Modelling control

Roberto Barbuti, Tiziana Castagnetti
Roberto Giacobazzi, Giorgio Levi

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
(barbuti,casta,giaco,levi)@dipisa.di.unipi.it

1 Introduction

The semantics of pure logic programs has traditionally been given in terms of the model theory of first-order logic. In real logic languages, like Prolog, some control features (like the ordering of clauses and the selection rule) are needed for implementation purposes. A formal semantics definition of Prolog is required for semantic-based analysis and program transformation techniques. For such class of languages, the model theoretic approach is inadequate for reasoning about the computational behavior of programs [7,6]. From a computational point of view, in fact, the usual model theoretic semantics ignores several behavioral aspects such as termination, the use of a sequential depth-first search strategy and controlling search constructs like cuts. The semantics of Prolog is therefore always given by encoding the operational semantics within different frameworks like dynamic algebras [3], deterministic transition systems [4] or denotational semantics [12,1,5].

The above approaches are not based on the first order logic semantics as we would like for logic language such as Prolog. In this paper we define a constraint logic language (τ) which allows to give a model theoretic and fixpoint characterization of the Prolog control features.

The τ-language inherits from the *cc* paradigm [14], some of the constraint features used for the synchronization in concurrent logic programming, like *ask* and *tell* constraints.

In the *cc* framework, a computation corresponds to a concurrent execution of agents which perform ask and tell constraints on a given global structures named store. Ask is performed by asking if a constraint is entailed by the current store. Differently from the concurrent framework, in the sequential language the ask constraint (used to represent a test on the unifiability of terms) is not allowed to generate a suspension . As usual, we perform computations by monotonically extending the store with the tell constraints.

From a semantic viewpoint, the τ-language naturally supports the notion of sequence of constraints. As stressed in [9], the semantic domains which characterize the behavior of actions in the *cc* paradigm are tree structures. Since we deal with a constraint language within a sequential framework, we can use these domains to characterize proof trees associated with Prolog programs. This is performed by translating the Prolog programs in the τ-language. Thus, in order to give a satisfactory semantics of Prolog, i.e. a model theoretic

and fixpoint semantics based on an immediate consequence operator, we can consider a corresponding semantics characterization given in terms of sequences of constraints in the τ framework.

Section 2 recalls some preliminaries on the constraint logic languages. The framework together with its fixpoint semantics is presented in Section 3. In Section 4 the transformation rule and some examples are presented.

2 Preliminaries

Let us recall some basic concepts about constraint logic programming (CLP). For a more complete treatment of the subject see [8,9,10,11,14].

We first define notational terminology for the first order language. Let us consider a *signature* of n-ary function symbols f in a finite set of sorts $SORT = \bigcup_i SORT_i$. Let Π be a set of predicate symbols such that $\Pi = \pi_c \cup \pi_P$ and $\pi_c \cap \pi_P = \emptyset$. An *atom* has the form $p(t_1, ..., t_n)$ where t_i is a term defined on the signature and $p \in \Pi$.

In general, for a constraint system, the basic notion is consistency in a given algebraic structure. Since we deal with a particular class of constraints (ask constraints), we extend the definition of a constraint system to support the notion of entailment. Let us consider a very simple class of first order constraint system.

Definition 2.1 *A simple first order constraint system is a quadruple* $\Re = \langle \Sigma, \mathcal{A}, Var, \Phi \rangle$ *where*

- Σ *is a many sorted first order vocabulary with associated sorts* S *(in a signature) and ranking function* ρ*, consisting of a denumerable collection of function symbols* $\mathcal{F}$ *and a denumerable collection of predicate symbols* π_c*.*
- $\mathcal{A}$ *is a* Σ*-structure defined on the alphabet of* Σ*, consisting of*
  1. *a collection of non-empty sets* $D\Re_s$*,* $s \in S$*;*
  2. *an assignment* $D\Re_{s_1} \times ... \times D\Re_{s_n} \rightarrow D\Re_s$ *to each n-ary function symbol* $f \in \mathcal{F}$*, where* $(s_1, ..., s_n, s)$ *is the signature of* f*;*
  3. *an assignment of a function* $D\Re_{s_1} \times ... \times D\Re_{s_n} \rightarrow \{true, false\}$ *to each n-ary predicate symbol* $p \in \pi_c$ *(apart from* $=$ *which is naturally interpreted as syntactic equality), where* $(s_1, ..., s_n)$ *is the signature of* p*.*
- Var *is an* S*-sorted set of variables,*
- Φ *is a set of admissible constraints, i.e. a non-empty subset of* (Σ, Var)*-formulas, closed under conjunction. The empty constraint is denoted by* true*.*

Thus we have interpreted predicates with fixed interpretation in the constraint system $\Re$, and uninterpreted ones whose meaning is implicitely defined in the program. An atomic constraint is then an atomic wff, whose predicate is an interpreted one. In the following we will denote any conjunction of atomic constraints in a set-theoretic form.

We will extend this traditional constraint definition, by allowing a new kind of constraint whose predicate symbols are uninterpreted ones. They correspond to a semidecidable test.

The previous definition of constraint system is very similar to the one used in Concurrent Constraint logic programming [14]. As usual, an $\mathcal{A}$-valuation is a mapping from Var to

the structure $\mathcal{A}$. In logic programming a widely used constraint system is the Herbrand system which interprets the vocabulary Σ on the free Σ-algebra of terms. In such a case the admissible constraints are possibly existentially quantified conjunctions of equations. We will interpret our constraint system in such a structure, by extending the range of admissible constraints in order to include the notion of entailment of disequations. A constraint will be interpreted as the possibly infinite set of valuations which verifies the constraint.

In the following section we introduce the basic constraint framework to describe an instance of CLP denoted by τ.

3 The Basic Framework

3.1 The Language of ask and tell constraints

The τ-language is a language in the class of Constraint logic languages where the domain is the Herbrand universe over a finite alphabet Σ and the constraints are first order formulas like equations ($s = t$) or disequations ($s \neq t$). In the following the symbol $\tilde{t}$ will often denote a finite sequence of symbols $t_1, ..., t_n$.

In the τ-language the set of admissible constraints can be reduced to three kind of constraints, local goals, Ask and Tell constraints. The first two define a general guard for the program while the third one correspond to telling a constraint to the store.

Definition 3.1 (τ**-constraints**) *A* τ*-constraint* c_τ *is a pair* $(g : C_{tell})$ *where* g *is a finite set of ask constraints* C_{ask} *and local goals* C_{lg}*, and* C_{tell} *is a finite set of tell constraints. Any empty set of constraints will be denoted by true.*

In the following we consider the meaning of a constraint with respect to the current store (which is a global set of constraints).

Definition 3.2 (Tell constraints) *A tell constraint is a finite set of equations of the form* $s = t$ *where* $var(s) \cap var(t) = \emptyset$*.*

An *ask constraint* is defined by using the predicate simbols $\neq$ and $=$, which are naturally interpreted in the $\Re$ constraint system. It is a universally quantified literal having the inequality as predicate or an equation of the form $s = t$ where s and t are terms defined in the signature.

Definition 3.3 (Ask constraints) *An ask constraint has the following form*

- $\forall X_1, \ldots, X_n . s \neq t$ *with* $\{X_1, \ldots, X_n\} = var(t)$ *and* $var(s) \cap var(t) = \emptyset$*.*
- $s = t$ *where* $var(s) \cap var(t) = \emptyset$*.*

As in the cc languages [14], the meaning of an ask constraint is based on the notion of entailment.

Definition 3.4 (Local goal) *A local goal is an object of the form* $\lfloor c_\tau \Box \tilde{B} \rfloor$ *or true, where* c_τ *is a* τ*-constraint with an empty local goal and empty ask constraint, and* $\tilde{B}$ *is a finite (possibly empty) set of atoms defined in* π_P*.*

A local goal is a new kind of constraint. It is defined on uninterpreted atoms and corrisponds to a test which does not change the global set of constraints σ.

Having defined our constraint structure, let us give the definition of a τ-program.

Definition 3.5 (program P^τ) *A program P^τ is defined over the sorted alphabets π_P and Σ, with ask, tell and local goal constraints. It is a set of clauses of the following form*

$$H : -g : C_{tell}\Box. \qquad or \qquad H : -g : C_{tell}\Box B_1, \ldots, B_n.$$

where H (the head) and $B_1, \ldots, B_n$ (the body) are π_P atoms, g is a finite set of ask constraints and local goals, and C_{tell} is a finite set of tell constraints. A goal is a program clause with an empty ask constraint, no head and a non-empty body.

Evaluation of a tell constraint.
Let $\Re$ be the structure and σ be the global store. A set of tell constraints C_{tell} is
- satisfied iff $\Re \models \exists(\sigma \wedge C_{tell})$, (i.e. the conjunction of the store and C_{tell} is consistent),
- failed iff $\Re \models \exists(\sigma \Rightarrow \neg C_{tell})$.

When the tell constraint is satisfied, the store is augmented with C_{tell} and the new store becomes $\sigma' = \sigma \wedge C_{tell}$. If the evaluation fails, the store is unchanged. In the following we will denote by [c] the set $\{c\theta \mid \theta \; is \; a \; \Re - solution \; of \; c\}$.

As in the cc languages [14], the meaning of an ask constraint is based on the notion of entailment.

Evaluation of an ask constraint.
Let $\Re$ be the interpretation structure and σ be the global store. A set of ask constraints C_{ask} is
- satisfied iff C_{ask} is entailed by (is a logical conseguence of) σ, $\Re \models \exists(\sigma \Rightarrow C_{ask})$,
- failed iff $\Re \not\models \exists(\sigma \Rightarrow C_{ask})$.

The evaluation does not change the state of the global store. As an example the (Σ, Var)-formula $\forall X.Y \neq f(X)$ is an ask constraint. It is important to note that if the ask constraint is not entailed by the current store but it is consistent with it, the ask constraint fails. In the concurrent constraint framework instead, this condition leads to a suspension, (and the ask constraint can later either succeed or fail [14]).

Evaluation of a local goal.
A local goal $\lfloor c_\tau \Box B \rfloor$ in a global store σ is
- satisfied iff the goal $c_\tau \Box B$ is satisfiable in the τ-program (see below for the definition), i.e. if the computation succeeds starting from σ,
- failed iff the evaluation of the goal fails.

The evaluation of a local goal is an action which does not affect the global store. In case of success, the final constraints c_{ans} (the binding for the variables in the goal) are not added to the store σ (i.e. the binding for the variables which are in the goal are not exported from the local to the external enviroment).

We define an empty local goal $\Box_{lg}$ as follows

- *true* is an empty local goal,
- $true : C_{tell}\Box$ is an empty local goal.

Note that an empty local goal does not contain any atom.

Evaluation of a τ constraint $(\{C_{lg}, C_{ask}\} : C_{tell})$.
The evaluation of a τ-constraint $(\{C_{lg}, C_{ask}\} : C_{tell})$ in a global store σ,
- succeeds iff each local goal $c_{lg} \in C_{lg}$ are satisfied in P^τ, $\Re \models \exists(\sigma \Rightarrow C_{ask})$ and $\Re \models \exists(\sigma \wedge C_{tell})$;
- fails iff there exists a local goal $c_{lg} \in C_{lg}$ which fails or $\Re \not\models \exists(\sigma \Rightarrow C_{ask})$ or $\Re \models \exists(\sigma \Rightarrow \neg C_{tell})$ (i.e. the evaluation of one of the constraints fails).

When the evaluation of $(\{C_{lg}, C_{ask}\} : C_{tell})$ succeeds, the new store is changed from σ to σ', with $\sigma' = \sigma \wedge C_{tell}$.

3.2 Operational and fixpoint Semantics

In the following we define the operational model for a τ-program. Let R be a computation rule, i.e. a function defined from a set of goals to a set of atoms, which, given a goal, returns the selected atom. We denote by $\longrightarrow_*$ any indefinite application of the operations (derivation path), defined below.

Definition 3.6 ($(P^\tau,\Re)$-derivation step) *Let P^τ be a τ-program and R a computation rule. A $(P^\tau, \Re)$-derivation step for a goal G*

$$\leftarrow true : C_{tell}\Box B_1, ..., B_j, .., B_n$$

with the selected atom B_j, results in a goal of the form

$$\leftarrow true : \overline{C}_{tell}\Box B_1, ..., B_{j-1}, \tilde{A}, B_{j+1}, ..., B_n$$

and is denoted by

$$(true : C_{tell}\Box B_1, ..., B_n) \overset{(P^\tau,\Re)}{\rightarrow} (true : \overline{C}_{tell}\Box B_1, ..., B_{j-1}, \tilde{A}, B_{j+1}, ..., B_n)$$

if there exists a variant of a clause in P^τ, $H \leftarrow \{C_{lg'}, C_{ask'}\} : C_{tell'}\Box\tilde{A}$, with no variables in common with the goal G, such that

$$(c_{lg'}) \overset{(P^\tau,\Re)}{\longrightarrow}_* (\Box_{lg}), \; for \; each \; local \; goal \; c_{lg'} \in C_{lg'}$$

where $\Box_{lg}$ is an empty local goal,

$$[C_{ask'}] \subseteq [C_{tell}, B_j = H]$$

and

$$\overline{C}_{tell} = \{C_{tell}, C_{tell'}, B_j = H\} \; is \; \Re - solvable.$$

A $(P^\tau, \Re)$ – *derivation* of a goal G is a finite or infinite sequence of goals such that every goal, apart from G, is obtained from the previous one by means of a single $(P^\tau, \Re)$ – *derivation* step.

A *successful* $(P^\tau, \Re)$ – *derivation* of a goal G is a finite sequence whose last element is a goal of the form $(true : c_{ans}\Box.)$ where c_{ans} is the answer constraint of the derivation. For semplicity, we will consider the left-to-right selection rule of Prolog.

