Fundamenta Informaticae 34 (1998) 1–22
IOS Press

Executing Suspended Logic Programs*

Robert A. Kowalski, Francesca Toni
Department of Computing
Imperial College of Science, Technology and Medicine
London SW7 2BZ, UK
{rak,ft}@doc.ic.ac.uk

Gerhard Wetzel
Logic Based Systems Lab, Department of Computer Science
Brooklyn College
NY 11210, USA
gw@sci.brooklyn.cuny.edu

Abstract. We present an extension of Logic Programming (LP) which, in addition to ordinary LP clauses, also includes integrity constraints, explicit representation of disjunction in the bodies of clauses and in goals, and suspension of atoms as in concurrent logic languages. The resulting framework aims to unify Constraint Logic Programming (CLP), Abductive Logic Programming (ALP) and Semantic Query Optimisation (SQO) in deductive databases.

We present a proof procedure for the new framework, simplifying and generalising previously proposed proof procedures for ALP.

We discuss applications of the framework, formulating traditional problems from LP, ALP, CLP and SQO.

Keywords: Logic Programming (LP), Constraint Logic Programming (CLP), Abductive Logic Programming (ALP), Semantic Query Optimisation (SQO) in Deductive Databases.

*The second author is supported by the EPSRC project "Logic-based multi-agent systems". The third author is supported by ONR grant N00014-96-1-1057. The authors are grateful to Hendrik Decker and Fariba Sadri for helpful comments on an earlier draft of this paper.

1. Introduction

Ordinary LP solves problems by representing problem-solving procedures by means of clauses of the form

$H \leftarrow L_1 \wedge \ldots \wedge L_m$

with $m \geq 0$, H an atom and each L_i a literal. Variables in H and L_i are implicitly universally quantified with scope the entire clause. H is called the *head* and $L_1 \wedge \ldots \wedge L_m$ is called the *body* of the clause. Clauses of this form are used backwards to unfold atoms in goals (existentially quantified conjunctions of literals). Negation is interpreted as negation as failure [4].

CLP, ALP and SQO are extensions of ordinary LP, each of which is oriented towards a different application area.

CLP [10] extends LP with *constraint predicates*, such as $\leq$, which are not processed as ordinary LP predicates by unfolding but are checked for satisfiability and simplified by means of a built-in, "black-box" constraint solver.

Example 1.1. (CLP)
The logic program

$\mathtt{order}(\mathtt{X_{1M}}, \mathtt{P_{1M}}, \mathtt{X_{2M}}, \mathtt{P_{2M}}) \leftarrow \mathtt{X_{1M}} + \mathtt{P_{1M}} \leq \mathtt{X_{2M}}$

$\mathtt{order}(\mathtt{X_{1M}}, \mathtt{P_{1M}}, \mathtt{X_{2M}}, \mathtt{P_{2M}}) \leftarrow \mathtt{X_{2M}} + \mathtt{P_{2M}} \leq \mathtt{X_{1M}}$

represents the problem of ordering two operations (on the same machine $\mathtt{M}$) with starting times $\mathtt{X_{1M}}$, $\mathtt{X_{2M}}$ and processing times $\mathtt{P_{1M}}$, $\mathtt{P_{2M}}$, respectively. This problem is characteristic of job-shop scheduling. Given a constraint solver for linear (in)equations, the goal

$\mathtt{order}(\mathtt{X_{11}}, 2, \mathtt{X_{21}}, 4) \ \wedge \ 0 \leq \mathtt{X_{11}} \leq 2 \ \wedge \ 0 \leq \mathtt{X_{21}} \leq 1$

fails, since if $\mathtt{X_{11}} + 2 \leq \mathtt{X_{21}}$ then $\mathtt{X_{21}} \geq 2$, and if $\mathtt{X_{21}} + 4 \leq \mathtt{X_{11}}$ then $\mathtt{X_{11}} \geq 4$. However, the goal

$\mathtt{order}(\mathtt{X_{11}}, 2, \mathtt{X_{21}}, 4) \ \wedge \ 0 \leq \mathtt{X_{11}} \ \wedge \ 0 \leq \mathtt{X_{21}} \leq 1$

succeeds with computed answer $\mathtt{X_{21}} + 4 \leq \mathtt{X_{11}} \wedge 4 \leq \mathtt{X_{11}} \wedge 0 \leq \mathtt{X_{21}} \leq 1$. Note that ordinary LP would require an explicit definition of $\leq$ in clausal form. The PROLOG built-in definition for $\leq$ would be inadequate.

Our generalisation of CLP, ALP and SQO is similar to Frühwirth's [7] "no-box" approach to CLP, in which the constraint solver is "programmed" explicitly by means of *constraint handling rules*. In the above example, these rules might take the form

$\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} + \mathtt{C} \leq \mathtt{Z} \rightarrow \mathtt{X} + \mathtt{C} \leq \mathtt{Z}$

$\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} < \mathtt{X} \rightarrow \mathtt{false}.$

Then, given the goal

$\mathtt{X_{11}} + 2 \leq \mathtt{X_{21}} \wedge 0 \leq \mathtt{X_{11}} \leq 2 \wedge 0 \leq \mathtt{X_{21}} \leq 1$

the first constraint handling rule adds $2 \leq \mathtt{X_{21}}$ as a conjunct to the goal and the second adds $\mathtt{false}$. (Similarly $\mathtt{false}$ is added to the goal $\mathtt{X_{21}} + 4 \leq \mathtt{X_{11}} \wedge 0 \leq \mathtt{X_{11}} \leq 2 \wedge 0 \leq \mathtt{X_{21}} \leq 1$.) Moreover, given the goal

$\mathtt{X_{21}} + 4 \leq \mathtt{X_{11}} \wedge 0 \leq \mathtt{X_{11}} \wedge 0 \leq \mathtt{X_{21}} \leq 1$

the first constraint handling rule adds to the goal the conjunct $4 \leq \mathtt{X_{11}}$, which is part of the answer.

ALP [12, 13] extends LP to perform abductive reasoning: given a logic program P and a goal (observation) G, ALP aims to find an "explanation" Δ of G, i.e. a set Δ such that

(A1) Δ contains *abducible* atoms only, i.e. atoms from a given set of candidate hypotheses,

(A2)) P together with Δ "entails" G, and

(A3) P together with Δ "satisfies" a given set of *integrity constraints* $\mathcal{IC}$.

The notion of entailment depends upon the chosen semantics for negation as failure. For negation-free logic programs, entailment is truth in the minimal Herbrand model of the program (see [12, 13]). Similarly, the notion of integrity constraint satisfaction can be understood in different ways. The weakest of these is the *consistency view*, i.e. the union of P, Δ and $\mathcal{IC}$ is consistent. The strongest is the *theoremhood view*, i.e. $\mathcal{IC}$ is entailed by the the union of P and Δ.

Example 1.2. (ALP)
Consider the logic program

`bird` $\leftarrow$ `albatross`

`bird` $\leftarrow$ `penguin`

with `penguin, albatross` and `flies` abducible, and the integrity constraint

`penguin` $\wedge$ `flies` $\rightarrow$ `false`.

Let the goal G be `bird` $\wedge$ `flies`. Then the explanation $\Delta_1 = \{\texttt{albatross}, \texttt{flies}\}$ satisfies (A1)–(A3) under the consistency view whereas the explanation $\Delta_2 = \{\texttt{penguin}, \texttt{flies}\}$ does not. Note that ordinary LP would fail to generate any answer for G as it fails on undefined predicates (like `flies`, `albatross` and `penguin`).

SQO [3] is aimed at optimising query answering in deductive databases, i.e. logic programs whose predicates are partitioned into *extensional* predicates, defined by unit clauses (clauses with $m = 0$), and *intensional* predicates, defined by non-unit clauses (clauses with $m > 0$). The extensional part of the database might be very large and queries to it might be computationally explosive. SQO uses the intensional part of the database together with integrity constraints, expressing "properties" of the database, to optimise queries before they are actually executed. This pre-processing might limit greatly or even avoid entirely the need to access the extensional part of the database.

Example 1.3. (SQO)
Let DB be a deductive database defining extensional predicates `employee`, `position` and `bonus`, and let

`position(X,manager)` $\wedge$ `bonus(X,B)` $\wedge$ `B=0` $\rightarrow$ `false`

be an integrity constraint expressing the property of DB that no manager can have a null bonus. Then, given the query

`employee(X)` $\wedge$ `position(X,manager)` $\wedge$ `bonus(X,B)` $\wedge$ `B=0`

SQO uses the integrity constraint to show that the query has no answers, without accessing DB. If ordinary LP were used to answer the query directly, every individual `employee` record would need to be accessed.

