Keith L. Clark, Peter J. Robinson, and Richard Hagen. Multi-threading and message communication in qu-prolog. *CoRR*, 1:283–301, 2001.
7. Vítor Santos Costa, David H. D. Warren, and Rong Yang. The andorra-i preprocessor: Supporting full prolog on the basic andorra model. In *ICLP*, pages 443–456, 1991.
8. Daniel Diaz and Philippe Codognet. *The GNU prolog systems and its implementation*. In ACM Symposium on Applied Computing, Como, Italy, 2000.
9. Jesper Eskilson and Mats Carlsson. SICStus MT — A multithreaded execution environment for SICStus prolog. In *PLILP'98*, volume 1490, pages 36–53, 1998.
10. Ivan Futo. *Prolog with communicating processes: from T-Prolog to CSR-Prolog*. MIT Press, Cambridge, MA, USA, 1993.
11. Manuel V. Hermenegildo and K. J. Greene. The &-prolog system: Exploiting independent and-parallelism. *New Generation Comput.*, 9(3/4):233–256, 1991.
12. E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H.D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The aurora or-parallel prolog system. *New Gen. Comput.*, 7(2-3):243–271, 1988.

13. R. Marques and J. Cunha. Pvm-prolog: A prolog interface to pvm, 1996.
14. Paulo Moura. *ISO/IEC DTR 13211-5:2007 Prolog Multi-Threading Support*. 2007.
15. Raymond Namyst and Jean-François Méhaut. *Parallel Computing: State-of-the-Art and Perspectives. Proceedings of the Intl. Conference ParCo '95, Ghent, Belgium, 19–22 September 1995*, volume 11 of *Advances in Parallel Computing*, chapter PM^2: Parallel Multithreaded Machine. A Computing Environment for Distributed Architectures, pages 279–285. Elsevier, February 1996.
16. Raymond Namyst and Jean-François Méhaut. *Marcel : Une bibliothèque de processus légers*. LIFL, Univ. Sciences et Techn. Lille, 1995.
17. Luis Moniz Pereira, Luis Monteiro, Jose Cunha, and Joaquim N Aparicio. *Delta Prolog: a distributed backtracking extension with events*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.
18. Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. YapOr: An or-parallel prolog system based on environment copying. In Pedro Barahona and José J. Alferes, editors, *Proceedings of the 9th Portuguese Conference on Progress in Artificial Intelligence (EPIA-99)*, volume 1695 of *LNAI*, pages 178–192, Berlin, September 21–24 1999. Springer.
19. Kish Shen. Exploiting dependent and-parallelism in prolog: The dynamic dependent and-parallel scheme (DDAS). In *JICSLP*, pages 717–731, 1992.
20. Jan Wielemaker. Native preemptive threads in SWI-prolog. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2003.
21. Michael J. Wise. Experience with PMS-Prolog: a distributed coarse-grain-parallel Prolog with processes, modules and streams. *Software—Practice and Experience*, 23(2):151–175, February 1993.

Multi-threading programming in Logtalk

Paulo Moura[1], Paul Crocker[1], and Paulo Nunes[2]

[1] Dep. of Computer Science, University of Beira Interior, Portugal
pmoura@di.ubi.pt crocker@di.ubi.pt
[2] Polytechnic Institute of Guarda, Portugal
pnunes@ipg.pt

Abstract. Logtalk, an object oriented logic programming language, provides experimental support for multi-threading programming with selected back-end Prolog compilers. By making use of core, low-level Prolog predicates that interface with operating-system native threads, Logtalk provides a high-level set of directives and predicates that allows programmers to easily take advantage of modern multi-processor and multi-core computers without worrying about the details of creating, synchronizing, or communicating with threads. Logtalk multi-threading programming features include support for concurrent calls akin to and-parallelism and or-parallelism, non-deterministic thread goals, asynchronous calls, and predicate synchronization. The integration with the Logtalk object-oriented features allows objects to send and receive both synchronous and asynchronous messages and to call local predicates concurrently. Logtalk multi-threading features are orthogonal to object-oriented concepts and can be useful even in the context of plain Prolog.

Introduction

In recent years, computers supporting multiple processors and multi-core processors have become mainstream. Major players in the hardware business such as Intel, AMD, or IBM provide complete lines of multi-core processors for desktop and portable computers. In fact, nowadays, we have to look hard to buy a single-core personal computer. Coupled with the support for multi-threading applications found on current operating systems, there is a strong incentive to migrate from pure sequential programs to programs that take performance and responsiveness advantages from using multiple threads.

Writing multi-threading applications implies using programming languages that provide the necessary support for thread creation, synchronization, and communication. One of the most commonly used multi-threading Application Programming Interface (API) is defined by the POSIX standard. The POSIX threads API or, as commonly know, *pthreads*, is a set of C functions dealing with thread management, mutual exclusion, and condition variables[3] [1, 2]. Given that most Prolog compilers are implemented in C or C++, pthreads is a common

[3] Condition variables allow the implementation of notification mechanisms where a thread suspends execution until some condition becomes true.

choice for providing core, low-level multi-threading built-in predicate support. However, despite threads being a powerful programming mechanism, it is easy to get into trouble when failing to properly synchronize threads accessing shared resources such as input/output streams and dynamic state. Although there are always tasks where a low-level multi-threading API is necessary, programming scenarios where a simpler, high-level interface is preferred are common. Recently, high level multi-threading programming constructs for imperative languages have become popular. For example, the OpenMP API [3, 4] implements high level programming constructs for shared-memory parallel applications, working as a pre-processor for C, C++ and Fortran. Another example is Intel's Threading Building Blocks [5], which provides high-level, task-based parallelism to C++. In the case of Prolog, earlier attempts to automate code parallelization proved difficult due to language semantic issues, e.g. order-dependency between goals. Nevertheless, extensive research [6] has resulted in a number of successful experimental systems, such as e.g. Andorra-I [7] and Muse [8]. These systems suffer from maintenance and portability problems, however, stemming from the complexity of their inner workings. Therefore, we cannot always rely on them for industrial applications. Logtalk [9, 10] takes a more pragmatic approach, striving for a simple and minimal set of directives and built-in predicates that allows programmers to easily take advantage of modern multi-processor and multi-core computers without worrying about low-level details of creating, synchronizing, or communicating with threads. Our work is motivated by past experiences with multi-agents systems (mostly using Logtalk with Peter Robinson's Qu-Prolog) and by a current project on the validation of large CAD/CAM data model files [11] where most steps are natural candidates for parallelization due to their independence and being side-effects free. Logtalk multi-threading development is guided by four main goals: (1) simple support for making concurrent calls, mostly for parallelizing independent computations; (2) support for asynchronous calls, where we can start a computing thread, perform some other tasks, and later retrieve the thread goal solutions; (3) simple directives for predicates that need to be synchronized due to side-effects; (4) a portable and robust implementation, capable of running with several back-end Prolog compilers in most operating-systems. Interestingly, these goals are orthogonal to Logtalk object-oriented features. Although objects provides an execution context for our multi-threading predicates and directives, where we take advantage of objects encapsulation and of objects local database, our results can also be applied in the context of plain Prolog (complementing, not replacing, core low-level multi-threading support).

This paper begins by describing the core support found on current Prolog compilers for multi-threading programming, used as a foundation for our work. Second, the Logtalk multi-threading programming features are presented and discussed. A brief comparison with related work follows. We conclude by discussing the current status of our work. Full documentation, complete code of the examples, and the implementation of the multi-threading features described in this paper are available with the current Logtalk distribution. The reader is invited to try out and give feedback on the actual system.

1 Starting point: Prolog multi-threading core support

Prolog compilers such as SWI-Prolog [12], YAP [13], Qu-Prolog [14, 15], BinProlog [16, 17], XSB [18, 19], or Ciao [20] provide a low-level, comprehensive set of built-in predicates supporting multi-threading programming. Most of these Prolog compilers make use of pthreads or, for some operating systems, of a suitable emulation. A recent ISO standardization proposal [21], started in April 2006, aims to specify a common core of low-level multi-threading programming support based on the semantics of POSIX threads[4]. We have decided to base the Logtalk high-level support for multi-threading programming on this common interface. The current Logtalk version supports multi-threading programming using both SWI-Prolog and YAP as back-end Prolog compilers; we expect soon to be able to support XSB and later Qu-Prolog, pending on-going work on the implementation of the current standardization proposal.

The current ISO standardization proposal specifies a comprehensive set of predicates for thread, mutex, and message queue management. It also includes a set of predicates for querying and setting thread creation default options. Most of these options deal with the different per-thread memory areas such as the stacks used by the Prolog implementation. The size of these memory areas is specially relevant for 32-bit architectures. Setting the default size values so that they cover most cases without the need of hand-tuning may result in a severe limitation on the maximum number of threads we can create before exhausting the memory address space[5]. Prolog implementations differ on their memory handling mechanisms. For heavily multi-threaded applications, implementations using *stack-shifters* for keeping default memory sizes small, dynamically expanding memory only when necessary, have an advantage over implementations that allocate large chunks of virtual memory space to simplify memory handling, relying on the operating system virtual memory mechanisms. This is important for a high-level multi-threading API such as the one provided by Logtalk, where it is not desirable to force the programmer to worry about such low-level details as the default thread stack size.

2 Logtalk multi-threading support: overview

Logtalk multi-threading programming is supported by a small set of built-in predicates and directives, which can be regarded as a high-level API complementing, not replacing, the core, lower-level API provided by selected Prolog compilers. This high-level API can be split in three groups of predicates and a set of directives. The first group contains a single predicate, `threaded/1`, which supports concurrent calls akin to and-parallelism and or-parallelism. The second group of

[4] The standardization group includes so far SWI-Prolog, YAP, Qu-Prolog, XSB, and Ciao developers. The proposal is currently being edited by the first author of this paper. Collaboration from other interested parties is most welcome.

[5] Note that we are talking about virtual memory space; actually used memory is often much less.

predicates provide support for asynchronous calls, here interpreted as separating proving a goal from the retrieval of the goal solutions. Two basic predicates are provided, `threaded_call/1` and `threaded_exit/1`, supporting non-deterministic thread goals. From these two predicates, we derive three specialized predicates: `threaded_once/1`, `threaded_ignore/1`, and `threaded_peek/1`. The third group of predicates allows thread synchronization using notifications, which are arbitrary, programmer-defined non-variable terms. Notifications are used as a peer-to-peer mechanism supported by the predicates `threaded_wait/1` and `threaded_notify/1`. The Logtalk multi-threading directives include two object directives, `threaded/0` and `synchronized/0`, and a predicate directive, `synchronized/1`, enabling an object to make multi-threading calls and supporting object and predicate-level synchronization. Logtalk multi-threading predicate calls always take place within the context of an object[6]. Thus, objects are able to send and receive both synchronous and asynchronous messages and to call local predicates concurrently. In the following sections, we provide a detailed account of Logtalk multi-threading support, illustrated with several examples, with an emphasis on the technical aspects of the current implementation.

3 Object message queues

Logtalk object message queues are used whenever an object defines predicates that make concurrent calls or asynchronous calls. In order to automatically create and set up an object message queue the `threaded/0` object directive is used:

```
:- threaded.
```

The object message queue is created when the object is compiled and loaded into memory or when created at runtime (the message queue for the pseudo-object *user* is automatically created at Logtalk startup). These message queues are used internally for storing replies to the threaded calls made from within the objects themselves and for exchanging thread notifications, as we will discuss later. The implicit use of object message queues for storing and exchanging thread results provides a cleaner and simpler alternative to the explicit use of blackboards or the dynamic database, as found on some other systems.

It is tempting to make this directive optional or simply forgo it, thus simplifying Logtalk multi-threading programming. In most cases, the Logtalk compiler could simply set up the creation of the object message queue when finding a call to a multi-threading built-in predicate in the body of an object predicate. However, it is always possible to construct new goals at runtime that call the multi-threading built-in predicates. In addition, an object may import a category[7] whose predicates make multi-threading calls (see section 6.3 for an example). Creating the object message queue on the fly is certainly possible but

[6] When at the Logtalk top-level interpreter, the execution context is the pseudo-object *user*.

[7] Logtalk categories are object building blocks (components), which can be virtually imported (without code duplication) by any object, irrespective of inheritance relations.

would lead to runtime errors if the back-end Prolog compiler does not support all the core multi-threading features Logtalk relies on. Therefore, we choose to make the `threaded/0` object directive mandatory. This allows us to both check at compile time for proper back-end Prolog compiler support and to cope with threaded goals generated at runtime in ways that cannot be anticipated by the Logtalk compiler.

4 Making concurrent calls

Logtalk provides a basic multi-threading built-in predicate, `threaded/1`, which supports concurrent calls akin to both and-parallelism and or-parallelism. In this context, and-parallelism and or-parallelism refers to using, respectively, a conjunction of goals and a disjunction of goals as a predicate argument. This built-in predicate is deterministic and opaque to cuts. Each goal in its argument is proved in its own thread (except when the argument is neither a conjunction not a disjunction of goals, in which case no threads are created for proving it and the predicate is equivalent to the standard Prolog built-in predicate `once/1`). Goals can be calls to local object predicates, messages to *self*, or messages to other objects. Thus, both local predicates and other object methods can be called concurrently.

4.1 And-parallelism

When the argument is a conjunction of goals, the `threaded/1` predicate call blocks the caller thread until either one of thread goals fails, rises an exception, or all the implicit thread goals succeed. A failure or an exception leads to the immediate termination of the other threads. The *and-parallelism* functionality of the `threaded/1` predicate covers a common programming pattern on multi-threading applications: parallelizing a set of independent computations. Here, independent computations translate to a conjunction of goals with no shared variables. Thus, each goal can be proved in parallel without worrying about synchronizing variable instantiations or suspending a thread goal until a variable is instantiated. Nevertheless, it turns out that forbidding the use of shared variables is over-restrictive and, with care, the programmer can sometimes use shared variables to further improve performance. For example, assume that we want to find all prime numbers in a given interval using two threads. We could write:

```
prime_numbers(N, M, Primes) :-
    M > N,
    N1 is N + (M - N) // 2,
    N2 is N1 + 1,
    threaded((
        prime_numbers(N2, M, [], Acc),
        prime_numbers(N, N1, Acc, Primes)
    )).
```

In this simple example, the two `prime_numbers/4` goals in the `threaded/1` predicate call share a variable (`Acc`) that acts as an accumulator, allowing us to avoid a call to an `append/3` predicate at the end (which would cancel part of the performance gains of using multi-threading). At a user level, sharing variables meets the expectations of a programmer used to single-threading programming and suggests easy parallelization of single-threaded code by simply wrapping-around goals in `threaded/1` predicate calls. At the implementation level, sharing variables between thread goals is problematic as the core Prolog thread creation predicates make a copy of the thread goal, thus loosing variable bindings. When a thread goal terminates, the variable bindings are reconstructed by Logtalk in the process of retrieving the goal solutions. I.e. shared variables are only synchronized after thread termination. A failure to synchronize shared variables results in the failure of the `threaded/1` call. Depending on how each goal uses the shared variables, their use may lead to other problems. For example, a predicate call may depend on a shared variable being instantiated in order to behave properly. This will not work as the thread goals are independently proved. Safe use of shared variables implies that the individual thread goals do not depend on their instantiation, as in the example above where the shared variable is used only as an accumulator. Research on these cases, which are examples of *non-strict independent and-parallelism*, is described on [22].

4.2 Competing threads: reinterpreting goal disjunction

The `threaded/1` predicate allows a disjunction of goals to be interpreted as a set of *competing* goals, each one running in its own thread. The first thread to terminate successfully leads to the termination of the other threads. Thus, the goals in a disjunction compete for a solution instead of being interpreted as possibly providing alternative solutions. This is useful when we have several methods to compute something, together with several processors or cores available, without knowing a priori which method will be faster or able to converge into a solution. For example, assume that we have implemented several methods for calculating the roots of real functions. We may then write:

```
find_root(Function, A, B, Error, Zero, Method) :-
    threaded((
            bisection::find_root(Function, A, B, Error, Zero),
            Method = bisection
        ;   newton::find_root(Function, A, B, Error, Zero),
            Method = newton
        ;   muller::find_root(Function, A, B, Error, Zero),
            Method = muller
    )).
```

The `threaded/1` call returns both the identifier of the fastest method and its result. Depending on the function and on the initial interval, one method may converge quickly into the function root while other method may simply diverge, never finding it. This is a pattern typical of other classes of algorithms (e.g.

graph path-finding methods or matrix eigenvalues calculation methods), making the `threaded/1` predicate useful for a wide range of problems.

It is important to stress that only the first successful goal on a disjunction can lead to the instantiation of variables on the original argument. Thus, we do not need to worry about the representation of multiple bindings of the same variable across different disjunction goals.

The effectiveness of this predicate relies on two factors: the ability to cancel the slower threads once a wining thread completes and the number of cores available. Canceling a thread is not always possible or as fast as desirable as a thread can be in a state where no interrupts are accepted. Aborting a thread is tricky in most multi-threading APIs, including pthreads. In the worst case scenario, some slower threads may run up to completion. Most current laptop and desktop computers contain two, four, or eight cores, making the possible waste of processing power by slower, non cancelable threads questionable. The number of cores per-processor is expected to rise steadily over the next few years with each new generation of processors. However, past experience have shown that performance does not grow linearly with the number of cores, due to increasing contention problems (e.g. memory access). Therefore, the real world usefulness of the `threaded/1` predicate *or-parallelism* functionality is an open question whose answer is most likely application-domain dependent.

5 Making asynchronous calls

Logtalk provides two basic multi-threading built-in predicates, `threaded_call/1` and `threaded_exit/1`, which allows us to make asynchronous calls and to later retrieve the corresponding results. Paired `threaded_call/1` and `threaded_exit/1` calls must be made from within the same object. An asynchronous call can be either a call to a local object predicate or a message sending call. Being asynchronous, a call to the `threaded_call/1` predicate is always true and results in the creation of a new thread for proving its argument. In addition, no variable binding occurs as a consequence of the call. The thread results (goal success, failure, or exception) are posted to the message queue of the execution context object. A simple example:

```
| ?- threaded_call(sort([3,1,7,4,2,9,8], Sorted)).

Sorted = _G189
yes

| ?- threaded_exit(sort([3,1,7,4,2,9,8], Sorted)).

Sorted = [1,2,3,4,7,8,9]
yes
```

This example shows how a `threaded_exit/1` call picks up the solutions from a `threaded_call/1` with a matching goal argument. When multiple threads run a matching goal, the `threaded_exit/1` call picks up the first thread to add a

goal solution to the message queue of the execution context object. Calls to the `threaded_exit/1` predicate block the caller until the object message queue receives the reply to the asynchronous call. Logtalk provides a complementary predicate, `threaded_peek/1`, which may be used to check if a reply is already available without removing it from the object message queue. The `threaded_peek/1` predicate call succeeds or fails immediately without blocking the caller. However, repeated use of this predicate is equivalent to polling a thread queue, which may severely hurt performance.

5.1 Non-deterministic goals

Asynchronous calls are often deterministic. Typically, they are used for performing some lengthy computation without blocking other aspects of an application. A common example is decoupling an interface from background computing threads. Nevertheless, Logtalk also allows non-deterministic asynchronous calls. The basic idea is that a computing thread suspends itself after providing a solution, waiting for a request for an alternative solution. For example, assuming a `lists` object implementing a `member/2` predicate, we could write:

```
| ?- threaded_call(lists::member(X, [1,2,3])).

X = _G189
yes

| ?- threaded_exit(lists::member(X, [1,2,3])).

X = 1 ;
X = 2 ;
X = 3 ;
no
```

In this case, the `threaded_call/1` and the `threaded_exit/1` calls are made within the pseudo-object *user*, whose message queue is used internally to store computed goal solutions. The implicit thread running the `lists::member/2` goal suspends itself after providing a solution, waiting for the request of an alternative solution; the thread is automatically terminated when the runtime engine detects that further backtracking to the `threaded_exit/1` call is no longer possible.

Supporting non-deterministic thread goals can be tricky as the thread is suspended between requests for alternative solutions: if a new request never occurs, the result could be a zombie thread. The current Logtalk implementation solves this problem by taking advantage of the `call_cleanup/2` built-in predicate found on some Prolog compilers such as SICStus Prolog, SWI-Prolog, YAP, B-Prolog and development versions of Qu-Prolog. This predicate allows us to call a *clean-up* goal as soon as the Prolog runtime detects that a goal is finished because it succeeded or failed deterministically or because its choice-points have been cut[8].

[8] This functionality cannot be implemented at the Prolog level, making the availability of this built-in predicate an additional requirement for running Logtalk multi-

There is one caveat when using the `threaded_exit/1` predicate that a programmer must be aware of, especially when using this predicate within failure-driven loops. When all the solutions have been found (and the thread generating them is therefore terminated), further calls to the predicate will generate an exception as the answering thread no longer exists. Note that failing instead of throwing an exception is not an acceptable solution as it could be misinterpreted as a failure of the thread goal.

For deterministic asynchronous calls, Logtalk provides a `threaded_once/1` built-in predicate that is more efficient when there is only one solution or when you want to commit to the first solution of the thread goal. In this case, the thread created for proving a goal stores the first solution on the message queue of the object making the `threaded_once/1` call and terminates. The solution thus becomes available for later retrieval by a call to the `threaded_exit/1` predicate.

5.2 One-way asynchronous calls

Sometimes we want to prove a goal in a new thread without caring about the results. This may be accomplished by using the built-in predicate `threaded_ignore/1`. For example, assume that we are developing a multi-agent application where an agent may send an *happy birthday* message to another agent. We could simply write:

```
..., threaded_ignore(agent::happy_birthday), ...
```

This call succeeds with no reply of the goal success, failure, or even exception ever being sent back to the message queue object making the call (note that this predicate implicitly implies a deterministic call of its argument).

6 Dealing with side effects: synchronizing predicate calls

Proving goals in a multi-threading environment may lead to problems when the goals imply side-effects such as input/output operations or modifications to an object database. For example, if a new thread is started with the same goal before the first one finished its job, we may end up with mixed output, a corrupted database, or unexpected goal failures.