In order to give a fixpoint and model-theoretic characterization of the operational semantics of τ-programs, we have to consider more complex semantic domain structures. In particular we need to keep track of the sequence of the set of τ-constraints involved in the refutation of a given goal. The semantic domains turn out to be very similar to those developed for denotational characterizations. For example, the semantics in [5] incorporates the sequential evaluation strategy used by standard Prolog evaluators. The meaning of a predicate is then a function from substitutions to a possibly infinite sequence of subststitutions. In this framework, the meaning of a sequence of literals $\{L, S\}$ is a stream of subtitutions obtained by first solving L with the input stream of substitutions and then piping the resulting stream to S. Analogously sequences capture the ordering of clauses. The answers obtained by using the first clause are sequentialized before those obtained by the remaining clauses. We can then argue that the notion of sequence is the main technical issue, which allows to define suitable domains for the denotational semantics of Prolog.

The notion of *sequence*, coming from the theory of concurrent logic languages [9,14], will be instantiated in order to find suitable denotations for τ-program behaviors. Since every test of a local goal constraint corresponds to a refutation of such a goal in the program (without changing the global set of constraints), we inductively define the notion of constrained atoms [9] as follows.

Definition 3.7 (constraint form) *A constraint form is a pair* $< \varphi_g : C_{tell} >$ *where* φ_g *(solved guard form) is a finite set of ask constraints, constraint forms, and* C_{tell} *is a finite set of tell constraints.*

As usual, we denote by *true* any empty solved guard form. Moreover, in the following we will denote by $\varphi_{ask} \subseteq \varphi_g$ and $\varphi_{lg} \subseteq \varphi_g$, such that $\varphi_{lg} \cap \varphi_{ask} = \emptyset$ and $\varphi_{lg} \cup \varphi_{ask} = \varphi_g$, the finite set of ask constraints and the finite set of constraint forms in φ_g respectively.

Thus each constraint form $< \varphi_g : C_{tell} >$ may be represented by a tree structure, where each node corresponds to a tell-constraint C_{tell} together with the corresponding set of ask constraints in φ_g and where each constraint form in φ_g, is a successor of the node itself.

Definition 3.8 (consistency) *A constraint form* η *is consistent w.r.t. a store* σ *iff*
- $\eta =< true : true >$ *or*
- *if* $\eta = < \varphi_g : C_{tell} >$ *then*

1. *each* $\eta_{lg} \in \varphi_g$ *is a consistent constraint form w.r.t.* σ,
2. *let* $\varphi_{ask} \subseteq \varphi_g$ *be the set of ask constraints in each solved guard form, then*
   $\Re \models (\exists)(\varphi_{ask} \wedge C_{tell})$
   $\Re \models (\exists)(\sigma \wedge \varphi_{ask} \wedge C_{tell})$,
   $\Re \models (\sigma \Rightarrow \varphi_{ask})$,

Definition 3.9 (merging constraint forms) *Given two constraint forms* $< \varphi_g : C_{tell} >$ *and* $< \varphi_{g'} : C_{tell'} >$, *the merging process*

$$Merge(< \varphi_g : C_{tell} >, < \varphi_{g'} : C_{tell'} >)$$

returns a constraint form η *such that,*

$$\eta =< \varphi_g \cup \nabla_{ask}(C_{tell}, \varphi_{ask'}) \cup \nabla_{lg}(< \varphi_g : C_{tell} >, \varphi_{lg'}) : C_{tell} \cup C_{tell'} >,$$

where

- $\nabla_{ask}(C_{tell}, \varphi_{ask'}) = \emptyset$ *if* $\Re \models \exists(C_{tell} \Rightarrow \varphi_{ask'})$
  $\nabla_{ask}(C_{tell}, \varphi_{ask'}) = \varphi_{ask'} \cup Closure(C_{tell}, Var(\varphi_{ask'}))$.
- $Closure(C_{tell}, Var(\varphi_{ask'})) = \emptyset$ *if* $Var(\varphi_{ask'}) \cap Var(C_{tell}) = \emptyset$
  $Closure(C_{tell}, Var(\varphi_{ask'})) = C_{tell_{/Var(\varphi_{ask'})}} \cup Closure(C_{tell}, Var(C_{tell_{/Var(\varphi_{ask'})}}))$
  where $C_{tell_{/Var(\varphi_{ask'})}} = \{c_{tell} \in C_{tell} | Var(c_{tell}) \cap Var(\varphi_{ask'}) \neq \emptyset\}$.
- $\nabla_{lg}(< \varphi_g : C_{tell} >, \varphi_{lg'}) = \{< \varphi_g''' : C_{tell}''' > \mid \exists < \varphi_g'' : C_{tell}'' > \in \varphi_{lg'} \wedge$
  $Merge(< \varphi_g : C_{tell} >, < \varphi_g'' : C_{tell}'' >) =< \varphi_g''' : C_{tell}''' >\}$.

Lemma 3.1 *Let* $< \varphi_{g_1}, C_{tell_1} >$ *and* $< \varphi_{g_2}, C_{tell_2} >$ *be two constraint forms. If* $Merge(< \varphi_{g_1}, C_{tell_1} >, < \varphi_{g_2}, C_{tell_2} >) =< \{\overline{\varphi_{lg}}, \overline{\varphi_{ask}}\} : \overline{C_{tell}} >$, *then*

- $\{\sigma \mid \eta_{lg} \in \varphi_{lg_1}$ *is conststent w.r.t.*$\sigma\} \cap \{\sigma \mid \eta_{lg} \in \varphi_{lg_2}$ *is conststent w.r.t.*$\sigma\} = \{\sigma \mid \eta_{lg} \in \overline{\varphi_{lg}}$ *is conststent w.r.t.*$\sigma\}$
- $\{\sigma \mid \Re \models \sigma \Rightarrow \varphi_{ask_1}\} \cap \{\sigma \mid \Re \models \sigma \wedge C_{tell_1} \Rightarrow \varphi_{ask_2}\} = \{\sigma \mid \Re \models \sigma \Rightarrow \overline{\varphi_{ask}}\}$
- $[C_{tell_1} \wedge C_{tell_2}] = [\overline{C_{tell}}]$

Corollary 3.2 *Let* $< \varphi_{g_1}, C_{tell_1} >$ *and* $< \varphi_{g_2}, C_{tell_2} >$ *be two constraint forms. Then* $Merge(< \varphi_{g_1}, C_{tell_1} >, < \varphi_{g_2}, C_{tell_2} >)$ *is a consistent constraint form w.r.t. a store* σ *iff by defining* $C_{tell_0} = true$,

1. *each* $\eta_{lg} \in \varphi_{lg_i}$ *is a consistent constraint form w.r.t.* $\sigma \wedge C_{tell_{i-1}}, \forall i = 1, 2$,
2. $\Re \models (\exists)(\varphi_{ask_1} \wedge C_{tell_1} \wedge \varphi_{ask_2} \wedge C_{tell_2})$
   $\Re \models (\exists)(\sigma \wedge \varphi_{ask_1} \wedge C_{tell_1} \wedge \varphi_{ask_2} \wedge C_{tell_2})$,
   $\Re \models (\sigma \wedge C_{tell_{i-1}} \Rightarrow \varphi_{ask_i})$, $\forall i = 1, 2$.

Definition 3.10 (constrained atoms) *A constrained atom is a constrained atom of the form* $\eta\Box p(\tilde{X})$, *where* η *is a constraint form and* $p(\tilde{X})$ *is a* π_P *atom. A constrained atom* $\eta\Box p(\tilde{X})$ *is consistent iff there exists a set of constraints* σ *on* $\tilde{X}$ *such that* η *is a consistent constraint form w.r.t.* σ.

Definition 3.11 (pre-order on constrained atoms) *Let* $\mathcal{A}$ *be the set of consistent constrained atoms* $\eta\Box p(\tilde{X})$ *for a program* P^τ *w.r.t a store* σ *defined on* $\tilde{X}$. *Given* $\eta_1\Box p(\tilde{X}_1)$ *and* $\eta_2\Box p(\tilde{X}_2)$ *two constrained atoms in* $\mathcal{A}$, *we define* $\eta_1\Box p(\tilde{X}_1) \preceq \eta_2\Box p(\tilde{X}_2)$ *iff there exists a variable renaming* δ *such that* $\eta_1 \sqsubseteq \eta_2\delta$, *where the relation* $\sqsubseteq$ *on constrains is defined as follows*

- $\forall \eta, < true : true > \sqsubseteq \eta$,
- $\forall \eta, \eta \sqsubseteq \eta$,
- *let* $\eta_1 =< \varphi^1_g : C^1_{tell} >$ *and* $\eta_2 =< \varphi^2_g : C^2_{tell} >$ *be two constraint forms. Then* $\eta_1 \sqsubseteq \eta_2$ *iff,*
  - $\varphi^1_{lg} = \varphi^2_{lg}$,
  - $\{\sigma \mid \Re \models \sigma \Rightarrow \varphi^1_{ask}\} \subseteq \{\sigma \mid \Re \models \sigma \Rightarrow \varphi^2_{ask}\}$
  - $[C^1_{tell}] \subseteq [C^2_{tell}]$.

Definition 3.12 (B^τ_p Base) *Let P^τ be a τ-program and let $\mathcal{A}$ be the set of consistent constrained atoms for P^τ. The Base of interpretations B^τ_p is the set $\mathcal{A}$ with the equivalence relation $\equiv$ induced by $\preceq$.*

Definition 3.13 (Interpretation) *An interpretation I is any subset of B^τ_p.*

An interpretation for a program P^τconsists in a set of consistent constrained atoms $\eta \Box p(\tilde{X})$ where η is a finite sequence path and $p(\tilde{X})$ is a predicate in the program.

Obviously, the sequentialization process strongly depends on the selection rule. We have no restriction on such a process but, in order to give a semantic characterization of P^τ programs, we will consider, for simplicity, a *Prolog* like selection rule.

Definition 3.14 (reduction to a solved guard form) *The reduction of a guard g to a solved guard form, in an interpretation I, w.r.t. a store σ,*

$$\mathcal{R}_{\mathcal{I},\sigma}(g) = \varphi_g$$

is defined as follow

- *if* $g = \{true\}$ *then* $\mathcal{R}_{\mathcal{I},\sigma}(g) = \{true\}$,
- *if* $g = \{c_{ask}\}$, *then* $\mathcal{R}_{\mathcal{I},\sigma}(g) = \{c_{ask}\}$,
- *if* $g = \{true : C_{tell} \Box B_1, ..., B_n\}$ *then*

  $$\mathcal{R}_{\mathcal{I},\sigma}(g) = \{\eta^n\}$$

  iff $\exists\ \eta'_1 \Box B'_1, ..., \eta'_n \Box B'_n \in I$, *where* $\eta'_i =< \varphi'_{g_i} : C'_{tell_i} >$,
  $\eta_i =< \varphi'_{g_i} : C'_{tell_i} \wedge B_{i+1} = B'_{i+1} >$, *if* $1 \leq i \leq n-1$,
  $\eta_n = \eta'_n$,
  $\eta^1 = Merge(< true : C_{tell} \wedge B_1 = B'_1 >, \eta_1)$,
  $\eta^n = Merge(\eta^{n-1}, \eta_n)$, *and such that the resulting solved guard form is consistent w.r.t. the store σ.*

- *if* $g = \{g_1, ..., g_m\}$ *then* $\mathcal{R}_{\mathcal{I},\sigma}(g) = \{\mathcal{R}_{\mathcal{I},\sigma}(g_1), ..., \mathcal{R}_{\mathcal{I},\sigma}(g_m)\}$

Thus, each $\eta_{lg} \in \varphi_g$ in a solved guard form, represents the sequence of constraints which has to be tested during the refutation of each local goal. The satisfaction of this kind of constraint, by a current store σ, corresponds infact to a refutation of the corresponding program goal $(true : C_{tell} \Box B_1, ..., B_n)$ in P^τ , starting with the store σ.

Definition 3.15 (truth) *Let I be an interpretation.*

- *A constrained atom $\eta \Box p(X)$ is true in I iff there exists $\eta' \Box p(X) \in I$ such that $\eta \Box p(X) \equiv \eta' \Box p(X)$*
- *A clause $H : -g : C_{tell} \Box B_1, ..., B_n$ is true in I iff there exists a set of constraints σ on the variables of H such that $\eta'_1 \Box B'_1, ..., \eta'_n \Box B'_n \in I$ and the constraint form η^n such that*
  $\eta^1 = Merge(< \mathcal{R}_{\mathcal{I},\sigma}(g) : C_{tell} \wedge B_1 = B'_1 >, \eta_1)$
  $\eta^n = Merge(\eta^{n-1}, \eta_n)$
  and where $\eta'_i =< \varphi'_{g_i} : C'_{tell_i} >$, *and*
  $\eta_i =< \varphi'_{g_i} : C'_{tell_i} \wedge B_{i+1} = B'_{i+1} >$, *if* $1 \leq i \leq n-1$
  $\eta_n = \eta'_n$,
  is consistent, implies that there exists $\eta' \Box H \in I$ such that
  $\eta^n \Box H \equiv \eta' \Box H$.

A model for a τ-program P^τ is any interpretation in which all the clauses of P^τ are true. Thus, the semantics for a p predicate in P^τ is obtained by the composition of the constraints coming from the body of any clause defining p.

In the following we give a characterization of the semantics in terms of a continuous immediate τ-consequence operator T^τ_P.