We define an extension of LP generalising CLP, ALP, and SQO. We distinguish between *ordinary* predicates (as in LP) and *external* predicates (corresponding to abducible predicates in ALP and extensional predicates in SQO). Constraint predicates in CLP are treated as ordinary predicates. Conceptually, all predicates, including both ordinary and external predicates, are defined by logic programs. We assume that the given logic programs are *locally stratified* [1], so that they admit a unique *perfect model* [23]. Consequently, entailment is understood as truth in the perfect model. As in ordinary LP, logic programs are executed backwards, to *unfold* atoms using their definitions.

As in SQO, the logic program defining the external predicates is not accessible. Instead, integrity constraints expressing properties of the inaccessible logic program may be used as approximations to the definitions. Integrity constraints are implications which are executed in a forward direction: if the condition holds, then derive (or *propagate*) the conclusion. Such integrity constraints are similar to condition-action (production) rules in artificial intelligence.

In LP, atoms of ordinary predicates might unify non-deterministically with the heads of several clauses. In such a case, as in SQO, propagating with carefully chosen integrity constraints instead of unfolding with logic programs might be able to reduce the amount of non-determinism. For this purpose, it is useful to *suspend* atoms [26, 24, 25, 22] to prevent their being unfolded. In our framework, an atom is *suspended* either if it is external or if it is an ordinary atom which non-deterministically unifies with the head of more than one clause. The notion of suspended atom generalises the notions of abducible atom in ALP, extensional atom in SQO, constraint atom in CLP as well as "frozen" atom in some variants of ordinary LP.

We allow different clauses for the same predicate to be written as one *iff-definition*, with disjunctions in the body, to control suspension and to allow a limited amount of user-controlled non-deterministic search. In example 1.1, the atom $\mathtt{order}(\mathtt{X}_{11}, 2, \mathtt{X}_{21}, 4)$ can be unfolded using two program clauses. If we rewrite the program in the form

$$\mathtt{order}(\mathtt{X}_{1M}, \mathtt{P}_{1M}, \mathtt{X}_{2M}, \mathtt{P}_{2M}) \leftrightarrow \mathtt{X}_{1M} + \mathtt{P}_{1M} \leq \mathtt{X}_{2M} \vee \mathtt{X}_{2M} + \mathtt{P}_{2M} \leq \mathtt{X}_{1M}$$

then the atom is an instance of the head of only one definition.

We allow multiple iff-definitions with the same predicate in the head. For example, the predicate $<$ can be defined by means of

$$\mathtt{0} < \mathtt{X} \leftrightarrow \mathtt{true}$$

$$\mathtt{s}(\mathtt{X}) < \mathtt{s}(\mathtt{Y}) \leftrightarrow \mathtt{X} < \mathtt{Y}.$$

We assume that the heads of iff-definitions defining the same predicate do not unify.

The use of iff-definitions also allows a direct representation of disjunction in goals giving a declarative representation to alternative branches in the search tree of ordinary LP goals (similarly to Muse Prolog [2]).

The paper is organised as follows. Section 2 presents the framework, called *Suspended Logic Programming* (SLP), and discusses its use for problem solving and knowledge representation. Section 3 presents the basic proof procedure for SLP. Section 4 describes applications. Section 5 discusses possible extensions of the basic proof procedure. Section 6 relates SLP to other approaches. Section 7 reports results using existing prototypes of SLP for some applications and concludes.

2. The framework

An SLP knowledge base consists of a set of iff-definitions and a set of integrity constraints, as defined below. We assume that the set of atoms, defined as usual, includes two special atoms, `true` and `false`.

Definition 2.1. An *iff-definition* is a formula of the form

$H \leftrightarrow D_1 \vee \ldots \vee D_n,\ n \geq 1$

where H is an atom (but not `true` or `false`) and each D_i (called a *disjunct*) is a conjunction of literals (called *conjuncts*). Variables in H are implicitly universally quantified with scope the entire definition, whereas variables in any disjunct D_i but not in H are existentially quantified with scope D_i.

H is called the *head* and $D_1 \vee \ldots \vee D_n$ is called the *body* of the iff-definition.

If H is the atom `p(t)`, for some tuple of terms `t`, then the iff-definition is said to *define* or to be an *(iff-)definition for* `p(t)`. Note that there may be zero, one or several iff-definitions for different atoms of the same predicate. We impose the restriction that the heads of different iff-definitions are not only different, but do not unify (see definition 2.2 below).

We will illustrate the framework by means of the n-queens problem, thus highlighting at the same time the use of the framework for representing (and solving) constraint satisfaction problems. In the n-queens problem, n queens have to be placed on an $n \times n$ chessboard, so that if a queen is placed at coordinates `(A,B)` and the rules of chess allow this queen to "move" from `(A,B)` to `(C,D)`, then there is no queen placed at coordinates `(C,D)`. First we define the predicate `move`:

Example 2.1. (move in chess: iff-definitions)
The following set of iff-definitions specifies the notion of queen move in chess:

$$\texttt{move((A,B),(C,D))} \leftrightarrow 1 \leq \texttt{A} \leq n \wedge 1 \leq \texttt{B} \leq n \wedge 1 \leq \texttt{C} \leq n \wedge 1 \leq \texttt{D} \leq n \wedge \texttt{move-row-col-diag((A,B),(C,D))}$$

$$\texttt{move-row-col-diag((A,B),(C,D))} \leftrightarrow [\texttt{not A=C} \wedge \texttt{B=D}] \vee [\texttt{not B=D} \wedge \texttt{A=C}] \vee [\texttt{C=A+X} \wedge \texttt{D=B+X} \wedge \texttt{not X=}\ 0].$$

Note that the given set of iff-definitions in this example is similar to part of the Clark *completion* [4] of the normal logic program consisting of the first iff-definition with $\leftrightarrow$ replaced by $\leftarrow$ and

$\texttt{move-row-col-diag((A,B),(C,B))} \leftarrow \texttt{not A=C}$
$\texttt{move-row-col-diag((A,B),(A,D))} \leftarrow \texttt{not B=D}$
$\texttt{move-row-col-diag((A,B),(A+X,B+X))} \leftarrow \texttt{not X=}\ 0.$

In general, in the Clark completion of a logic program, iff-definitions are in homogenised form (i.e. all terms in the head are mutually distinct variables) and there is exactly one definition per predicate. Instead, in SLP, iff-definitions do not have to be in homogenised form and there may be multiple iff-definitions for the same predicate.

Example 2.2. The set of iff-definitions in example 2.1 could be replaced by the set of iff-definitions

$\texttt{move((1,1),(1,2))} \leftrightarrow \texttt{true}$
$\texttt{move((1,1),(1,3))} \leftrightarrow \texttt{true}$
$\vdots$

In example 2.1, the iff-definitions defining `move` (and the subsidiary predicate `move-row-col-diag`) rely upon definitions of the predicates $\leq, =$. CLP distinguishes between domain-specific predicates, like `move`, and constraint predicates, like $\leq, =$. We treat constraint predicates as *ordinary predicates*, defined by iff-definitions. We will assume that $=$ belongs to the set of ordinary predicates, defined by $\texttt{t=t} \leftrightarrow \texttt{true}$, for all variable-free terms `t`.

In addition to ordinary predicates, SLP allows *external predicates*, whose iff-definitions are inaccessible. External predicates correspond to abducibles in ALP and extensional predicates in SQO. We regard iff-definitions for external predicates as inaccessible rather than as non-existent, in the same way that SQO regards definitions for extensional predicates as inaccessible during the query optimisation process.

Example 2.3. (n-queens problem: external predicates)
The placement of a queen at coordinates `(A,B)` can be represented by the atom `queen(A,B)`. A possible iff-definition for `queen`, for $n = 4$, is

$\texttt{queen}(\texttt{X},\texttt{Y}) \leftrightarrow (\texttt{X} = 1 \wedge \texttt{Y} = 2) \vee (\texttt{X} = 2 \wedge \texttt{Y} = 4) \vee (\texttt{X} = 3 \wedge \texttt{Y} = 1) \vee (\texttt{X} = 4 \wedge \texttt{Y} = 3).$

An alternative iff-definition is

$\texttt{queen}(\texttt{X},\texttt{Y}) \leftrightarrow (\texttt{X} = 1 \wedge \texttt{Y} = 3) \vee (\texttt{X} = 2 \wedge \texttt{Y} = 1) \vee (\texttt{X} = 3 \wedge \texttt{Y} = 4) \vee (\texttt{X} = 4 \wedge \texttt{Y} = 2).$

Each such definition is a solution to the 4-queens problem, and should be thought of as inaccessible. In other words, the predicate `queen` is external.