The usual solution for synchronizing calls is to use semaphores, mutexes, or some other similar mechanism. In the case of the multi-threading ISO standardization proposal, a set of built-in predicate for working with mutexes is already specified. We could certainly use them to synchronize predicate calls. However, working at this level, implies naming, locking, and unlocking mutexes. This is a task best performed by the compiler and the language runtime rather than the programmer who should only need to worry about declaring which predicate calls should be synchronized.

threading applications with a specific back-end Prolog compiler. Standardization of this predicate is currently being discussed.

In Logtalk, predicates (and grammar rule non-terminals) with side-effects can be simply declared as synchronized by using either the `synchronized/0` object directive or the `synchronized/1` predicate directive. Together, these two directives allows from object-level synchronization to predicate-level synchronization. Proving a query to a synchronized predicate (or synchronized non-terminal) is protected internally by a mutex, thus allowing for easy thread synchronization.

6.1 Object-level synchronization

The `synchronized/0` object directive allows us to synchronize all object predicates using the same mutex:

```
:- synchronized.
```

This directive provides the simplest possible synchronization solution; it is useful for small objects where all or most predicates access the same shared resources.

6.2 Predicate-level synchronization

When fine-grained synchronization is preferred, the `synchronized/1` predicate directive allows us to synchronize subsets of an object predicates or a single object predicate. For example, the following two directives:

```
:- synchronized([write_buffer/1, read_buffer/1]).
:- synchronized(random/1).
```

will make calls to the `write_buffer/1` and `read_buffer/1` predicates synchronized using the same mutex while the predicate `random/1` will use a different mutex.

6.3 Synchronizing predicate calls through notifications

Declaring a set of predicates as synchronized can only ensure that they are not executed at the same time by different threads. Sometimes we need to suspend a thread not on a synchronization lock but on some condition that must hold true for a thread goal to proceed. I.e. we want a thread goal to be suspended until a condition becomes true instead of simply failing. The built-in predicate `threaded_wait/1` allows us to suspend a predicate execution (running in its own thread) until a notification is received. Notifications are posted using the built-in predicate `threaded_notify/1`. A notification is a Prolog term that a programmer chooses to represent some condition becoming true. Any Prolog term can be used as a notification argument for these predicates. Related calls to the `threaded_wait/1` and `threaded_notify/1` must be made within the same object as its message queue is used internally for posting and retrieving notifications. Each notification posted by a call to the `threaded_notify/1` predicate is consumed by a single `threaded_wait/1` predicate call, i.e. these predicates implement a peer-to-peer mechanism. Care should be taken to avoid deadlocks when two (or more) threads both wait and post notifications to each other.

To see the usefulness of this notification mechanism consider the *dining philosophers* problem [23]: five philosophers sitting at a round table, thinking and eating, each sharing two chopsticks with its neighbors. Chopstick actions (picking up and putting down) can be easily synchronized using a notification such that a chopstick can only be handled by a single philosopher at a time:

```
:- category(chopstick).

    :- public([pick_up/0, put_down/0]).

    pick_up :-
        threaded_wait(available).

    put_down :-
        threaded_notify(available).

:- end_category.
```

There are five chopsticks, therefore we need to define the corresponding five objects. The code of all of them is similar. E.g.:

```
:- object(cs1,
    imports(chopstick)).

    :- threaded.
    :- initialization(threaded_notify(available)).

:- end_object.
```

This and other examples of the use of notifications to synchronize threads are provided with the current Logtalk distribution for the interested reader. Common usage patterns are generate-and-test scenarios where size-limited buffers are used for intermediate storage of candidate solutions. In these scenarios, a producer thread needs to suspend when the buffer is full, waiting for the consumer thread to notify it of available spots. Likewise, a consumer thread needs to suspends when the buffer is empty, waiting for the producer thread to notify it that new items are available for consumption.

7 Performance

Preliminary tests show that the performance of Logtalk multi-threading applications scales as expected with the number of threads used, bounded by the number of processing cores. The following table shows the speedup as we increase the number of threads in three simple benchmark tests: calculating primes numbers and sorting lists using the merge sort and the quicksort algorithms.

Benchmark · Number of threads	**1**	**2**	**4**
Prime numbers (in the interval [1, 500000])	1.00	1.65	3.12
Merge sort (20000 float random numbers)	1.00	1.87	2.87
Quicksort (20000 float random numbers)	1.00	1.43	1.82

The corresponding multi-threading examples can be found on the current Logtalk distribution. The tests are performed on an Apple MacPro Dual 3.0GHz Dual-Core Intel Xeon 5100 with 2GB of RAM, running MacOS X 10.4.10. The back-end Prolog compiler used was SWI-Prolog 5.6.37.

Use of multi-threading features is interesting for problems where the computation costs surpasses the overhead of thread creation and management. Part of this overhead is operating-system dependent. E.g. we found that, on the hardware described above, Linux provide the fastest thread creation and thread join results, followed by Windows XP SP2, and than MacOS X 10.4. For practical applications, experimentation is necessary in order to fine-tune a multi-threading solution given the problem complexity, the number of processing cores, the back-end Prolog compiler, and the operating-system.

8 Related work

Besides the Prolog compilers currently implementing the ISO standardization proposal, a number of other Prolog compilers provide alternative implementations of multi-threading concepts. Two of these compiler are BinProlog and Ciao Prolog, which we briefly discuss below. A prototype multi-threading version of SICStus Prolog is described in [24]. Outside the scope of Prolog compilers, Erlang [25, 26] is one of the best known examples of declarative programming languages supporting concurrent (and distributed) systems.

8.1 BinProlog

BinProlog provides a set of multi-threading built-in predicates, ranging from simple, high-level predicates to lower-level predicates that give increasing control to the programmer. As this paper deals essentially with high-level predicates, two BinProlog predicates stand out. The predicate `bg/1` allows a goal to be proved in its own thread. The predicate `synchronize/1` uses an implicit mutex to prevent two threads of executing its argument concurrently.

Most BinProlog multi-threading examples use a blackboard for storing and retrieving thread goal results. The programmer must use the blackboard explicitly. In contrast, the use of object message queues by Logtalk for exchanging thread goal results is transparent to the programmer.

BinProlog supports thread synchronizing using *thread guards*. Thread guards, which work as mutexes, can be generated by calling the `new_thread_guard/1` predicate and used with the predicates `thread_wait/1`, `thread_notify/1`, and `thread_notify_all/1`. Despite the name similarity, these predicates are quite different from the Logtalk `threaded_wait/1` and `threaded_notify/1` predicates where notifications are arbitrary non-variable Prolog terms chosen by the programmer.

8.2 Ciao Prolog

Ciao Prolog supports a concept of *engines*, which are used for proving a goal using a separate set of memory areas. These engines can use an operating-system

thread for proving a goal, therefore providing support for concurrency. Similar to other Prolog compilers, goals are copied into engine, thus loosing variable bindings. Ciao provides a set of predicates for managing engines that rely on the use of goal identifiers to reference a specific engine. Goal identifiers play a role similar to the thread identifiers found on other Prolog compilers. The Prolog database is shared between threads and is used as the primary means of thread communication and synchronization. Ciao makes use of *concurrent* predicates, which are dynamic predicates that allow thread execution to be suspended until a new clause is asserted. Ciao supports non-deterministic thread goals, providing a `eng_backtrack/2` predicate to backtrack over thread goals. When a thread goal fails, the engine is not automatically terminated; the programmer must explicitly call a `eng_release/1` predicate. This contrasts with Logtalk where a thread is automatically terminated when the thread goal fails. It is possible that the implementation in Ciao of a functionality similar to the one provided by the `call_cleanup/2-3` predicate would also allow transparent engine release.

8.3 SWI-Prolog high-level multi-threading library

As we finish writing this paper, a new high-level multi-threading library was just committed to the SWI-Prolog CVS server. This library provides two predicates, `concurrent/3` and `first_solution/3`, which provide functionality similar to the Logtalk predicate `threaded/1`. The `concurrent/3` predicate allows easy concurrent execution of a set of goals. The caveats listed in the SWI-Prolog library documentation are basically the same that apply to Logtalk and to every other Prolog compiler making a copy of a goal when using a thread: users of this predicate are advised against using shared goal variables. This seems to be more of a cautious advise for safe use the `concurrent/3` predicate than an implementation limitation (note that both this SWI-Prolog library and Logtalk rely on the same core multi-threading built-in predicates). The predicate `first_solution/3` runs a set of goals concurrently and picks the first one to complete, killing the other threads. This predicate shares with the Logtalk `threaded/1` predicate the same potential thread cancelation problem: a thread may be in a state where no signals are being processed, delaying or preventing thread cancelation. The SWI-Prolog library predicates allows the user to specify a list of options that will be used by the underlying calls to the core `thread_create/3` predicate. Thus, simpler predicates using the default thread creation options are trivial to implement.

9 Conclusions and future work

Logtalk currently uses a Prolog system as a back-end compiler, including for core multi-threading services. The features described in this paper could be implemented at a lower level, arguably with some performance gains (e.g. by minimizing the overhead of thread creation). The downside would be loosing the broad compatibility of Logtalk with Prolog compilers. Although currently only

a small number of Prolog compilers provide the necessary interface to POSIX threads (or a suitable emulation), we expect its number to grow in the future.

Logtalk shares with some Prolog implementations the goal of finding useful high-level multi-threading primitives. Most high-level multi-threading predicates are supported by the same or similar core, low-level features. Therefore, a convergence and cross-fertilization of research results is expected and desirable. For example, Logtalk predicates such as `threaded/1` and the `synchronized/0-1` directives would be useful even in plain Prolog. Further experimentation and real-world usage will eventually show which high-level multi-threading predicates are worthwhile to implement across systems.

Our current work focus on documenting functionality, developing programming examples, and testing our implementation for robustness and compatibility across Prolog compilers and operating systems. The specification of the multi-threading predicates and directives is considered stable. The Logtalk multi-threading features will soon drop their experimental status to become available for using in production systems.

Work on the ISO draft standard proposal for Prolog multi-threading support [27] is progressing steadily. The current Logtalk implementation uses only a small subset of the proposed thread predicates. An improved implementation may be possible using a more complete Prolog interface to POSIX threads. In fact, the major reason for the Logtalk multi-threading features to be classified as experimental is due to the lack of a final standard specification that can be relied on for all the compliance testing necessary for writing robust portable code.

10 Acknowledgments

We are grateful to Peter Robinson, Jan Wielemaker, Vitor Santos Costa, Terrance Swift, and Rui Marques for their ground work implementing Prolog multi-threading core support and for helpful suggestions and discussions on the subject of this paper. This work has been partially funded by the Fundação para a Ciência e Tecnologia, Portugal.

References

1. ISO/IEC. *International Standard ISO/IEC 9945-1:1996. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API).* ISO/IEC, 1996.
2. David R. Butenhof. *Programming with POSIX Threads.* Professional Computing Series. Addison Wesley, 1997.
3. Rohit Chandra, Leonardo Dagum, David Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2001.
4. OpenMP: Simple, Portable, Scalable SMP Programming. `http://www.openmp.org/`.
5. Intel Threading Building Blocks. `http://threadingbuildingblocks.org/`.

6. Gopal Gupta, Enrico Pontelli, Khayri Ali, Mats Carlsson, and Manuel Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.
7. Vítor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. *SIGPLAN Not.*, 26(7):83–93, 1991.
8. Khayri Ali and Roland Karlsson. The Muse Or-parallel Prolog model and its performance. In *Proceedings of the 1990 North American conference on Logic programming*, pages 757–776, Cambridge, MA, USA, 1990. MIT Press.
9. Paulo Moura. *Logtalk – Design of an Object-Oriented Logic Programming Language*. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal, September 2003.
10. Paulo Moura. Logtalk web site. `http://logtalk.org/`.
11. Paulo Moura and Vincent Marchetti. Logtalk Processing of STEP Part 21 Files. In S. Etalle and M. Truszczyński, editors, *International Conference on Logic Programming 2006*, number 4079 in Lecture Notes in Computer Science, pages 453–454, Berlin Heidelberg, August 2006. Springer-Verlag.
12. Jan Wielemaker. Native preemptive threads in SWI-Prolog. In Catuscia Palamidessi, editor, *Practical Aspects of Declarative Languages*, pages 331–345, Berlin, Germany, December 2003. Springer Verlag. LNCS 2916.
13. Vitor Santos Costa. YAP Home Page. `http://www.ncc.up.pt/~vsc/Yap/`.
14. Keith L. Clark, Peter Robinson, and Richard Hagen. Multi-threading and Message Communication in Qu-Prolog. *Theory and Practice of Logic Programming*, 1(3):283–301, 2001.
15. Peter Robinson. Qu-prolog web site. `http://www.itee.uq.edu.au/~pjr/HomePages/QuPrologHome.html`.
16. Paul Tarau. *BinProlog 2006 11.x Professional Edition – Advanced BinProlog Programming and Extensions Guide*, 2006.
17. Paul Tarau. BinNet Corporation. BinProlog Home Page. `http://www.binnetcorp.com/BinProlog/`.
18. The XSB Research Group. *The XSB Programmer's Manual: version 3.0.1*, 2007.
19. The XSB Research Group. XSB Home Page. `http://xsb.sourceforge.net/`.
20. Manuel Carro and Manuel Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *International Conference on Logic Programming*, pages 320–334, 1999.
21. Paulo Moura (editor). ISO/IEC DTR 13211–5:2007 Prolog Multi-threading predicates. `http://logtalk.org/plstd/threads.pdf`.
22. Manuel V. Hermenegildo and Francesca Rossi. Strict and Nonstrict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
23. Wikipedia. Dining philosophers problem. `http://en.wikipedia.org/wiki/Dining_philosophers_problem`.
24. Jesper Eskilson and Mats Carlsson. SICStus MT—A Multithreaded Execution Environment for SICStus Prolog. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1490 of *Lecture Notes in Computer Science*, pages 36–53. Springer-Verlag, 1998.
25. Pekka Hedqvist. A Parallel and Multithreaded Erlang Implementation. Master's thesis, Uppsala University, Uppsala, Sweden, June 1998.
26. Erlang Home Page. `http://www.erlang.org/`.
27. Jonathan Hodgson. ISO/IEC/ JTC1/SC22/WG17 Official Home Page. `http://www.sju.edu/~jhodgson/wg17/wg17web.html`.

Towards High-Level Execution Primitives for And-parallelism: Preliminary Results

Amadeo Casas[1] Manuel Carro[2] Manuel V. Hermenegildo[1,2]
{amadeo, herme}@cs.unm.edu
{mcarro, herme}@fi.upm.es

[1] Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, USA.
[2] School of Computer Science, Univ. Politécnica de Madrid, Spain.

Abstract. Most implementations of parallel logic programming rely on complex low-level machinery which is arguably difficult to implement and modify. We explore an alternative approach aimed at taming that complexity by raising core parts of the implementation to the source language level for the particular case of and-parallelism. Therefore, we handle a significant portion of the parallel implementation mechanism at the Prolog level with the help of a comparatively small number of concurrency-related primitives which take care of lower-level tasks such as locking, thread management, stack set management, etc. The approach does not eliminate altogether modifications to the abstract machine, but it does greatly simplify them and it also facilitates experimenting with different alternatives. We show how this approach allows implementing both restricted and unrestricted (i.e., non fork-join) parallelism. Preliminary experiments show that the amount of performance sacrificed is reasonable, although granularity control is required in some cases. Also, we observe that the availability of unrestricted parallelism contributes to better observed speedups.

Keywords: Parallelism, Virtual Machines, High-level Implementation.

1 Introduction

Parallel computers have finally become mainstream with the advent of multicore processors. As a result, there is a renewed interest in languages and tools to simplify the task of writing parallel programs. The use of declarative languages and, among them, logic programming ones is considered an interesting approach for obtaining increased performance through parallel execution on multicore architectures, including multicore embedded systems. The high-level nature of these languages allows coding in a style that is closer to the application and thus preserves more of the original parallelism for automatic parallelizers to uncover. Their amenability to semantics-preserving automatic parallelization [20] is also due, in addition to this high level of abstraction, to their relatively simple semantics, and the separation between the control component from the declarative specification. This makes it possible for the evaluator to execute some operations in any order (which in some cases subsumes parallel execution), without affecting the meaning of the program. In addition, logic variables can be assigned at most one value, and thus it is not necessary to check for some types of flow dependencies or to perform SSA transformations when compared to imperative languages. At the same time, in most other respects the presence of dynamic data structures with "declarative pointers" (logical variables), irregular computations, or complex control

makes the parallelization of logic programs a particularly interesting case that allows tackling the more complex parallelization-related challenges in a formally simple and well-understood context [11].

A wealth of research on parallel execution of logic programs has been done so far (see, e.g., [10] and its references). Two main forms of goal-level parallelism (i.e., parallelism which happens between different goals of a search tree, as opposed to, e.g., unification parallelism) have been exploited in the execution of logic programs. *Or-parallelism* parallelizes the execution of different clauses of a predicate (and their continuations) and is naturally applicable to programs featuring implicit search. Some notable systems that exploit or-parallelism are Aurora [18] and MUSE [2]. *And-parallelism* refers to the parallel execution of different goals in the resolvent. It arises naturally in different kinds of applications (independently of whether there is implicit search or not), of which divide-and-conquer algorithms are notable cases. Examples of systems that have exploited and-parallelism are &-Prolog [12], DDAS [24], and others. Also, some systems such as &ACE [21], AKL [16], and Andorra [23] have exploited certain combinations of both and- and or-parallelism.

The basic ideas of the &-Prolog model have been adopted by many other systems (e.g., &ACE and DDAS). It consists of two components: a parallelizing compiler which detects the possible runtime dependencies between goals in clause bodies and annotates the clauses with expressions to decide whether parallel execution can be allowed at runtime, and a run-time system that exploits that parallelism. The run-time system is based on an extension of the original WAM architecture and set of instructions, and was originally implemented, as most of the other systems mentioned, on shared-memory multiprocessors, although distributed implementations were also taken into account. We will follow the same overall architecture and assumptions herein, and concentrate as well on (modern) shared-memory, multicore processors.

While these models and their implementations have been shown very effective at exploiting parallelism efficiently and obtaining significant speedups, most of them are based on quite complex, low-level machinery which makes implementation and maintenance of these systems inherently hard. In this paper we explore an alternative approach to the implementation of parallelism in logic programs that is based on raising components of the implementation to the source language level and keeping at low level only selected operations related to, e.g., thread handling and locking. We expect of course a performance hit, but hope that this division of concerns will make it possible to more easily explore variations on the execution schemes. While doing this, another objective of our proposal is to be able to easily exploit non-restricted and-parallelism, i.e., parallelism that is not restricted to fork-join operations.

2 Classical Approaches to And-Parallelism

In goal-level and-parallelism, a key issue is which goals to select for parallel execution in order to avoid situations which lead to incorrect execution or slowdown [15, 11]. Not only errors but also significant inefficiency can arise from the simultaneous execution of computations which depend on each other since, for example, this may trigger more backtracking than in the sequential case. Thus, goals are said to be independent if their parallel execution will not perform additional search and will not produce incorrect results. Very general notions of independence have been developed, based on constraint theory [9]. However for simplicity we discuss only those based on variable sharing.

In *Dependent and-parallelism (DAP)* goals are executed in parallel even if they share variables, and the competition to bind them has to be dynamically dealt with using notions such as sequencing bindings from producers to consumers. Unfortunately this usually implies substantial execution overhead. In *Strict Independent and-parallelism (SIAP)* goals are allowed to execute in parallel only when they do not share variables, which guarantees the correctness and no-slowdown. *Non-strict independent and-parallelism (NSIAP)* is a significant extension, also guaranteeing the no-slowdown property, in which goals are parallelized even if they share variables, provided that at most one goal binds a shared variable or the goals agree in the possible bindings for shared variables. Compile-time tools have been devised and implemented to statically detect cases where this holds, thus making the runtime machinery lighter and faster. Undetermined cases can, if deemed advantageous, be checked at runtime.

Another issue is whether any restrictions are posed on the patterns of parallelization. For example, *Restricted and-parallelism (RAP)* constrains parallelism to (nested) fork-join operations. In the &-Prolog implementation of this model conjunctions which are to be executed in parallel are often marked by replacing the sequential comma (`,/2`) with a parallelism operator (`&/2`).

In this paper we will focus on the implementation of IAP and NSIAP parallelism, as both have practically identical implementation requirements. Our objective is to exploit both restricted and unrestricted, goal-level and-parallelism.

Once a method has been devised for selecting goals for parallel execution, an obviously relevant issue is how to actually implement such parallel execution. One usual implementation approach used in many and-parallel systems (both for IAP [12, 21] and for DAP [24]) is the *multi-sequential, marker model* introduced by &-Prolog. In this model parallel goals are executed in different *abstract machines* which run in parallel. In order to preserve sequential speed, these abstract machines are extensions of the sequential model, usually the Warren Abstract Machine (WAM) [26, 1], which is the basis of most efficient sequential implementations. Herein we assume for simplicity that each (P)WAM has a parallel thread (an "agent") attached and that we have as many threads as processors. Thus, we can refer interchangeably to WAMs, agents, or processors. Within each WAM, sequential fragments appear in contiguous stack sections exactly as in the sequential execution.[3] The new data areas are [12]:

Goal Stack: A shared area onto which goals that are ready to execute in parallel are pushed. WAMs can pick up goals from other WAMs' (or their own) goal stacks. Goal stack entries include a pointer to the environment where the goal was generated and to the code starting the goal execution, plus some additional control information.

Parcall Frames: They are created for each parallel conjunction and hold the necessary data for coordinating and synchronizing the parallel execution of the goals in the parallel conjunction.

Markers: They separate stack sections corresponding to different parallel goals. When a goal is picked up by an agent, an *input marker* is pushed onto the choicepoint stack. Likewise, an *end marker* is pushed when a goal execution ends. These are

[3] In some proposals this need not be so: *continuation markers* [25] allow sequential execution to spread over non-contiguous sections. We will not deal with that issue here.