Definition 3.16 *Let P^τ be a τ-program. The mapping T^τ_P on the set of interpretations B^τ_p is defined as follows*

$$\begin{aligned} T^\tau_P(I) \;=\; \{\ & [\eta^n \Box p(X_1, ...X_n)]_\equiv \in B^\tau_p \mid \\ & \exists p(X_1, ...X_n) : -g : C_{tell} \Box B_1, .., B_n \in P^\tau, \\ & \exists\ \eta'_1 \Box B'_1, ..., \eta'_n \Box B'_n \in I, \\ & \exists\ \sigma, \textit{ set of constraints on the variables } \{X_1, ..., X_n\} \textit{ such that} \\ & \eta^1 = Merge(< \mathcal{R}_{\mathcal{I},\sigma}(g) : C_{tell} \wedge B_1 = B'_1 >, \eta_1) \\ & \eta^n = Merge(\eta^{n-1}, \eta_n), \\ & where \\ & \quad \eta'_i =< \varphi'_{g_{i_1}} : C'_{tell_{i_1}} > \textit{ and} \\ & \quad \eta_i =< \varphi'_{g_{i_1}} : C'_{tell_{i_1}} \wedge B_{i+1} = B'_{i+1} >, \textit{ if } 1 \leq i \leq n-1 \\ & \quad \eta_n = \eta'_n; \\ & \textit{is a consistent constrained form w.r.t. } \sigma\}. \end{aligned}$$

The immediate consequence operator T^τ_P is monotonic and continuous on the complete lattice of interpretations (under the set inclusion ordering relation).

Theorem 3.3 *T^τ_P is continuous.*

Theorem 3.4 *An interpretation I is a model for a program P^τ iff $T^\tau_P(I) \subseteq I$.*

Hence, there exists the least fixpoint of T^τ_P

$$lfp(T^\tau_P) = lub_{n \in \omega} T^{\tau\,n}_P(\emptyset) = T^\tau_P \uparrow \omega$$

Definition 3.17 (fixpoint semantics) *The fixpoint semantics of a τ-program P^τ, is the least fixpoint of the transformation T_P^τ associated with P^τ.*

Theorem 3.5 (equivalence of fixpoint and operational semantics) *Let P^τ be a τ-program. The goal $G = : -true : C_{tell} \Box B_1, ..., B_n$ succeeds in P^τ, with an answer constraint c_{ans}, iff*
$\exists\ \eta_1' \Box B_1', ..., \eta_n' \Box B_n' \in lfp(T_P^\tau)$, where $\eta_i' =< \varphi_{g_i}' : C_{tell_i}' >$, and by defining
$\eta_i =< \varphi_{g_i}' : C_{tell_i}' \wedge B_{i+1} = B_{i+1}' >$, if $1 \leq i \leq n-1$
$\eta_n = \eta_n'$,
the constraint form $\eta^n =< \overline{\varphi}_g : \overline{C}_{tell} >$ such that

$$\eta^1 = Merge(< true : C_{tell} \wedge B_1 = B_1' >, \eta_1),$$
$$\eta^n = Merge(\eta^{n-1}, \eta_n)$$

is consistent w.r.t. the empty store $\sigma = \emptyset$, and $[\overline{C}_{tell}] = [c_{ans}]$.

4 Modelling Prolog Control

We will now show how a Prolog program P (in a constraint form), can be transformed in a corrisponding τ- program P^τ. The trasformed P^τ program captures some operational behaviors of the source program P, like the ordering of clauses and the use of a sequential depht-first search strategy. The transformation captures an approximation of the termination of the source program. Each clause of the program P is translated into a set of clauses in the τ-language. Each clause of the τ-program specifies by means of suitable set of ask, tell and local goal constraints, under which conditions the corresponding clause in P is called during the computation. Moreover, the program P^τcontains a set of clauses, one for each clause in P which approssimate when those clauses in P lead to finite derivations (either success or failure).

Example 4.1 *Let P_a be the following source Prolog program in a constraint form*

$p(Y) : -Y = s(X) \Box p(X).$ $\quad c1$
$p(Y) : -Y = 0 \Box.$ $\quad c2$
$q(Y) : -Y = 0 \Box.$ $\quad c3$

The following set of clauses is the result of the compilation of the Prolog computation rule into a τ-program P^τ

$p(Y) : -true : Y = s(X) \Box p(X).$
$p(Y) : -\{\forall X.Y \neq s(X)\} : Y = 0 \Box.$
$p(Y) : -\{\lfloor true : Y = s(X) \Box p\_t_1(s(X)) \rfloor\} : Y = 0 \Box.$
$p\_t_1(Y) : -\{\lfloor true : Y = s(X) \Box p\_t(X) \rfloor\} : true \Box.$
$p\_t_1(Y) : -\{Y \neq s(X)\} : true \Box.$
$p\_t_2(Y) : -true : Y = 0 \Box.$
$p\_t_2(Y) : -\{Y \neq 0\} : true \Box.$
$p\_t(Y) : -\{\lfloor true : true \Box p\_t_1(Y) \rfloor, \lfloor true : true \Box p\_t_2(Y) \rfloor\} : true \Box.$
$p\_t(Y) : -\{\forall X.Y \neq s(X), Y \neq 0\} : true \Box.$
$q(Y) : -true : Y = 0 \Box.$
$q\_t(Y) : -\{\lfloor true : true \Box q(Y) \rfloor\} : true \Box.$
$q\_t(Y) : -\{Y \neq 0\} : true \Box.$

The first clause in P^τcorresponds to the application of the clause $c1$ in P_a. The second clause in P^τ corresponds to the application of clause $c2$ in P_a when clause $c1$ is not applicable (the ask constraint specifies this condition). The third clause in P^τ corresponds to the applications of clause $c2$ when clause $c1$ is applicable and terminates (this condition is represented by the local goal). Terminations of a clause application means that the clause either finitely fails or generates a finite number of answers. The clauses defining the predicate p_t_1 (p_t_2) specify the termination condition for the application of clause c1 (c2). The predicate p_t defines the termination conditions for any call to predicate p. In particular the least clause for p_t characterizes the case where no clauses for p are applicable. Similar considerations apply to the clauses for q.

When a Prolog program is transformed in the τ- language, each tuple of terms has to be considered and handled as a single term, by transforming it using a tupling functor. Therefore programs in the constraint representation are composed by monadic predicates only and can be transformed into τ-programs as regular monadic programs.

Let P be a Prolog program. We denote by P^{con} the corresponding monadic constraint Prolog program. The program P^{con} is obtained

1. by replacing each n-adic predicate $p(t_1, ..., t_n)$ $\quad, n > 1$, in P with the monadic predicate $p(t)$ where $t =< t_1, .., t_n >$;
2. by replacing each clause $p(t) : -B_1, ..., B_r.$ $\quad, r > 0$, in P with the clause $p(Y) : -Y = t \Box B_1, ..., B_r.$ $\quad$ where $Y \notin \{var(t), var(B_1), ..., var(B_r)\}$.

Example 4.2 *The Prolog program*

$p(0, s(X)) : -p(0, X).$
$p(s(X), 0).$

will be described by the following equivalent monadic program in the constraint language

$p(Y) : -Y =< 0, s(X) > \Box p(0, s(X)).$
$p(Y) : -Y =< s(X), 0 > \Box.$

We give now the general transformation rule. The resulting τ-program is obtained by translating the Prolog computation rule (ordering of clauses and left to right selection rule) in a set of conditions which can be suitable expressed in our constraint framework. Such a transformation can always be applied to any positive Prolog program.

Definition 4.1 (the transformation rule from P^{con} to P^τ) *Let P^{con} be a Prolog program described by means of a constraint language with monadic predicates. Let pred(P^{con}) denote the set of predicate symbols. For each predicate $p \in pred(P^{con})$ let $\mathcal{D}_p^k = \{\pi_1, ..., \pi_k\}_p$ be the ordered set of clauses defining the predicate p in P^{con}. Moreover given a clause $\pi_i \in P^{con}$, where*

$$\pi_i = \ p(Y) : -Y = t_i \Box q_1(t_1), ..., q_r(t_r)$$

let
$\Gamma_{lg_i} = \{\lfloor true : Y = t_j \Box p\_t_j(t_j) \rfloor \mid j < i\}$

$\Gamma_{ask_i} = \{\forall \tilde{X}_j.Y \neq t_j \mid j < i,\ \tilde{X}_j = var(t_j)\}$

$\Gamma_{tell_i} = \{Y = t_i\}$

The τ-program P^τ is obtained by the following transformation rule

1. $\forall \pi_i \in \{\pi_1, ..., \pi_k\}_p$ *in* P^{con}
   - $\forall < C_{lg}, C_{ask}, C_{tell} > \in 2^{\Gamma_{lg_i}} \oplus_i 2^{\Gamma_{ask_i}} \times \Gamma_{tell_i}$ *generate a clause in* P^τ *of the form*

     $p(Y) : -g : C_{tell} \Box q_1(t_1), ..., q_r(t_r).$

     where
     $C_{tell} = \Gamma_{tell_i}$
     $g = true \qquad if\ i = 1$
     $g = 2^{\Gamma_{lg_i}} \oplus_i 2^{\Gamma_{ask_i}} \quad if\ i > 1$
     with
     $2^{\Gamma_{lg_i}} \oplus_i 2^{\Gamma_{ask_i}} = \{< \{lg_j \mid lg_j \in \gamma_{lg} \ \wedge\ \gamma_{lg} \in 2^{\Gamma_{lg_i}}\}, \{ask_k \mid ask_k \in \gamma_{ask} \ \wedge\ \gamma_{ask} \in 2^{\Gamma_{ask_i}}\} >$ *such that* $j \neq k \ \wedge\ \mid \gamma_1 \cup \gamma_2 \mid = i - 1\}$

     $\oplus_i$ *is the disjoint union of two sets with respect to the indexes of the elements of its argument and generates sets of cardinality i-1.*

   - *if* $k > 1$ *then generate*
     $p\_t_i(Y) : -\{\lfloor true : \Gamma_{tell_i} \Box q_1\_t(t_1), ..., q_r\_t(t_r) \rfloor\} : true\Box.$
     $p\_t_i(Y) : -\{\forall \tilde{X}_i.Y \neq t_i\} : true\Box. \qquad \tilde{X}_i = var(t_i)$

2. $\forall p \in pred(P^{con})$, *generate the following set of clauses*
   if $(k = 1 \wedge r = 0)$ *then*
   $p\_t(Y) : -\{\lfloor true : true \Box p(Y) \rfloor\} : true\Box.$
   if $(k = 1 \wedge r > 0)$ *then*
   $p\_t(Y) : -\{\lfloor true : true \Box q_1\_t(t_1), ..., q_r\_t(t_r) \rfloor\} : true\Box.$
   if $k > 1$ *then*
   $p\_t(Y) : -\{\{\lfloor true : true \Box p\_t_i(Y) \rfloor \mid i = 1, ..., k\}\} : true\Box.$
   $p\_t(Y) : -\{\{\forall \tilde{X}_i.Y \neq t_i \mid \pi_i \in \mathcal{D}_p^k\}\} : true\Box.$

3. *If* π_i *is a Prolog goal,* $c \Box G_1, ..., G_n$ *then generate a* τ*-goal*

   $true : C_{tell} \Box G_1, ..., G_n$ *where* $C_{tell} = c.$

Example 4.3 *P_b is obtained from the program of example 4.1, by permuting the first two clauses.*

$p(Y) : -Y = 0\Box.$
$p(Y) : -Y = s(X)\Box p(X).$
$q(Y) : -Y = 0\Box.$

The correspondent τ-program P^τ is

$p(Y) : -true : Y = 0\Box.$
$p(Y) : -\{Y \neq 0\} : Y = s(X)\Box p(X).$
$p(Y) : -\{\lfloor true : Y = 0 \Box p\_t_1(0) \rfloor\} : Y = s(X)\Box p(X).$
$p\_t_1(Y) : -true : Y = 0\Box.$
$p\_t_1(Y) : -\{Y \neq 0\} : true\Box.$
$p\_t_2(Y) : -\{\lfloor true : Y = s(X)\Box p\_t(X) \rfloor\} : true\Box.$
$p\_t_2(Y) : -\{Y \neq s(X)\} : true\Box.$
$p\_t(Y) : -\{\lfloor true : true \Box p\_t_1(Y) \rfloor, \lfloor true : true \Box p\_t_2(Y) \rfloor\} : true\Box.$
$p\_t(Y) : -\{Y \neq 0, \forall X.Y \neq s(X)\} : true\Box.$
$q(Y) : -true : Y = 0\Box.$
$q\_t(Y) : -\{\lfloor true : true \Box q(Y) \rfloor\} : true\Box.$
$q\_(Y) : -\{Y \neq 0\} : true\Box.$

Let us consider the two programs P_a and P_b in example 4.1 and 4.3. respectively. The two programs have the same semantics as pure logic programs, however, with the Prolog search rule they have different behaviors. As an example, the goals ?-p(X). and ?-p(s(X)). do not terminate in P_a, while the goals ?-p(0). and ?-p(s(0)). return "yes".
We can note that the Lifting lemma does not hold any more. Moreover in P_a the goal ?-p(X),q(X). does not terminate with the usual Prolog selection rule, while it terminates successfully computing the answer X=0 with the rightmost selection rule. Thus, for Prolog programs, the indipendence from the selection rule does not hold too.

By considering the P_b program instead, all the goals ?-p(X)., ?-p(s(X))., ?-p(0). and ?-p(s(0)). succeed with the Prolog search rule. The problem is related to the different order of the clauses into the two programs.

By "excuting" a goal in the model we can capture the two different behavio rs of P_a and P_b. Each model associated with a τ-program is in fact a set of sequenced atoms $\eta \Box p(X)$ which represents, by means of sequences of constraints, the computational behaviors goals.