Definition 2.2. Given two disjoint sets of predicates (ordinary and external predicates, respectively) with $=$ belonging to the set of ordinary predicates, an *SLP theory* is a set $\mathcal{T}$ of iff-definitions whose heads do not unify and such that $\mathcal{T} = \mathcal{T}_o \cup \mathcal{T}_e$, with $\mathcal{T}_o$ and $\mathcal{T}_e$ defining the ordinary and external predicates, respectively.

$\mathcal{T}_o$ is called the *accessible part* of $\mathcal{T}$.
$\mathcal{T}_e$ is called the *inaccessible part* of $\mathcal{T}$.

The assumption that the set of external predicates is disjoint from the set of ordinary predicates is analogous to the assumption in ALP that the set of abducible predicates is disjoint from the set of ordinary predicates. Note that this assumption can be made without loss of generality, as for ALP [12, 13].

Since we assume that heads of iff-definitions do not unify, every (ordinary) atom is an instance of the head of at most one (accessible) iff-definition.

Definition 2.3. We call an atom A *reducible* if there exists exactly one accessible definition such that A is an instance of the head of the definition. Otherwise, we call the atom A *suspended*.

Therefore, external atoms are always suspended. Ordinary atoms are suspended whenever no definition exists whose head unifies with the given atom (analogous to failure in ordinary LP) or multiple definitions exist whose head unifies with the given atom (analogous to non-determinism in ordinary LP).

Suspension is an important feature of concurrent LP languages [26] and concurrent CLP [24, 25], where it is often used to eliminate search non-determinism entirely. The concept of suspension is also related to *delay* mechanisms in NU-PROLOG [22]. In this and several other PROLOG dialects (including CHIP [28]) a control statement of the form

`delay p(`$a_1, \ldots, a_n$`)`

where each a_i is either a "+" or a "−", specifies that an atom `p(`$t_1, \ldots, t_n$`)` can be selected only if all arguments corresponding to a "+" are variable-free. Suspension in most concurrent LP languages and in SLP is more general.

In SLP, a controlled form of search non-determinism is allowed by the use of disjunction in the bodies of iff-definitions. The user can control the extent of search non-determinism by controlling the form in which the iff-definitions of ordinary predicates are written. In particular, if a predicate is defined by a single iff-definition in homogenised form, then all atoms of the predicate are reducible. On the other hand, if a predicate is defined by a set of iff-definitions each with a variable-free head, then all non-variable-free atoms of the predicate are suspended.

Constraint predicates are ordinary predicates. We assume, as given, a set $\mathcal{T}_c \subseteq \mathcal{T}_o$ of iff-definitions for the constraint predicates, such that, for any constraint predicate `c`, the atom `c(t)` is suspended whenever accessing the iff-definitions for c in $\mathcal{T}_c$ would be combinatorially explosive. For example, accessing any correct definition for `3` $\leq$ `X` would generate infinitely many (combinatorially explosive) alternative instantiations of `X`, `X=3`, `X=4`, ... We will assume that $\mathcal{T}_c$ is such that atoms such as `X` $\leq$ `3`, `3` $\leq$ `X`, `X` $\leq$ `Y` and `plus(2,X,Y)` [1] are suspended, whereas such atoms as `3` $\leq$ `10`, `2` $\leq$ `0`, `plus(2,2,4)`, `plus(2,2,X)`, `plus(X,2,3)` and `plus(2,X,5)` are reducible.

Similarly to SQO and ALP, SLP uses integrity constraints as approximations of the iff-definitions whenever the iff-definitions can not be used, either because they are inaccessible (as in the case of external predicates) or more generally because it would be combinatorially explosive to execute them.

Definition 2.4. An *integrity constraint* is a formula of the form

$L_1 \wedge \ldots \wedge L_n \rightarrow A$

where $n \geq 0$, each L_i is a literal, different from (`not`) `false` and (`not`) `true`, and A is an atom, different from `true`. All variables are implicitly universally quantified with scope the entire formula.

$L_1 \wedge \ldots \wedge L_n$ is called the *condition* and A is called the *conclusion* of the integrity constraint.

Note that the empty condition is allowed. For example, the integrity constraint `queen(1,2)` forces a queen to be located at `(1,2)`. Note also that `true` is not allowed as conclusion of an

[1] Note that any formula `F[X+Y]` can be rewritten in terms of a relational representation using the predicate `plus`.

integrity constraint. Indeed, we will see in section 3 that any integrity constraint with `true` as conclusion would be of no use.

Example 2.4. (n-queens problem: integrity constraints)
The kernel of the n-queens problem can be expressed by a single integrity constraint

`queen(A,B)` $\wedge$ `queen(C,D)` $\wedge$ `move((A,B),(C,D))` $\rightarrow$ `false`

where `queen` is an external predicate (as in example 2.3) and `move` is an ordinary predicate (as in example 2.1).

Integrity constraints provide useful approximations not only for (inaccessible) external predicate definitions but also for (accessible) ordinary predicate definitions, to process atoms that are suspended. For example, the integrity constraint

$X \leq Y \wedge Y \leq Z \rightarrow X \leq Z$

can be used to add the reducible atom $21 \leq 10$ to the goal

$21 \leq Y \wedge Y \leq 10$

consisting of suspended atoms only. The newly added reducible atom can then be reduced to `false`.

As mentioned in the introduction, we will assume that the ordinary logic program obtained from the given theory by rewriting each iff-definition as a set of ordinary clauses is locally stratified [1]. Such locally stratified logic programs admit a unique perfect model [23]. To guarantee that integrity constraints are sound approximations of the given theory, we will assume they are true in the *perfect model of the theory*, defined as the perfect model of the logic program obtained from the theory by rewriting each iff-definition as a set of ordinary LP clauses. The perfect model can be regarded as an extensional representation of the meaning of the theory.

Definition 2.5. An SLP *knowledge base* is a pair $\langle \mathcal{T}, \mathcal{IC} \rangle$ where $\mathcal{T}$ is a locally stratified SLP theory and $\mathcal{IC}$ is a set of integrity constraints such that $\mathcal{T} \models \mathcal{IC}$ and $\mathcal{T} \models CET$ (the Clark Equality Theory [4]), where $\mathcal{T} \models A$ means that A is true in the perfect model of $\mathcal{T}$.

Note that $\mathcal{T} \models CET$ follows from the assumption that `t=t` $\leftrightarrow$ `true` belongs to $\mathcal{T}$ for all variable-free terms `t`.

In any given application it is necessary to decide what knowledge should be represented in the form of iff-definitions and what should be represented by means of integrity constraints. This problem is partly addressed by the different semantics of definitions and integrity constraints: definitions define predicates, whereas integrity constraints are true properties of the predicates so defined.

The problem can also be addressed procedurally: given a desired algorithmic behaviour, definitions can be used to obtain the backward-reasoning, goal-reduction component of the behaviour, whereas integrity constraints can be used to obtain the forward-reasoning component.

The proof procedure defined in the next section combines backward and forward reasoning to derive a sequence of goals starting from an initial goal.

3. The basic proof procedure

Definition 3.1. An *SLP initial goal* is a conjunction of literals. All variables in an initial goal are free.

Example 3.1. (n-queens problem: initial goal)
The initial goal

$\texttt{queen(1,}\mathtt{X}_1\texttt{)} \wedge \ldots \wedge \texttt{queen(}n\texttt{,}\mathtt{X}_n\texttt{)}$

stands for the problem of placing a queen in some column $\mathtt{X}_i$ of every row i.

The proof procedure derives a sequence of goals starting from the given initial goal, with the aim of deriving a goal which is disjunction of "answers". These answers might or might not determine values for the free variables of the initial goal. The disjunction of answers is *equivalent* to the initial goal in the sense that, in the perfect model of $\mathcal{T}$, the initial goal and the disjunction are satisfied by the same assignments to the free variables.

To simplify the treatment of quantifiers, both iff-definitions and integrity constraints are assumed to be *range-restricted*, i.e. all variables in the head/conclusion must appear in at least one atom in the body/condition, and this atom must not be an equality between two variables. This guarantees that every variable in goals derived by the proof procedure is free or existentially quantified.

In the simplest (and most common) case, goals derived by the proof procedure are disjunctions of "flat goals":

Definition 3.2. A *flat goal* is a conjunction

$G = C_1 \wedge \ldots \wedge C_n \wedge D_1 \wedge \ldots \wedge D_m$

where $n, m \geq 0$ and $n+m > 0$, the C_i are atoms or "implications" (called the *global constraints* of G), and the D_i are disjunctions of conjunctions of atoms and "implications" (called the *local constraints* of the disjunct to which they belong). [2]

Note that every initial goal is a flat goal.