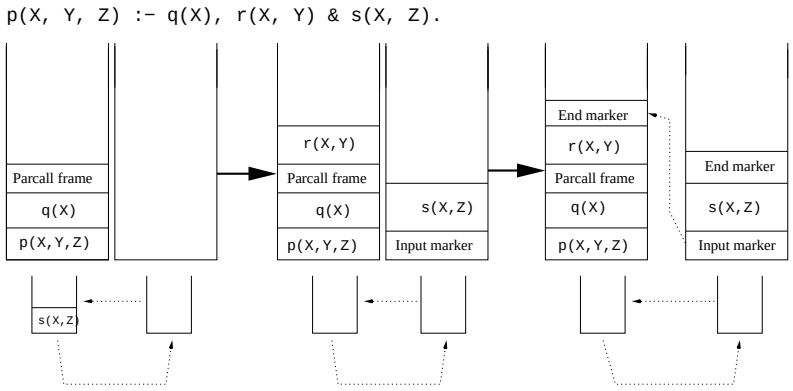


Fig. 1. Sketch of data structures layout using the marker model.

linked to ensure that backtracking will happen following a logical (i.e., not physical) order.

Figure 1 sketches a possible stack layout for a program such as:

```
p(X, Y, Z) :- q(X), r(X, Y) & s(X, Z).
```

with query `p(X, Y, Z)`. We assume that `X` will be ground after calling `q/1`. Different snapshots of the stack configurations are shown from left to right. Note that in the figure we are intermixing parcall frames and markers in the same stack. Actual implementations have chosen to place them in different parts of the available data areas.[4]

When the first WAM executes the parallel conjunction `r(X, Y) & s(X, Z)`, it pushes a parcall frame onto its stack and a goal descriptor onto its goal stack for the goal `s(X, Z)` (i.e., a pointer to the WAM code that will construct this call in the argument registers and another pointer to the appropriate environment), and it immediately starts executing `r(X, Y)`. A second WAM, which is looking for jobs, picks `s(X, Z)` up, pushes an input marker into its stack (which references the parcall frame, where data common to all the goals is stored, to be used in case of *internal failure*) and constructs and starts executing the goal. An end marker is pushed upon completion. When the last WAM finishes, it will link the markers (so as to proceed adequately on backtracking and unwinding), and execution will proceed with the continuation of `p/3`.

Classical implementations using the marker model handle the `&/2` operator at the abstract machine level: the compiler recognizes `&/2` and compiles it by issuing specific WAM instructions, which are executed by a modified WAM implementation. These modifications are far from trivial, although relatively isolated (e.g., unification instructions are usually not changed, or changed in a generic, uniform way).

As mentioned in the introduction, one of our objectives is to explore an alternative implementation approach based on raising components to the source language level and keeping at low level only selected operations. Also, we would like to avoid modifications to the low-level compiler. At the same time, we want to be able to easily exploit non-restricted and-parallelism, i.e., parallelism that is not restricted to fork-join operations. These two objectives are actually related in our approach because, as we

[4] For example, in &ACE parcall frames are not pushed onto the environment stack but on a different stack, and their slots are allocated in the heap, to simplify the memory management.

will see in the following section, we will start by decomposing the parallelism operators into lower-level components which will also allow supporting non-restricted and-parallelism.

3 Decomposing And-Parallelism

It has already been reported [5, 4] that it is possible to construct the and-parallel operator `&/2` using more basic yet meaningful components. In particular, it is possible to implement the semantics of `&/2` using two end-user operators, `&>/2` and `<&/1`, defined as follows:[5]

- `G &> H` schedules goal `G` for parallel execution and continues with the code after `G &> H`. `H` is a *handler* which contains (or *points to*) the state of goal `G`.
- `H <&` waits for the goal associated with `H` (`G`, in the previous item) to finish. At that point all bindings `G` could possibly generate are ready, since `G` has reached a solution. Assuming goal independence between `G` and the calls performed while `G` was being executed, no binding conflicts should arise.

`G &> H` ideally takes a negligible amount of time to execute, although the precise moment in which `G` actually starts depends on the availability of resources (primarily, free agents/processors). On the other hand, `H <&` suspends until the associated goal finitely fails or returns an answer. It is interesting to note that the approach shares some similarities with the concept of *futures* in parallel functional languages. A future is meant to hold the return value of a function so that a consumer can wait for its complete evaluation. However, the notions of "return value" and "complete evaluation" do not make sense when logic variables are present. Instead, `H <&` waits for the moment when the producer goal has completed execution, and the "received values" (a tuple, really) will be whatever (possibly partial) instantiations have been produced by such goal.

Note that with the previous definitions, the `&/2` operator can be expressed as:

```
A & B :- A &> H, call(B), H <&.
```

Concrete implementations will of course expand `A & B` at compile time using the above definition in order not to pay the price of an additional call and the meta-call. The same can be applied to `&>` and `<&`.

However, these two new operators can additionally be used to exploit more and-parallelism than is possible with `&/2` alone [8]. We will just provide some intuition by means of a simple example (a performance evaluation is included in Section 5.) [6]
Consider predicate `p/3` defined as follows:

```
p(X,Y,Z) :- a(X,Z), b(X), c(Y), d(Y,Z).
```

whose (strict) dependencies (assuming that `X,Y,Z` are free and do not share on entry) are shown in Figure 2. A classical fork-join parallelization is shown in Figure 3,

[5] We concentrate on forward execution. For backtracking behavior see [5, 4] and Section 4.5. Also, although exception handling is beyond our current scope, in general exceptions uncaught by a parallel goal surface at the corresponding `<&/1`, where they can be captured by the parent.

[6] Note that the `&>/2` and `<&/1` operators do not replace the fork-join operator `&/2` at the language level due to its conciseness in cases in which no extra parallelism can be exploited with `&>/2` and `<&/1`.

while an alternative (non fork-join) parallelization using the new operators is shown in Figure 4. We assume here that solution order is not relevant.

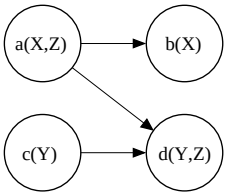


Fig. 2. Dependency graph for `p/3`.

It is obvious that it is always possible to parallelize programs using `&>/2` and `<&/1` and obtain the same parallelism as with `&/2` (since `&/2` can be defined in terms of `&>/2` and `<&/1`). The converse is not true. Furthermore, there are cases (as in Figure 4) where the parallelizations allowed by `&>/2` and `<&/1` can be expected to result in shorter execution times, for certain goal execution times [8]. In our example, the annotation in Figure 3 misses the possible parallelism between the subgoals `c/1` and `b/1`, which the code in Figure 4 allows: `c/1` is scheduled at the beginning of the execution, and it is waited for in `Hc <&`, just after `b/1` has been scheduled for parallel execution.

In addition to `&>/2` and `<&/1`, we propose specialized versions in order to obtain additional functionality or more efficiency. In particular, `&!>/2` and `<&!/1` are intended to perform the same actions as `&>/2` and `<&/1` respectively, but only for single-solution, non-failing goals, where there is no need to anticipate backtracking during forward execution. These primitives allow the parallelizer to flag directly in the parallelization goals that analysis has detected to be deterministic and non-failing (see [14]), and this can result in important simplifications in the implementation.

4 Sketch of a Shared Memory Implementation

Our proposed implementation divides responsibilities among several layers. User-level parallelism and concurrency primitives intended for the programmer and parallelizers are at the top and written in Prolog. Below, goal publishing, searching for available goals, and goal scheduling are written at the Prolog level, relying on some low-level support primitives for, e.g., locking or low-level goal management. with a Prolog interface but written in C.

In our current implementation for shared-memory multiprocessors, and similarly to [12], agents wait for work to be available, and execute it if so. Every agent is created as a thread attached to an (extended) WAM stack set. Sequential execution proceeds as usual, and coordination with the rest of the agents is performed by means of shared data structures. Agents make new work available to other agents through a *goal list* which is associated with every stack set and which can be consulted by all the agents.

In the following subsections we will introduce the library with the (deterministic) low-level parallelism primitives and we will present the design (and a sketch of the actual code, simplified for space reasons) of the main source-level algorithms used to

```
p(X, Y, Z):-
        a(X, Z) & c(Y),
        b(X) & d(Y, Z).
```

Fig. 3. Nested fork-join annotation.

```
p(X, Y, Z) :-
        c(Y) &> Hc,
        a(X, Z),
        b(X) &> Hb,
        Hc <&,
        d(Y, Z),
        Hb <&.
```

Fig. 4. Using the new operators.

run deterministic, non-failing goals in parallel. We will conclude with some comments on the execution of nondeterministic goals in parallel.

4.1 Low-Level Parallelism Primitives

The low-level layer has been implemented as a Ciao library ("**apll**") written in C which provides basic mechanisms to start threads, wait for their completion, push goals, search for goals, access to O.S. locks, etc. Most of these primitives need to refer to an explicit goal and need to use some information related to its state (whether it has been taken, finished, etc.). Hence the need to pass them a `Handler` data structure which abstracts information related to the goal at hand.

The current (simplified) list of primitives follows. Note that this is not intended to be a general-purpose concurrency library (such as those available in Ciao and other Prolog systems ——in fact, very little of what should appear in such a generic library is here), but simply a list of primitives suitable for efficiently implementing at a higher-level different approaches to exploiting independent and-parallelism.

`apll:push_handler(+Goal,+Det,-Handler)` atomically creates a unique *handler* (an opaque structure) associated to `Goal` and publishes `Goal` in the goal list for any agent to pick it up. `Handler` will henceforth be used in any operation related to `Goal`. `Det` describes whether `Goal` is deterministic or not.

`apll:find_handler(-Handler)` searches for a goal published in some goal list. If one exists, `Handler` is unified with a handler for it; the call fails otherwise, and it will succeed at most once per call. Goal lists are accessed atomically so as to avoid races when updating them.[7]

`apll:at_goal_list(+Handler)` succeeds if `Handler` has not been picked up yet, and fails otherwise.

`apll:retrieve_goal(+Handler,-Goal)` unifies `Goal` and the goal initially associated to `Handler`.

`apll:goal_finished(+Handler)` succeeds if the execution state of the goal associated to `Handler` is finished, and fails otherwise.

`apll:set_finished(+Handler)` sets to finished the execution state of the goal associated to `Handler`.

`apll:waiting(+Handler)` succeeds when the execution state of the publishing agent of the goal associated to `Handler` is suspended and fails otherwise.

Additionally, a set of locking primitives is provided to synchronize thread executions and to obtain mutual exclusion at the Prolog level. Every agent uses two different locks: one which is used to ensure mutual exclusion when dealing with shared data structures (i.e., when adding new goals to the list), and another one which is used to synchronize the agent waking up when `<& / 1` is waiting for either more work to be available, or the execution of a goal picked up by some other agent to finish. Both can be accessed with specific (`*_self`) predicates to specify the ones belonging to the calling agent. Otherwise, they are accessed through a goal `Handler` to refer to the locks belonging to the agent which created the goal `Handler` refers to (i.e., its *creator*).

[7] Different versions exist of this primitive which can be used while implementing different goal scheduling strategies.

apll:suspend suspends the execution of the calling thread.
apll:release(+Handler) releases the execution of the agent which created `Handler` (which could have suspended itself with the above described predicate).
apll:release_some_suspended_thread selects one out of any suspended threads and resumes its execution.
apll:enter_mutex(+Handler) attempts to enter into a mutual exclusion to access shared variables of the agent associated to `Handler`.
apll:enter_mutex_self same as above, with the agent's own mutex.
apll:exit_mutex(+Handler) signals the locks in the realm of the creator of `Handler` to exit mutual exclusion.
apll:exit_mutex_self same as above with the calling thread.

4.2 High-level Goal Publishing

Based on the previous primitives, we develop the user-level ones. We will implement a particular strategy in which rather than, e.g., having idle agents busily looking for work, such agents are suspended and resumed in a more organized way depending on availability of work (this strategy is also the one used in our experiments).

```
Goal &!> Handler :-
   apll:push_handler(Goal,det,Handler),
   apll:release_some_suspended_thread.
```

Fig. 5. Publishing a (deterministic) parallel goal.

A call to `&!>/2` (or `&>/2` if the goal is nondeterministic) implies publishing the goal in the goal list managed by the agent, i.e., making it available for other agents to pick up for execution. Figure 5 shows the (simplified) Prolog code implementing this functionality (again, the code shown can be expanded in line but is shown as a meta-call for clarity). First, a pointer to the goal generated is inserted in the goal list. Second, the current thread will signal any suspended agents that there is new work available. As we will see later, the agent receiving the signal will resume its execution, pick up the new parallel goal, and start its execution.

After executing `Goal &!> H`, the *handler* `H` will hold the state of `Goal`, which can be inspected both by the thread which publishes `Goal` and by any thread which picks up `Goal` to execute it. Therefore, in some sense, `H` takes the role of the *parcall frame* in [12], but it is not placed in the environment — it goes to the heap instead. Threads can communicate and synchronize through the handler in order to consult and update the state of `Goal`. This is especially important when executing `H <&!`.

4.3 Performing Goal Joins

Figure 6 provides code implementing `<&!/1` (the deterministic version of `<&/1`). First, the thread needs to check whether the goal has been picked up by some other thread, using `apll:at_goal_list/1`. If the goal has not been picked up yet by another agent then the publishing agent will execute it locally and `<&!/1` will succeed trivially.

If the goal has been picked up by another agent and its execution has finished then `<&!/1` will automatically succeed. The bindings made during goal execution are, naturally, available (we are dealing with a shared-memory implementation). If the goal execution has not finished yet then the thread will not suspend right away. Instead, it will search for more work in order to keep itself busy, and it will only suspend if there is definitely no work to perform at the moment. This ensures that overall efficiency is kept

```
Handler <&! :-
    apll:enter_mutex_self,
    (
        apll:at_goal_list(Handler) ->
        apll:retrieve_goal(Handler,Goal),
        apll:exit_mutex_self,
        call(Goal)
    ;
        apll:exit_mutex_self,
        perform_other_work(Handler)
    ).
```

```
perform_other_work(Handler) :-
    apll:enter_mutex_self,
    (
        apll:goal_finished(Handler) ->
        apll:exit_mutex_self
    ;   (
            find_goal_and_execute ->
            true
        ;
            apll:exit_mutex_self,
            apll:suspend
        ),
        perform_other_work(Handler)
    ).
```

Fig. 6. Goal join with continuation.

at a reasonable level, as we will see in Section 5. Races when accessing shared variables are avoided using locks for mutual exclusion and conditional synchronization.

Figure 7 presents the source code which searches for a goal available and executes it. `find_goal_and_execute/0` will fail if there is no goal available. If one is found then the thread will pick it up, execute it, mark it as finished and resume the execution of the publishing agent, if suspended. In that case, the publishing agent (suspended in `eng_suspend/0`) will check which situation applies after resumption and act accordingly after recursively invoking the predicate `perform_some_work/1`.

4.4 Agent Creation

Agents are generated using the `create_agents/1` predicate which launches a number of O.S. threads using the `start_thread/0` predicate imported from a generic concurrency library. Every of these threads execute continuously the `agent/0` code which takes care of searching for more work or suspending, if that is the case (Figure 8). Thus, during normal execution agents are either sleeping because there is nothing to execute or working on some goal. We assume for simplicity that agent creation is in general performed at system startup or just before starting a parallel execution. Higher-level predicates are however provided in order to manage threads in a more flexible way. For instance, `ensure_agents/1` makes sure that a given number of executing agents is available. In fact, agents can be created lazily, and added or deleted dynamically as needed, depending on machine load. However, this interesting issue of thread throttling is beyond the scope of this paper.

```
find_goal_and_execute :-
    apll:find_handler(Handler),
    apll:exit_mutex_self,
    apll:retrieve_goal(Handler,Goal),
    call(Goal),
    apll:enter_mutex(Handler),
    apll:set_finished(Handler),
    (
        apll:waiting(Handler) ->
        apll:release(Handler)
    ;
        true
    ),
    apll:exit_mutex(Handler).
```

Fig. 7. Finding a parallel goal and executing it.

```
create_agents(0) :- !.
create_agents(N) :-
    N > 0,
    conc:start_thread(agent),
    N1 is N - 1,
    create_agents(N1).

agent :-
    apll:enter_mutex_self,
    (
        find_goal_and_execute ->
        true
    ;
        apll:exit_mutex_self,
        apll:suspend
    ),
    agent.
```

Fig. 8. Creating parallel agents.

4.5 Towards Non-determinism

For simplicity we have left out of the discussion and also of the code the support for backtracking, which clearly complicates things. We have made significant progress in our implementation towards supporting backtracking following the marker model, so that for example the failure-driven top level is used unchanged and memory is recovered orderly at the end of parallel executions. However, completing the implementation of backtracking is still the matter of current work.

There are interesting issues both at the design level and also at the implementation level. An interesting point at the design level is for example deciding whether backtracking happens when going over `&>/2` or `<&/1` during backward execution. Previous work [5, 4] leaned towards the latter, which is also probably easier to implement; however, there are also reasons to believe that the former may in the end be more appropriate. For example, in parallelized loops such as:

```
p([X|Xs]):- b(X) &> Hb, p(Xs), Hb <&.
```

spawning `b(X)` and keeping the recursion local and not the other way around is important because task creation is the real bottleneck. However, if backtracking occurs at `<&` solution order would not be preserved, whereas it would if backtracking occurs at `&>`. Note that in such loops the loss of LCO is only of relative importance, since if there are several solutions to either `b/1` or `p/1`, LCO could not be applied anyway.

At the implementation level, there is for example the issue of avoiding the "trapped goal" and "garbage slots" problems [13]. One approach we are considering to this end is to move trapped stack segments (sequential sections of execution) to the top of the stack set in case backtracking is needed from a trapped section. We can also later compact sections which become empty to avoid garbage slots. In order to express this at the Prolog level, we foresee the need of additional primitives, still the subject of further work, to manage stack segments as first-class citizens.

Another fundamental idea in the approach that we are exploring is not to create markers explicitly, but use instead, for the same purpose, standard choice points built by creating alternatives (using alternative clauses) directly in the control code (in Prolog) that implements backtracking.

5 Experimental Results

We now present performance results obtained after executing a selection of well-known benchmarks with independent and-parallelism. As mentioned before, we have implemented the proposed approach in Ciao [3], an efficient system designed with extension capabilities in mind. All results were obtained by averaging ten runs on a state-of-the-art multiprocessor, a Sun Fire T2000 with 8 cores, and 8 Gb of memory. While each core is capable in theory of running 4 threads in parallel, and in theory up to 32 threads could run simultaneously on this machine we only show speedups up to 8 agents. Our experiments (see the later comments related to Figure 11), show that speedups with more than 8 threads stop being linear even for completely independent computations (i.e., 32 totally independent threads do not really speed up as if 32 independent processors were available), as threads in the same core compete for shared resources such as integer pipelines, etc. Thus, beyond 8 agents, it is hard to know whether reduced speedups are due to our parallelization and implementation or to limitations of the machine.

AIAKL	Simplified *AKL* abstract interpreter.	**Hamming**	Calculates *Hamming* numbers.
Ann	Annotator for and-parallelism.	**Hanoi**	Solves *Hanoi* puzzle.
Boyer	Simplified version of the *Boyer-Moore* theorem prover.	**MergeSort**	Sorts a 10000 element list.
Deriv	Symbolic derivation.	**MMatrix**	Multiplies two 50×50 matrices.
FFT	Fast Fourier transform.	**Palindrome**	Generates a palindrome of 2^{14} elements.
Fibonacci	Doubly recursive *Fibonacci*.	**QuickSort**	Sorts a 10000 element list.
FibFun	Functional *Fibonacci*.	**Takeuchi**	Computes *Takeuchi*.
		WMS2	A work scheduling program.

Table 1. Benchmarks for restricted and unrestricted IAP.

Although most of the benchmarks we use are quite well-known, Table 1 provides a brief description. Speedups appear in Tables 2 (which contains only programs parallelized using restricted [N]SIAP, as in Figure 3) and 3 (which additionally contains unrestricted IAP programs, as in Figure 4). The speedups are with respect to the sequential speed on one processor of the original, unparallelized benchmark. Therefore, the columns tagged *1* correspond to the slowdown coming from executing a parallel program in a single processor. Benchmarks with a *GC* suffix were executed with granularity control with a suitably chosen threshold and benchmarks with a *DL* suffix use difference lists and require NSIAP for parallelization. All the benchmarks in the tables were automatically parallelized using CiaoPP [14] and the annotation algorithms described in [8] (*TakeuchiGC* needed however some unfolding in order to uncover and allow exploiting more parallelism using the new operators, as discussed later).

It can be deduced from the results that in several benchmarks the *natural* parallelizations produce small granularity. This, understandably, impacts our implementation since a sizable part of it is written in Prolog, which implies additional overhead in the preparation and execution of parallel goals. Thus, it is not possible to perform a fair comparison of the speedups obtained with respect to previous (lower-level) and-parallel systems. The overhead implied by the proposed approach produces comparatively low performance on a single processor and in some cases with very fine granularity, such as Boyer and Takeuchi, speedups are shallow (below $2\times$) even over 8 processors. In these examples execution is dominated by the sequential code of the scheduler and agent management in Prolog. However, even in these cases, setting a granularity threshold based on a measure of the input argument size [17] much better results can be obtained. Figure 9 depicts graphically the impact of granularity control in some benchmarks. Annotating the parallelized program to take into account granularity measures based on size of the input arguments, and automatically finding out the optimal threshold for a given platform, can be done automatically in many cases [17, 19].

Table 3 shows a different comparison: some programs have traditionally been executed under IAP using the restricted (nested fork-join) annotations, and can be annotated for parallelism using the more flexible `&>/2` and `<&/1` operators, as in Figures 3 and 4. In some cases those programs obtain little additional speedup, but interestingly in other cases the gains are very relevant. An interesting example is the Takeuchi function which underwent a manual (but mechanical) transformation involving an *unfolding* step, which produced a clause where non-nested fork-join can be taken advantage of, producing a much better speedup. This can be clearly seen in Figure 10. Note that the speedup curve did not seem to stabilize even when the 8 processor mark was reached.