The two models M_a^τ and M_b^τ for the τ programs P_a and P_b are

$M_a^\tau = \{\ \forall(X).Y \neq s(X) : Y = 0 \Box p(Y)$
$\forall(X).Y \neq s(X), Y \neq 0 : true \Box p(Y)$
$< \forall(Z).X \neq s(Z), Y = s(X) : Y = s(X), X = 0 > \Box p(Y)$
$....$
$true : Y = 0 \Box q(Y)\ \}$

$M_b^\tau = \{\ true : Y = 0 \Box p(Y)$
$Y \neq 0, \forall(X).Y \neq s(X) : true \Box p(Y)$
$< Y \neq 0 : Y = s(X), X = 0 > \Box p(Y)$
$....$
$true : Y = 0 \Box q(Y)\}$

Let us consider the following set of representative Prolog goals

$? - Y = 0 \Box p(Y).$

which returns "yes" in both the models. The ask and the tell constraints in the first atom in the models, are respectively entailed by and consistent with the store Y=0. The goal

$? - Y = s(0) \Box p(Y).$

returns "yes" in both the models, while the goal

$$? - true \Box p(Y).$$

does not succeed in M_a^τ because none of the ask constraints can be satisfied. On the contrary, it succeeds in M_b^τ, since the ask constraint in the first atom in the model is entailed by *true* and the tell constraint is consistent with the store.

Finally, the goal

$$? - true \Box p(s(Y)).$$

does not succeed in M_a^τ while it succeeds in M_b^τ.

5 Conclusion

In this paper we have shown that the Prolog control behavior can be modelled by a constraint language. By means of a program transformation rule we obtain a constraint logic program associated with each Prolog program. Thus, the model theoretic and fixpoint semantics of Prolog can be obtained into a different linguistic framework. This transformational approach can be extended to give a fixpoint characterization of Prolog meta-level predicates too, like *assert* and *retract* or of negation.

This linguistic framework is useful for semantics based analysis of Prolog programs. The abstract interpretation technique presented in [2] is based on a fixpoint characterization of the semantics. Thus we can obtain bottom-up analysis of Prolog programs by defining a suitable abstract domain of sequences and an abstract immediate consequence operator for the τ-language. The key point is the definition of sequence path consistency with respect to an abstract constraint system. As a consequence, by using a bottom-up abstract interpretation technique we can statically derive information on the real Prolog computational behavior without the need of complex operational or denotational semantics definitions of Prolog [13].

References

[1] B. Arbab and D.M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4:309–330, 1987.

[2] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. Technical Report TR 20/89, Dipartimento di Informatica, Università di Pisa, 1989. Revised version 1990.

[3] E. Börger. A logical operational semantics of full Prolog. In E. Börger, H. Kleine Büning, and M. Richter, editors, *CSL 89. 3rd workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1990.

[4] A. de Bruin and E. de Vink. Continuation semantics for Prolog with cut. In J. Diaz and F. Orejas, editors, *Proc. CAAP 89*, volume 351 of *Lecture Notes in Computer Science*, pages 178–192. Springer-Verlag, Berlin, 1989.

[5] S.K. Debray and P. Mishra. Denotational and operational semantics for Prolog. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 245–269. North-Holland, Amsterdam, 1987.

[6] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[7] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. Technical Report TR 32/89, Dipartimento di Informatica, Università di Pisa, 1989.

[8] M. Gabbrielli and G. Levi. Modeling answer constraints in Constraint Logic Programs. Technical report, Dipartimento di Informatica, Università di Pisa, 1990.

[9] M. Gabbrielli and G. Levi. Unfolding and Fixpoint Semantics of Concurrent Constraint Programs. In H. Kirchner and W. Wechler, editors, *Proc. Second Int'l Conf. on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 204–216. Springer-Verlag, Berlin, 1990.

[10] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Department of Computer Science, Monash University, June 1986.

[11] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[12] N.D. Jones and A. Mycroft. Stepwise development of operational and d enotational semantics for Prolog. In Sten-Åke Tarnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 281–288, 1984.

[13] N.D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In C. Hankin S. Abramsky, editor, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, Chichester, 1987.

[14] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. Seventeenth Annual ACM Symp. on Principles of Programming Languages*. ACM, 1990.

Andorra Prolog in FCP(:)

M. Falaschi[1,3] Y. Kesten[2] E.Shapiro[1]

[1] Department of Applied Mathematics and Computer Science
The Weizmann Institute of Science, Rehovot 76100, Israel

[2] Department of Mathematics and Computer Science
Ben Gurion University, Beer-Sheva 84120, Israel

[3] Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56100 Pisa, Italy

Abstract

In this paper, we describe an algorithm for implementing Andorra in FCP(:). Andorra is a language whose definition has been informally given by David Warren. Andorra entitles an execution of PROLOG programs which combines both AND and OR parallelism. The aim of this paper is both for a better understanding of the Andorra model of computation and for an easy, efficient and correct implementation of an initial Andorra prototype in FCP(:).

We first give a formal operational definition of Andorra in the Plotkin's structural operational style and introduce Andorra search trees to represent the Andorra search space. Then we describe the algorithm and its implementation in FCP(:).

Finally, we provide an asymptotic analysis of the parallel performance of our algorithm, in terms of unification steps. The efficiency of the interpreter depends on the FCP(:) basic mechanism of distributed atomic unification. The atomic guard unification allows a straightforward implementation of the determinacy analysis.

1 Introduction

The Andorra computation model is a parallel execution model for logic programs first suggested by D.Warren. The model exploits both And-parallelism (dependent and independent) and Or-parallelism inherent in logic programs. In Andorra, execution alternates between two phases:

1. The *And-parallel phase*, in which all deterministic atoms (i.e. atoms which can be reduced by one clause only) are evaluated concurrently. This phase terminates when either the evaluation of one of the deterministic atoms fails (failure), the goal is empty (successful termination), or the goal contains only nondeterministic atoms (start of phase 2).

2. The *Or-expansion phase*, in which one nondeterministic atom within the goal is expanded, i.e., for each unifiable clause, a new goal is created and executed in and-parallel.

The pure model does not specify which of the nondeterministic atoms is to be chosen for Or-expansion. It is worth noting though, that in the presence of and-parallelism, one should differentiate between two types of nondeterministic atoms, *stable* and *unstable* ones.

An unstable nondeterministic atom is an atom for which further instantiations can reduce the number of clauses it can reduce with. Such an atom could later become deterministic. A stable nondeterministic atom is one which is nondeterministic and not unstable. Whatever the Heuristic for choosing an atom for or-expansion, the stable nondeterministic are good candidate to be expanded first.

Another open question is how to perform the analysis to find out the deterministic atoms to be reduced in the AND parallel phase (determinacy analysis).

Thus, although the model of Andorra is very simple, there are aspects of it which need to be investigated, in a formal context.

We start this investigation by giving first a formal definition of Andorra in terms of a structural operational semantics (section 2). Then we define the Andorra search space (section 3). In section 4 we describe an interpreter of Andorra in the concurrent logic language FCP(:). In section 5 we analyse the performance of our interpreter and in section 6 we conclude and talk about future research.

There are several other works on Andorra-like machines. The basic Andorra model, as suggested by D. Warren, is currently beeing implemented by R. Yang [Yan89], for shared memory multiprocessors. This implementation (Andorra-I), is based on Aurora, an or-parallel prolog implementation, itself an extension of the WAM [L*88].

Pandora [BG89], is a nondeterministic language which extends the basic Andorra model. The language is implemented as an extension to Parlog, adding deadlock detection mechanism to Parlog machine and a nondeterministic fork primitive.

Haridi and Brand [HB88] define a language 'Andorra' which is an extension of the original Warren's definition. Our approach is different as we want to keep the language as small and easy as it is possible. While Haridi and Brand in [HB88] look for a greater generality, combining techniques which stem from concurrent logic languages as well as from prolog-like languages. See also [HJ90] for an extension of the original Andorra model based on constraints. We do not deal with possible extension of the model for non independent and-parallelism [BH91].

2 Syntax and Operational Semantic

The syntax of an Andorra program is similar to Prolog's. Edinburgh syntax is used for logical variables, terms and predicates. The usual definitions for *term, atom, clause* and *program* [Sha89] are assumed.

Definition: *candidate-clause, unit-goal, goal.*
Let P be an Andorra program, and $P_{/a}$ the set of all clauses in P with head predicate equal to the head predicate of atom a. Then:

- A clause $c \in P$ is a *candidate-clause* of an atom a if a is unifiable with c, i.e. the head of c and a are unifiable.
- A *unit-goal* is a pair $(a, \mathcal{C})$, where a is an atom and $\{c \mid c \in P_{/a}$ and c is a candidate clause for $a\} \subseteq C \subseteq \{c \mid c \in P_{/a}\}$. A unit-goal (a, C) can be:

  a. *Failing* – a unit-goal with no candidate clauses ($C = \emptyset$).
  b. *Deterministic* – a unit goal with exactly one candidate clause ($C = \{c\}$).
  c. *Nondeterministic* – a unit-goal with more than one candidate clause ($C = \{c_1, \ldots, c_n\}$ where $n > 1$).

- A *goal* G, is a sequence of unit-goals:

  $$G = (a_1, C_1), \ldots, (a_n, C_n)$$

  A goal is *failing*, or *deterministic* if one of its unit-goals is failing or deterministic respectively.

The semantic of an Andorra program is specified using a transition system, following the structural approach to the presentation of operational semantic [Plo81,Plo83].

Definition: *transition system for an Andorra program.*
With every Andorra program P, we associate a nondeterministic transition system, which consists of:

- A *set of states.*
  A state is a multiset of pairs $\{\langle G_i; \theta_i \rangle \mid i = 1, \ldots, n\}$, where G_i is either a goal or $G_i \in \{fail, true\}$, and θ_i is a substitution.
- A *set of transitions.*
  A transition (t) is a function from a state to a set of states ($S \rightarrow 2^S$).

Definition: *enabled-transition, terminal-state, computation.*

- A transition t is *enabled* on state S if $t(S)$ is non-empty.
- A *terminal state* is a state on which no transition is enabled.
  The terminal states of an Andorra computation are multisets of the form $\{\langle \alpha_i; \theta_i \rangle \mid i = 1, \ldots, n\}$ where $\alpha_i \in \{fail, true\}$ for $i = 1, \ldots, n$ and $n > 0$.
  A goal $\langle fail; \theta \rangle$ is called a *failing goal*, and $\langle true; \theta \rangle$ a *success goal.*
- A *computation* is a sequence of states $\mathbf{C} = S_1, S_2, \ldots, S_k$ satisfying:
  a. Initiation: $S_1 = \langle G; \varepsilon \rangle$, where P is an Andorra program, G is a goal of the form $(a_1, P_{/a_1}), \ldots, (a_n, P_{/a_n})$ and ε– the empty substitution.
  b. Consecution: For each k, $S_{k+1} \in t(S_k)$ for some transition t.
  c. Termination: $\mathbf{C}$ is finite and of length k if S_k is terminal.

The following constitute the set of transitions of an Andorra computation:

Transitions on unit-goals

1. Reduce$^d_{UG}$

$$\langle (a, \{c\}); \theta \rangle \xrightarrow{Reduce^d_{UG}} \langle b\theta'; \theta \circ \theta' \rangle$$

where $c = a' \leftarrow b_1, \ldots, b_m$, $\theta' \in mgu(a, a')$,
$b = ((b_1, P_{/b_1}), \ldots, (b_m, P_{/b_m}))\theta'$ if $m \geq 1$, $b = true$ if $m = 0$

and $((b_1, P_{/b_1}), \ldots, (b_m, P_{/b_m}))\theta'$ is a shorthand for $(b_1\theta', P_{/b_1}), \ldots, (b_m\theta', P_{/b_m})$.

2. Eliminate-clause$_{UG}$

$$\langle (a, C); \theta \rangle \xrightarrow{Eliminate-clause_{UG}} \langle (a, C'); \theta \rangle$$

where $C' = \{c \mid c \in C$ and a is unifiable with $c\}$.

3. Fail$_{UG}$

$$\langle (a, \emptyset); \theta \rangle \xrightarrow{Fail_{UG}} \langle fail; \theta \rangle$$

4. Or-expand$_{UG}$

$$\langle (a, C); \theta \rangle \xrightarrow{Or-expand_{UG}} \langle (a, \{c_1\}); \theta \rangle, \ldots, \langle (a, \{c_m\}); \theta \rangle$$

where $C = \{c_1, \ldots, c_m\}$, $m > 1$ and c_i is unifiable with a for $i = 1, \ldots, m$.

Transitions on goals

In what follows,u, v will represent a single unit-goal, and s – a (possibly empty) sequence of unit-goals, i.e., $s = u_1, \ldots, u_n = (a_1, C_1), \ldots, (a_n, C_n)$ where $n \geq 0$. We assume $true$ to be the neutral element w.r.t. the sequence juxtaposition ",", i.e. $s, true = true, s = s$.

1. Reduce$_G$

$$\frac{\langle u; \theta \rangle \xrightarrow{Reduce^d_{UG}} \langle s'\theta'; \theta \circ \theta' \rangle}{\langle s_1, u, s_2; \theta \rangle \xrightarrow{Reduce_G} \langle (s_1, s', s_2)\theta'; \theta \circ \theta' \rangle}$$

2. Eliminate-clause$_G$

$$\frac{\langle u; \theta \rangle \xrightarrow{Eliminate-clause_{UG}} \langle u'; \theta \rangle}{\langle s_1, u, s_2; \theta \rangle \xrightarrow{Eliminate-clause_G} \langle s_1, u', s_2; \theta \rangle}$$

3. Fail$_G$

$$\frac{\langle u; \theta \rangle \xrightarrow{Fail_{UG}} \langle fail; \theta \rangle}{\langle s_1, u, s_2; \theta \rangle \xrightarrow{Fail_G} \langle fail; \theta \rangle}$$

4. Or-expand$_G$

$$\frac{\langle u; \theta \rangle \xrightarrow{Or-expand_{UG}} \{\langle u_k; \theta \rangle \mid k = 1, \ldots, m\}}{\langle s_1, u, s_2; \theta \rangle \xrightarrow{Or-expand_G} \{\langle s_1, u_k, s_2; \theta \rangle \mid k = 1, \ldots, m\}}$$

if for all $u' \in \{v \mid v$ in s_1 or v in $s_2\}$, $Or - expand_{UG}$ is the only transition enabled on u'.