Negative literals $\texttt{not}\,\texttt{p}$ which are *not* in the condition of an implication are treated as implications $\texttt{p} \rightarrow \texttt{false}$. Therefore, implications are either obtained from negative literals, or are integrity constraints in $\mathcal{IC}$, or are obtained from them by any number of applications of the inference rules of the proof procedure, given in definition 3.3 below.

Flat goals are a declarative representation of and-or trees of depth two, with atoms or implications as leaves of the trees. Until section 5, we will allow goals derived by the proof procedure to be more general formulae, corresponding to and-or trees of any depth, with atoms or implications as leaves.

The proof procedure is a rewriting procedure, consisting of a number of inference rules, each of which replaces a goal G_i by a goal G_{i+1} which is equivalent to G_i in the theory $\mathcal{T}$ of the given SLP knowledge base $\langle \mathcal{T}, \mathcal{IC} \rangle$. Each inference rule of the proof procedure is applied to a sub-formula of G_i which is a conjunction of atoms and implications.

[2] *Implications* have the same syntax of integrity constraints, except that in addition to universally quantified variables, they can also contain existentially quantified or free variables, occurring elsewhere in the same goal.

Definition 3.3. A *derivation* of a goal G_n from an initial goal G_0 is a finite sequence of goals $G_0, G_1, \ldots, G_n$ where $G_1 = G_0 \wedge \mathcal{IC}$ and, for $0 < i < n$, G_{i+1} is obtained from G_i by applying one of the following inference rules to a sub-formula S of G_i which is a conjunction of atoms and implications:

- *Unfolding*: given a reducible atom A in S
  and an accessible definition $H \leftrightarrow D_1 \vee \ldots \vee D_m$ in $\mathcal{T}$
  such that $A = H\sigma$ for some substitution σ, then
  - if A is a conjunct of S then, G_{i+1} is G_i with A in S replaced by $(D_1 \vee \ldots \vee D_m)\sigma$;
  - if A is a conjunct in the condition of an implication $L_1 \wedge \ldots \wedge A \wedge \ldots \wedge L_k \rightarrow C$ in S, then G_{i+1} is G_i with the implication in S replaced by
    $[L_1 \wedge \ldots \wedge D_1\sigma \wedge \ldots \wedge L_k \rightarrow C] \wedge \ldots \wedge [L_1 \wedge \ldots \wedge D_m\sigma \wedge \ldots \wedge L_k \rightarrow C]$
- *Propagation*: given implications
  $C_1 \rightarrow p(t_1, \ldots, t_n)$ with C_1 a (possibly empty) conjunction of suspended atoms
  and $p(s_1, \ldots, s_n) \wedge C_2 \rightarrow B$, both of which are conjuncts in S,
  then G_{i+1} is G_i with the *resolvent* $C_1 \wedge s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge C_2 \rightarrow B$ added as a conjunct of S.
- *Logical equivalence transformations*: G_{i+1} is G_i with
  $[A \vee B] \wedge C$ replaced by $[A \wedge C] \vee [B \wedge C]$ (called *splitting*)
  $\texttt{not}\ A_1 \wedge \ldots \wedge \texttt{not}\ A_n \rightarrow B$ replaced by $B \vee A_1 \vee \ldots \vee A_n$ (called *negation rewriting*)
  $A \wedge \texttt{false}$ replaced by $\texttt{false}$
  $A \vee \texttt{false}$ replaced by A, etc.
- *Equality rewriting*, implementing the unification algorithm of [21] and the application of substitutions.

These inference rules are similar to those of the IFF proof procedure of Fung and Kowalski [8] and SLDNFA of Denecker and De Schreye [6]. Unfolding and propagation are similar to unfolding and propagation in the IFF procedure. Logical equivalence and equality rewriting are as in the IFF procedure. The IFF procedure has two additional inference rules (see section 5).

As in ordinary LP, unfolding is a form of backward reasoning. However, whereas in ordinary LP atoms and heads of definitions in the logic program are unified, in SLP atoms need to be instances of heads of accessible iff-definitions. Therefore, unfolding in SLP is a form of one-way unification. Note that atoms are suspended iff they cannot be unfolded.

Propagation is a form of resolution. Like resolution, it is used to test formulae (in our case, goals) for satisfiability (or consistency). Consequently, there is no guarantee that propagation will always terminate. However, restrictions can be placed upon propagation to increase the likelihood of termination and to eliminate redundancies. The most natural restriction is to apply propagation only when C_1 is empty. This is a form of forward reasoning (also called P_1-resolution). Note that the equalities in the implication resulting from propagation may allow instantiation of suspended atoms in C_1 so that they become reducible.

Logical equivalence transformations are used to simplify goals. Splitting is a declarative representation of branching in ordinary LP.

Equality rewriting can be regarded, conceptually, as propagation with CET and unfolding with the iff-definitions `t=t` $\leftrightarrow$ `true`, for all variable-free terms `t`, augmented with garbage collection of redundant equalities. Moreover, equality rewriting applies substitutions, for example by rewriting an implication $\mathtt{X} = \mathtt{a} \rightarrow \mathtt{p}(\mathtt{X})$, with `X` universally quantified with scope the implication, as $\mathtt{p}(\mathtt{a})$.

Repeated applications of unfolding can generate multiple nestings of disjunctions. For example, consider the iff-definition

$\mathtt{even(X)} \leftrightarrow \mathtt{X{=}0} \lor \mathtt{[X{=}s(X')} \land \mathtt{even(X')]}$

and the initial goal $\mathtt{even(10)} \land \mathtt{G'}$, for some conjunction of literals $\mathtt{G'}$. Then, repeatedly unfolding gives

$\mathtt{(\ 10{=}0} \lor \mathtt{[10{=}s(X')} \land \mathtt{even(X')]\ )} \land \mathtt{G'}$

$\mathtt{(\ 10{=}0} \lor \mathtt{[10{=}s(X')} \land \mathtt{(\ X'{=}0} \lor \mathtt{[X'{=}s(X'')} \land \mathtt{even(X'')]\ )]\ )} \land \mathtt{G'}$

$\vdots$

It is easy to refine the notion of derivation to ensure that every goal in a derivation is a disjunction of flat goals. Indeed, it is sufficient to impose the restriction that every unfolding step is followed by a splitting step. The definition above of unfolding atoms in the condition of implications implicitly incorporates a logical equivalence transformation step, to guarantee that the formulae replacing the original implication are implications in turn.

In the previous section 2, we have assumed, as given, a set $\mathcal{T}_c \subseteq \mathcal{T}_o$ of iff-definitions for the constraint predicates. However, rather than provide $\mathcal{T}_c$ explicitly, we may assume, equivalently, that a submodel $\mathcal{M}_c$ of the perfect model of $\mathcal{T}$ is given as their "definition". Thus, for any constraint predicate `p`, the atom `p(t)` is unfolded to `true` if `t` is variable-free and `p(t)` is true in $\mathcal{M}_c$, `p(t)` is unfolded to `false` if `t` is variable-free and `p(t)` is false in $\mathcal{M}_c$, `p(t)` is unfolded to a conjunction of equalities σ for the variables in `t` if $\mathtt{p(t)}\sigma$ is the *unique* instance of `p(t)` that is true in $\mathcal{M}_c$, and `p(t)` is suspended otherwise. For example, $\mathtt{3} \leq \mathtt{10}$ and $\mathtt{2} \leq \mathtt{0}$ are unfolded to `true` and `false`, respectively, and $\mathtt{X} \leq \mathtt{3}$, $\mathtt{3} \leq \mathtt{X}$ and $\mathtt{X} \leq \mathtt{Y}$ are suspended. Furthermore, `plus(2,3,0)`, `plus(2,2,4)`, `plus(2,2,X)`, `plus(X,2,3)` and `plus(2,X,5)` are unfolded to `false`, `true`, `X=4`, `X=1` and `X=3`, respectively, and `plus(2,X,Y)` is suspended.

Definition 3.4. A *successful derivation* for an initial goal G_0 is a derivation $G_0, \ldots, G_n = D \lor Rest$ such that $D \neq$`false` and such that no inference rule can be applied to D. [3]

Given a successful derivation $G_0, \ldots, G_n = D \lor Rest$ for an initial goal G_0, then the set of all atoms and *denials* (implications with `false` as conclusion) in D is a *computed answer* to G_0.