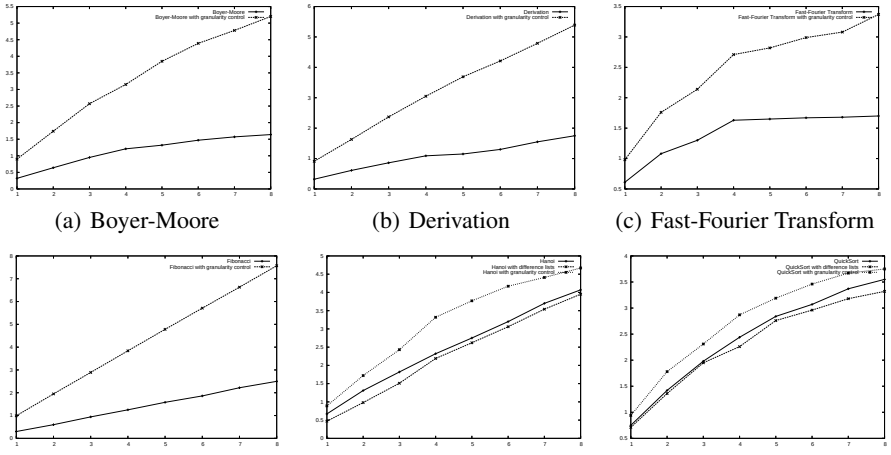
Benchmark	Number of processors								
	Seq.	1	2	3	4	5	6	7	8
AIAKL	1.00	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
Ann	1.00	0.98	1.86	2.65	3.37	4.07	4.65	5.22	5.90
Boyer	1.00	0.32	0.64	0.95	1.21	1.32	1.47	1.57	1.64
BoyerGC	1.00	0.90	1.74	2.57	3.15	3.85	4.39	4.78	5.20
Deriv	1.00	0.32	0.61	0.86	1.09	1.15	1.30	1.55	1.75
DerivGC	1.00	0.91	1.63	2.37	3.05	3.69	4.21	4.79	5.39
FFT	1.00	0.61	1.08	1.30	1.63	1.65	1.67	1.68	1.70
FFTGC	1.00	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
Fibonacci	1.00	0.30	0.60	0.94	1.25	1.58	1.86	2.22	2.50
FibonacciGC	1.00	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Hamming	1.00	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
Hanoi	1.00	0.67	1.31	1.82	2.32	2.75	3.20	3.70	4.07
HanoiDL	1.00	0.47	0.98	1.51	2.19	2.62	3.06	3.54	3.95
HanoiGC	1.00	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
MergeSort	1.00	0.79	1.47	2.12	2.71	3.01	3.30	3.56	3.71
MergeSortGC	1.00	0.83	1.52	2.23	2.79	3.10	3.43	3.67	3.95
MMatrix	1.00	0.91	1.74	2.55	3.32	4.18	4.83	5.55	6.28
Palindrome	1.00	0.44	0.77	1.09	1.40	1.61	1.82	2.10	2.23
PalindromeGC	1.00	0.94	1.75	2.37	2.97	3.30	3.62	4.13	4.46
QuickSort	1.00	0.75	1.42	1.98	2.44	2.84	3.07	3.37	3.55
QuickSortDL	1.00	0.71	1.36	1.95	2.26	2.76	2.96	3.18	3.32
QuickSortGC	1.00	0.94	1.78	2.31	2.87	3.19	3.46	3.67	3.75
Takeuchi	1.00	0.23	0.46	0.68	0.91	1.12	1.32	1.49	1.72
TakeuchiGC	1.00	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63

Table 2. Speedups for restricted IAP.

Benchmark	Parallelism	Number of processors								
		Seq.	1	2	3	4	5	6	7	8
FFTGC	Restricted	1.00	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	Unrestricted	1.00	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
FibFunGC	Restricted	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Unrestricted	1.00	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Hamming	Restricted	1.00	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	Unrestricted	1.00	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
TakeuchiGC	Restricted	1.00	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	Unrestricted	1.00	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72
WMS2	Restricted	1.00	0.99	1.01	1.01	1.01	1.01	1.01	1.01	1.01
	Unrestricted	1.00	0.99	1.10	1.10	1.10	1.10	1.10	1.10	1.10

Table 3. Speedups for unrestricted IAP.

The FibFun benchmark is also an interesting case. A definition of Fibonacci was written in Ciao using the functional package [7] which implements a rich functional syntactic layer via compilation to the logic programming kernel. The automatic translation into predicates does not produce however the same Fibonacci program that programmers usually write (input parameters are calculated right before making the recursive calls), and it turns out that it cannot be directly parallelized using existing order-preserving annotators and restricted IAP. On the other hand it can be automatically



(a) Boyer-Moore (b) Derivation (c) Fast-Fourier Transform

(d) Fibonacci (e) Hanoi (f) QuickSort

Fig. 9. Speedups for some selected benchmarks with and without granularity control.

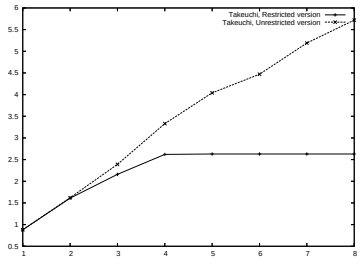


Fig. 10. Restricted and unrestricted IAP versions of *Takeuchi*.

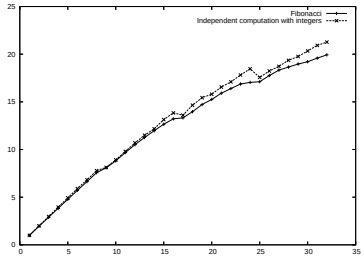


Fig. 11. *Fibonacci* with granularity control vs. maximum speedup in a real machine.

parallelized (including the translation from functional to logic programming notation) using the unrestricted operators.

Despite our observation that the T2000 cannot produce linear speedups beyond 8 processors even for independent computations, we wanted to try at least a Prolog example using as many threads as natively available in the machine, and compare its speedup with that of a C program generating completely independent computations. Such C program provides us with a practical upper bound on the attainable speedups. The results are depicted in Figure 11 which shows both the ideally parallel C program and a parallelized Fibonacci running on our implementation. Interestingly, the speedup obtained is only marginally worse than the best possible one. In both curves it is possible to observe a sawtooth shape, presumably caused by tasks filling in a row of units in all cores and starting to use up additional thread units in other cores, which happens at 1×8, 2×8, and 3×8 threads.

6 Conclusions

We have presented a new implementation approach for exploiting and-parallelism in logic programs with the objectives of simpler machinery and more flexibility. The ap-

proach is based on raising the implementation of some components to the source language level by using more basic high-level primitives than the fork-join operator and keeping only some relatively simple operations at a lower level. Our preliminary experimental results show that reasonable speedups are achievable with this approach, although the additional overhead, at least in the current implementation, makes it necessary to use granularity control in many cases in order to obtain good results. In addition, recent compilation technology and implementation advances [6, 22] provide hope that it will eventually be possible to recover a significant part of the efficiency lost due to the level at which parallel execution is expressed. Finally, we have observed that the availability of unrestricted parallelism contributes in practice to better observed speedups. We are currently working on improving the implementation both in terms of efficiency and of improved support for backtracking. We have also developed simultaneously specific parallelizers for this approach, which can take advantage of the unrestricted nature of the parallelism which it can support [8].

7 Acknowledgments

This work was funded in part by the IST program of the European Commission, FP6 FET project IST-15905 *MOBIUS*, by the Ministry of Education and Science (MEC) project TIN2005-09207-C03 *MERIT-COMVERS* and by the Madrid Regional Government CAM project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo and Amadeo Casas were also funded in part by the Prince of Asturias Chair in Information Science and Technology at UNM.

References

1. Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.
2. K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.
3. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.
4. D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.
5. D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.cliplab.org/`.
6. M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo. High-Level Languages for Small Devices: A Case Study. In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.
7. A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *FLOPS'06*, Fuji Susono (Japan), April 2006.

8. A. Casas, M. Carro, and M. Hermenegildo. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, The Technical University of Denmark, August 2007. Springer-Verlag.
9. M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems*, 22(2):269–339, March 2000.
10. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.
11. M. Hermenegildo. Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. *Parallel Computing*, 26(13–14):1685–1708, December 2000.
12. M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
13. M. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 40–55. Imperial College, Springer-Verlag, July 1986.
14. 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, October 2005.
15. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
16. Sverker Janson. *AKL. A Multiparadigm Programming Language*. PhD thesis, Uppsala University, 1994.
17. P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.
18. E. Lusk et al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.
19. E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.
20. K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.
21. E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–572. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.
22. Vítor Santos-Costa. Optimising Bytecode Emulation for Prolog. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *LNCS*, pages 261–277. Springer-Verlag, 1999.
23. Vítor Manuel de Morais Santos-Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I*. PhD thesis, University of Bristol, August 1993.
24. K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.
25. K. Shen and M. Hermenegildo. Flexible Scheduling for Non-Deterministic, And-parallel Execution of Logic Programs. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 635–640. Springer-Verlag, August 1996.
26. D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

Dealing with large predicates: exo-compilation in the WAM and in Mercury

Bart Demoen[&], Phuong-Lan Nguyen[@],
Vítor Santos Costa[#], and Zoltan Somogyi[*]

& Department of Computer Science, K.U.Leuven, Belgium
@ Institut de Mathématiques Appliquées, UCO, Angers, France
# LIACC/FCUP, Universidade do Porto, Portugal
* Department of Computer Science and Software Engineering,
University of Melbourne, Australia
bmd@cs.kuleuven.be, nguyen@ima.uco.fr, vsc@dcc.fc.up.pt, zs@csse.unimelb.edu.au

Abstract. Logic programming systems often need to deal with large but otherwise regular predicates, e.g. wide ground facts. Such predicates can be treated as any other predicate by the compiler, but there are good reasons to treat them specially, the most important being that separating the code from the data really pays off. We call the technique exo-compilation: it reduces the memory needed for the code to about one third of the normal WAM compilation schema without undue slowdown. As a bonus, queries with lots of void variables get a significantly better treatment. We first introduce the idea of exo-compilation by an example and present its implementation in hProlog. We show how other optimisations can be built on top of it, and evaluate how it performs in practice. We then show how the same ideas have been applied to the compilation of Mercury, whose implementation is based on very different principles.

1 Introduction

In some applications, logic programming systems must run queries over predicates consisting of a large set of *wide facts*, i.e. facts where the arity of the predicate is large. As one example, such predicates appear in Machine Learning *datasets* ranging from gene expression databases, where the biological study may include a large number of activity observations for the same gene, to medical reports, where the doctor may annotate a large number of possible conditions and parameters of interest, to film databases, where communities may store a large number of fields of interest per film and performer, and so on. Wide predicates also have been used to represent properties input stream, or to represent tables for scanner and parser generators.

One key observation is that most often these facts are *typed and moded nicely*, i.e. the facts are ground and the arguments are all constants. Traditional, WAM-based Prolog implementations do not exploit this situation. Moreover, such predicates are often called with only a few arguments instantiated (typically, but not

always, a key or sub-key of the relation), and most arguments are *void*. Note that in WAM parlance one says a variable is *void* if it is singleton in the query. The WAM does have limited support for void variables that can be recognised at compile-time. However, in this case, different queries will have different void variables. Therefore, the WAM must process void variables as it processes any other logical variable, forcing both the caller and the callee side to do an amount of work that is linear in the number of these variables. We would prefer to avoid that work altogether.

We show in Section 2 how both issues (wide facts and void variables) can be addressed by a technique where data and code are separated and which was named *exo-compilation* by the third author. We discuss next how exo-compilation was implemented in hProlog (see [6]): Section 3 contains an experimental evaluation thereof. Section 4 discuss how exo-compilation can be applied to other forms of regular code. The ideas of exo-compilation also apply to other logic programming systems. Section 5 shows how Mercury deals with large predicates: there is a great deal of similarity between the emulator and compiler approaches, but it is worth showing the details in both contexts. Section 6 discusses related work and concludes.

2 Exo-compilation

We first introduce some conventions that will be useful when showing abstract machine code.

- when an instruction refers to the i^{th} WAM argument register, we denote that by A(i), as in `getatom A(3), foo`
- the instruction *try* (and others) takes as argument a number that represents an arity, say 3 - we denote this as `try arity(3)`

Other operands are adorned in similarly way, to make clearer what they stand for.

When an atom (like foo) or a functor (bla/3) is used as an operand of an instruction - and in an exo-table (see later) - we actually mean the internal tagged representation of the atom or functor. Such a representation typically fits in a machine word.

We use $@x$ to denote an address labeled x.

2.1 The basic idea

We start from a predicate p/3 which consists of 4 facts and whose arguments are atoms:

```
p(a1,b1,c1).
p(a2,b2,c2).
p(a3,b3,c3).
p(a4,b4,c4).
```

For explanatory reasons, we go through some steps before arriving at the final code we want to generate: in the final code the instructions are separated from the data. For now, we ignore both indexing and instruction merging: they are orthogonal issues. The WAM compiles the above predicate to code such as can be seen in the left column below:

```
                                |       set_exo_pointer @t ------------------> a1 b1 c1
    try_me_else arity(3) @2     |       try_me_else_exo arity(3) @2           a2 b2 c2
    getatom A(1) a1             |       getatom_exo A(1)                      a3 b3 c3
    getatom A(2) b1             |       getatom_exo A(2)                      a4 b4 c4
    getatom A(3) c1             |       getatom_exo A(3)
    proceed                     |       proceed
@2: retry_me_else arity(3) @3   |   @2: retry_me_else_exo arity(3) @3
    getatom A(1) a2             |       getatom_exo A(1)
    getatom A(2) b2             |       getatom_exo A(2)
    getatom A(3) c2             |       getatom_exo A(3)
    proceed                     |       proceed
@3: retry_me_else arity(3) @4   |   @3: retry_me_else_exo arity(3) @4
    getatom A(1) a3             |       getatom_exo A(1)
    getatom A(2) b3             |       getatom_exo A(2)
    getatom A(3) c3             |       getatom_exo A(3)
    proceed                     |       proceed
@4: trust_me_else arity(3)      |   @4: trust_me_else_exo arity(3)
    getatom A(1) a4             |       getatom_exo A(1)
    getatom A(2) b4             |       getatom_exo A(2)
    getatom A(3) c4             |       getatom_exo A(3)
    proceed                     |       proceed
```

The code left above is very repetitive: we just scan each fact symbol by symbol. To show this idea clearly, we can re-arrange the code to separate walking through the arguments from the actual constants. The new code is shown in the right column. It relies on the following new WAM instructions:

- *set_exo_pointer* has one argument @t: it is a pointer to *a table* with the constants occurring in the facts. The table is nicely rectangular and compact. A WAM register *exo_pointer* is set to this pointer.
- *try_me_else_exo* acts as the *try_me_else* instruction in the WAM and also stores the current exo_pointer in its choice point.
- *retry_me_else_exo arity(N) @alt* fetches the exo_pointer from the choice point, adds `N` to it and stores that value in the choice point. The other WAM actions associated to *retry_me_else* are also performed.
- *trust_me_else_exo arity(N)* fetches the exo_pointer from the choice point and adds `N` to it. The other WAM actions associated to *trust_me_else* are also performed.
- *getatom_exo A(i)* fetches the i^{th} element from the current row in the exo-table (the exo_pointer points to that row now) and unifies it with Argument register i.

Note that the arity in the `(re)try/trust_me_else_exo` instruction is also the width of the exo table, so we could have denoted that operand as width(3). In fact, the width and the arity can be processed independently, and thus could be given separately.

At this point, have we gained anything? The amount of space needed for code+data has not decreased, and the instructions have a small extra overhead

in fetching the constants from the table and manipulating the exo_pointer. On the other hand, the code for p/3 is now generic, i.e. it suffices to make the exo_pointer point to a different table - say

```
u1 v1 w1
u2 v2 w2
u3 v3 w3
u4 v4 w4
```

to see that all the code except for setting the exo_pointer can be reused for executing a different set of facts.

Clearly every fact consists of the same code: three `getatom_exo` instructions and a `proceed`. We exploit that by generating the following (final) code:

```
    try_exo arity(3) @a @e @t ----------> a1 b1 c1
@a: keep_trying_exo arity(3)              a2 b2 c2
@e: getatom_exo A(1)                      a3 b3 c3
    getatom_exo A(2)                      a4 b4 c4
    getatom_exo A(3)                      NULL
    proceed
```

We have added a NULL entry to the table as sentinel, so that we can check whether we have reached the end of the table. (We could have used a count of rows or entries; as we will see, Mercury uses the latter.)

The new instructions act as follows:

- *try_exo N @a @e @t* sets exo_pointer to point to the table @t, creates a choice point, saves exo_pointer in it and sets the alternative field to @a, and then transfers control to @e.
- *keep_trying_exo N* fetches exo_pointer from the choice point, adds N to it, and stores the resulting value in the choice point; the alternative in the choice point is not updated. If (exo_pointer+N) points to the NULL sentinel, the choice point is discarded. Either way, this instruction restores the argument registers.

These new instructions give us a major benefit of *exo-compilation*: they allow us to eliminate all but one copy of the fact handling code, which can mean a potentially huge memory saving, while preserving the genericity of the code.

2.2 Void Variables

In the context of ILP - but also in general in the database context - one is often confronted with wide facts that are queried by goals containing lots of void variables, i.e. fields in which one is not interested during a particular query. E.g. for a fact p/12, the query could be *?- p(bruce,willis,_,_,_,_,_,_,_,Salary,_).*[1] Exo-compilation suggests dealing with void variables by generating a *specialised* predicate `p_1_2_11/3` whose code is:

[1] The point is that only the first, second and eleventh argument take part in the query, not that the first two arguments are instantiated or manifest.

```
    try_exo arity(3) @a @e @t(p/12) --------------> table for p/12
@a: keep_trying_select width(12) arity(3)
@e: getatom_exo A(1)
    getatom_exo A(2)
    getatom_exo_offset A(3), offset(11)
    proceed
```

The instruction *keep_trying_select* is a variant if *keep_trying_exo*, but where the width of the exo-table is different from the choice-point's arity.

getatom_exo_offset A(3), offset(11) unifies the third argument register with the atom to be found at offset 11 in the row currently pointed to by exo_pointer.

Replacing the original query by *?- p_1_2_11(bruce,willis,Salary)* eliminates all the unnecessary overhead of the void variables: initialisation, unification, trailing/untrailing, and storing in/restoring from the choice point.

In the context of ILP the above goal is typically generated dynamically and as part of a conjunction. In that case, before executing the conjunction, a void variable detection analysis can be performed and the appropriate transformation can be carried out. Since other analyses/transformations are already performed on such conjunctions (subsumption testing, once-transformation, ...) this seems reasonable - see for instance [9]. The code above must be generated at runtime. This is feasible as well, as other approaches have dealt with compiling (totally, partially, on the fly, and just in time) such code. See for instance [2].

2.3 Instruction merging and specialisation

The code for the facts can benefit from instruction merging: both in the original WAM and in the exo-compilation approach, we can easily collapse a sequence of get_atom instructions. We can even exploit the fact that the argument registers to be unified with table elements are consecutive and invent one new instruction like get5atoms_exo which needs no arguments at all, leading to a further reduction of the memory needed to represent the code and less argument fetching. Yap [10] performs such an instruction merging in ordinary WAM code: a sequence of up to 6 get_atom instructions from consecutive argument registers (and starting from 1) is compressed [8]. hProlog merges any sequence of up to 3 get_atom instructions, irrespective of the argument register they refer to. We have performed its analogue for the getatom_exo instruction.

Instruction specialisation is also applicable: hProlog and Yap (as many other systems) have specialised versions of the try/retry/trust_me_else instructions for several arities. hProlog does this specialisation up to arity 5, Yap up to arity 4. The same can be done for the analogous exo-instructions. We did not do this specialisation in our exo-compiler, because we are mainly interested in much higher arities.

3 Experiments in hProlog

Exo-compilation was implemented in hProlog as follows: for arities up to 15, there are predefined predicates (generated at startup) with code of the following form, which is for arity 5:

```
keep_trying_exo arity(5)          keep_trying_exo arity(5)
getatom_exo A(1)                  getatom3_exo A(1) A(2) A(3)
getatom_exo A(2)                  getatom2_exo A(4) A(5)
getatom_exo A(3)                  proceed
getatom_exo A(4)
getatom_exo A(5)
proceed
```

where the left half shows the code without instruction merging and the right with instruction merging. For larger arities, these would need to be generated on the fly.

This code acts as entry points for the code for an exo-compiled set of facts. This compilation is currently integrated in the compiler as follows: when the *prolog_flag* named *exo_compiling* is on, and the predicate to be compiled contains only facts with an atom for each argument, the predicate is exo-compiled. The compiler constructs the exo_table. An exo-predicate is compiled to one try_exo instruction which sets the exo_pointer and then transfers control to the appropriate pre-defined predicate.

Instruction merging of the exo-instructions was made into a command line hProlog option, so it is easy to run the benchmarks with and without instruction merging.

Generating code for the void specialisations is done by calling a new built-in predicate create_void_specialisation/3 which takes as arguments

- the exo-predicate - so that the exo-table can be retrieved
- the name/arity of the new predicate
- a description of which arguments of the original exo-predicate need to be selected

Because of the application we have in mind (dynamically generated queries in ILP), the user needs to call this built-in, but nothing prevents the compiler to do so as well. E.g., *?- create_void_specialisation(foo/5, gee/3, [2,4,5]).* generates the following code for gee/3:

```
@gee_3: try_exo arity(3) @a @e @t(foo/5)
    @a: keep_trying_select width(5) arity(3)
    @e: getatom3_exo_offset A(1) offset(2) A(2) offset(4) A(3) offset(5)
        proceed
```

when instruction merging is on.

We use $@t(foo/5)$ to denote the address of the exo-table for foo/5: it is known at load/link time.

The timings in Tables 1, 2 and 3 were obtained on a PC with a 1.8 GHz Pentium 4 CPU running Debian. Times are given in milliseconds. We used hProlog 2.7, Yap 5.1.1 and SICStus 3.12.0.