Transitions on disjunctions of goals

$$\frac{\langle G; \theta \rangle \xrightarrow{t_G} SS'}{SS \rightarrow (SS \setminus \{\langle G; \theta \rangle\}) \cup SS'}$$

where:

$$\begin{cases} t_G \in \{\text{Reduce}_G, \text{Eliminate-clause}_G, \text{Fail}_G, \text{Or-expand}_G\} \\ SS, SS' \text{ are multisets of goals} \\ \langle G; \theta \rangle \in SS \end{cases}$$

3 The Andorra search space

In the following, we introduce two trees, the *Andorra AND Tree* and *Andorra Search Tree*, to represent an Andorra computation and its search space. We refer to $\langle G; \theta \rangle$, a single element of the multiset constituting a state of the Andorra computation, as and *and-state*. Let $l_k(A)$ be the sequence of all nodes at depth k in the tree A.

Definition Let P be an Andorra program and $\langle G; \theta \rangle$ be an and-state. An *Andorra AND Tree* (AA-tree) A for $P \cup \{\langle G; \theta \rangle\}$ is defined as follows:

(a) Each node of the tree is a tuple $\langle g; \theta' \rangle$ where g is a sequence of unit-goals.
Let $l_k(A) = \langle g^1_k; \theta_k \rangle, \ldots, \langle g^n_k; \theta_k \rangle$. Then, the function $And - st$ is defined by
$And - st(l_k(A)) = \langle g^1_k, \ldots, g^n_k; \theta_k \rangle$, i.e. the and-state represented by level k of the tree.

(b) The root is an and-state, initially $\langle G; \theta \rangle$.
Set k to 1.

(c) Perform the following algorithm

(i) For each unit-goal $u \in And - st(l_k(A)) = \langle G_k; \theta_k \rangle$ such that
$\langle u; \theta_k \rangle \xrightarrow{Eliminate-clause_{UG}} \langle u'; \theta_k \rangle$, replace the occurrence of u in level k of A by u'.

If $And - st(l_k(A))$ contains a failing goal, or all its unit-goals are either nondeterministic or $true$,
then STOP, the construction terminates.
$And - st(l_k(A))$ is the *final state* of A.
Else, go to (ii).

(ii) For every unit-goal u in every $n = \langle s_1, u, s_2; \theta \rangle \in l_k(A)$,

If $\langle u; \theta \rangle \overset{Reduce^d_{UG}}{\longrightarrow} \langle u'; \theta \circ \theta' \rangle$
then a. n has a child $n' = \langle u'; \theta \circ \theta' \rangle$.
b. apply θ' to all nodes of the tree, but n'
(where θ' applied to $\langle G; \theta \rangle$ is $\langle G\theta'; \theta \circ \theta' \rangle$).
Else if u is nondeterministic then n has a child $n' = \langle u; \theta \rangle$,
which is marked "copied".

(iii) Increment k and go to (i).

◇

The size of an AA-tree A is the number of nodes in A which are not marked "copied". The height of an AA-tree represents the number of parallel reductions required to evaluate the tree.

Note that STOP at (c.i) is never reached iff the AA-tree is infinite. A *Correspondence function* $And - corr(\langle G; \theta \rangle)$ is a mapping from an and-state $S = \langle G; \theta \rangle$ into an AND tree whose root is S, if one such finite tree exists, $\perp$ otherwise. Roughly speaking, $\perp$ represents And computations which are infinite. Let *Fin* be a mapping defined over $AA - trees \cup \{\perp\}$. $Fin(A) = \perp$ if $A = \perp$ and $Fin(A) = final - state - of - A$ if $A \neq \perp$.

The final state of an AA-tree $\in \{\langle true; \theta \rangle, \langle fail; \theta \rangle, \langle G; \theta \rangle\}$ where G is a goal in which all unit-goals are nondeterministic.

Definition Let P be an Andorra program, G a goal and $And - corr$ a correspondence function. An *Andorra Search Tree* (AS-tree) A for $P \cup \{G\}$ w.r.t. $And - corr$ is a tree defined as follows:

(a) Each node is a pair (S_1, S_2), where S_1 is an and-state and $S_2 = Fin(And - corr(S_1))$.

(b) The root is $(\langle G; \epsilon \rangle, Fin(And - corr(\langle G; \epsilon \rangle)))$.

(c) If n is a node of the form $(S, \langle s_1, u, s_2; \theta \rangle)$,
and $\langle s_1, u, s_2; \theta \rangle \overset{OR-expand_G}{\longrightarrow} \{\langle s_1, u_k, s_2; \theta \rangle \mid k = 1, \ldots, m\}$, then n has m children $(\langle s_1, u_k, s_2; \theta \rangle, Fin(And - corr(\langle s_1, u_k, s_2; \theta \rangle)))$, $k = 1, \ldots, m$.

◇

An AS-tree is an alternative to the SLD-tree [Llo87] to represent the search space of an Andorra program P w.r.t. a goal G. Each node $n = (S_1, S_2)$ such that $S_2 \neq \perp$ has a *correspondent* AND tree, given by $And - corr(S_1)$. Let $n = (S_1, S_2)$ be a leaf. If $S_2 = \perp$ the branch represents an Andorra computation with an infinite And phase. If $S_2 = \langle fail; \theta \rangle$ the branch represents a successful computation and θ is a computed answer substitution for G.

In SLD-trees, an Andorra And phase would be represented by a linear path, which hides the parallel evaluation. In AS-trees, the AND-phase is represented by a single node. The $AND - corr$ function maps this single node representation into an AA-tree, in which the And parallel evaluation is represented appropriately.

4 The Algorithm and its FCP(:) Implementation

The algorithm is based on the algorithm developed for the or-parallel Prolog interpreter in FCP [Sha86]. Similar to the Or-Parallel algorithm, all paths in the Andorra search tree of the given goal are searched in parallel. Each evaluated path has its own copy of the goal, and does not share logical variables with processes evaluating other paths. If an explored path is successful, its instantiated copy of the goal is added to the set of solutions.

Unlike the Or-parallel model in which each path is explored by a single process, in the current algorithm a path is explored by a pool of processes executing in parallel. To organize the search such that each path is explored exactly once, a technique similar to [Sha86] is used here, with one basic difference; since a path is now explored by a pool of processes rather than a single one, a *path* in the Or-Parallel model becomes a *tree* in the Andorra model. The *single* search tree representing the Or-Parallel Prolog computation becomes a *set* of trees. Accordingly, the *list* of clause indices used to represent a path in the Or-parallel computation is replaced by a *tree* of indices (called the trace-tree) in the Andorra computation.

The Algorithm. There are three types of processes: *controllers*, *reducers*, and *tracers*. Each path in the Andorra search tree is explored by a dynamically changing set of reducers chained to a single controller. The entire tree is explored by a dynamically changing set of controller/reducers chains (Figure 1).

A *reducer* is responsible for the evaluation of a single unit-goal. If its unit-goal is deterministic, the reducer initiates its reduction. It then invokes a set of reducers, one for every unit-goal within the body of the reducing clause, chain them all to the path's controller and terminates. If the unit-goal fails, the reducer terminates, sending a *fail* message along the controller/reducers chain, to propagate failure termination of the path. If the unit-goal is nondeterministic, the reducer suspends. Reducers of *unstable* nondeterministic goals will get unsuspended when their goal becomes deterministic. If deadlock is reached before then, they may get Or-expanded. Reducers of *stable* nondeterministic goals remain suspended until Or-expanded.

A *controller* controls the evaluation of the path to which it is assigned. It senses the And-parallel phase termination (suspension of all reducers on its chain, bringing the evaluation of the path to a *deadlock*) and initiates Or-expansion . The controller also senses both successful and failing termination of the path evaluation, adding its instantiated copy of the goal to the set of solutions upon success.

Upon Or-expansion, new controller/reducers chains are invoked. The controller initiating the expansion (the *initiator*) proceeds with the leftmost branch. All other branches are evaluated by first tracing the computation up to the branchpoint, using a fresh copy of the goal.

Tracing for a single new branch is performed in parallel by a dynamically changing set of *tracers* chained to a single controller. The tracers repeat all reductions performed up to the current branchpoint, following the trace-tree. Tracing also produces a copy of the trace-tree, to be used by the new controller/reducers chain. To avoid redundant tracing, branches at a branchpoint are traced one at a time. This insures a maximum of one redundant tracing per branchpoint.

To maximize parallelism, the initiator proceeds in parallel to the tracing for sibling

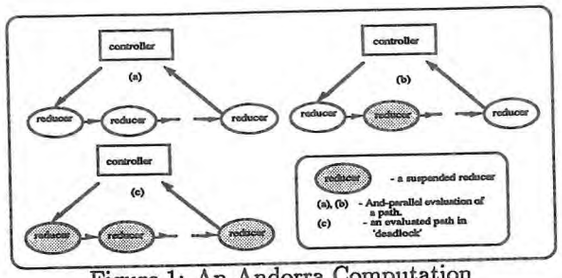

Figure 1: An Andorra Computation

branches. This creates a problem since the initiator and the tracers share the initiator's trace-tree, i.e., the trace-tree continues to grow while being used for tracing. The problem is solved by marking the nodes at the frontier of the trace-tree with the branchpoint's timestamp, prior to Or-expansion. Note that a trace-tree will usually contain several timestamped frontiers, one for each branchpoint on the evaluated path.

Tracing continues up to the nodes marked with the current branchpoint timestamp. At this point, tracers of all but the leftmost unit-goal become (suspended) reducers, and the tracer of the expanded goal reduces the goal with its next reducible clause, invoking reducers for its body. Once traced, all branches are independent, each with its own copy of the initial goal, set of unit-goals and trace-tree. They are thus good candidates for parallel execution on distributed processors.

The forward exploration of a path can be viewed as a distributed computation with partial centralized control. *Centralized*, since most of the control functions are performed by one central process (the controller). *Partial*, since one important control function, namely, suspension/unsuspension of unstable nondeterministic goals is performed by the Ask-suspension and distributed-unification mechanisms of the FCP machine [KMS89].

The FCP(:) implementation. The FCP(:) implementation closely matches the algorithm. Solutions are collected via difference-lists by which the controllers are connected. The problems of distributed phased termination detection and quiescence detection, arising in both the forward evaluation of a path and its retracing, are solved with the short circuit protocol, as discussed in sec. 7.3 of [Sha89].

The implementation assumes the program to be accessible via three predicates, *iclause/3*, *dclause/10*, and *tclause/5*.

iclause(Index,Goal,Body) is used for tracing. It contains one clause for each clause in the original program. It returns the Body of a given clause (Index) after unifying Goal with the head of that clause.

dclause(Goal,Body,Index,...) performs the determinacy analysis of a given Goal. In the current implementation, the most naive run time analysis is performed. For each procedure in the Andorra program $\{h_1 \leftarrow B_1, \ldots, h_m \leftarrow B_m\}$, we have a corresponding set of dclauses as follows:

$dclause(G, Body, Index, \ldots) :-$
$\quad G = \backslash = h_1, \ldots, G = \backslash = h_{i-1}, G = \backslash = h_{i+1}, \ldots, G = \backslash = h_m : G = h_i, Index = i \mid$
$\quad Body = B_i, \ldots \qquad$ for all $i = 1, \ldots, m$

$dclause(G, Body, Index, \ldots) :-$

$G = \backslash = h_1, \ldots, G = \backslash = h_m : Index = failed \mid \ldots$

$dclause(G, Body, Index, \ldots) :-$
$\quad otherwise \mid$ artificial suspension,...

The *otherwise* clause is shared by all procedures, thus $2m+1$ *dclause* clauses are used to represent m andorra clauses.

If the goal G is deterministic, it is reduced, returning both $Index$ and $Body$ of the reducing clause. If G fails, $Index$ is set to *failed*. If G is nondeterministic, *dclause* suspends. Dclause processes of *unstable* nondeterministic goals suspend on two or more of the guard tests $Goal = \backslash = h_i$[1]. They will unsuspend and be reevaluated if and when the goal gets further instantiated. Dclause processes of *stable* nondeterministic goals fail, and are dealt with by the *otherwise* clause. This could be used to separate between stable and unstable nondeterministic goals, as discussed in section 1.

tclause(ContIn,Index,ContOut,Goal,Body) is used for Or-expansion. It contains one clause for each clause in the original program. It finds the first clause (of the original program) with $Index \geq ContIn$ that can reduce Goal, and returns its body ($Body$), its index ($Index$), and an index ($ContOut$) to the Goal's next reducible clause.

5 Complexity

In the following we analyze the overhead and speedup of the algorithm. To measure the speedup, we compare a standard sequential implementation of Prolog, with a parallel implementation of Andorra's execution algorithm. The main factors affecting the cost of an exhaustive search with Andorra computation model, as opposed to Prolog, are *Or-expansion* and the run-time cost of the *determinacy analysis* on the negative side, and the *Or-parallel evaluation* and *reduction of the search space* on the positive side.