Suspension is used not only to control non-determinism, but also to stop the computation at a point where the derived answer is more compact and more informative than it would be if it were unfolded further. For example, the answer $\mathtt{0} \leq \mathtt{X} \leq \mathtt{10}$ is more compact than the disjunction

[3]Here we intend that in a successful derivation any application of propagation to a pair of implications is performed at most once.

of the answers `X=0`, ..., `X=10`. The IFF proof procedure of [8] defines a computed answer for G_0 to be the set of all atoms in a disjunct D with all variables in D instantiated to variable-free terms in such a way that all denials in D are satisfied. We do not instantiate variables in answers because this would lead to less informative answers. As a consequence, we also keep denials as part of the answer.

The proof procedure is sound in the following sense:

Theorem 3.1. *If D is a computed answer for the initial goal G, then*

(S1) *D is a conjunction of suspended atoms and denials (with only suspended atoms in their condition)*
(S2) $\mathcal{T} \models \tilde{\forall}[D \rightarrow G]$
(S3) $\mathcal{IC} \cup CET \cup \tilde{\exists} D$ *is consistent.*

Condition (S1) holds trivially by definition of computed answer. Condition (S2) follows from the assumption that $\mathcal{T} \models \mathcal{IC}$ and from the fact that each of the inference rules preserves equivalence with respect to the given SLP theory $\mathcal{T}$. Condition (S3) follows from the fact that propagation verifies consistency.

Condition (S1) generalises condition (A1) in ALP, that abductive explanations consist of abducible (external) atoms only. However, whereas abducible atoms in abductive answers must be variable-free, answers in SLP can contain variables. Condition (S2) generalises condition (A2) in ALP, that the goal is entailed by the given theory extended with the abductive answer. Condition (S3) is a weak version of the "consistency view" of integrity satisfaction in condition (A3) of ALP, that the union of the given theory, the integrity constraints and the abductive answer is consistent. Indeed, the direct generalisation of (A3) in SLP would be that $\mathcal{T} \cup CET \cup \mathcal{IC} \cup \tilde{\exists} D$ is consistent, or equivalently, since $\mathcal{T} \models \mathcal{IC}$ and $\mathcal{T} \models CET$, that

(S3′) $\mathcal{T} \cup \tilde{\exists} D$ is consistent.

However, this condition is too strong for our proof procedure because suspended atoms are tested for consistency relative to the integrity constraints, which only approximate the theory. For example, given the goal

$\texttt{X} \leq \texttt{10} \wedge \texttt{X} \geq \texttt{20}$

and no integrity constraints, the computed answer is the goal itself, which is inconsistent with every correct theory for $\leq$.

Condition (S3′) is guaranteed to hold, however, if the integrity constraints provide a complete axiomatisation of the theory (in CLP terminology, they are *satisfaction complete*). This is not usually the case.

4. Applications

Possible applications of SLP include constraint satisfaction problems (such the already discussed n-queens problems), configuration problems, operations research problems (such as job-shop scheduling and warehouse location problems), and temporal reasoning and theories of action in artificial intelligence.

Example 4.1. (Configuration)
Consider the problem of configuring a computer system. The possible choices for the components (processor, monitor, memory, operating system, etc.) can be specified by iff-definitions, e.g.

$\texttt{processor(X)} \leftrightarrow \texttt{X = pentium} \vee \texttt{X = sparc} \vee \ldots$

$\texttt{operating\_system(X)} \leftrightarrow \texttt{X = os2} \vee \texttt{X = unix} \vee \ldots$

These definitions might be accessible or not. System constraints and personal preferences can be represented by integrity constraints, in the form of both positive requirements and denials, e.g.

$\texttt{processor(sparc)} \rightarrow \texttt{operating\_system(unix)}$

$\texttt{processor(sparc)} \wedge \texttt{operating\_system(os2)} \rightarrow \texttt{false}.$

If the definition of `processor` is accessible, then the query

$\texttt{processor(X)} \wedge \texttt{operating\_system(os2)}$

results in the computed answer `X=pentium`. The query

$\texttt{processor(sparc)} \wedge \texttt{operating\_system(os2)}$

results in no computed answer. If the definition of `processor` is inaccessible, then the query

`processor(sparc)`

results in the computed answer $\texttt{processor(sparc)} \wedge \texttt{operating\_system(unix)}$.

Example 4.2. (Job-shop scheduling)
An operation corresponds to the execution of a job on a machine. The problem of ordering two operations (on the same machine M) with starting times $\texttt{X}_{\texttt{1M}}$ and $\texttt{X}_{\texttt{2M}}$ and with processing times $\texttt{P}_{\texttt{1M}}$ and $\texttt{P}_{\texttt{2M}}$, respectively, can be expressed by the iff-definition

$\texttt{order}(\texttt{X}_{\texttt{1M}},\texttt{P}_{\texttt{1M}},\texttt{X}_{\texttt{2M}},\texttt{P}_{\texttt{2M}}) \leftrightarrow (\ \texttt{X}_{\texttt{1M}} + \texttt{P}_{\texttt{1M}} \le \texttt{X}_{\texttt{2M}}\ ) \ \vee\ (\ \texttt{X}_{\texttt{2M}} + \texttt{P}_{\texttt{2M}} \le \texttt{X}_{\texttt{1M}}\ ).$

Suppose there are two jobs and two machines, and therefore four operations `(1,1)`, `(2,1)`, `(1,2)` and `(2,2)`. Suppose the operations have starting times $\texttt{X}_{11}, \texttt{X}_{21}, \texttt{X}_{12}, \texttt{X}_{22}$ and processing times 2, 4, 6, 5, respectively. Finally, suppose that 8 is the latest starting time for operation `(1,1)`, 5 is the earliest starting time for `(2,1)`, 0 is the earliest starting time for `(2,2)` and operation `(1,2)` must be executed before operation `(1,1)`. Then, the problem of ordering the operations can be expressed by the initial goal

$\texttt{order}(\texttt{X}_{11}, 2, \texttt{X}_{21}, 4) \wedge \texttt{order}(\texttt{X}_{12}, 6, \texttt{X}_{22}, 5) \wedge \texttt{C}$

where `C` is

$\texttt{X}_{11} \le 8 \ \wedge\ 5 \le \texttt{X}_{21} \ \wedge\ 0 \le \texttt{X}_{22} \ \wedge\ \texttt{X}_{12} + 6 \le \texttt{X}_{11}.$

Let $\mathcal{IC}$ be

$\texttt{C}_1 \le \texttt{X} \wedge \texttt{X} + \texttt{C}_2 \le \texttt{Y} \rightarrow \texttt{C}_1 + \texttt{C}_2 \le \texttt{Y}$

$\texttt{X} \le \texttt{Y} \wedge \texttt{Y} < \texttt{X} \rightarrow \texttt{false}.$

Unfolding reduces the initial goal conjoined with $\mathcal{IC}$ to

JSS-1: $(\ \texttt{X}_{11} + 2 \le \texttt{X}_{21} \ \vee\ \texttt{X}_{21} + 4 \le \texttt{X}_{11}\ ) \ \wedge \texttt{order}(\texttt{X}_{12}, 6, \texttt{X}_{22}, 5) \ \wedge\ \texttt{C} \ \wedge\ \mathcal{IC}.$

Splitting gives

$(\mathtt{X}_{11} + 2 \leq \mathtt{X}_{21} \ \wedge\ \mathtt{order}(\mathtt{X}_{12}, 6, \mathtt{X}_{22}, 5) \ \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}) \vee$
$(\mathtt{X}_{21} + 4 \leq \mathtt{X}_{11} \ \wedge\ \mathtt{order}(\mathtt{X}_{12}, 6, \mathtt{X}_{22}, 5) \ \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}).$

Propagation in the second disjunct gives

$(\mathtt{X}_{11} + 2 \leq \mathtt{X}_{21} \ \wedge\ \mathtt{order}(\mathtt{X}_{12}, 6, \mathtt{X}_{22}, 5) \ \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}) \vee$
$(\mathtt{X}_{21} + 4 \leq \mathtt{X}_{11} \ \wedge\ 9 \leq \mathtt{X}_{11} \ \wedge\ \mathtt{false} \ \wedge\ \mathtt{order}(\mathtt{X}_{12}, 6, \mathtt{X}_{22}, 5) \ \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}).$

Logical equivalence transformation in the second disjunct gives

JSS-2: $\mathtt{X}_{11} + 2 \leq \mathtt{X}_{21} \ \wedge\ \mathtt{order}(\mathtt{X}_{12}, 6, \mathtt{X}_{22}, 5) \ \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}.$