We start with an experiment in which a set of predicates p/n, n=1..15, each with 150 facts and with all atom arguments is called with free, unshared arguments. We do this in Yap and in hProlog, both in a version with and without instruction merging. The table also contains the timings for SICStus. In this way, one gets a better view on the performance. In this and following tables, we have added the ratio between subsequent columns between brackets. Table

Arity	Yap merging	Yap no merging	hProlog merging	hProlog no merging	SICStus merging
1	664	676 (0.98)	552	568 (0.97)	2720
3	948	2032 (0.46)	1464	1993 (0.73)	3480
5	1620	2905 (0.55)	1752	2596 (0.67)	5490
7	3040	3604 (0.84)	3053	3324 (0.91)	6560
9	4284	4812 (0.89)	3904	4373 (0.89)	8230
11	5085	5592 (0.90)	4613	4784 (0.96)	10130
13	5884	6248 (0.94)	5024	5648 (0.88)	12580
15	6304	6793 (0.92)	5761	6457 (0.89)	12860

Table 1. Performance on plain WAM code

1 shows that without instruction merging Yap and hProlog have close performance. With instruction merging, the figures show that up to arity 6, the Yap compression is superior, while the hProlog merging is better for larger arities. SICStus performs significantly worse on all arities. As far as we know, SICStus merges two consecutive get_atom instructions, as well as a (get_atom proceed) combination.

The general trend in Table 1 is clear: timings become larger with larger arities.

Arity	hProlog no merging	hProlog exo no merging	hProlog merging	hProlog exo merging
1	568	1412 (0.4)	552	1476 (0.37)
3	1993	2372 (0.84)	1464	2056 (0.71)
5	2596	2868 (0.9)	1752	2072 (0.84)
7	3324	3433 (0.96)	3053	3040 (1.00)
9	4373	4816 (0.9)	3904	3744 (1.04)
11	4784	4889 (0.97)	4613	4452 (1.03)
13	5648	5656 (0.99)	5024	4908 (1.02)
15	6457	6364 (1.01)	5761	5485 (1.05)

Table 2. hProlog in plain WAM mode and in exo mode

Table 2 shows that for hProlog exo-compilation starts paying off from arity 7 (actually 6 with the full data) with merging, but only from arity 13 without merging. There is indeed an overhead in the getatom_exo instruction, probably because of the lack of registers: there is no spare register for the pointer to the exo-table.

In Table 3, we show hProlog on the same set of benchmarks, but with a query with only three non-void arguments. The *exo void* columns take advantage of that in the way described in Section 2.2, while the *plain exo* columns follow the plain exo-compilation schema. The difference is clear: the left column is close to constant (as it should be), while the right column's runtime increases linearly with the arity. It is nice to see that the break even point is close to three. All the code was generated beforehand, i.e. not dynamically as would be needed in an ILP context where the queries are not known in advance.

Arity	hProlog exo void no merging	hProlog plain exo no merging	hProlog exo void merging	hProlog plain exo merging
3	2312	2328 (0.99)	1540	1564 (0.98)
5	2288	2964 (0.77)	1492	1820 (0.81)
7	2284	3672 (0.62)	1492	2805 (0.53)
9	2256	4284 (0.52)	1565	3760 (0.41)
11	2272	4725 (0.48)	1504	4605 (0.32)
13	2353	5528 (0.42)	1580	5188 (0.30)
15	2384	6156 (0.38)	1688	5724 (0.29)

Table 3. Optimising queries with void variables

4 Generalising exo-compilation

Exo-compilation as described above exploits a specific form of regularity of code. The generalisation to other atomic types besides atoms is straightforward. An example shows how facts with structured (ground) terms can be dealt with:

```
p(foo(a,b(c))).      p(gee(x,y(z))).
```

These facts have enough in common to treat them by exo-compilation, especially if there are many thousands of them. The generalisation of the getatom_exo instruction to functors is

```
get_struct_exo_offset A(i), offset(j)
```

with obvious meaning. Other instructions need an exo version as well.

The above p/1 would be translated to

```
    try_exo arity(1) @a @e @t ----------------> foo/2 a b/1 c
@a: keep_trying_select arity(1) width(4)        gee/2 x y/1 z
@e: get_struct_exo_offset A(1), offset(1)       NULL
    unify_atom_exo_offset offset(2)
    unify_struct_exo_offset offset(3)
    unify_atom_exo_offset offset(4)
    proceed
```

The meaning of the instructions should be clear, and dealing with void variables in a call to such non-flat facts is clearly feasible. The next steps in generalising exo-compilations are: allow variables, allow lists (of different length), allow

clauses with similar bodies, and allow arbitrary ground terms; the first few of those are described in more detail in [4]. The same reference also contains some preliminary ideas about combining indexing with exo-compilation in the context of the WAM.

5 How Mercury handles large predicates

5.1 Large predicates without indexing

In Mercury, every predicate (or function) has one or more modes. Each mode specifies, for each argument, whether that argument is input or output in that mode, and also specifies a *determinism*, which specifies upper and lower bounds on the number of solutions expected in that mode. For example, *append(in,in,out)* has determinism *det*, meaning calls to *append* in that mode will have exactly one solution, while *append(out,out,in)* has determinism *multi*, meaning calls to *append* in that mode will have one or more solutions.

The Mercury compiler generates a separate piece of code for each mode: each mode of a predicate is called a *procedure*. The easiest modes to generate good code for when the predicate body is defined by a large set of facts are the modes in which all arguments are output (and whose determinism is therefore *multi*). The C code generated by the compiler for this mode of the predicate p/3 with four facts from Section 2.1, will look something like this, after discarding some irrelevant details:

```
p_3_0:
    mkframe(1, local_label_2);
    temp1 = &common_table_0[0];
    framevar1 = 3;
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed();

local_label_2:
    r4 = framevar1;
    if (r4 >= 9) goto local_label_3;
    framevar1 = framevar1 + 3;
    temp1 = &common_table_0[r4];
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed();

local_label_3:
    temp1 = &common_table_0[r4];
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed_discard();
```

The first block of the code handles the first solution, the last block of code handles the last solution, and the middle block handles all the others in between, rather like a try/retry/trust chain in the WAM in which all retry's are collapsed into one block of code.

The *mkframe* macro allocates a frame on the nondet stack; this frame functions as a combination choice point and environment. The environment part holds one slot (containing the equivalent of the WAM's exo-pointer) and the backtrack point is *local_label_2*. The common_table_0 is Mercury's internal name for what was named exo table in Section 2.1, and it stores the data which in this case is the 12 atoms. The Mercury calling convention for *multi* procedures requires argument n to be returned in register r_n, so the assignments to r_1, r_2 and r_3 pick up the value of each argument of the first solution from a table containing all the solutions. The *succeed* macro then returns to the return address recorded in the stack frame by the *mkframe* macro.

After backtracking causes execution to reach *local_label_2*, the code checks whether the next solution is the last one. If not, the stack frame is updated to show that this solution was returned but otherwise it is left intact. If the last solution is reached, execution branches to code that uses the *succeed_discard* variant of the *succeed* macro: it discards the stack frame after picking up the return address from it.

Unlike most Prolog implementations, the Mercury compiler does not convert procedure bodies to disjunctive normal form. The goal form representing a table facts (a disjunction in which each disjunct is a conjunction of unifications that create static terms) may therefore appear inside other goals, e.g. an if-then-else, either in the program as written or after inlining. In such cases, the generated code is very similar code to the code above, the main difference being that it won't have to create a stack frame (the surrounding code having already created one), and the mechanism used to record the next alternative will be slightly different [3]. The reasons why the names of the tables don't refer to procedure names is that a single procedure may refer to more than one of these tables, while a table may be referred to from more than one procedure if the predicate whose data is contained in the table is inlined at more than once call site. This works because Mercury does not support dynamic predicates.

In Mercury, the exo tables can contain any type of ground term. For ground terms, there is never any need to copy them: the exo table contains the term exactly as it would be laid out in the heap, so all what needs to be returned as a result is a pointer to the term. The same technique can be used in the WAM, but would require minor changes in the garbage collector (which Mercury's use of a conservative collector renders moot). In fact, ECLiPSe has implemented static terms at least since 1990.

5.2 Large predicates with indexing

Mercury uses indexing quite aggressively, i.e., not just on arguments in the head of a predicate, but also in explicit disjunctions. In the past, the implementation of indexing was based on using a table (e.g. a hash table) to map the value of the switched-on variable to a code address, with the code address giving the start of the code for handling a particular switch arm. Remember that Mercury generates C code. This indexing schema works fine for small and even medium-sized code. But when the size of the predicate increases, and as a consequence the size of the

generated C code also increases, this becomes problematic: it exposes quadratic behaviour in the C optimiser, as well as limits in the Mercury compiler's own low level optimiser. Exo-compilation avoids the problem of generated code size explosion: only the size of the generated C data arrays increases.

Indexing kicks in when enough input is available to reduce the alternatives that can return answers. In the context of exo-compilation, and for simplicity of explanation, we focus on a predicate consisting of facts, and for which there is one input argument that is an integer. We assume also that the range of this integer is dense in some range starting from 0, so that the input can be used as in index in an array without further manipulation.[2] We work by example, and distinguish three cases, namely that the procedure is (1) det, (2) semidet and (3) nondet or multi.

A det procedure: This corresponds to a predicate like

```
:- pred p(int:in, term1:out, term2:out).
p(0,a0,b0).
p(1,a1,b1).
p(2,a2,b2).
p(3,a3,b3).
p(4,a4,b4).
```

It is clear that a two dimensional table of the form

```
a0 b0
a1 b1
a2 b2
a3 b3
a4 b4
```

can be addressed directly with the input argument to retrieve the necessary output. The generated code's size is independent of the number of facts: this will be true for the other cases as well.

A semidet procedure: This occurs when there is no solution for one or more input values, as in

```
:- pred q(int:in, term1:out, term2:out).
q(0,a0,b0).
q(2,a2,b2).
q(3,a3,b3).
q(4,a4,b4).
```

The compiler generates a bit vector, which at place i indicates whether the corresponding input argument corresponds to a solution. A successful test causes a jump to the same code as in the det case; an unsuccessful test causes failure.[3]

[2] Mercury uses appropriate generalisations of these conditions.

[3] Mercury has long generated lookup tables for predicates like p and q, but it has traditionally generated a separate vector for each output argument. The generation of a two-dimentional table like the one shown above is new, as is every other exocompilation technique we present in the paper.

A nondet or multi procedure: This occurs for instance in the following predicate:

```
:- pred r(int:in, term1:out, term2:out).
r(0,a0,b0).
r(0,c0,d0).
r(0,e0,f0).
r(0,g0,h0).
r(2,a2,b2).
r(3,a3,b3).
r(4,a4,b4).
r(4,c4,d4).
r(4,e4,e4).
```

The compiler generates two tables, the first one of which is addressed by using the input argument as an index. The table contains $n + 2$ columns if there are n output arguments - in our example $n = 2$ and the table is shown in Figure 1. The first entry in each column contains an indication of whether the input value

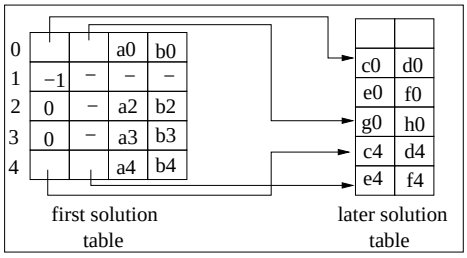


Fig. 1. First and later solution table for a nondet procedure

corresponds to a switch arm with no solution (-1), exactly one solution (0) or more than one solution (any value above 0). Code tests this value and branches to the appropriate continuation, which consists in failure in the first case.

If the selected switch arm has one solution, the indexing code will pull the values of the output variables out of the last n columns of the selected row, put them where the code after the switch (which may be the procedure epilogue) expects them, and then jump to that code.

If the selected switch arm has more than solution, the indexing code will also pull the values of the output variables out of the last n columns and put them where they are expected, but before jumping to the code after the switch, it will set up the return of the later solutions on backtracking.

Part of this involves saving the values of the first and second columns in the current stack frame (as *framevar1* and *framevar2* respectively in the example code below). Each switch arm that has more than one solution stores all those solutions except the first in a contiguous region of the second table. The first column of the first-solution table row points to (contains the index of) the start of the first of these solutions in the later-solution table, while the second column points to the start of the last of these solutions.

The other part is directing execution to code that uses these saved values. How this is done depends on the context. The required code may be as expensive as pushing a temporary frame (effectively a mini choicepoint) on the nondet stack, as cheap as simply updating the backtrack code pointer of the current nondet stack frame, or something in between (see [3] for more info, through some details have changed since then).

If the switch is the entirety of a procedure body, the code executed on backtracking will look like the following, which is a generalised version of the code at local labels 2 and 3 above (*common_table_1* refers to the later solutions table).

```
local_label_5:
    r4 = framevar1;
    if (r4 >= framevar2) goto local_label_6;
    framevar1 = framevar1 + 3;
    temp1 = &common_table_1[r4];
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed();
```

```
local_label_6:
    temp1 = &common_table_1[r4];
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed_discard();
```

5.3 Semantic analysis of large predicates

Code generation, is not the only area in which large predicates pose challenges. Many semantic analysis algorithms (which Prolog does not need but Mercury does) have behavior that is quadratic (or worse) in the number of clauses, in the sizes of terms, or both. The Mercury compiler certainly contained many such algorithms. We had to find and try to fix them one by one (unfortunately, some fixes require more work than we have funding for). The most ubiquitous problem was algorithms that added something to the end of a list after each disjunct; doing $O(n)$ work at disjunct n yields a quadratic algorithm. We changed these to instead return a list of those somethings (which differ from analysis to analysis), and then used code patterned after balanced mergesort to generate the final output.

6 Discussion, related work, and conclusion

Large datasets occur in a wide range of applications. Our experience includes

- knowledge bases processed by an inductive logic programming system;
- tables generated by scanner and parser generators; and
- large databases of structured patterns to look for in an input stream.

There are doubtless many more. While each one of these may be only a niche, put together they are significant enough to be worthy of attention.

Most current logic programming systems do not have any support for large datasets. The reason for this is mostly historical: such systems have traditionally been called deductive databases. As a result, the handling of large datasets by most logic programming systems has left a lot to be desired. Each of us bumped into these problems in our daily work, and decided to fix the situation.

As it happened, though we started working on different systems (Yap, hProlog and Mercury) and got our driving motivations from different application areas, we ended up with techniques that are quite similar. [4] Thought both Prolog and Mercury compilers traditionally generate code for each part of the program, two of us (Santos Costa and Somogyi) independently decided that the efficient implementation of large tables of facts requires breaking this rule, and generating generic table lookup code instead, with all the information specific to the program confined to the tables. Although this is a fairly standard programming technique used in databases, parsers and in many other programs, it *is* unusual to find it in code generated automatically by a compiler.

The most obvious benefit of this approach, which we call exo-compilation, is of course the large reduction in the amount of code required. However, it also yields speedups, which is a more important benefit, though it is also much less obvious. For some people, it may even be counter-intuitive, because exo-compilation actually *increases* the number of memory accesses, due to accesses to data in tables requiring an extra level of indirection. Though this extra indirection reduces locality a bit, eliminating redundant copies of WAM instructions and putting the arguments of each fact closer together improves locality by far more, and it is this latter effect that dominates in our applications. Our experiments show performance improvements, and detailed cache simulations (performed with the cachegrind option of `valgrind-3.2.0-Debian`) have revealed that D1 misses and L2 references can drop by a factor of 5, even though overall I and D references increase by about 6%: these figures were obtained for running the totality of the benchmarks in Tables 1 and 2.

Exo-compilation also brings other benefits. In the ILP context often each example is represented by some predicates each of which consists of a few small facts only. Treating an individual example by exo-compilation might seem senseless. However, there are often too many examples to keep them all in memory and ILP tools need to switch between examples frequently. For that reason, hipP (an ILP dedicated cousin of hProlog) has an intricate module for switching between examples, where each example is pre-compiled fully to WAM code. Applying exo-compilation to the examples as a whole, would result in generic code that can be used for each example and switching between examples would consist of switching between exo-tables. Since these are smaller than fully compiled code, it allows more examples simultaneously in memory, it reduces the amount of memory traffic when an example needs to be (re)loaded, and as a result it increases performance and the size of the datasets that can be handled by the system. These considerations were in fact a major motivation for exploring exo-compilation in the context of Prolog.

[4] Our different choices were partly dictated by differences in the languages. partly motivated by differences in the existing technology bases: Prolog supports dynamic predicates and dynamic loading of predicates while Mercury doesn't, and likewise for variables in facts; the Mercury abstract machine and the WAM are quite dissimilar, and the WAM is usually interpreted while Mercury compiles to C.

Exo-compilation also serves as the basis for further optimisations. For example, dynamic indexing fits in nicely with exo-compilation, and it would be more difficult to implement without separating the operands from the code. Or consider the optimisation we presented in Section 2.2 which was intended to remove the overhead of void variables in queries. This overhead can be reduced by the Vienna Abstract Machine (see [7]) or by a tagging schema that caters for void variables (as in Beer [1] which caters for uninitialised variables), but these techniques still perform actions that are linear in the number of void variables, while our technique does not.

We have just started using exo-compilation in live applications such as ILP. The future will show how well the idea actually works in practice, but the initial indications are quite promising.

Acknowledgements

We thank Research Foundation-Flanders (FWO-Vlaanderen) for supporting Bart Demoen.

References

1. J. Beer. The occur-check problem revisited. *J. Log. Program.*, 5(3):243–261, 1988.
2. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.
3. T. Conway, F. Henderson, and Z. Somogyi. Code generation for mercury. In *ILPS*, pages 242–256, 1995.
4. V. Costa, B. Demoen, and P.-L. Nguyen. Big facts, void variables, the WAM and exo-compilation. Report CW 486, Department of Computer Science, K.U.Leuven, Leuven, Belgium, April 2007. URL: http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW486.abs.html.
5. L. Damas, V. Santos Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1989.
6. B. Demoen. hProlog. http://www.cs.kuleuven.be/ bmd/hProlog/.
7. A. Krall. The Vienna Abstract Machine. *Journal of Logic Programming*, 29(1-3):85–106, 1996.
8. H. Nässén, M. Carlsson, and K. Sagonas. Instruction merging and specialization in the SICStus Prolog virtual machine. In *Principles and Practice of Declarative Programming (PPDP01)*, 2001.
9. V. Santos Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4(Aug):465–491, August 2003.
10. V. Santos Costa. *Prolog Performance on Larger Datasets*, In *PADL07*, 2007.

Some Improvements over the Continuation Call Tabling Implementation Technique

Pablo Chico de Guzmán[1] Manuel Carro[1] Manuel V. Hermenegildo[1,2]
Cláudio Silva[3] Ricardo Rocha[3]

pchico@clip.dia.fi.upm.es
{mcarro, herme}@fi.upm.es
herme@cs.unm.edu
ccaldas@dcc.online.pt
ricroc@dcc.fc.up.pt

[1] School of Computer Science, Univ. Politécnica de Madrid, Spain
[2] Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, USA
[3] DCC-FC & LIACC, University of Porto, Portugal,

Abstract. Tabled evaluation has been proved an effective method to improve several aspects of goal-oriented query evaluation, including termination and complexity. Several "native" implementations of tabled evaluation have been developed which offer good performance, but many of them need significant changes to the underlying Prolog implementation. More portable approaches, generally using program transformation, have been proposed but they often result in lower efficiency. We explore some techniques aimed at combining the best of these worlds, i.e., developing a portable and extensible implementation, with minimal modifications at the abstract machine level, and with reasonably good performance. Our preliminary results indicate promising results.

1 Introduction

Tabling [16, 2, 15] is a resolution strategy which tries to *memoize* previous calls and their answers in order to improve several well-known shortcomings found in SLD resolution. It brings some of the advantages of bottom-up evaluation to the top-down, goal-oriented evaluation strategy. In particular, evaluating logic programs under a tabling scheme may achieve termination in cases where SLD resolution does not (because of infinite loops —for example, the tabled evaluation of bounded term-size programs is guaranteed to always terminate). Also, programs which perform repeated computations can be greatly sped up. Program declarativeness is also improved since the order of clauses and goals within a clause is less relevant, if at all. Tabled evaluation has been successfully applied in many fields, such as deductive databases [11], program analysis [17, 3], reasoning in the semantic Web [19], model checking [9], and others.

In all cases the advantages of tabled evaluation stem from checking whether calls to *tabled predicates*, i.e., predicates which have been marked to be evaluated using tabling, have been made before. Repeated calls to tabled predicates consume answers from a table, they suspend when all stored answers have been consumed, and they fail when no more answers can be generated. However, the advantages are not without drawbacks. The main problem is the complexity of some (efficient) implementations of tabled resolution, and a secondary issue is the difficulty in selecting which predicates to table in order not to incur in undesired slow-downs.

Two main categories of tabling mechanisms can be distinguished: *suspension-based* and *linear* tabling mechanisms. In suspension-based mechanisms the computation state of suspended tabled subgoals has to be preserved to avoid backtracking over them. This is done either by *freezing* the stacks, as in XSB [13], by copying to another area, as in CAT [5], or by using an intermediate solution as in CHAT [6]. Linear tabling mechanisms maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations. The computation of the (local) fixpoint is performed by repeatedly looping subgoals until no more solutions can be found. Examples of this method are the linear tabling of BProlog [18] and the DRA scheme [7].

Suspension-based mechanism have achieved very good performance results but, in general, deep changes to the underlying Prolog implementation are required. Linear mechanisms, on the other hand, can usually be implemented on top of existing sequential engines without major modifications. One of our theses is that it should be possible to find a combination of the best of both worlds: a suspension-based mechanism that is efficient and does not require complex modifications to the underlying Prolog implementation, thus contributing to maintainability. Also, we would like to avoid introducing any overhead that would reduce the execution speed for SLD execution.

Our starting point is the *Continuation Call Mechanism* presented by Ramesh and Chen in [12]. This approach has the advantage that it indeed does not need deep changes to the underlying Prolog machinery. On the other hand it has shown up to now worse efficiency than the more "native" suspension-based implementations. Our aim is to analyze the bottlenecks of this approach, explore variations of it, and propose solutions in order to improve its efficiency without losing much in implementation simplicity and portability.

2 Tabling Basics

We will now sketch how tabled evaluation works from a user point of view (more details can be found in [2, 13]) and then we briefly describe the continuation call mechanism implementation technique proposed in [12] on which we base our work.