Let P be an Andorra program with m average number of clauses per procedure, and G – a goal with an Andorra-search-tree T. Let T be an AS-tree in which all nodes correspond to AA-trees of the same height h_a. Assume the longest path from the root to a leaf is the rightmost path in T, whose length is h_o and the branching factor at each node on that path is b. Let D represent the cost of a deterministic reduction, and N – the cost of suspending a nondeterministic goal.
Then, assuming enough processors, the algorithm will evaluate each AA-tree in $O(h_a D)$ parallel reductions. Or-expansion at the first node will require $O(bh_a + N)$ parallel reductions, and the number of parallel reductions required for Or-expansion at the $i-th$ node (of any path in T) is $O(bih_a + N)$.
The algorithm evaluates the entire tree T in $O(h_A D + bh_a h_o^2 + h_o N)$ parallel reductions, where $h_A = h_o h_a$ is the sum of the heights of the AA-trees on the rightmost path of T.

Thus, for an AS-tree with a single node (a successful AA-tree), the algorithm will evaluate the single solution in $O(h_A D)$ parallel reductions, where D represents the overhead of determinacy analysis. With this tree, the best performance is with a thick tree, i.e. a tree whose size (n) is logarithmic in its height. For this tree, the speedup of the algorithm

[1] $= \backslash =$ is an FCP(:) guard primitive [YKS90]. X = \ = Y succeeds if X and Y are not unifiable. It fails if no further instantiation of global variables can make it succeed and suspends otherwise

is $O(\frac{n}{D \log n})$. The worst performance will be with a thin AA-tree ($h_A = n$), for which the speedup is $O(1/D)$.

For a pure OR tree ($h_a = 1$ for all nodes in the AS-tree), all solutions will be evaluated in $O(bh_o^2 + h_oN)$ parallel reductions, where bh_o^2 represents the overhead of retracing (the factor b resulting from the fact that branches at a branchpoint are retraced one at a time), and h_oN – the overhead of reevaluating the suspension of all unstable nondeterministic goals, at the end of each retracing phase. Again, the best performance with a pure Or-tree is with a thick tree, where the speedup is $O(\frac{n}{blog^2n+N \log n})$, and the worst performance – a thin tree for which the speedup is $O(\frac{1}{bn+N})$.

The analysis of the speedup for the general Andorra search tree, i.e. a tree combining both And and Or parallelism, has to deal with the effect of the computation model on the search space, and is not performed here.

The run-time cost of a deterministic reduction (D) in the current implementation is $O(m^2)$, but is reduced to $O(m)$ by compiling the FCP representation of the Andorra program (i.e. *iclause*, *dclause* and *tclause*), using the Decision Tree Compilation Technique described in [KS88]. With this technique, each FCP procedure is compiled into a decision tree. The compilation of each *dclause* procedure, representing an Andorra procedure P_i, results in a decision tree T_{p_i} of size $O(m^2)$. T_{p_i} is a binary tree (since *dclause* clauses contain guard tests only), with $(m + \binom{m}{2} + 1)$ terminal nodes, of which $(m + 1)$ represent ***successful transitions*** and $\binom{m}{2}$ represent ***failure transitions***. The successful transitions correspond to the m clauses of *dclause* representing a deterministic goal G, and the single clause representing a failing G. The failure transitions correspond to the $\binom{m}{2}$ cases of stable nondeterministic goals, i.e. G that unifies with at least 2 (out of m) clauses in P_i without instantiating any of its variables. Unstable nondeterministic goals are not represented by any terminal nodes; They suspend on one of the guard tests $G = \backslash = H_i$, constituting the internal nodes of T_{p_i}. The height (h) of T_{p_i} is m for all successful transitions (i.e.-deterministic and failing goals), and $2 \leq h \leq m$ for failure transitions (i.e. nondeterministic goals). Thus, both D and N are $O(m)$.

The algorithm involves three major operations besides reductions: generating copies of prefixes of the trace-tree, generating copies of the initial goal for every new branch at a branchpoint, and the detection of distributed phased termination and deadlock.

6 Conclusion

We have presented a formal definition of Andorra and its search space. Then we have introduced an interpreter of Andorra in the concurrent logic language FCP(:), and studied its performance. As we have discussed in section 5, determinacy analysis and tracing are the main factors which affect the performance of our interpreter. Thus, we are currently investigating about techniques to perform determinacy analysis at compile time, and allow parallel retracing of branches. We believe that an approach similar to Ueda [Ued87a,Ued87b] can be successfully applied to our framework. We are also investigating about the proof of correctness for our interpreter.

6.1 Acknowledgments

This work was performed while the first author was visiting the Department of Computer Science at the Weizmann Institute of Science. He gratefully acknowledges the department of Computer Science for making this possible.

References

[BG89] R. Bahagat and S. Gregory, Pandora: Non-Deterministic Parallel Logic Programming, *Proceedings of the Sixth Int. Conf. on Logic Programming.* (G. Levi and M. Martelli, eds.), MIT press, Lisbon, 1989, pp. 471–485.

[BH91] F. Bueno and M. Hermenegildo, Towards a Translation Algorithm from Prolog to the Andorra Kernel Language, this volume, 1991.

[HB88] S. Haridi and Per Brand, Andorra Prolog, an integration of prolog and committed choice languages, *Proceedings FGCS'88*, ICOT, 1988.

[HJ90] S. Haridi and Sverker Janson, Kernel Andorra Prolog and its Computation Model, *Proceedings ICLP'90*, MIT press, 1990.

[KMS89] A. Kleinman, Y. Moses, and E. Shapiro, *A Distributed Unification Algorithm*, Technical Report, The Weizmann Institute of Science, 1989.

[KS88] S. Kliger and E. Shapiro, *A Decision Tree Compilation Algorithm for FCP(|,:,?)*, *Proceedings ICLP'88*, MIT press, 1988, pp. 1315–1336.

[L*88] E. Lusk et al., The Aurora Or-Parallel Prolog System, *Proceedings FGCS'88*, ICOT, 1988, pp. 819–830.

[Llo87] J.W Lloyd, *Foundations of logic programming*, Springer-Verlag, 1987.

[Plo81] G. D. Plotkin, *A structural approach to operational semantics*, Technical Report DAIMI FN-19, Aarhus University, 1981.

[Plo83] G. D. Plotkin, An operational semantics for CSP, *IFIP TC2 Working Conference on Formal Description of Programming Concepts - II* (D. Bjorner, ed.), North-Holland, Amsterdam, 1983, pp. 199–225.

[Sha86] E. Shapiro, Or-Parallel Prolog in Flat Concurrent Prolog, *Concurrent Prolog Collected Papers*, chapter 34, pp. 415–441, MIT Press, 1986.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21-3:412–510, September 1989.

[Ued87a] K. Ueda, Making exhaustive search programs deterministic, *New Generation Computing* 5:1, 1987, p. 29.

[Ued87b] K. Ueda, Making Exhaustive Search Programs Deterministic, Part II, *Proceedings of the Fourth International Logic Programing Conference*, MIT Press, 1987.

[Yan89] R. Yang, *PEPMA 6-monthly report*, Technical Report, Department of Computer Science, University of Bristol, 1989.

[YKS90] E. Yardeni, S. Kliger, and E. Shapiro. The languages FCP(:,?) and FCP(:). *New Generation Computing*, 7(2,3):89–107, 1990.

Appendix

The following programs are written in a concise notation. For each predicate definition (procedure) the head is written in full only on the first line and its parameters are considered implicitely repeated in the following lines of the definition. The same applies to the recursive calls. The recursive calls sometimes consist only of the name of the predicate. It is implicit that all the variables which are not modified in the body of the clause are just repeated in the recursive call. While the variables which are binded in the body will be replaced in the call by the variables which bind the old ones.

For example consider the following program:
procedure p(X,Y).

p :- X = [a | X'] | q(X), p.

This program stands for

p(X,Y) :- X = [a | X'] | q(X), p(X',Y).

A Andorra boolean satisfiability program and its FCP(:) representation.

Program 1: An Andorra Boolean satisfiability program.

```
satis(value(true)).
satis(and(X,Y)) :- satis(X), satis(Y).
satis(or(X,Y)) :- satis(X).
satis(or(X,Y)) :- satis(Y).
satis(not(X)) :- invalid(X).

invalid(value(false)).
invalid(or(X,Y)) :- invalid(X), invalid(Y).
invalid(and(X,Y)) :- invalid(X).
invalid(and(X,Y)) :- invalid(Y).
invalid(not(X)) :- satis(X).
```

% A test program for Andorra interpreter; Boolean satisfiability.
Program 2: representation of the Andorra Boolean satisfiability program.

```
-language([implicit,colon]).

procedure iclause(Index,Goal,Body).          % for recomputation of deterministic reductions
                                             % and previous or-choices.
iclause :-
        Index=1 |
        Goal=satis(value(true)), Body=true.
iclause :-
        Index=2 |
        Goal=satis(and(X,Y)), Body=(satis(X), satis(Y)).
iclause :-
        Index=3 |
        Goal=satis(or(X,Y)), Body=satis(X).
iclause :-
        Index=4 |
        Goal=satis(or(X,Y)), Body=satis(Y).
iclause :-
        Index=5 |
        Goal=satis(not(X)), Body=invalid(X).
iclause :-
        Index=6 |
        Goal=invalid(value(false)), Body=true.
iclause :-
        Index=7 |
        Goal=invalid(or(X,Y)), Body=(invalid(X), invalid(Y)).
iclause :-
        Index=8 |
        Goal=invalid(and(X,Y)), Body=invalid(X).
iclause :-
        Index=9 |
        Goal=invalid(and(X,Y)), Body=invalid(Y).
iclause :-
        Index=10 |
        Goal=invalid(not(X)), Body=satis(X).

procedure tclause(ContIn,Index,ContOut,Goal,Body).
                                             % for or-expansion
tclause :-
        ContIn=<1 : Goal=satis(value(true)), Index=1 |
        ContOut=fail, Body=true.
tclause :-
        ContIn=<2 : Goal=satis(and(X,Y)), Index=2 |
        ContOut=3, Body=(satis(X), satis(Y)).
tclause :-
        ContIn=<3 : Goal=satis(or(X,Y)), Index=3 |
        ContOut=4, Body=satis(X).
tclause :-
        ContIn=<4 : Goal=satis(or(X,Y)), Index=4 |
        ContOut=5, Body=satis(Y).
tclause :-
        ContIn=<5 : Goal=satis(not(X)), Index=5 |
        ContOut=6, Body=invalid(X).
tclause :-
        ContIn=<6 :Goal=invalid(value(false)), Index=6 |
        ContOut=7, Body=true.
tclause :-
        ContIn=<7 : Goal=invalid(or(X,Y)), Index=7 |
        ContOut=8, Body=(invalid(X), invalid(Y)).
tclause :-
        ContIn=<8 : Goal=invalid(and(X,Y)), Index=8 |
        ContOut=9, Body=invalid(X).
tclause :-
        ContIn=<9 : Goal=invalid(and(X,Y)), Index=9 |
        ContOut=10, Body=invalid(Y).
tclause :-
        ContIn=<10 : Goal=invalid(not(X)), Index=10 |
        ContOut=fail, Body=satis(X).
tclause :-
        otherwise : Index=fail |
        ContOut=fail, Body=false.


procedure dclause(Goal,Body,Index,L,R,DL,MT,NL,NR,NMT).

dclause :-
        Goal=\=satis(and(X,Y)),Goal=\=satis(or(X,Y)),
```

```
        Goal=\=satis(not(X)):
        Goal=satis(value(true)), Index=1 |
        Body=true, NL=L, NR=R, NMT=MT.
dclause :-
        Goal=\=satis(value(true)),Goal=\=satis(or(X,Y)),
        Goal=\=satis(not(X)) :
        Goal=satis(and(X,Y)), Index=2 |
        Body=(satis(X),satis(Y)), NL=L,NR=R, NMT=MT.
dclause:-
        Goal=\=satis(value(true)),Goal=\=satis(and(X,Y)),
        Goal=\=satis(or(X,Y)),Goal=\=satis(not(X)) :
        Goal=satis(or(X,Y)), Index=3 |
        Body = satis(X),NL=L, NR=R, NMT=MT.
dclause:-
        Goal=\=satis(value(true)),Goal=\=satis(and(X,Y)),
        Goal=\=satis(or(X,Y)),Goal=\=satis(not(X)) :
        Goal=satis(or(X,Y)), Index=4 |
        Body = satis(Y),NL=L,NR=R, NMT=MT.
dclause :-
        Goal=\=satis(value(true)),Goal=\=satis(and(X,Y)),
        Goal=\=satis(or(X,Y)) :
        Goal=satis(not(X)), Index=5 |
        Body=invalid(X), NL=L, NR=R, NMT=MT.
dclause :-
        Goal=\=invalid(or(X,Y)),Goal=\=invalid(and(X,Y)),
        Goal=\=invalid(not(X)) :
        Goal=invalid(value(false)), Index=6 |
        Body=true, NL=L, NR=R, NMT=MT.
dclause :-
        Goal=\=invalid(value(false)),Goal=\=invalid(and(X,Y)),
        Goal=\=invalid(not(X)) :
        Goal=invalid(or(X,Y)), Index=7 |
        Body=(invalid(X),invalid(Y)), NL=L, NR=R, NMT=MT.
dclause :-
        Goal=\=invalid(value(false)), Goal=\=invalid(or(X,Y)),
        Goal=\=invalid(and(X,Y)), Goal=\=invalid(not(X)) :
        Goal=invalid(and(X,Y)), Index=8 |
        Body=invalid(X), NL=L, NR=R, NMT=MT.
dclause :-
        Goal=\=invalid(value(false)), Goal=\=invalid(or(X,Y)),
        Goal=\=invalid(and(X,Y)), Goal=\=invalid(not(X)) :
        Goal=invalid(and(X,Y)), Index=9 |
        Body=invalid(y), NL=L, NR=R, NMT=MT.
dclause:-
        Goal=\=invalid(value(false)), Goal=\=invalid(or(X,Y)),
        Goal=\=invalid(and(X,Y)) :
        Goal=invalid(not(X)), Index=10 |
        Body=satis(X), NL=L, NR=R, NMT=MT.
dclause:-
                % one clause per procedure in the original program.
        Goal=satis(_), % for failing goals.
        Goal=\=satis(value(true)),Goal=\=satis(and(X,Y)),
        Goal=\=satis(or(X,Y)),Goal=\=satis(not(X)) :
        Index=failed |
        NL=L, NR=R.
dclause:-
                                % failing goal.
        Goal=invalid(_),
        Goal=\=invalid(value(false)), Goal=\=invalid(or(X,Y)),
        Goal=\=invalid(and(X,Y)), Goal=\=invalid(not(X)) :
```



```
        Index=failed |
        NL=L, NR=R.
dclause :-
        otherwise , Index=abort | true.
                                % artificial suspension
                                % of stable nondeterministic goals.
dclause:-
        L=[dl(Message,N)|L'] |
        N':=N+1,
        Andorra#deadlock_message(DL,Message,N',R,R'),
        DL'=deadlock,
        dclause.
dclause:-
                % suspended
        L=[failed|_] |
        NR=R, NL=L.
dclause:-
                % suspended
        L=[branch(BranchTime,_)|_] : info(5,BranchTime) |
        MT=[mark(BranchTime)| MT'],  % broadcast suspended unit-goals to mark the frontier
        NR=R, NL=L, NMT=MT'.        % the broadcaster is the leftmost suspended unit-goal
dclause:-
        MT=[mark(BranchTime)| MT'] | %request to mark the trace-tree sent to the reducer
        NL=[mark(BranchTime) |NL'],  % by each of the suspended unit-goals
        L=[Done|L'], R=[Done|R'],
        dclause.
```