Unfolding gives

$\mathtt{X}_{11} + 2 \leq \mathtt{X}_{21} \ \wedge\ (\mathtt{X}_{12} + 6 \leq \mathtt{X}_{22} \ \vee\ \mathtt{X}_{22} + 5 \leq \mathtt{X}_{12}) \ \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}.$

Splitting gives

$(\mathtt{X}_{11} + 2 \leq \mathtt{X}_{21} \ \wedge\ \mathtt{X}_{12} + 6 \leq \mathtt{X}_{22} \ \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}) \vee$
$(\mathtt{X}_{11} + 2 \leq \mathtt{X}_{21} \ \wedge\ \mathtt{X}_{22} + 5 \leq \mathtt{X}_{12} \ \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}).$

Three applications of propagation in the second disjunct give

$(\mathtt{X}_{11} + 2 \leq \mathtt{X}_{21} \ \wedge\ \mathtt{X}_{12} + 6 \leq \mathtt{X}_{22} \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}) \vee$
$(\mathtt{X}_{11} + 2 \leq \mathtt{X}_{21} \ \wedge\ \mathtt{X}_{22} + 5 \leq \mathtt{X}_{12} \ \wedge\ \mathtt{X}_{22} + 11 \leq \mathtt{X}_{11} \wedge 11 \leq \mathtt{X}_{11} \wedge\ \mathtt{false} \ \wedge\ \mathtt{C} \ \wedge\ \mathcal{IC}).$

Logical equivalence transformation in the second disjunct gives the computed answer

$(\mathtt{X}_{11} + 2 \leq \mathtt{X}_{21} \ \wedge\ \mathtt{C} \ \wedge\ \mathtt{X}_{12} + 6 \leq \mathtt{X}_{22}).$

In order to simulate efficient operational research algorithms for job-shop scheduling, the basic proof procedure needs to be extended with a more sophisticated form of propagation ("CPD", see section 5).

Example 4.3. (Temporal reasoning)
The event calculus [18] is a logic programming formalism for temporal reasoning. A simplified version of the event calculus can be expressed as an SLP knowledge base with iff-definitions:

$\mathtt{holds\_at}(\mathtt{P}, \mathtt{T}_2) \leftrightarrow \mathtt{happens}(\mathtt{E}, \mathtt{T}_1) \wedge \mathtt{T}_1 < \mathtt{T}_2 \wedge \mathtt{initiates}(\mathtt{E}, \mathtt{P}) \wedge \mathtt{not\ broken}(\mathtt{T}_1, \mathtt{P}, \mathtt{T}_2)$
$\mathtt{broken}(\mathtt{T}_1, \mathtt{P}, \mathtt{T}_2) \leftrightarrow \mathtt{happens}(\mathtt{E}, \mathtt{T}) \wedge \mathtt{terminates}(\mathtt{E}, \mathtt{P}) \wedge \mathtt{T}_1 < \mathtt{T} < \mathtt{T}_2.$

The first expresses that a property P holds at some time T_2 if it is initiated by an event E at some earlier time T_1 and is not broken (i.e. persists) from T_1 to T_2. The second expresses that a property P is broken (i.e. does not persist) from a time T_1 to a later time T_2 if an event E that terminates P happens at a time T between T_1 and T_2.

The predicate `happens` is external, with integrity constraint

$\mathtt{happens}(\mathtt{E}, \mathtt{T}) \wedge \mathtt{preconditions}(\mathtt{E}, \mathtt{T}, \mathtt{P}) \wedge \mathtt{not\,holds\_at}(\mathtt{P}, \mathtt{T}) \rightarrow \mathtt{false}$

expressing that an event E cannot happen at a time T if the preconditions P of E do not hold at time T.

The predicates `preconditions`, `initiates` and `terminates` are ordinary, e.g.

$\mathtt{preconditions}(\mathtt{carry\_umbrella}, \mathtt{T}, \mathtt{P}) \leftrightarrow \mathtt{P} = \mathtt{own\_umbrella} \vee \mathtt{P} = \mathtt{borrowed\_umbrella}$
$\mathtt{initiates}(\mathtt{rain}, \mathtt{raining}) \leftrightarrow \mathtt{true}$
$\mathtt{terminates}(\mathtt{sun}, \mathtt{raining}) \leftrightarrow \mathtt{true}.$

Additional integrity constraints might be given to represent reactive behaviour of agents, e.g.

$\mathtt{happens}(\mathtt{rain}, \mathtt{T}) \rightarrow \mathtt{happens}(\mathtt{carry\_umbrella}, \mathtt{T} + 1)$

and to prevent concurrent execution of actions (events):

$\mathtt{happens(E,T)} \wedge \mathtt{happens(E',T)} \rightarrow \mathtt{E} = \mathtt{E'}.$

Atoms in an answer computed by the proof procedure can be interpreted as actions to be performed to achieve given goals and/or to react to given events. Denials in an answer can be interpreted as prohibitions. For example, given the goal

$\mathtt{holds\_at(raining,3)} \wedge \mathtt{happens(sun,2)}$

the proof procedure computes the only answer

$\mathtt{happens(rain,T)} \wedge \mathtt{T} < 3 \wedge (\mathtt{T} < 2 \rightarrow \mathtt{false}) \wedge \mathtt{happens(sun,2)}.$

Indeed, unfolding $\mathtt{holds\_at(raining,3)}$ gives

$\mathtt{happens(rain,T)} \wedge \mathtt{T} < 3 \wedge (\mathtt{broken(T,raining,3)} \rightarrow \mathtt{false}).$

Then, two applications of unfolding, starting with broken(T,raining,3), gives

$\mathtt{happens(rain,T)} \wedge \mathtt{T} < 3 \wedge (\mathtt{happens(sun,T')} \wedge \mathtt{T} < \mathtt{T'} < 3 \rightarrow \mathtt{false}).$

Then, propagating with happens(sun,2) gives the answer.

Given the goal

$\mathtt{happens(rain,5)}$

the proof procedure computes $\mathtt{happens(carry\_umbrella,6)}$.

The proof procedure presented in this paper has been used as the inference engine of the agent architecture of [17]. There, the proof procedure is interleaved with interaction with the environment so that reactive behaviour can be achieved.

5. Improving the proof procedure

Propagation, which checks the consistency of goals with integrity constraints, may not terminate. For example, given the goal (possibly obtained by several applications of inference rules)

$\mathtt{G} = (\mathtt{V} > 2 \rightarrow \mathtt{false}) \wedge (\mathtt{X} > \mathtt{Y} \wedge \mathtt{Y} > \mathtt{Z} \rightarrow \mathtt{X} > \mathtt{Z})$

unrestricted propagation as defined in section 3 would generate an infinite sequence of goals

$\mathtt{G} \wedge (\mathtt{V} > \mathtt{Y} \wedge \mathtt{Y} > 2 \rightarrow \mathtt{false})$

$\mathtt{G} \wedge (\mathtt{V} > \mathtt{Y} \wedge \mathtt{Y} > 2 \rightarrow \mathtt{false}) \wedge (\mathtt{V} > \mathtt{Y'} \wedge \mathtt{Y'} > \mathtt{Y} \wedge \mathtt{Y} > 2 \rightarrow \mathtt{false})$

$\vdots$

Such non-termination can be avoided, for example by restricting SLP knowledge bases so that both iff-definitions and integrity constraints are recursion-free. In general, however, propagation needs to be refined both to improve termination and to improve efficiency. Since propagation is a form of resolution, any resolution refinement can be used. For example, using P_1-resolution, propagation could be applied only when C_1 is empty. Other resolution strategies, such as subsumption and ordering strategies, can also be employed. In the general case, however, because satisfiability is not semi-decidable, termination can not always be guaranteed.

Additional inference rules are needed to guarantee the practicality of the proof procedure. Given the goal

$\mathtt{position(tom,X)} \wedge (\mathtt{X} = \mathtt{manager} \rightarrow \mathtt{bonus(tom,20)})$

no inference rule can be applied, and the atoms in the goal are returned as an answer. Fung and Kowalski [8] employ an additional inference rule (*case analysis*) giving two answers

$(\texttt{position}(\texttt{tom}, \texttt{X}) \wedge \texttt{X} = \texttt{manager} \wedge \texttt{bonus}(\texttt{tom}, \texttt{20}))$ and

$(\texttt{position}(\texttt{tom}, \texttt{X}) \wedge (\texttt{X} = \texttt{manager} \rightarrow \texttt{false}))$.