2.1 Tabling by Example

Let us use as running example the program in Figure 1, taken from [12], whose purpose is to determine reachability of nodes in a graph We ignore for now the `:- tabled path/2` declaration (which instructs the compiler to use tabled execution for the designated predicate), and assume that SLD resolution is to be used. Then, a query such as `?- path(a, N).` will never terminate since there is a left-recursive clause which generates a goal with the same instantiation as the initial call.

Adding the `:- tabled` declaration forces the compiler and runtime system to distinguish the first occurrence of a tabled goal (the *generator*) and subsequent calls which are identical up to variable renaming (the *consumers*). The generator applies resolution using the program clauses to derive answers for the goal. Consumers *suspend* the current execution path (using implementation-dependent means) and move to a different

```
edge(a, b).
edge(b, c).
edge(b, d).

:− tabled path/2.

path(X, Y):−
    path(X, Z),
    edge(Z, Y).
path(X, Y):−
    edge(X, Y).
```

Fig. 1. A simple tabled program.

```
path(X, Y):− slg(path(X, Y)).

slg_path(path(X, Y), Id):−
    slgcall(Id, path(X, Z), path_cont).
slg_path(path(X, Y), Id):−
    edge(X, Y),
    answer(Id, path(X, Y)).

path_cont(Id, path(X, Z)):−
    edge(Z, Y),
    answer(Id, path(X, Y)).
```

Fig. 2. Program in Figure 1 transformed for tabled execution.

branch. When such an alternative branch finally succeeds, the answer generated for the initial query is inserted in a table associated with the original goal. This makes it possible to reactivate suspended calls and to continue execution at the point where it was stopped. Thus, consumers do not use SLD resolution, but obtain instead the answers from the table where they have been previously inserted by the producer. Predicates not marked as tabled are executed following SLD resolution, hopefully with (minimal or no) overhead due to the availability of tabling in the system.

2.2 The Continuation Call Technique

The continuation call technique [12] implements tabling by a combination of program transformation and side effects in the form of insertions to and reads from an internally-maintained table which relates calls, answers, and the continuation code to be executed after consumers read answers from the table. We will now sketch how the mechanism works using the `path/2` example shown in Figure 1. The original code is transformed into the program in Figure 2 which is the code actually executed.

Roughly speaking, the transformation for tabling is as follows: a bridge predicate for `path/2` is introduced so that calls to `path/2` made from regular Prolog execution do not need to be aware of `path/2` being tabled. The call to the `slg/1` primitive will ensure that its argument is executed to completion and will return, on backtracking, all the solutions found for the tabled predicate. `slg/1` also inserts the call in the answer table and generates an identifier for it. Control is then passed to a new distinct predicate (in this case, `slg_path/2`) by constructing a goal from `path(X, Y)` (which is passed as an argument to `slg/1`) and then calling this term, suitably instantiated, from inside the implementation of `slg/1`.[4] The first argument contains the variables in the original call to `path/2` and the second one is the identifier generated for the parent call, which is used to relate operations on the table with this initial call. Each clause of `slg_path/2` is derived from a clause of the original `path/2` predicate by:

[4] The new term has been created in the example simply by prepending the prefix `slg_` to the argument passed to `slg/1`. Any means of constructing a new unique predicate symbol based on the original one is acceptable. Our implementation performs at compile time as much of this work as possible.

```
:− tabled path/2.

path(X, Z):−
    edge(X, Y),
    path(Y, Z).

path(X, Z):−
    edge(X, Z).
```

Fig. 3. A program which needs to keep an environment.

```
path(X, Y):− slg(path(X, Y)).

slg_path(path(X, Y), Id):−
    edge(X, Y),
    slgcall(Id , [X], path(Y, Z), path_cont_1).
slg_path(path(X, Y), Id):−
    edge(X, Y),
    answer(Id, path(X, Y)).

path_cont_1(Id , [X], path(Y, Z)):−
    answer(Id, path(X, Z)).
```

Fig. 4. The program in Figure 3 after being transformed for tabled execution.

- Adding an `answer/2` primitive at the end of each clause resulting from a transformation and which is not a *bridge* to call a continuation predicate. `answer/2` is responsible for checking for redundant answers and executing whatever continuations (see the following item) there may be associated with that call identified by its first argument.
- Instrumenting recursive calls to `path/2` using the `slgcall/3` primitive. If the term passed as an argument (i.e., `path(X, Y)`) has already been inserted in the table, `slgcall/3` creates a new consumer which reads answers from the table. Otherwise, the term is inserted in the table with a new call identifier and execution follows using the `slg_path/2` program clauses to derive new answers. In the first case, `path_cont/2` is recorded as (one of) the continuation(s) of `path(X, Y)` and `slgcall/3` fails. In the second case `path_cont/2` is only recorded as a continuation of `path(X, Y)` if the tabled call cannot be completed. The `path_cont/2` continuation will be called from `answer/2` after inserting a new answer or erased upon completion of the `path(X, Y)` subgoal.
- The body of `path_cont/2` encodes what remains of the clause body of `path/2` after the recursive call. It is constructed in a similar way to `slg_path/2`, i.e., applying the same transformation as for the initial clauses and calling `slgcall/3` and `answer/2` at appropriate times.

This strategy tries to complete subgoals as soon as possible, failing whenever new answers are found, and thus implements the so-called *local scheduling* [13]. This implementation uses the same completion detection algorithm as the SLG-WAM.

Figures 3 and 4 illustrate how additional modifications are required in the translation for some programs in order to pass on additional variables to continuations. Note that in the program in Figure 3 an answer to `?- path(X, Y)` may need to provide a value to variable `X` which does not appear in the recursive call to `path/2`. If the simple translation of Figure 2 is performed, this variable will not be available at the point in the code where the answer is inserted in the table. The solution adopted in this case is to explicitly carry a set of variables when preparing the call to the continuation. This set is also inserted in the table, and is passed to the continuation call when resumed. The translation is shown in Figure 4. Note that the call to `slgcall/4`

```
answer(callid Id, term Answer) {
  insert Answer in answer table
  If (Answer ∉ answer table)
    for each continuation call C
          of tabled call Id {
        call (C) consuming Answer;
    }
  return FALSE;
}
```

Fig. 5. Pseudo-code for `answer/2`.

```
slgcall ( callid Parent, term Bindings,
          term Call , term CCall) {
   Id = insert Call into answer table;
   if (Id.state == READY) {
      Id.state = EVALUATING;
      call the transformed clause of Call;
      check for completion;
   }
   consume answers for Id;
   if (Id.state != COMPLETE) {
        Id depends on Parent;
        add a new continuation
            call (CCall, Bindings) to Id;
   }
   return FALSE;
}
```

Fig. 6. Pseudo-code for `slgcall/4`.

in `path_cont_1` includes a list containing variable `X`. This list is, on resumption, received by `path_cont_1` and used to construct and insert in the table an answer which includes `X`. A safe approximation of the variables which should appear in this list is the set of variables which appear in the clause before the tabled goal and which are used in the continuation, including the `answer/2` primitive if there is one in the continuation —this is the case in our example. Variables appearing in the tabled call itself do not need to be included, as they will be passed along anyway.

The list of bindings is a means to recover the environment existing when a call is suspended. Other approaches recover this environment using, e.g., lower-level mechanisms, such as the forward trail of SLG-WAM plus freeze registers [13]. The continuation call approach, has, however, the nice property that several of the operations are made at the Prolog level through program transformation, which simplifies the implementation (and helps portability). On the other hand, the primitives which insert answers in the table and retrieve them are usually, and for efficiency reasons, written using some lower-level language and accessed using a suitable interface.

The pseudo-code for `answer/2` and `slgcall/4` is shown in Figures 5 and 6, respectively. The pseudo-code for `slg/1` is similar to that of `slgcall/4` but, instead of consuming answers, they are returned by backtracking and it finally fails when all the stored answers have been exhausted.

2.3 Issues in the Continuation Call Mechanism

We have identified two performance-related issues when implementing the technique sketched in the previous section. The first one is rather general and related to the heavy use of the interface from C to Prolog (and back) that the implementation needs to make, and which adds an overhead which cannot be neglected.

The second one is the repeated copying of continuation calls. Continuation calls (Prolog predicates with an arbitrarily long list of variables as an argument) are com-

pletely copied from Prolog memory to the table for every consumer found. Storing a pointer to these structures in memory is not enough, since `slg/1` and `slgcall/3` fail immediately after associating a continuation call with a table call in order to force the program to search for more solutions and complete the tabled call. Therefore, the data structures created during forward execution may be removed on backtracking and not be available when needed. When continuations are resumed by `answer/2`, it is necessary to reconstruct them as Prolog terms from the data stored in the table to be able to call them as a goal. This can also clearly have a negative impact on performance.

Finally, an issue found with the baseline implementation that we used as a starting point [14], is that it did not allow backtracking over Prolog predicates called from C and this compromised extensibility. In particular, this makes it difficult to implement other scheduling strategies. Since this shortcoming may appear also in other C interfaces, it is a clear candidate for improvement.

3 An Improvement over the Continuation Call Technique

We now propose some improvements to the different limitations of the original design and implementation that we discussed in Section 2.3.

3.1 Using a Lower-Level Interface

Calls from C to Prolog were initially performed using a relatively high-level interface similar to those commonly found in current state of the art logic programming systems: operations to create and traverse Prolog terms appear to the programmer as regular C functions, and details of the internal data representation are hidden to the programmer. This interface imposed a noticeable overhead in our implementation, as the calls to C functions had to allocate environments, pass arguments, set up Prolog environments to call Prolog from C, etc.

Since the low-level code which constructs Prolog terms and performs calls from C is the same regardless of the program being executed, we decided to skip the programmer interface and call directly macros available in the engine implementation. Given that the complexity of the C code involved is certainly manageable, that was a not a difficult task to do and it sped the execution up by a factor of 2.5 on average.

3.2 Calling Prolog from C

A relevant issue when using a C-to-Prolog interface is the need to call Prolog goals from C efficiently. This is needed both by `slgcall/3` and `answer/2` in order to invoke continuations of tabled predicates. As mentioned before, we want to design a solution which relies as little as possible on non-widely available characteristics of C-to-Prolog interfaces (to simplify portability), but which keeps the efficiency as high as possible.

The solution we have adopted is to move calls to continuations from the C level to the Prolog level. Continuations are stored in a (Prolog) list which is pointed to from the corresponding table entry, and they are returned one at a time on backtracking using an extra argument of `slgcall/3` and `answer/2`. These continuations are then called

```
path(X,Y) :-
    slgcall(path(X, Y), Sid,
            true, Pred),
    (
       nonvar(Pred) ->
          (call(Pred) ;
           test_complete(Sid))
    ;
       true
    ),
    consume_answer(path(X, Y), Sid).

slg_path(path(X, Y), Sid) :-
     edge(X, Y),
     answer(path(X, Y), Sid, CCall, 0),
     call(CCall).
```

```
slg_path(path(X, Y),Sid) :-
    edge(X, Z),
     slgcall(path(Z, Y), NewSid,
             path_cont_1, Pred),
    (
       nonvar(Pred) ->
          (call(Pred);
           test_complete(NewSid))
    ;
       true
    ),
    read_answers(Sid, NewSid, [X], CCall, 0),
    call(CCall).

path_cont_1(path(X, Y), Sid, [Z]) :-
    answer(path(Z, Y), Sid, CCall, 0),
    call(CCall).
```

Fig. 7. New program transformation for right-recursive definition of `path/2`.

from Prolog.[5] Failure happens when there is no pending continuation call. New continuations found during program execution can be destructively inserted at the end of the list of continuations transparently to Prolog.

In Figure 7 (which shows the translation we propose now for the code in Figure 3), `answer/4`, `read_answers/5`, and `slgcall/4` return in variables `Pred` and `CCall` the continuations of a tabled call that are to be called as Prolog goals. This avoids using up C stack space due to repeated Prolog $\rightarrow$ C $\rightarrow$ Prolog $\rightarrow$ ... calls, which may exhaust the available space. Additionally, the C code is somewhat simplified (e.g., there is no need to set up a Prolog environment to be used from C) which makes using the lower-level, faster interface less of a burden. The last unused argument of `answer/4` (and `read_answers/5`) implements a trick to make the corresponding choicepoint have an extra, unused slot (corresponding to a WAM argument), which will be used to hold a pointer to the rest of the list of continuations. Having such a slot avoids changing the structure of choicepoints and how they are managed. This pointer is destructively updated every time a continuation call is handed to the Prolog level.

We would like to clarify how some of the primitives used in Figure 7 work for this case. Note that the functionality of `slgcall/3` (`slg/1` when called from SLD-type execution) has been split across `slgcall/3`, `test_completion/1` and `read_answer/5` (`consume_answers/2` when associated with `slg/1`) in order to be able to perform calls to continuations from Prolog. `slgcall/5`, as in the original definition, checks if a call to a tabled goal is a new one. If so, `Pred` is unified with a goal whose main functor is `slg_path/2` and whose arguments are appropriately instantiated. A free variable is returned otherwise. `test_complete/1` always succeeds but performing a side effect: it tests if the tabled goal identified by `Sid` can be

[5] In our implementation this exploits being able to write non-deterministic predicates in C. If this feature is not available in a given system, a list of continuations can always be returned instead which is then traversed on backtracking using `member/2`.

marked as complete, and marks it in that case. `read_answers/5` consumes actual answers for the call identified by `NewSid` and then associates a new continuation call with `NewSid` if the tabled call is not completed. Its first argument, `Sid`, is needed to mark dependencies between tabled calls. `consume_answer/2` returns the answers stored in the table one at a time and on backtracking if the tabled call is completed. Otherwise, it behaves internally as `read_answers/5`.

3.3 Freezing Continuation Calls

In this section we will sketch some proposals to reduce the overhead associated with the way continuation calls are handled in the original approach.

Overhead related to resuming consumers: The original continuation call technique saved a binding list to reinstall the environment of consumers instead of copying or freezing the stacks and using a forward trail, as CAT, CHAT, or SLG-WAM. This is a relatively non-intrusive technique, but it requires copying terms back and forth between Prolog and the table where calls are stored. Restarting a consumer needs to construct a term whose first argument is the new answer (which is stored in the heap), the second one the goal identifier (an atomic item), and the third one a list of bindings (which may be arbitrarily large). If the list of bindings has N elements, constructing the continuation call requires creating $\approx 2N + 4$ heap cells. If a continuation call is resumed often and N is high, the efficiency of the system can degrade quickly.

The technique we propose constructs all the continuation calls in the heap as a regular Prolog term. This makes calling the continuation a constant time operation, since `answer/4` only has to unify its third argument with the continuation call. Since that argument is a variable at run time, full unification is not needed. However, the fragment of code which constructs this call performs backtracking as it fails after every success of `answer/4`. This would remove the constructed call from the heap, thereby forcing us to construct it again. Protecting that term would make it possible to construct it only once. The solution we propose can be seen as a variant of the approach taken by CHAT, but without having to introduce new abstract machine instructions.

In the explanation of our proposed *freezing* technique we will use the following notation. `H` denotes the pointer to the top of the heap. `B` is the pointer to the most recent choicepoint. To distinguish different kinds of choicepoints (borrowing from [6]) we will use $\mathtt{B_T}$, where `T` can be `G`, `C` or `P`, which stand for generator, consumer, or Prolog, respectively. The pointer to the heap stored in a choicepoint will be denoted as $\mathtt{B_T[H]}$.

In CHAT the heap pointer is not reset on backtracking (as the WAM does with the assignment `H := B`$_\mathtt{P}$`[H]`) by manipulating the heap pointer field $\mathtt{B_P[H]}$ of the Prolog choicepoints between the (newly created) consumer choicepoint and the choicepoint corresponding to its generator so that they all point to the current top of the heap `H`: $\mathtt{B_P[H]} := \mathtt{B_C[H]}$. Therefore, forward execution will continue building terms on the heap on top of the previous solutions.

This solution can generate garbage in the heap, which is not a serious problem as garbage collection can eventually free it. A more critical problem is the need to traverse an arbitrarily long series of choicepoints, which could make the system efficiency decrease. A solution for this problem has been proposed [4], which for us has

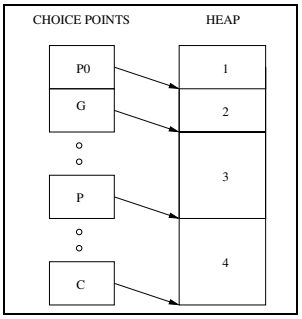


Fig. 8. Initial state.

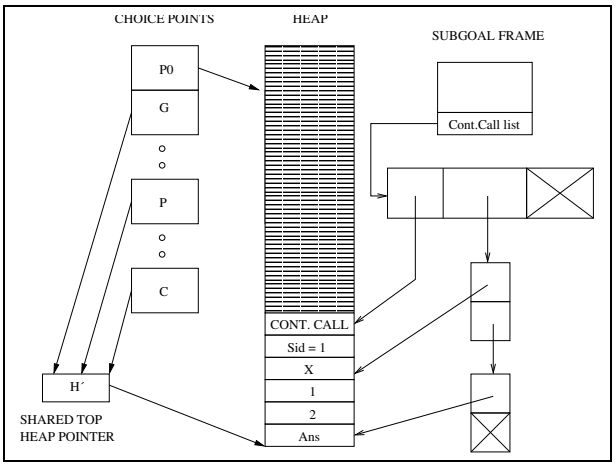


Fig. 9. Frozen continuation call.

the drawback of needing new WAM-level instructions and adding a new field to some choicepoints. As an alternative solution, we update the `B[H]` fields of the choicepoints between a new consumer and its generator so that they point to a pointer `H'` which in turn points to the heap top. Whenever we need to change again the `B[H]` field for these choicepoints, we simply update `H'` plus the choicepoints pushed since the last adjustments. Determining whether `B[H]` points to the heap or to `H'` is easy and simply entails deciding whether it falls within the heap limits. This needs changing the WAM instructions used for backtracking in a very localized way which, in our experience, has an unmeasurable impact over SLD execution performance.

Figure 8 shows the state of the choicepoint stack and heap before freezing a continuation call. On the left of Figure 9 all `B[H]` fields of the choicepoints `G`, `P`, and `C` have changed to a common pointer `H'` to the heap top. Thus, the continuation call `(C, [X,1,2], Ans)` is frozen.

Trail management to recover a continuation call state: The same term T corresponding to a continuation call C can be used several times to generate multiple answers to a query. This is in general not a problem as answers are in any case saved in a safe place (e.g., the answer table), and backtracking would undo the bindings to the free variables in T. There is, however, a particular case which needs special measures. When a continuation call C_1, identical to C, is resumed within the scope of C, and it is going to read a new answer, the state of T has to be reset to its frozen initial state. The variables which may have been bound by C (Figure 10) are reset to unbound by using a list of free variables collected when this term was copied to the heap (Figure 9, at the right). Since C_1 is using the same term T as C, we say that C_1 is a *reusing* call. This approach to call reuse avoids repeatedly copying several times the same continuation call to the heap.

When C_1 finishes and execution has to continue with C, the state of T has to be restored to the one existing just before starting C_1, i.e., that in Figure 10, where some initially free variables were bound. This is done by constructing a *value trail* (Figure 11)

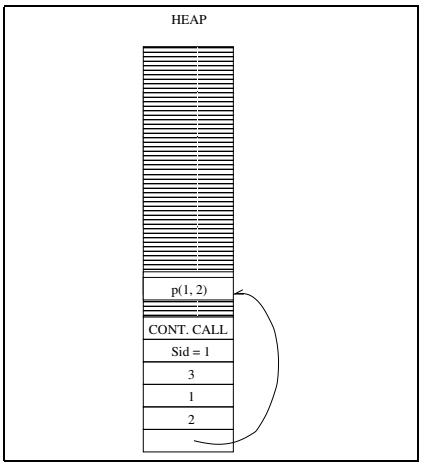


Fig. 10. Before reusing a cont. call.

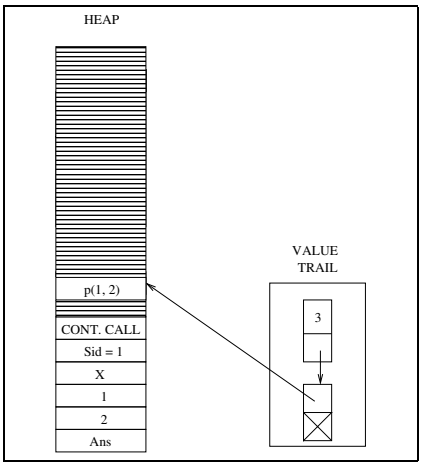


Fig. 11. Setting up the value trail.

just before untrailing T prior to calling C_1. This value trail is used to put back in T the bindings generated by C up to the point in which it was interrupted. Value trails are pointed to from the choicepoints associated with `answer/4`.

Other systems like CHAT or SLG-WAM also spend some extra time while preparing a consumer to be resumed, as they need to record bindings in the forward trail in order to later reinstall them. This is done for every resumption, and not only for reusing calls.

3.4 Freezing Answers

When a consumer is found or when `read_answers/5` is executed a continuation call is created and its third argument needs to be instantiated using the answers found so far to continue execution. These answers are, in principle, stored in the table (i.e., `answer/4` inserted them), and they have to be constructed on the heap so that the continuation call can access them and proceed with execution.

The ideas in Section 3.3 can be reused to freeze the answers and avoid the overhead of building them again. In fact, since there are no reused answers, trail management is not needed for them. As done with the continuation calls, a new field is added to the table pointing to a (Prolog) list which holds all the answers found so far for a tabled goal. When a continuation for some tabled goal is to be executed, the elements of the answer list are unified with the corresponding argument of the continuation call. The list head is, again, accessed through a pointer which is saved in a slot of the corresponding choicepoint and which is updated on backtracking.

In spite of this freezing operation, answers to tabled goals are stored in the table in addition to being linked in a list. There are two reasons for this: the first one is that when some tabled goal is completed, all the answers have to be accessible from outside the derivation tree of the goal. The second one is that the table (which is a trie in our implementation, following [10]) makes checking for duplicate answers faster.

lchain X	Left-recursive path program, unidimensional graph.
lcycle X	Left-recursive path program, cyclic graph.
rchain X	Right-recursive path program (this generates more continuation calls), unidimensional graph.
rcycle X	Right-recursive path program, cyclic graph.
numbers X	Find arithmetic expressions which evaluate to some number N using all the numbers in a list L.
numbers Xr	Same as above, but all the numbers in L are all the same (this generates a larger search space).