B Andorra Interpreter in FCP(:)

Program 3: An Andorra Interpreter in FCP(:).

```
-language([implicit,colon,typed]).

procedure solve(Program,Goal,Sols).
% Sols are the instances of the melted FrozenGoal, solved with Program.


solve :-
        freeze(Program#Goal,FrozenGoal,_),
        melt(FrozenGoal,Program'#Goal',_),
        control(FrozenGoal,Goal',Trace,Sols\[],L,R),
        reduce(Program',Goal',Trace,[dl(first,0)|L],R,MT).

procedure control(FrozenGoal,Goal,Trace,SolsDL,L,R).

control :-  % Using Dijkstra's algorithm for deadlock detection.
        R=[dl(nodeadlock,N)|R'], N>0 | L=[dl(first,0)|L'],
        control.
control :-
        R=[dl(first,N)|R'], N>0 | L=[dl(second,0)|L'],
        control.
control :-
        R=[dl(second,N)|R'], N>0 |      % Deadlock detected.
                % Prepare for or-expansion
        L=[branch(BranchTime,Cont),dl(first,0)|L'],
        R'=[done|R"],
        SolsDL=Sols1\Sols3,
        SolsDL'=Sols1\Sols2,
        control,                        % Continue with one branch
```

```
        trace(FrozenGoal,Trace,Sols2\Sols3,BranchTime,Cont).
        % Trace other Or-branches, starting at clause-index Cont.
control :-
        R=[dl(_Message,0)|_R'] |        % No more goals to reduce on this Or-branch
        SolsDL=[Goal|X]\X.
control :-
        R=[failed|_R'] |                % No solution on this Or-branch.
        SolsDL=X\X.

procedure trace(FrozenGoal,Trace,SolsDL,BranchTime,Cont).

trace :-                        % no more branches
        Cont=fail |
        SolsDL=X\X.
trace :-
        integer(Cont) |
        SolsDL=Sols1\Sols3,
        SolsDL'=Sols1\Sols2,
        melt(FrozenGoal,Program#Goal1,_),
        trace'(Program,Goal1,Trace,TraceOut,
                [done,dl(first,0)|L],[done|R],MT,BranchTime,Cont,Cont'),
        control(FrozenGoal,Goal1,TraceOut,Sols2\Sols3,L,R),
        trace.

procedure trace'(Program,Goal,TraceIn,TraceOut,DoneL,DoneR,MT,
        BranchTime,ContIn,ContOut).
trace' :-
        Goal=true, TraceIn=true |
        TraceOut=true, DoneL=DoneR.
trace' :-
        Goal=(A,B),
        TraceIn=(TraceInA,TraceInB) |
        TraceOut=(TraceOutA,TraceOutB),
        trace'(Program,A,TraceInA,TraceOutA,DoneL,M,MT,BranchTime,ContIn,ContOut),
        trace'(Program,B,TraceInB,TraceOutB,M,DoneR,MT,BranchTime,ContIn,ContOut).
trace' :-
        TraceIn=trace(Index,TraceIn') |
        Program#iclause(Index,Goal,Goal'),
        TraceOut=trace(Index,TraceOut'),
        trace'.
trace' :-
        TraceIn=mark(Time,TraceIn'),
        Time=BranchTime |
        DoneL=[Done|L], DoneR=[Done|R],
        reduce(Program,Goal,TraceOut,L,R,MT).
trace':-
        TraceIn=mark(Time,TraceIn'),
        Time<BranchTime |
        trace'.
trace':-
        TraceIn=expand(Time,TraceIn'),
        Time<BranchTime |
        trace'.
trace':-
        TraceIn=expand(Time,TraceIn'),
        Time=BranchTime,
        DoneL=[done|L], DoneR=[done|R] | % of all deterministic reductions.
        Program#tclause(ContIn,BranchIndex,ContOut,Goal,Goal'),
```



```
        TraceOut=trace(BranchIndex,TraceOut'),
        reduce(Program,Goal',TraceOut',L,R,MT).

procedure reduce(Program,Goal,Trace,L,R,MT).

reduce :-
        Goal=true |
        Trace=true,
        L=R.
reduce :-
        Goal=false |
        Trace=false,
        R=[failed].
reduce :-
        Goal=(A,B) |
        Trace=(TraceA,TraceB),
        reduce(Program,A,TraceA,L,M,MT),
        reduce(Program,B,TraceB,M,R,MT).
reduce :-               % for a non-deterministic unit-goal, both dclause and reduce'
                        % processes will suspend.
        Goal=true, Goal=false, Goal=(_,_) |
        Program#dclause(Goal,Goal',Index,L,R,nodeadlock,MT,NL,NR,NMT),
        reduce'(Goal',Index,Program,Goal,Trace,NL,NR,NMT).


procedure reduce'(Body,Index,Program,Goal,Trace,L,R,MT).

reduce' :-
        L=[failed|_] |
        Trace=false,
        R=[failed].
reduce' :-
        known(Body) |
        Trace=trace(Index,Trace'),
        reduce(Program,Body,Trace',L,R,MT).
reduce' :-
        Index=failed |
        R=[failed],
        Trace=false.
reduce':-
        L=[mark(BranchTime)|L'] |
        Trace=mark(BranchTime,Trace'),
        reduce'.
reduce' :-
        L=[branch(BranchTime,Cont)|L'],
        R=[done|R'] |
        Program#tclause(0,BranchIndex,Cont,Goal,Goal'),
        Trace=expand(BranchTime,trace(BranchIndex,Trace')),
        reduce(Program,Goal',Trace',L',R',MT).

procedure deadlock_message(DL,DeadlockType,N,R,R1).

deadlock_message :-
        DL=nodeadlock | R=[dl(nodeadlock,N)|R1].
deadlock_message :-
        DL=deadlock | R=[dl(DeadlockType,N)|R1].
```

POSTERS

LOGIC MODELING OF VLSI CIRCUITS: THE ITALTEL EXPERIENCE

Massimo Bombana, Patrizia Cavalloro, Umberto Pelizzari, Giuseppe Zaza
ITALTEL SIT - CLTE - 20019 Settimo Milanese
MILANO - ITALY
email:cavallor@settimo.italtel.it

Abstract

In the Design and Automation Laboratory of the Research and Development Department of ITALTEL we have tested the expressiveness of functional and logical languages to describe complex VLSI circuits. This modeling constitutes the base of the utilisation of formal techniques for an exhaustive correctness evaluation of implemented designs. In addition it allows the use of innovative tools to derive alternative architectures from algorithmic descriptions. In our opinion this represents an interesting and promising area for logical applied research. Conclusions are drawn on the expressiveness and utility of this modeling when applied to complex devices typical of our current production line.

1. Innovative features of the application

Any industrially useful CAD design flow must include means of exhaustly evaluating the correctness and reliability of implemented designs. Theorem provers have been used in the academic world to validate devices of different complexities, but have always been applied to rather unrealistic examples. We believe that these techniques are now mature enough for testing in an industrial environment. In our case we wished to evaluate their applicability to complex devices typical of the advanced communication sector. Even the simple inclusion of logic modeling [1] into a basically functional frame of reference represents a very substantial innovation for the traditional CAD design environment. It also implies a change in the mentality of designers and of the people working in this field.

After testing the feasability and advantages of the integration [2] between a proprietory object-oriented CAD environment [3] and OTTER, a first order logic theorem prover, we extended our approach to allow the use of higher order logic. To this aim, the functional language ML has been choosen because it is widely supported by automatic tools. LAMBDA and HOL, theorem provers based on this description language, can so be applied both for the correctness evaluation of designs and for the extraction of architectures from algorithms. This last point is of paramount importance for DSPs (Digital Signal Processors).

2. VLSI design environment and logic modeling: a useful synergy

Logical modeling of very complex hardware designs is a composite task and comprises various steps:

1.translations of designs from a hardware description language, such as VHDL, into ML
2.evaluation of the correctness of the translation
3.utilisation of theorem provers for correctness evaluation of the implementation and architectural synthesis

The first point has been approached defining a subset of ML whose expressiveness could match that of VHDL. Such a one-way translation is not sufficient in itself: the true characterising point is the insertion and exploitation of those peculiar features of ML which are not present in

VHDL due to its algorithmic nature.
ML descriptions are characterised by the following major topics:

a. abstract data type definitions to allow the modeling of signals and buses
b. logical modeling of time through recursion
c. definition of internal states and functional modeling through them

A Universal Asynchronous Receiver Transmitter (UART) can be divided in functional blocks. Each of them is described by:

Forall t: (out(t+m) = BLK_COMB(xstat t–1);;
xstat (t) = BLK_SYNC (xin t, xstat t–1);;)

where BLK_COMB and BLK_SYNC model respectively the combinatory and synchronous parts of the block, xin and xstat are the sets of input data and internal state variables, and m is a delay due to the physical implementation. The complete description of the device proved that ML has at least the same expressiveness of a hardware description language.

The second point has been approached developing a translation environment comprising the following steps:

a. implementation of a basic utility environment in ML to support device specifications
b. input pattern selection and code simulation of the VHDL description
c. implementation of a utility environment in ML to support ML code simulation
d. automatic comparisons of the output generated patterns

This environment has been applied to various devices of different complexity.

The third point concerns still ongoing activities. We started describing algorithms used in DSPs and other devices , such as the Policing Unit to monitor the rate of ATM (Asynchronous Transfer Mode) connections. We intend to utilise this modeling to test different and alternative architectures using the LAMBDA tool.

3. Conclusions and references

Our activity has shown the great interest and importance of logical modeling and logical tools for the definition of an innovative CAD design environment which will be able to match the requirements of the future. The introduction of these new methodologies is by no means straightforward and implies also a change of mentality for designers. Anyway we believe that these tools will cover an important role in this field, provided they are well integrated into the traditional CAD design flow.

Future activities will be focused on the application of theorem provers, such as LAMBDA and HOL, to this modeling with the aim of both providing new correctness evaluation methodologies and alternative architectural synthesis.

[1] M. Bombana, P. Cavalloro, R. Vaghini, G. Zaza "CAD objects and logic modeling: an industrial experience"– TOOLS'91– Paris, March 4–8, 1991

[2] M. Bombana, P. Cavalloro, R. Vaghini, G. Zaza "The integration of OTTER into a design environment" Italtel Technical Report n. 92090/5/90"– Accepted at the poster session of CHDL'91 – Marseille, April 22–24, 1991

[3] M. Bombana " Design Rule Independent Module Generation" 14th ESPRIT CAVE Workshop – Kolding 1989

DEVELOPMENT OF A REAL-TIME DIAGNOSTIC SYSTEM FOR DATA ACQUISITION

R. Campanini, I. D'Antone, G. Di Caro, G. Giusti

Dipartimento di Fisica, Università di Bologna
Via Irnerio, 46 - 40126 Bologna - Italy

ABSTRACT

We develop an expert system written in CS-Prolog on an array of transputers to make a diagnosis on data acquisition systems. The aim of this work is to put in evidence the break elements of the sperimental apparatus with the use of some rules of production and with the structural knowledge about the connections between these elements. The system is logically separated in two blocks with specific functions. The first is directly interfaced with the electronic acquisition of external apoparatus. This block make a pre-elaboration of the signal that comes during the run, then send the collected information to the second block. This information is composed by a list of number that identify the not-working unit of the system (a "unit" is the elementary element that may be checked by the electronics, the "element" is a cluster of a unit that compound the system). The second block with this list make an exhaustive search on all the elements that refer to each number and with the inference between the rules and the previous information give a diagnosis.

In this first step of our work we have focused the attention on the second block, the first is performed by a simulation that send the required lidst of number to the second group. This is written in CS-Prolog and the interface with the external world is made with few routines in 3L Parallel C.

The fixed part of knowledge base is built with production rules that define the damage condition for each element, anc with the structural information concerning the geometrical relations between the "units" and the "elements". This run-time knowledge base is updated with the information derived from the diagnosis.