Another useful operation, which generalises subsumption, is *deletion*, i.e. elimination of logically redundant information that can be recovered at any time. For example, given the goal

$\texttt{X} > 3 \wedge \texttt{X} > 100 \wedge (\texttt{X} > \texttt{Y} \wedge \texttt{Y} > \texttt{Z} \rightarrow \texttt{X} > \texttt{Z})$

the atom $\texttt{X} > 3$ is logically entailed by $\texttt{X} > 100$ and transitivity of $>$. Therefore, $\texttt{X} > 3$ can be deleted from the goal. We have not yet identified appropriate criteria to distinguish when integrity constraints are to be used for propagation and when for deletion.

The splitting rule can give rise to an exponential explosion in the size of goals. A way to limit or even avoid splitting completely is to apply propagation within and across disjunct boundaries. This is a form of non-clausal resolution. We call our variant of such non-clausal resolution *Constraint Propagation with Disjunctions (CPD)*.

Let us illustrate CPD via the job-shop scheduling problem of section 4. Consider the goal JSS-1. Propagating the global constraints (see definition 3.2) C with the second disjunct of the first conjunct using $\mathcal{IC}$ avoids splitting completely and gives (after logical equivalence transformations) JSS-2. We call this kind of CPD *global-to-local*. Global-to-local CPD does not avoid splitting in many cases. However, its application might generate, within the disjuncts, tighter constraints to be used for further propagation immediately after splitting takes place.

Propagation as defined in the basic proof procedure can be thought of as *local-to-local* CPD if applied within a disjunct, and *global-to-global* CPD if applied within the global constraints.

A fourth form of CPD, *local-to-global* CPD, can be used to extract local information from disjuncts and make it global to the disjunction. For example, consider the job-shop scheduling problem in the previous section, and the goal

$(\ \texttt{X}_{12} + 6 \leq \texttt{X}_{22}\ \vee \texttt{X}_{22} + 5 \leq \texttt{X}_{12}\ )\ \wedge \texttt{C}'$

where $\texttt{C}'$ is

$\texttt{X}_{12} + 6 \leq \texttt{X}\ \wedge\ \texttt{X}_{22} + 5 \leq \texttt{X}\ \wedge\ 0 \leq \texttt{X}_{22}\ \wedge\ 0 \leq \texttt{X}_{12}$.

X can be thought of as the starting time of a dummy operation that must follow all other operations. Any lower bound for X is therefore a lower bound for the time of the overall schedule. Two steps of global-to-local CPD introduces $11 \leq \texttt{X}$ in each disjunct. Local-to-global CPD then factors $11 \leq \texttt{X}$ out of the disjuncts and adds it to $\texttt{C}'$. This information might be useful, in the context of an extended goal, to decide another disjunction using global-to-local CPD without splitting.

As an additional example, consider

$(\ \texttt{X} \leq 3 \vee 10 \leq \texttt{X}\ )\ \wedge\ (\ (\ 5 \leq \texttt{X} \wedge \texttt{X} \leq 8\ )\ \vee\ 100 \leq \texttt{X}\ )$.

Local-to-global CPD using the first conjunct can be used to generate the additional, top-most level conjunct

$\neg\ (\ 3 < \texttt{X} \wedge \texttt{X} < 10\ )$.

Global-to-local CPD introduces false into the first disjunct of the second conjunct and thus renders $100 \leq \texttt{X}$ a global constraint. Further global-to-local CPD introduces false into the first disjunct of the first conjunct in the initial goal. In this case, the use of CPD produces an answer without performing any (expensive) splitting. Of course, this is at the expense of employing (and controlling) the use of more powerful inference rules. We have not yet completely determined automatic control strategies for the application of these rules. However, experience with our prototypes (see section 7) and more generally with the use of similar rules in automated theorem-proving encourages us to believe that such efficient control strategies may be possible.

The formalisation of CPD is simplified when goals are disjunctions of flat goals.

Definition 5.1. Let

$G = C_1 \wedge \ldots \wedge C_n \wedge D_1 \wedge \ldots \wedge D_m$

be a flat goal, with global constraints $C_1, \ldots, C_n$ and disjunctions $D_1, \ldots, D_m$, and let

$D = A_1 \wedge \ldots \wedge A_r$

be a disjunct of some D_i. Then

- *Global-to-global CPD* is propagation between two global constraints C_s and C_t. The resolvent becomes an additional conjunct of G.
- *Global-to-local CPD* is propagation between a global constraint C_s and a local constraint A_t. The resolvent becomes an additional conjunct of D.
- *Local-to-local CPD* is propagation between two local constraints A_s and A_t. The resolvent becomes an additional conjunct of D.
- If all disjuncts of the disjunction D_i contain the same local constraint A_s, then, by *(simple) local-to-global CPD*, A_s becomes an additional conjunct of G.

Simple local-to-global CPD is the opposite of splitting. More complex forms of local-to-global CPD are also possible, as illustrated earlier.

6. Comparisons

The proof procedure originates from work on ALP and SQO. Comparisons with other approaches to ALP and SQO can be found in [8, 30]. Here we concentrate on comparisons with CLP.

The CPD methods can be viewed as a natural implementation, and in some sense a generalisation, of several methods of handling CLP *disjunctive constraints*, i.e. disjunctions of atomic constraints. These methods include CHIP forward-checking and look-ahead techniques [28], Generalised Propagation [20] and Constructive Disjunction [29, 11].

CLP(X) [9], the traditional approach to CLP, distinguishes between ordinary and constraint predicates. In our approach, constraint predicates and ordinary predicates are both treated as ordinary, accessible predicates defined by iff-definitions. Atoms of ordinary predicates are suspended when unfolding them would be combinatorially explosive. Both kinds of predicates can also be approximated by integrity constraints, which can be applied when atoms are suspended. As we have seen, constraint predicates can also be understood as being defined model-theoretically, as in CLP(X).

In CLP(X) knowledge is represented in the form of clauses of the form

$H \leftarrow B \wedge C$

where H is an atom of an ordinary predicate, B a conjunction of literals of ordinary predicates and C is a conjunction of literals of constraint predicates. Answers to goals (existentially quantified conjunctions of literals) are sets of constraints obtained by "unfolding" atoms of ordinary predicates and collecting any constraints encountered in the process in a separate store (initially empty) checked for consistency and simplified by a given *constraint solver*. Different instances of CLP(X) employ different constraint solvers. For example, a constraint solver for CLP($\mathcal{R}$) (the instance of CLP(X) for linear real arithmetic) might rely upon special-purpose built-in algorithms for solving systems of linear equations and inequalities — like Gaussian elimination or the Simplex algorithm.

In SLP, propagation with integrity constraints and evaluation in the submodel $\mathcal{M}_c$ of the perfect model of $\mathcal{T}$ play the role of the constraint solver in CLP(X). However, whereas the user has no knowledge and no control over the internal organisation of the constraint solver in CLP(X), i.e. CLP(X) is a *black-box* approach to CLP, users can choose their own set of integrity constraints in SLP.

A major theoretical disadvantage of black-box approaches to CLP is that to fully understand the meaning of a computed answer (or lack thereof), it is necessary to know the implementation details of the constraint solver. Moreover, the techniques integrated into the constraint solver may not be the most appropriate to solve a given problem efficiently. A number of non-black-box approaches to CLP have been proposed to overcome these deficiencies. *Glass-box* approaches rely upon the user to guide the constraint solver in finding a solution more efficiently. For example, the `cc(FD)` language [29] allows several (higher-level) operators (such as implication, propagation and cardinality) which the user can employ in the program. Similarly, CHIP [28] allows *disjunctive constraints* (i.e. disjunctions of constraints) to be labelled as *forward-checking* or *look-ahead* constraints. *Transparent* constraint solvers, whose organisation is known to the user, are further examples of the glass-box approach. The treatment of equality in LP and in SLP, via equality rewrite rules, is a concrete example of a transparent constraint solver. The *no-box* approach, a term coined by Frühwirth [7], constitutes the most complete departure from the original black-box approach, as it abandons the idea of a built-in constraint solver. Instead, the user specifies a set of constraint handling rules which implement a constraint solver explicitly, similarly to our integrity constraints. However, our approach uses a submodel (for constraint predicates) of the perfect model to evaluate reducible constraint atoms, and for this reason, it is not purely no-box.

SLP combines CLP and ALP. Other such combinations have been proposed, e.g. the language of [14]. This differs from ours in the same way that the instance of SLP for CLP differs from the conventional, black-box approach to CLP.

7. Conclusions

We have presented Suspended Logic Programming, an extension of ordinary LP including, in addition to ordinary LP clauses, integrity constraints, direct representation of disjunction in the bodies of clauses and in goals, and suspension of atoms as in concurrent languages. We have argued that SLP unifies and generalises Constraint Logic Programming, Abductive Logic Programming and Semantic Query Optimisation. We have presented a proof procedure for the new framework, based upon previously proposed proof procedures for ALP.