Table 1. Terse description of the benchmarks used.

4 Performance Evaluation

We have implemented the proposed techniques as an extension of the Ciao system [1]. Tabled evaluation is provided to the user as a loadable *package* that provides the new directives and user-level predicates, performs the program transformations, and links in the low-level support for tabling. We have implemented and measured three variants: the first one is based on a direct adaptation of the implementation presented in [14], using the standard, high-level C interface. We have also implemented a second variant in which the lower-level and simplified C interface is used, as discussed in Sections 3.1 and 3.2. Finally, a third variant incorporates the proposed improvements to the model discussed in Sections 3.3 and 3.4.

We have then evaluated the performance of our proposal using a series of benchmarks which are briefly described in Table 1. The results are shown in Table 2, where times are given in milliseconds. All measurements have been made using Ciao-1.13 and XSB 3.0.1 compiled with local scheduling and disabling garbage collection in all cases (this in the end did not impact execution times very much). We used `gcc 4.1.1` to compile both systems, and we executed them on a machine with Fedora Core Linux, kernel 2.6.9.

For reference, we have made an attempt to also compare with the execution times reported in [12]. Due to the difference in technology (Prolog system, C compilers, CPUs, available memory, etc.) it is not possible to compare directly with those execution times. Instead, we took those graph benchmarks which can be executed using SLD resolution and measured their execution times on Ciao-1.13. We then compared these times to those reported in [12] (which were originally executed using the then current version of SICStus Prolog) and obtained a speed ratio. Finally, we applied this ratio in order to estimate the execution time that would be obtained for other (tabled) programs by the original implementation in our platform. These *predicted* times for the original continuation call-based execution (when available) are presented in the second column of Table 2.

The three following columns in the table provide the execution times for the three variants implemented as explained at the beginning of this section. It is reassuring to note that the execution times predicted from those in [12] are within reasonable range (and with a relatively consistent ratio) when compared to those obtained from our first, baseline version. We are quite confident, therefore, that they are in general terms compa-

Benchmark	Original	Ciao Ccal	Lower C itf.	Copying
lchain 1024	8.65	7.12	2.85	2.07
lcycle 1024	8.75	7.32	2.92	2.17
rchain 1024	-	2620.60	1046.10	603.44
rcycle 1024	-	8613.10	2772.60	1150.54
numbers 5	-	1691.00	676.40	772.10
numbers 5r	-	3974.90	1425.48	986.00

Table 2. Comparison of original implementation and Ciao implementations.

rable, despite the difference in the base system, C compiler technology, implementation of answer tables, etc.

Lowering the level of the C interface and improving the transformation for tabling and the way calls are performed have a clear impact. It should also be noted that the latter improvement seems to be specially relevant in non-trivial programs which handle data structures (the larger the data structures are, the more re-copying we avoid) as opposed to those where little data management is done. On average, we consider the version reported in the rightmost column to be the implementation of choice among those we have developed, and this is the one we will refer to in the rest of the paper.

Table 3 tries to determine how our implementation of tabling compares with a state-of-the-art one —namely, the latest available version of XSB at the time of writing. In the table we provide, for several benchmarks, the raw time (in milliseconds) taken to execute them using tabling and, when possible, SLD resolution, the speedup obtained when using tabling, for Ciao and XSB, and the ratio of the execution time of XSB vs. Ciao using SLD resolution and tabling.

It should be taken into account that XSB is somewhat slower than Ciao when executing programs using SLD resolution —at least in those cases where the program execution is large enough to be really significant (between 1.8 and 2 times slower for these non-trivial programs). This is partly due to the fact that XSB is, even in the case of SLD execution, prepared for tabled resolution, and thus the SLG-WAM has an additional overhead (reported to be around 10% [13]) not present in other Prolog systems and also presumably that the priorities of their implementors were understandably more focused on the implementation of tabling.

The speedup obtained when using tabling with respect to SLD resolution (the columns marked $\frac{\text{SLD}}{\text{Tabling}}$) is, in general, favorable to XSB, specially for benchmarks which are tabling-intensive but do not resume so many consumers (e.g., the transitive closure without cycles), confirming, as expected, the advantages of the native implementation of tabling in XSB. However, and interestingly, the difference in the speedups between XSB and Ciao tends to reduce as the programs get more complex, mix in more SLD execution, the XSB forward trail gets larger, and consumers are resumed more times, especially if the answers are large and there are no reusing continuation calls.

For example, in the `rchain X` and `numbers X` benchmarks, the speed relation between XSB and Ciao is roughly constant independently of the value of `X`. On the other hand, in `rcycle X` and `numbers Xr` this relation is more favorable to Ciao the larger the execution is. We attribute this to two reasons. The first one is that XSB

does not resume consumers immediately after finding new answers, so it has to pay an extra cost during completion to traverse the list of suspended consumers, and this traversal may have to be repeated several times. The second one is the forward trail that XSB uses: when repeatedly resuming consumers, XSB needs to keep track of the bindings and reinstall them, while our implementation only performs an initial copy between two memory areas (to have a continuation ready to execute) and, since there are no reusing continuation calls in these programs, it can resume continuations in constant time, having better asymptotic behavior. Since the number of resumptions of `rchain X` and `numbers X` is linear in the value of `X`, their behavior is not affected. Besides, answers for `numbers Xr` are relatively large (they are arithmetic expressions) and our implementation freezes them when evaluating a tabled call, while XSB has to reconstruct them whenever a consumer is resumed.

It is also interesting to note that `rchain X` and `rcycle X` are faster in XSB than in Ciao because their execution is tabling intensive. However, in non-trivial benchmarks like `numbers X` and `numbers Xr`, which at least in principle should reflect more accurately what one might expect in larger applications, execution times are in the end somewhat favorable to Ciao. This is probably due in part to the faster raw speed of the basic engine in Ciao but it also implies that the overhead of the approach to tabling used is reasonable after the proposed optimizations. More work is in any case needed to compare further not only with XSB but also with other modern systems supporting tabling. In this context it should be noted that in these experiments we have used the baseline, bytecode-based compilation and abstract machine, but turning on global analysis and the optimizing, low-level compiler [8] can further improve the speed of the SLD part of the computation.

The results are also encouraging to us because they appear to be another example supporting the "Ciao approach" of starting from a fast and robust, but extensible LP-kernel system and then including additional characteristics by means of pluggable components whose implementation must, of course, be as efficient as possible but which in the end benefit from the initial base speed of the system.

5 Conclusions

We have reported on the design and efficiency of some improvements made to the continuation call mechanism of Ramesh and Chen. We argue that the resulting mechanism is still easier to add to an existing, WAM-based system than implementing the SLG-WAM, as it requires relatively small changes to the underlying execution engine. In fact, almost everything is implemented within a fairly reusable C library, and the engine has to be changed only to conditionally reinterpret the `B[H]` field when backtracking.

Our experimental results show that in general the speedups that the SLG-WAM obtains with respect to SLD execution are, as expected, better than the ones obtained by our implementation. However, the difference in raw speed between the systems makes Ciao have sometimes better results in the absolute, as well as sometimes better convergence results.

Our main conclusion is that using an external module for implementing tabling is a viable alternative for adding tabled evaluation to Prolog systems, especially if coupled

Program	Ciao			XSB			$\frac{XSB}{Ciao}$	
	SLD	Tabling	$\frac{SLD}{Tabling}$	SLD	Tabling	$\frac{SLD}{Tabling}$	SLD	Tabling
rchain 64	0.02	2.54	0.0080	0.02	0.9	0.027	1.00	0.35
rchain 256	0.11	37.01	0.0027	0.11	14.4	0.008	1.00	0.39
rchain 1024	0.48	603.44	0.0008	0.42	216.1	0.002	0.88	0.36
rcycle 64	-	4.98	-	-	2.1	-	-	0.42
rcycle 256	-	72.13	-	-	35.2	-	-	0.49
rcycle 1024	-	1150.54	-	-	650.9	-	-	0.56
numbers 3	0.56	0.63	0.88	1.0	0.7	1.43	1.79	1.11
numbers 4	24.89	25.39	0.98	44.4	28.7	1.55	1.78	1.13
numbers 5	811.08	772.10	1.05	1465.9	868.7	1.69	1.81	1.13
numbers 3r	1.62	1.31	1.24	3.3	1.8	1.83	2.04	1.37
numbers 4r	99.74	33.43	2.98	197.7	49.3	4.01	1.98	1.47
numbers 5r	7702.03	986.00	7.81	15091.0	1500.1	10.6	1.96	1.52

Table 3. Comparing the speed of our (Ciao) implementation and XSB.

with the proposed optimizations. It is also an approach that ties in well with the modular approach to extensions which is an integral part of the design of the Ciao system. As a result, the modifications have already been integrated in the Ciao repository and will thus also appear in upcoming distributions.

6 Acknowledgments

This work was funded in part by the IST program of the European Commission, FP6 FET project IST-15905 *MOBIUS*, by the Spanish Ministry of Education and Science (MEC) project TIN2005-09207-C03 *MERIT-COMVERS* and by the Government of the Madrid Region (CAM) Project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo is also funded in part by the Prince of Asturias Chair in Information Science and Technology at the U. of New Mexico, USA. Ricardo Rocha and Cláudio Silva were partially funded by Myddas project (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.
2. Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
3. S. Dawson, C.R. Ramakrishnan, and D.S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 117–126, New York, USA, 1996. ACM Press.

4. Bart Demoen and K. Sagonas. CHAT is θ(SLG-WAM). In D. Mc. Allester H. Ganzinger and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *Lectures Notes in Computer Science*, pages 337–357. Springer, September 1999.
5. Bart Demoen and Konstantinos Sagonas. CAT: The Copying Approach to Tabling. In *Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 1998.
6. Bart Demoen and Konstantinos F. Sagonas. Chat: The copy-hybrid approach to tabling. In *Practical Applications of Declarative Languages*, pages 106–121, 1999.
7. Hai-Feng Guo and Gopal Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, pages 181–196, 2001.
8. J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.
9. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 143–154. Springer Verlag, 1997.
10. I. V. Ramakrishnan, Prasad Rao, K. F. Sagonas, Terrance Swift, and David Scott Warren. Efficient tabling mechanisms for logic programs. In *ICLP*, pages 697–711, 1995.
11. Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
12. R. Ramesh and Weidong Chen. A Portable Method for Integrating SLG Resolution into Prolog Systems. In Maurice Bruynooghe, editor, *International Symposium on Logic Programming*, pages 618–632. MIT Press, 1994.
13. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, May 1998.
14. C. Silva, R. Rocha, and R. Lopes. An External Module for Implementing Linear Tabling in Prolog. In S. Etalle and M. Truszczyński, editors, *International Conference on Logic Programming*, number 4079 in LNCS, pages 429–430, Seattle, Washington, USA, August 2006. Springer-Verlag.
15. H. Tamaki and M. Sato. OLD resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, 1986. Lecture Notes in Computer Science, Springer-Verlag.
16. D.S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.
17. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.
18. Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), October 2001.
19. Youyong Zou, Tim Finin, and Harry Chen. F-OWL: An Inference Engine for Semantic Web. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 238–248. Springer Verlag, January 2005.

A Control Flow Graph for Mercury

François Degrave and Wim Vanhoof

University of Namur
Faculty of Computer Science
Rue Grandgagnage 21, B-5000 Namur, Belgium
e-mail: {fde,wva}@info.fundp.ac.be

Abstract. In this work, we define how one can build a control flow graph for programs written in the logic programming language Mercury. We formally relate paths in this graph to program executions, including failure and backtracking. We illustrate how our graph could be used for program analysis by defining a simple (un)reachability analysis for Mercury.

1 Introduction

Control flow graphs are well-known and widely used structures in software development tools such as compilers and debuggers. Their main interest lies in the fact that they provide an explicit representation of a program's control flow structure which makes them well-suited as a building block for implementing program analyses and optimizations such as dead-code elimination, branch predicate, loop transformations, etc. [5, 1]. Moreover, the fact that they can easily be visualised makes that they are frequently used in debugging and (semi) automatic test-case generation (e.g. [10]).

Notwithstanding these applications, the construction and use of control flow graphs for logic programs have received little attention. Some notable exceptions include [4, 3, 2]. This should not be surprising, given that in logic programming languages control information is far less explicit in programs and hence more difficult to catch in a static structure.

In this work, we define how one can build and use a control flow graph for the logic programming language Mercury. The fact that Mercury is a *moded* language makes it easier to extract control flow information from a program than it would be the case for an unmoded language such as Prolog. Nevertheless, the resulting structure is a non-trivial extension of its counterpart for imperative programs, since it needs to allow for reasoning about success and failure of goals, backtracking, and multiple answers.

Our paper is structured as follows. After a short introduction to Mercury, we define in Section 2 how one can build a control flow graph for programs written in the language. In Section 3 we formalise the relation between a program execution – possibly failing or returning multiple answers – and a path in the statically created graph. Before concluding, we show in Section 4, how such a control flow graph could advantageously be used for program analysis by defining a simple (un)reachability analysis for Mercury.

2 A Control Flow Graph for Mercury

Mercury is a statically typed logic programming language which offers a *mode system* describing how the instantiation of a variable changes over the execution of a goal. Each predicate argument is classified as either input to the call, denoted by *in* (the argument is a ground term before and after the call) or output by the call which is denoted by *out* (the argument is a free variable that will be instantiated to a ground term at the end of the call). A predicate may have more than one mode, each mode representing a particular usage of the predicate. Each such mode is called a *procedure* in Mercury terminology. Each procedure has a declared (or inferred) *determinism* which states the number of solutions it can generate and whether it can fail. Determinisms supported by Mercury include `det` (indicating that a call to the procedure will succeed exactly once), `semidet` (a call will either succeed once or fail), `multi` (a call will generate at least one solution but possibly more), and `nondet` (a call can either fail or generate one or more solutions). There exist other instantiation states and determinisms but these are outside the scope of this paper; we refer to [9] for details. Let us consider for example the definition of the well-known `append/3` and `member/2` predicates. We provide two mode declarations for each predicate, reflecting their most common usages:

```
:- pred append (list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.

append([], Y, Y).
append([E|Es], Y ,[E|Zs]):- append(Es, Y, Zs).

:- pred member(T, list(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.

member(X, [X|_]).
member(X, [Y|T]) :- not (X=Y), member(X, T).
```

For `append/3`, either the first two arguments are input and the third one is output in which case the call will succeed exactly once (det), or the third argument is input and the first two are output in which case the call may generate multiple solutions (multi). Note that no call to `append/3` in either of these modes can fail. For `member/2`, either both arguments are input and the call will either succeed once or fail (semidet), or only the second argument is input, in what case the call can fail, or generate one or more solutions (nondet).

In this work, we consider a program as being a set of procedures. We assume thus that every mode of a predicate has been translated into a different procedure and that, in addition, every procedure is well-typed and well-moded. Such a procedure can be translated to *superhomogeneous form* [9]. A procedure in superhomogeneous form consists of a single clause (usually a disjunction) in which

the arguments in the head of the clause and in procedure calls in the body are all distinct variables. Explicit unifications are generated for these variables in the body, and complex unifications are broken down into several simpler ones. The following definition gives a *labelled syntax* for procedures in superhomogeneous form. That is, we associate a distinct *label* to a number of program points of interest. These labels – which are written in subscripts and attached to the left and/or the right of a goal – are intended to indentify the nodes of the program's control flow graph.

Definition 1. *Let Π denote the set of procedures symbols, Σ the set of function symbols and $\mathcal{V}$ and $\mathcal{L}$ respectively the set of variables and labels in a given program P. The syntax of a program in labelled superhomogenous form is defined as follows:*

$$
\begin{array}{lll}
LProc & ::= & p(X_1, \ldots, X_k) : -LConj. \\
LConj & ::= & {}_lG_{l'} \\
 & | & {}_lG, C \\
LDisj & ::= & C'; C'' \\
 & | & D; C \\
LGoal & ::= & Atom \\
 & | & Disj \\
 & | & not(C) \\
 & | & if\ C\ then\ C'\ else\ C'' \\
Atom & ::= & X == Y \mid X \Rightarrow f(Y_1, ..., Y_n) \mid X \Leftarrow f(Y_1, ..., Y_n) \\
 & | & Z := X \mid p(X_1, ..., X_n)
\end{array}
$$

where $C, C', C'' \in LConj$, $D \in LDisj$, $G \in LGoal$, $a \in Atom$, X, Y, Z and $X_i, Y_i (0 \leq i \leq n) \in \mathcal{V}, p/k \in \Pi, f \in \Sigma, l, l' \in \mathcal{L}$. All labels within a given program are assumed to be distinct.

The body of a procedure is defined as a conjunction of goals. A goal is either an atom or a number of goals connected by *disjunction*, *if then else* or *not*. An atom is either a procedure call or an unification. As a result of mode analysis, each unification is characterized either as a *test* denoted by X == Y (where both X and Y are of atomic type and input), an *assignment* denoted by Z := X, a *construction* denoted by X $\Leftarrow f(Y_1, \ldots, Y_n)$ (where X is output, $Y_1, \ldots, Y_n$ are input variables or a *deconstruction* denoted by X $\Rightarrow f(Y_1, \ldots, Y_n)$ (where X is input, $Y_1, \ldots, Y_n$ are output variables). Moreover, the conjunctions are reordered such that values are produced before they are consumed when the predicate is executed by a left-to-right selection rule [9]. For a procedure p we denote by $head(p)$ and $body(p)$ respectively the head atom and body of p's definition.

Example 1. The `append(in,in,out)`, `member(in,in)` and `member(out,in)` procedures in labelled superhomogeneous form would look like:

$$\texttt{append}(\texttt{X} :: \texttt{in}, \texttt{Y} :: \texttt{in}, \texttt{Z} :: \texttt{out}) \ :-$$
$$\quad {}_{1_1}({}_{1_2}\texttt{X} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}]_{,1_3}\ \texttt{append}(\texttt{E}_\texttt{s}, \texttt{Y}, \texttt{W})_{,1_4}\ \texttt{Z} \Leftarrow [\texttt{E}|\texttt{W}]_{1_5}\ ;\ {}_{1_6}\texttt{Z} = \texttt{Y}_{1_7})_{1_8}.$$

$$\texttt{member}(\texttt{X} :: \texttt{in}, \texttt{Y} :: \texttt{in}) \ :-$$
$$\quad {}_{1_1}\texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}]_{,1_2}\ ({}_{1_3}\texttt{X} == \texttt{E}_{1_4}\ ;\ {}_{1_5}\texttt{member}(\texttt{X}, \texttt{E}_\texttt{s})_{1_6})_{1_7}.$$

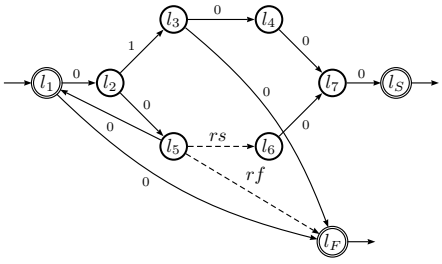
$$\texttt{member}(\texttt{X} :: \texttt{out}, \texttt{Y} :: \texttt{in}) \ :-$$
$$\quad {}_{1_1}\texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}]_{,1_2}\ ({}_{1_3}\texttt{X} := \texttt{E}_{1_4}\ ;\ {}_{1_5}\texttt{member}(\texttt{X}, \texttt{E}_\texttt{s})_{1_6})_{1_7}.$$

Note that our labelled syntax is conceived in such a way that each individual subgoal is surrounded by two labels. In what follows we will use the function `first`, defined on labelled conjunctions and disjunctions, which returns the label preceding a given goal.

$$\begin{aligned}
\texttt{first}\ ({}_lG_{l'}) &= l \\
\texttt{first}\ ({}_lG, C) &= l \\
\texttt{first}\ (C; C') &= \texttt{first}(C) \\
\texttt{first}\ (D; C) &= \texttt{first}(D)
\end{aligned}$$

In what follows we translate a program into a control flow graph such that a path in this graph corresponds to a particular execution of the program. The nodes of the directed graph are the labels occuring in the program, together with two special labels: a *success label* l_S and a *failure label* l_F. These two labels represent the two possible end points of an execution success or failure. In a control flow graph, we will make a distinction between three different types of edges. A *regular* arc (l, l') denotes a possible transition from the program point labelled l to the one labelled l'. Regular arcs are annoted by a natural number n, which we will write in superscript along the arc, as in $(l, l')^n$. This will allow later on, during path construction, to impose an order between different arcs that leave a single node representing a choicepoint. In what follows we will often refer to that number as the *priority* of the arc. A *return-after-success* or *return-after-failure* arc, denoted $(l, l')^{rs}$ respectively $(l, l')^{rf}$, implies that the atom preceded by l is a procedure call and points out the label l' where the execution should be resumed upon success, respectively failure, of the call.

Example 2. Let us consider `member(in,in)`, defined in Example 1. The control flow graph corresponding to a program containing only this procedure would look as follows:



For example, the first arc (l_1, l_2) denotes the success of the atom $Y \Rightarrow [E|E_s]$. The arc (l_1, l_F) denotes the fact that this deconstruction can fail. In order not to overload the figure, we only depict priorities when relevant, i.e. for subgraphs representing a disjunction : the edge initiating the first disjunct is annotated by the highest value, the edge initiating the last disjunct by the lowest value – which will always be zero. As such these values (or priorities) reflect the order of the corresponding disjuncts in the source code. A node from which leave several edges having different priorities represents effectively a *choicepoint*. When walking through the graph in order to collect execution paths, the highest priority must be chosen first. This denotes the fact that, operationally, in a disjunction $(D_1; D_2)$ the disjunct D_1 is executed first, and D_2 is executed only if a backtracking occurs.