The parallelization of the problem is made with the creation of a set of worker-processes that the suystem assign univocally to every element as defined previously. Each of this workers run on a different transputer, as much distributed as prossible on the arrat. A worker perform a study on its element and send the result, the diagnosis, to a amster-process that displays it. In this way every kind of element is logically localized on a processor and the relative information is only on its memory. So each worker is like an expert specialized on a particular element, in its memory there are only the rules, the information and the derived diagnosis related to this element.

We think that the originality and the main interest of this work is due to the possibility to have a bidirectional exchange of information between Prolog-system and datat acquisition, this make possible a direct control of the damage condition.

UN GENERATORE DI CODICE SORGENTE PER APPLICAZIONI TRANSAZIONALI SU BASI DI DATI REALIZZATO IN PROLOG.

S. Dulli , R. Sprugnoli , L. Veronese
Dipartimento di Matematica Pura e applicata
Via Belzoni 7, 35131 Padova

Si è esaminata la possibilità di utilizzare il linguaggio PROLOG per progettare generatori di codice sorgente di programmi in grado di effettuare transazioni su basi di dati.

La base di partenza è stata la definizione di un modello concettuale del sistema informativo che ne mettesse in luce l'aspetto della dinamica temporale. Questo risultato è stato ottenuto definendo le varie transazioni come componenti integrative del modello concettuale dei dati. Tale modello concettuale è poi stato specificato come meta-linguaggio utilizzando la sintassi del linguaggio PROLOG. In questo modo è stato possibile specificare un sistema informativo come una base di conoscenza facilmente interrogabile ed incrementabile nel tempo.

Un generatore di codice sorgente può essere scritto in modo agevole progettando un meta-interprete PROLOG che in ingresso accetta il nome di un programma descritto nella base di conoscenza e che termina con successo la propria valutazione se il programma è generabile a partire dalla base di conoscenza fornita dall'utente. Come effetto collaterale della valutazione, i predicati che costituiscono il meta-interprete producono il codice sorgente del programma applicativo fornito come obiettivo in un linguaggio dato.

L'architettura del generatore è basata sulla definizione di insiemi di predicati, detti moduli, specializzati nella scrittura di classi di istruzioni diverse: vi sono predicati specializzati nella scrittura di istruzioni per la valutazione di espressioni, per la lettura di informazioni dalla base di dati associata al sistema informativo, per l'aggiornamento della stessa, ecc. ...

Tali moduli comunicano tra di loro scambiandosi dei messaggi utilizzando le primitive di *assert* e *retract*. Questa tecnica consente di affrontare agevolmente i problemi di correttezza ed efficienza del codice generato.

L'architettura adottata inoltre consente una facile estensione del sistema e rende semplice la conversione di un generatore di codice che genera codice in un dato linguaggio, in un altro generatore basato sullo stesso meta-linguaggio che produce programmi scritti in un linguaggio diverso.

Il generatore sviluppato ha molte caratteristiche innovative che gli derivano dal fatto di essere stato realizzato in PROLOG; interessante è il fatto che il linguaggio consenta di affrontare il problema della progettazione di un generatore di codice di questo tipo ad un livello di astrazione elevato. Infatti poichè le transazioni sono descritte come meta-procedure (cioè procedure che operano sulle entità descritte nel modello concettuale), e tali meta-procedure sono specificate come termini PROLOG, la generazione di codice consiste fondamentalmente nella esplorazione ricorsiva di una meta-procedura. Si sono dovuti affrontare tuttavia una serie di problemi legati alla necessità di scrivere le istruzioni del programma generato nella sequenza corretta ed in maniera da ottenere un programma efficiente; questi problemi sono stati risolti sfruttando le tecniche classiche di progettazione dei compilatori, come la generazione del codice a più passi.

La caratteristica principale che differenzia questo generatore di codice da quelli tradizionali è che non si tratta di una semplice traduzione da un linguaggio procedurale ad un altro, ma è implicato un passo ulteriore di inferenza, nel quale le procedure vengono costruite dinamicamente a partire dall'appplicazione di regole alla conoscenza specificata dall'utente in forma dichiarativa.

In sintesi il lavoro coinvolge problematiche relative ad aree diverse dell'informatica, e cioè

- nella Computer Aided Software Engineering (CASE), poichè si pone come strumento di automazione del ciclo di sviluppo dei sistemi informativi;
- nel Data Base, poichè affronta le problematiche legate alla progettazione dei sistemi transazionali;
- nella Programmazione logica, poichè PROLOG viene utilizzato sia come linguaggio di specifica, che come motore inferenziale.

Il sistema realizzato è quindi caratterizzato da:

- una proposta di estensione del modello Entity-Relationship in modo da poter trattare dinamicamente le transazioni;
- un linguaggio per la specificazione dei sistemi informativi;
- l'uso di PROLOG come linguaggio di specifica;
-l'uso di PROLOG per la realizzazione di un meta-interprete che opera sul linguaggio precedentemente definito;
- indipendenza dell'architettura del generatore da uno specifico linguaggio cliente.

L'implementazione è stata realizzata in Prolog-2 ; si sono sviluppate a titolo esemplificativo applicazioni gestionali producendo codice sorgente in dBASE IV e in RPG.

A FRIENDLY INTERACTIVE TUTORIAL OF THE WARREN ABSTRACT MACHINE

Julio García-Martín Juan José Moreno-Navarro
Dep. de L.S.I.I.S. - Facultad de Informática - Universidad Politécnica de Madrid
Campus de Montegancedo s/n - Boadilla del Monte - 28660 Madrid (SPAIN)

ABSTRACT

The present paper shows a PROLOG implementation based on the Warren abstract machine with the capability to show to the user its internal behaviour. The tool provides an interactive interface with the machine components allowing to follow its execution step by step and with detail in the degree selected by the user. The implementation is based on a formal specification of the WAM developed by the authors. This specification has been described by stepwise refinement.

1. MOTIVATION

An important part of the success of PROLOG as a (real) programming language comes from the work of [Warren 83]. Warren designed an abstract machine (usually called **WAM** -[Warren **A**bstract **M**achine]) for the efficient execution of PROLOG. The quality of this work has meant that the WAM is the usual reference point for PROLOG compilers. In the literature, there is only a small number of papers having the goal of explaining how the WAM works. Unfortunately, most of them are not successfully written. So, the original Warren's work [Warren 83], the tutorial developed in the Argonne National Laboratory [Gabriel et al. 85], the "real tutorial" by [Aït-Kaci 90], the book written by Maier and Warren [Maier, Warren 88], or another related work is the formal verification of the correctness of the WAM in [Russinof 89]. None of these works can be considered fully satisfactory. The main reason: the nonsense that supposes to explain the implementation of a very abstract language as PROLOG with a lot of concrete details. Besides this lack of abstraction there is another problem: most of the previous works answer the question "how does it work?" but what about other questions like "why does it work?" or "how can we get it?". WAM can be seen as a successful experimental result but not a successful design.

Usual descriptions of the WAM are informally and partially described. This is a drawback for new PROLOG implementors, because they have not any "non-ambiguous" definition of the WAM. The only serious attempts in this line come from [Kursawe 87] and [Hanus 88]. People interested in understanding the computational behaviour of the WAM have only one choice: make themselves its own toy implementation. But this solution could need a great effort compared with the scope of the original problem. As a result of all the problem discussed previously, we found the WAM very complicate to understand without new helps. Readers interested in having a general knowledge about PROLOG compilation have no chance to get it. Our proposal to solve, maybe partially, all these problem is twofold: **First**, we are involved in the development of a gradual definition of the WAM by using stepwise refinement. In a certain sense, the elements of the machine and their optimizations are "deduced" instead of directly shown. **Second**, we have developed a concrete implementation of the previous WAM specification. Our tool is able to compute PROLOG programs by showing to the user all the components of the machine and their behaviour. Each element is displayed at the level of abstraction the user decides. For the purpose of reading this paper we suppose a basic knowledge about the WAM.

2. A STEPWISE REFINEMENT DESCRIPTION OF THE WAM

We are developing a stepwise refinement of the WAM with three goal in mind:

- Justify the design of the WAM.

- Offer a gradual definition at different levels of abstraction.
- Make a formal (hence non-ambiguous) description.

All the elements of the data area are treated as abstract data types, following the methodology where the details are hidden. A list of the data types and a briefly description of their operations is the following:

* HEAP: with operations to initialize, construct a structure, consult the arguments of a structure, bind a variable, etc.
* TRAIL: with operations to initialize, push a variable, mark a trail in a certain moment and undo a trail from the top to a mark inside.
* ENVIRONMENT: with operations for creation, storage and consult of particular information.
* CHOICE-POINT: with similar operations as ENVIRONMENT.
* STACK: with operations to initialize, push and pop elements, consult the element at the top of the stack, etc. The and-stack and the or-stack are instances of this structure.
* REGISTERS: with operations to store or consult information of one of them. The temporary registers and the argument registers are instances of this structure.

All the general registers are part of the data structures they refer. Refinement of their implementations yields to more efficient and precise definitions of the data area. For instance, it is more simple to consider an environment as a distinguished element. A refinement of this structure can consider it as some contiguous memory positions in the and-stack. We have presented how the data area is handled. A similar discussion can be made about the instruction set and code generation. The structures PROLOG-PROGRAM and WAM-PROGRAM must be defined together with a functional abstraction which relates them: the TRANSLATION function. Most of the tricky optimizations of the WAM can be delayed to the last steps of refinement because they are not important to understand its behaviour.

3. PLAYING AND LEARNING WITH THE WAM

The previous stepwise specification is supported by an interactive tool. The tool allows to execute a PROLOG program but looking inside its execution. For this purpose a very flexible window system is used in order to display the information associated to each component of the WAM. There are two special windows for the PROLOG program and the WAM program. The tool has three general execution modes that can be selected by the user using the general menu: quick execution, delayed execution and the step by step mode. The user is able to select the instructions in which the program must stop and he can also select which structures (and at what level of abstraction) he wants to see on the screen. This menu is also available to change our selection (i.e. increase or decrease the abstraction level for the structure). If we have decided to display the programs we obtain the result of figure 1. Notice that the PROLOG clause selected for an atom is emphasized as well as the corresponding WAM instruction (at the level of abstraction selected). Furthermore, the element of the clause in execution is shown in a special video mode. The components of the data area follow a similar criteria.

They are displayed at the level of specification selected by the user. Any structure can be traversed to look at its elements. A special feature of the tool is that it helps to find and locate an element in the machine. For instance, it is possible to ask for the binding of a variable in the trail or the content of a register. In both cases, we obtain a representation of the term they refer to and the structure (usually the

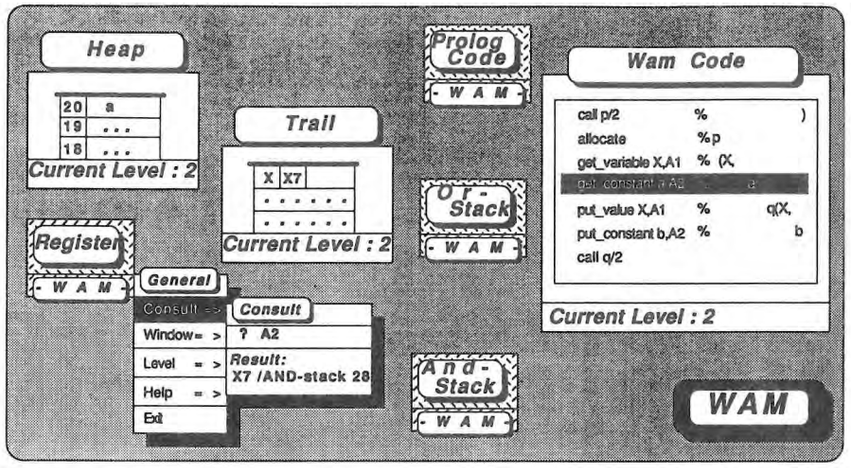

Figure 1: Example of execution

heap) and the point where they are stored. The tool is implemented in Modula-2 which allows us to make a clean program that is quite close to our specification (the abstract data types are implemented as independent modules). The windows are supported by the X-Windows interface. Currently, we have a prototype running on personal computers under DOS loosing some of the facilities described here. Another version for SUN workstations is in progress and will be finished in the next months.

4. CONCLUSIONS

The combination of a stepwise specification of the Warren abstract machine with an interactive tool that supports this definition provides a very friendly tutorial to learn how the WAM works. The tool could be a useful either for people who ignores everything about PROLOG compilers or for expert WAM students who want to go deeper into some of its details.

REFERENCES

[Aït-Kaci 90] Aït-Kaci H.: "The WAM: A (Real) Tutorial". Digital Paris Research Laboratory, January 1990.

[Gabriel et al. 85] Gabriel J., Lindholm T., Lusk E.L., Overbeck R.A.: "A Tutorial on the WAM for Computational Logic". ANL-64-84, Argonne Nat. Lab., 1985.

[García, Moreno 90] García-Martín, J., Moreno-Navarro, J.J.: "A Stepwise Definition of the Warren Abstract Machine", Internal Report, in preparation, 1990.

[Hanus 88] Hanus, M.: "A Formal Specification of the Warren Abstract Machine", ESOP 88, LNCS, Springer Verlag 1988.

[Kursawe 87] P. Kursawe: "How to Invent a PROLOG Machine", New Gen. Comp., 5, 1989.

[Maier, Warren 88] Maier D., Warren D.S.. "Computing with Logic: Logic Programming with Prolog". Ed. Benjamin Cummings 1988.

[Russinoff 89] Russinoff, D.M. : "A Verified Prolog Compiler for the Warren Abstract Machine". MCC Tech. Report N. ACT-ST-292-89, Austin, Texas, July, 1989.

[Warren 83] Warren D.H.D. : "An Abstract Prolog Instruction Set". Technical Note 309 (SRI Project 4776), October, 1983.