Formally, we can define a control flow graph of a program thus as follows:

Definition 2. *Let P be a labelled program and let $\mathcal{L}^+$ denote the set of labels occuring in P augmented by the dedicated success and failure labels, i.e. $\mathcal{L}^+ = \mathcal{L} \cup \{l_S, l_F\}$. The* control graph of P*, denoted by G_P, is a directed graph $(\mathcal{L}^+, E)$ with $E \subseteq \mathcal{L}^+ \times \mathcal{L}^+$ in which each edge in E is annotated as either* return-after-success*,* return-after-failure *or with a priority number.*

The control flow graph for a program P can be constructed by considering each procedure in isolation. If we let $\mathcal{A}rc$ denote the set of all annotated edges over $\mathcal{L}^+$, we can define a function $\texttt{graph} : LGoal \times \mathcal{L}^+ \times \mathcal{L}^+ \times \mathbb{N} \rightarrow \mathcal{P}(\mathcal{A}rc)$ that builds the set of edges associated with a labelled goal. The two label parameters represent, respectively, the points at which execution should resume upon success or failure of the given goal, and the natural number parameter represents the priority given to the arc initiating the next disjunct when treating a disjunction. As such, we can define the set of edges of the control flow graph of a complete program as the union of the sets of edges associated to the body goals of the individual procedures:

$$G_P = (\mathcal{L}^+, \bigcup_{p \in \Pi} \texttt{graph}(body(p), l_S, l_F, 0)).$$

where l_S and l_F represent the special success and failure labels of the control flow graph. The function `graph` is then defined by induction on the structure of a labelled goal, as follows:

$$\begin{aligned} &\texttt{graph}\ (_l\ G\ _{l'},\ l_s,\ l_f,\ nb) = \{(l', l_s)^0\} \cup arcs_G \qquad (1)\\ &\textbf{where}\ arcs_G = \texttt{graph}\ ((_lG),\ l',\ l_f,\ nb) \end{aligned}$$

In case of a goal surrounded by two labels l and l', we build the graph for $_lG$ using l' as success resume point, and we connect l' to l_s such that upon success of G, execution will be resumed at l_s. A conjunction is treated as follows:

$$\begin{aligned} &\texttt{graph}((_lG, C),\ l_s,\ l_f,\ nb) = arcs_C \cup arcs_G \qquad (2)\\ &\textbf{where}\ arcs_C = \texttt{graph}(C,\ l_s,\ l_f,\ nb)\\ &\qquad\quad arcs_G = \texttt{graph}((_lG, \texttt{first}(C),\ l_f,\ nb) \end{aligned}$$

In a conjunction of the form $({}_l G, C)$ (2), the failure of any conjunct leads to the failure of the entire conjunction. The success of the conjunct G results in the execution of C – that's why the local graph of G is built by using the first label of C as success resume label.

Let a denote a unification, then we define :

$$\begin{array}{l}\texttt{graph}\ ({}_l\ a,\ l_s,\ l_f,\ nb) = \\ \quad \begin{cases} \{(l,l_s)^0\} & \textbf{if } a = \ Z := X |\ Z \Leftarrow f(Y_1,\ldots,Y_n) |\ Z := X \oplus Y \\ \{(l,l_s)^0, (l,l_f)^0\} & \textbf{otherwise} \end{cases}\end{array} \tag{3}$$

Case 3 is one of the base cases of the `graph` function. It treats the case of a unification a preceded by a label. If the unification is bound to succeed, we only add a single regular arc, linking the label preceding a to the success resume label l_s. If it can fail, then we also add another regular arc, linking the label before a to the failure resume label l_f.

$$\begin{array}{l}\texttt{graph}\ ({}_l\ p(X_1,...,X_n),\ l_s,\ l_f,\ nb) = \ \{(l,l_p)^0, (l,l_s)^{rs}, (l,l_f)^{rf}\} \\ \textbf{where } l_p = \texttt{first}(body(p))\end{array} \tag{4}$$

In case of a predicate call (4), we link the label appearing in front of the call to the entry label of p's body goal. We add a return-after-success arc linking l to l_s and, likewise, a return-after-failure arc linking l to l_f. Note that this return-after-failure arc does not need to be added if the determinism of p is either `det` or `multi` meaning that the call cannot fail.

$$\begin{array}{l}\texttt{graph}\ ({}_l\ \textbf{if } C \textbf{ then } C' \textbf{ otherwise } C'',\ l_s,\ l_f,\ nb) = \\ \qquad \{(l, \texttt{first}(C))^0\} \cup arcs_1 \cup arcs_2 \cup arcs_3 \\ \textbf{where } arcs_1 = \texttt{graph}(C,\ \texttt{first}(C'),\ \texttt{first}(C''),\ nb) \\ \qquad arcs_2 = \texttt{graph}(C',\ l_s,\ l_f,\ nb) \\ \qquad arcs_3 = \texttt{graph}(C'',\ l_s,\ l_f,\ nb)\end{array} \tag{5}$$

In the *if-then-else* construction (5), the success of the *test* conjunction C leads to the execution of the *then* conjunction C', and its failure leads to the execution of the *else* conjunction C''. For this reasons, we build the local graph of C using the first label of C' as the success resume label and the first label of C'' as the failure resume label.

$$\begin{array}{l}\texttt{graph}\ ({}_l not(C),\ l_s,\ l_f,\ nb) = \{(l, \texttt{first}(C))^0\} \cup arcs_C \\ \textbf{where } arcs_C = \texttt{graph}(C,\ l_f,\ l_s,\ nb)\end{array} \tag{6}$$

The graph of the goal *not(C)* (6) is obtained by inverting the success label and the failure resume labels when constructing C's graph. By doing this, failure of C will lead to a success for $not(C)$, and vice versa.

$$\begin{array}{l}\texttt{graph}({}_l(D;C),\ l_s,\ l_f,\ nb) = \{(l, \texttt{first}(C))^{nb}\} \cup arcs_D \cup arcs_C \\ \textbf{where } arcs_D = \texttt{graph}({}_l D,\ l_s,\ l_F,\ (nb+1)) \\ \qquad arcs_C = \texttt{graph}(C,\ l_s,\ l_f,\ 0)\end{array} \tag{7}$$

In a disjunction (7), failure of the first disjunct D implies backtracking to the second one C. Therefore, when constructing D's graph, the failure label used is the global failure label l_F. On the other hand, the failure of this second disjunct C leads to the failure of the whole expression; the failure label used for C's graph is then the current failure label. Also note that the label l can be seen as a choicepoint; the graph corresponding to a disjunction will contain as many arcs starting from l as there are disjuncts in the disjunction. Every such arc will be labelled with a number denoting the priority in execution of the disjunct it corresponds to – the highest the number, the highest the priority. We construct the graph of ${}_l(C; C')$ in a similar way:

$$\begin{aligned}
&\texttt{graph}({}_l(C;C'),\ l_s,\ l_f,\ nb) = \\
&\qquad \{(l, \texttt{first}(C'))^{nb}, (l, \texttt{first}(C))^{nb+1}\} \cup arcs_C \cup arcs_{C'} \qquad (8)\\
&\textbf{where } arcs_C = \texttt{graph}(C,\ l_s,\ l_F,\ 0)\\
&\qquad\quad arcs_{C'} = \texttt{graph}(C',\ l_s,\ l_f,\ 0)
\end{aligned}$$

3 Extracting execution paths

In the following paragraph, we will show how program execution can be modeled using the defined control flow graph.

Formally, we will represent an execution path as a sequence of *execution segments*; each segment being itself a sequence of labels denoting a partial execution – that is a sequence of computation steps ending in success or failure that do not include backtracking.

Definition 3. *Let $G_P = (\mathcal{L}^+, E)$ be a control flow graph, we define an* execution segment *in G_P as a sequence of labels over $\mathcal{L}^+$. We say that an execution segment $S = \langle l_1, \ldots, l_k \rangle$ is* valid *iff the following conditions hold:*

1. *l_k is either l_S or l_F; and*
2. *$\forall i : 1 \leq i < k$ we have that*
   (a) *either $(l_i, l_{i+1})^{nb} \in E$;*
   (b) *or $l_i = l_S$ and $\exists\ l_m \in S$ $(m < i)$ such that $(l_m, l_{i+1})^{rs} \in E$ and $\langle l_{m+1}, \ldots, l_i \rangle$ is a valid execution segment.*
   (c) *or $l_i = l_F$ and $\exists\ l_m \in S$ $(m < i)$ such that $(l_m, l_{i+1})^{rf} \in E$ and $\langle l_{m+1}, \ldots, l_i \rangle$ is a valid execution segment and $\neg\exists (l_j, l_v)^{nb} \in E$ with $nb > 0$ and $m < j < i$*

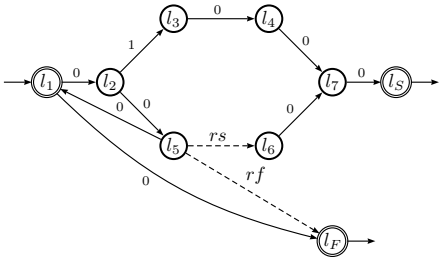
The execution segment S is complete *iff $\#\{l \in S \mid (l, l')^{rs} \in E\} = \#\{l' \in S \mid (l, l')^{rs} \in E\}$.*

A valid execution segment basically constitutes a sequence of labels ending with either l_S or l_F (condition 1) such that each pair of successive labels (l_i, l_{i+1}) with $l_i \neq l_S$ and $l_i \neq l_F$ is connected in the control flow graph (condition 2a). If the segment contains l_S (or l_F) in a position other than the last one, this

must denote success (or failure) of a procedure and hence the following label in the sequence must be the endpoint of an appropriate return arc (conditions 2b and 2c). Moreover, the subsequence $\langle l_{m+1}, \ldots, l_i \rangle$, representing the execution between the call and return, must itself be a valid execution segment. However, the return after the *failure* of a call can be performed only if the corresponding call definitely failed, i.e. it is impossible to perform backtracking to a choicepoint created after the call that would make the latter succeed; this condition means, in terms of path construction, that all pairs of consecutive labels between the call and the return – i.e. between l_m and l_i – are linked with an arc with a priority equal to zero. An execution segment is complete if and only if every call has a corresponding return.

Example 3. Let us consider `member(in,in)`, defined in example 1. The sequence of labels $s = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ represents a valid and complete execution segment in the control flow graph depicted in Example 2. Note that it corresponds to the execution of a call in which the deconstruction of the list into a first element and a tail succeeds, as does the equality test between the former element and the call's first argument, leading to the success of the predicate. In other words, the execution segment s represents a call `member(X,Y)` in which the element X figures at the first position in the list Y.

Example 4. Let us now consider the nondeterministic `member(out,in)` procedure, also defined in Example 1. Its control flow graph is as follows :



The sequence of labels $s = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ represents a valid and complete execution segment. The execution segment s represents the execution leading to the first solution of a call `member(X,Y)` in which the list Y is not empty.

An execution path is a sequence of execution segments, each of them being itself a sequence of labels denoting a partial execution ending in success or failure. As such, in a path $\langle S_1, ..., S_n \rangle$, a segment S_i $(i \neq 1)$ represents the continuation of the execution of S_{i-1} by backtracking. Consequently, the first label of S_i $(i \neq 1)$ must necessarily be the label of a node that is connected with a choicepoint.

Example 5. Let us consider again the nondeterministic `member(out,in)` procedure. The execution path

$$p = \langle \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle, \langle l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle, \langle l_5, l_1, l_F, l_F, l_F \rangle \rangle$$

corresponds to the execution of a call $member(X, Y)$ in which a first solution is produced by assigning the first element of the list Y to X and returning from the

call (expressed by the first segment of the path, ending in l_S). A second solution is produced by backtracking, continuing the execution at l_5, performing a recursive call, assigning the second element of the original list to X, and performing the return (the second segment, also ending in l_S). The execution continues by backtracking and continuing at l_5 and performing a recursive call in which the deconstruction of the list argument fails. In other words, the execution path p corresponds to a call to member in which the second argument is instantiated to a list containing only two elements.

Since a segment S_i ($i \neq 1$) represents the continuation of the execution of S_{i-1} by backtracking, concatenating the relevant part of S_1 (i.e. representing the execution up to the choicepoint used for backtracking) with S_2 should give us a new valid segment, denoting a partial execution in which a different choice was commited.

Definition 4. *Let $G_P = (\mathcal{L}^+, E)$ be a control flow graph for a program P and S_1, S_2 two execution segments. We define the* fusion *of the relevant part of the execution segment S_1 with the execution segment S_2 as follows:*

$$\texttt{fusion}(\langle l_1^1, ..., l_n^1\rangle, \langle l_1^2, ..., l_m^2\rangle) = \langle l_1^1, ..., l_j^1, l_1^2, ..., l_m^2\rangle,\ j < n$$

such that :

1. $(l_j^1, l_{j+1}^1)^{nb} \in E,\ nb > 0$
2. $(l_k^1, l_{k+1}^1)^{nb'} \notin E,\ k > j,\ nb' > 0$
3. $(l_j^1, l_1^2)^{nb-1} \in E$

In this definition, l_j refers to the last choicepoint appearing in S_1 having at least 1 remaining choice left (1 and 2). The first label of the second segment must be the next choice at this choicepoint (3). This new definition allows us to define an *ordered path*, which is an execution path in which all the possible backtrackings are performed in the appropriate order.

Definition 5. *We define an* ordered path *as a sequence of execution segments $\langle S_1, \ldots, S_n\rangle$ such that the following conditions hold:*

1. *if $n = 1$ then S_1 is valid and complete and, if (1) the mode of the predicate under consideration is* `multi` *or* `nondet`*, or (2) S_1 ends with l_F, then there must not exist $l_i, l_{i+1} \in S_1$ such that $(l_i, l_{i+1})^{nb} \in E,\ nb > 0$*
2. *if $n > 1$ then :*
   (a) $\forall 1 \leq i \leq n$, S_i is a valid execution segment
   (b) $\texttt{fusion}(S_1, S_2)$ exists and is a valid execution segment
   (c) the path defined as $\langle \texttt{fusion}(S_1, S_2), \ldots, S_n\rangle$ is an ordered path

If an execution path consists of a single segment, this segment must be valid and complete. This condition is sufficient if the predicate under consideration is either `det` or `semidet` and the segment ends with l_S; indeed, this means that the execution has reached the only solution, if it exists. Otherwise, we must add the condition that none of the involved labels must be the departure point of an arc

with a priority higher than 0 in the graph (1). Indeed, if this were the case, it would mean that there exists a choicepoint in the segment having an alternative that has not been exploited.

Moreover, we have to pay attention to the fact that every segment composing a path must appear in the correct order, i.e. the segment S'_1 obtained by concatenating the relevant part of S_1 with S_2 is valid, and the path $\langle S'_1, ..., S_n \rangle$ obtained by replacing S_1 and S_2 by S'_1 in the original path is ordered itself (2b).

Consequently, if we consider a predicate having an infinity of solutions, the only execution paths considered as valid for this predicate would have an infinite length. In practice, such a path can easily be "cut" after a given number of solutions.

Definition 6. *Let $G_P = (\mathcal{L}^+, E)$ be a control flow graph for a program P and p a procedure defined in P, we define an* execution path *for p in G_P as a sequence of execution segments $\langle S_1, \ldots, S_n \rangle$ such that the following conditions hold:*

1. *if $S_1 = \langle l_1, \ldots, l_n \rangle$ then $l_1 = \texttt{init}(p)$*
2. *$\langle S_1, \ldots, S_n \rangle$ is an ordered path and if $S_i = \langle l^i_1, \ldots, l^i_n \rangle$ then $\forall 1 \leq j < n$: if $(l^i_j, l^i_{j+1})^{nb} \in E$ then $(l^i_j, l_k)^{nb+1} \notin E$*

Since in a disjunction, all the disjuncts except the first one can only be reached by performing a backtracking – either after a failure of the disjuncts preceding them or after a success of the execution –, a label preceding a disjunct which is not the first of a disjunction will appear in a path only as the first label of a segment. Therefore, if a choicepoint l_c appears in a segment, the label following it l_{c+1} must necessarily be the first label of the first disjunct of the corresponding disjunction, i.e. the arc $(l_c, l_{c+1})^{nb}$ is the arc having the highest priority among the arcs starting from the choicepoint l_c.

4 Simple unreachability analysis

The control flow graph we defined in the previous sections allows us to easily perform some analysis on Mercury programs. As an illustration we will devise a simple "unreachability" analysis for Mercury. That is, given a label l, the analysis will try to detect whether l cannot be reached by any execution and, as such, represents dead code in the program. Let us define the concept of primitive path. Intuitively, a primitive path is an execution path calculated from a graph without taking into account the mechanisms of backtracking, priorities in disjunctions and procedure calls.

Definition 7. *Let $G_p = (\mathcal{L}^+, E)$ be a control flow graph for a procedure p, we define a* primitive path *for p in G_P as a sequence of labels $\langle l_1, ..., l_n \rangle$ such that :*

1. *$l_1 = \texttt{init}(p)$*
2. *$l_n = l_S$ or $l_n = l_F$*
3. *$\forall i, j : i \neq j \Rightarrow l_i \neq l_j$*
4. *$\forall l_i \in \mathcal{L}^+,\ i < n$, $(l_i, l_{i+1})^{nb} \in E$ or $(l_i, l_{i+1})^{rs} \in E$ or $(l_i, l_{i+1})^{rf} \in E$.*

A primitive path is just a path along the graph, starting from the first label of the procedure and ending at either the success or the failure label, and where all the labels are distinct (a primitive path cannot cycle). Therefore, there is only a finite number of such paths that we can extract from a given graph.

Example 6. Let us consider again the nondeterministic `member(out,in)` procedure, defined in Example 1. The primitive paths of this procedure are :

$$\langle l_1, l_2, l_3, l_4, l_7, l_S\rangle, \langle l_1, l_F\rangle, \langle l_1, l_2, l_5, l_6, l_7, l_S\rangle, \langle l_1, l_2, l_5, l_F\rangle$$

In what follows we assume that, in the control flow graph, edges originating from a label associated to an atom are annotated by a corresponding constraint, depending of the kind of atom and whether it succeeds or fails, as follows:

a	(l, l')	$(l, l'')\ l' \neq l''$
${}_lX := Y_{l'}$	$x = y$	$true$
${}_lX == Y_{l'}$	$x = y$	$x \neq y$
${}_lX <= f(Y_1, \ldots, Y_n)_{l'}$	$x = f(y_1, \ldots, y_n)$	$true$
${}_lX => f(Y_1, \ldots, Y_n)_{l'}$	$x = f(y_1, \ldots, y_n)$	$x \neq f(y_1, \ldots, y_n)$

In order to collect the constraints associated to an primitive path, the basic idea is to walk the segment and collect the constraints associated to the corresponding edges. We formalise this process by defining a function $\mathcal{U}$ that, given an primitive path returns the constraints collected along the segment.

Definition 8. *Let $G_p = (\mathcal{L}^+, E)$ be a control flow graph and let $\langle l_1, \ldots, l_n\rangle$ be a (prefix of a) valid primitive path. We define $\mathcal{U}(\langle l_1, \ldots, l_n\rangle)$ as the set of constraints C obtained as follows:*

- *If $((l_1, l_2)^{nb} \in E$ then let c be the constraint associated to the edge (l_1, l_2). Let $C' = \mathcal{U}(\langle l_2, \ldots, l_n\rangle)$, then we define $C = \{c\} \cup C'$.*

For the base case of the recursion, we have that $\mathcal{U}(\langle l\rangle, M) = \emptyset$.

Definition 9. *Let $G_p = (\mathcal{L}^+, E)$ be a control flow graph for a procedure p, and $\mathcal{P}_p$ the set of primitive paths of G_p. The label $l \in \mathcal{L}^+$ is* unreachable *if $\forall pp \in \mathcal{P}_p$ such that $l \in pp$, the set of constraints $\mathcal{U}(pp')$ is not satisfiable, where pp' is the unique prefix of pp ending with l.*

This definition denotes the fact that a program point cannot be reached if every possible execution leading to this point necessarily fails. Uniqueness of the prefix is guaranteed by the fact that all labels of a primitive path are distinct. Of course, this condition for unreachability is sufficient but by no means necessary, since the problem is in general undecidable.

5 Conclusion

In this work, we have defined how one can build a control flow graph for programs written in Mercury, and we have formally shown how program executions

relate to paths in our graph. We have also illustrated, by defining a simple unreachability analysis, how such a graph could be used when analysing Mercury programs. While the analysis defined in Section 4 is rather simple, we conjecture that also more involved analyses can fruitfully use such control flow graphs. The described construction has been implemented, and in our current work we are using the graphs for driving an algorithm for automatic test case generation for Mercury [6].

To the best of our knowledge, little work exists in which control flow graphs are formally defined and used in a logic programming setting. In [3], the authors present a visual debugging environment for Mercury programs, which employs a layered AND-OR tree representation of the program to provide a way of visualizing the execution of a program. Similar tree representations have been previously used in the context of Prolog, for example in the Transparent Prolog Machine [2]. Other work on Prolog includes [4], in which the author makes the control flow of Prolog more explicit by applying program transformations. These transformations introduce, for example, predicates that save and restore the program state at choicepoints. The results of a program analysis are subsequently used to contruct a control flow graph.

Although our own work is particularly targeted to Mercury, it would be interesting to see whether and how the defined notions could be generalised for use in other logic programming languages, in particular Prolog.

Acknowledgments

The authors would like to thank Baudouin Le Charlier for interesting and productive discussions on the subject of this paper. We thank the anonymous referees for their valuable input.

References

1. Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.
2. Mike Brayshaw and Marc Eisenstadt. A practical graphical tracer for Prolog. *International Journal of Man-Machine Studies*, 35(5):597–631, 1991.
3. M. Cameron, M. Garcia de la Banda, K. Marriott, and P. Moulder. Vimer: A visual debugger for Mercury. In *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2003.
4. Thomas Lindgren. Control flow analysis of Prolog. In *International Logic Programming Symposium*, pages 432–446, 1995.
5. Steven Muchnick. *Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
6. Nathalie Mweze and Wim Vanhoof. Automatic generation of test inputs for Mercury programs (abstract), 2006. Preproceedings of LOPSTR 2006.
7. A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

8. Z. Somogyi, H. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1), 1996.
9. Z. Somogyi, H. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(4), 1997.
10. Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.