In the presentation of the proof procedure we have concentrated on its logic and ignored control issues. Any concrete implementation of the proof procedure needs to employ specific control strategies (Algorithm = Logic + Control [15]). In our experience, any such strategies should delay splitting as much as possible, in favour of simple propagation and CPD.

To date, two prototype implementations of SLP have been developed [30]. The implementations include several of the extensions proposed in section 5 and have been tested on example applications. The table below shows some computational results for two version of the n-queens problem, the first using the single integrity constraint given in example 2.4 and the second using the two integrity constraints

`queen(A,B)` $\wedge$ `move((A,B),(C,D))` $\rightarrow$ `attacked(C,D)`

`queen(C,D)` $\wedge$ `attacked(C,D)` $\rightarrow$ `false`

with `attacked` an external predicate.

n	#S	1-r., no CPD		1-r., g-t-l CPD			2-r., full CPD	
4	2	0.03	45	0.03	5	(5)	0.14	3
5	10	0.11	176	0.14	13	(13)	1.43	9
6	4	0.48	745	0.45	39	(39)	5.44	20
7	40	2.20	3072	1.98	107	(107)	53.95	69
8	92	10.80	13755	8.59	415	(383)	386.63	252
9	352	56.00	64336	43.54	1641	(1477)	n/a	n/a
10	724	304.90	313335	205.88	6665	(5715)	n/a	n/a

The first two columns specify the number of queens n and the corresponding number of possible solutions #S, respectively. The next three columns give the processing times (in seconds on a Sun UltraSparc) and numbers of splitting operations for, respectively, a solution using the single integrity constraint (1-r), first without making use of CPD, then with global-to-local CPD, and a solution using the two integrity constraints (2-r), with full application of all CPD methods including local-to-global CPD. The numbers in brackets in the fourth column are the numbers of splitting operations performed when employing the heuristic to always choose for splitting the disjunction with the smallest number of disjuncts. This strategy implements the well-known "most-constrained variable" heuristic which has been successfully used in CLP to speed up the solving of many constraint satisfaction problems. Note that such heuristics are not expressed in the framework itself, rather they are control matters that are part of the implementation, which can be made options for the user to switch on or off.

The computational results obtained in this and similar experiments indicate that a careful usage of the CPD methods cuts down both the size of the search tree (quite dramatically) and the computation time (not quite so dramatically). The table also shows that the benefits of using global-to-local CPD in the one-rule version increase with the problem size — computation times are almost the same for $n = 4, \ldots, 6$, but for $n = 8, \ldots, 10$ the gain is, roughly, 20%, 25% and 30%, respectively. Similarly, the pruning factor of the search tree (i.e. the node count for the one-rule version without CPD divided by the node count for the one-rule version with global-to-local CPD) increases from 33 for $n = 8$ to 43 for $n = 9$ and 54 for $n = 10$. For $n = 11$ (results not reported in the table) the node count is more than 60 times higher without CPD and the time saved is about 60%.

A full application of the CPD methods, including local-to-local and local-to-global CPD, may further prune the search tree, but at least in the presently available implementations, this is accompanied by increasing computation times. For $n = 9$ and $n = 10$ no results were obtained at all because the program ran out of memory (this is indicated by "n/a", or "not available", in the table).

A more efficient implementation of SLP (based on the prototypes) is currently being developed and already yields significantly better results than those reported above [31].

Preliminary results [27] seem to indicate the suitability of the approach presented in this paper to express operational research algorithms. Future work should explore this area further.

We have assumed that SLP theories are locally stratified. More work is needed to study the non-locally stratified case.

Finally, SLP and concurrent LP employ on a similar notion of suspension. Unifying the two approaches might be another interesting topic for future research.

References

[1] Apt, K. R., Blair, H., Walker, A.: "Towards a theory of declarative knowledge", J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1988.

[2] Carlsson, M.: *Design and Implementation of an OR-Parallel Prolog Engine*, SICS Dissertation Series 02, The Royal Institute of Technology, Stockholm, Sweden, 1990.

[3] Chakravarthy, U. S., Grant, J. and Minker, J.: "Logic-Based Approach to Semantic Query Optimization", *ACM Transactions on Database Systems* **15(2)**, 1990, 162–207.

[4] Clark, K. L.: "Negation as failure", Gallaire, H. and Minker, J. (eds.), *Logic and Data Bases*, Plenum Press, 1978, 292–322.

[5] Console, L., Theseider Dupré, D. and Torasso, P.: "On the relationship between abduction and deduction", *Journal of Logic and Computation* **2(5)**, Oxford University Press, 1991, 661–690.

[6] Denecker, M. and De Schreye, D.: "SLDNFA: an abductive procedure for abductive logic programs", *Journal of Logic Programming* **34(2)**, Elsevier, 1997, 111–167.

[7] Frühwirth, T.: "Constraint Handling Rules", Podelski, A. (ed.), *Constraint Programming: Basic and Trends*, LNCS 910, Springer Verlag, 1995, 90–107.

[8] Fung, T. H. and Kowalski, R. A.: "The Iff Proof Procedure for Abductive Logic Programs", *Journal of Logic Programming* **33(2)**, Elsevier, 1997, 151–165.

[9] Jaffar, J. and Lassez, J.-L.: "Constraint Logic Programming", *Proc. of the 14^{th} ACM Symp. on the Principles of Programming Languages*, 1987, 111–119.

[10] Jaffar, J. and Maher, M.: "Constraint Logic Programming: A Survey", *Journal of Logic Programming* **19/20**, Elsevier, 1994, 503–581.

[11] Jourdan, J. and Sola, T.: "The Versatility of Handling Disjunctions as Constraints", Bruynooghe, M. and Penjam, J. (eds.), *Proc. of the 5^{th} Intern. Symp. on Programming Languages Implementation and Logic Programming*, Springer Verlag, 1993, 60–74.

[12] Kakas, A. C., Kowalski, R. A. and Toni, F.: "Abductive Logic Programming", *Journal of Logic and Computation* **2(6)**, Oxford University Press, 1992, 719–770.

[13] Kakas, A. C., Kowalski, R. A. and Toni, F.: "The role of abduction in logic programming", *Handbook of logic in Artificial Intelligence and Logic Programming* **5**, Oxford University Press, 1998, 235–324.

[14] Kakas, A. C. and Michael, A.: "Integrating Abductive and Constraint Logic Programming", Sterling, L. (ed.), *Proc. of the 12^{th} Int. Conf. on Logic Programming*, MIT Press, 1995, 399–413.

[15] Kowalski, R.A.: *Logic for problem solving*, Elsevier, 1979.

[16] Kowalski, R. A.: "A dual form of logic programming", Lecture Notes of the Workshop in Honour of Jack Minker, University of Maryland, 1992.

[17] Kowalski, R.A. and Sadri, F.: "Towards a unified agent architecture that combines rationality with reactivity", *Proc. International Workshop on Logic in Databases*, San Miniato (PI), Italy, LNCS 1154, Springer Verlag, 1996.

[18] Kowalski, R.A. and Sergot, M.: "A logic-based calculus of events", *New Generation Computing* **4**, 1986, 67–95.

[19] Kowalski, R. A., Toni, F. and Wetzel, G.: "Towards a declarative and efficient glass-box CLP language", Fuchs, N. and Gottlob, G. (eds.), *Workshop Logische Programmierung*, Zurich, 1994.

[20] Le Provost, T. and Wallace, M.: "Generalised Propagation Over the CLP Scheme", ECRC Technical Report 92-01, 1992.

[21] Martelli, A. and Montanari, U.: "An efficient unification algorithm", *ACM Trans. on Prog. Lang. and Systems* **4(2)**, 1982, 258–282.

[22] Naish, L.: "An Introduction to NU-PROLOG", Technical Report TR 82/2, University of Melbourne, 1982.

[23] Przymusinski, T.: "On the Semantics of Stratified Databases", J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1988.

[24] Saraswat, V. A. and Rinard, M.: "Concurrent Constraint Programming", *Proc. of the 17^{th} ACM Symp. on the Principles of Programming Languages*, 1990, 232–245.

[25] Schulte, C. and Smolka, G.: "Encapsulated Search for Higher-order Concurrent Constraint Programming", Bruynooghe, M. (ed.), *Proc. of the Int. Logic Programming Symposium*, MIT Press, 1994, 505–520.

[26] Shapiro, E.: "The Family of Concurrent Logic Programming Languages", *ACM Computing Surveys* **21 (3)**, 1989, 413–510.

[27] Toni, F.: "A theorem-proving approach to job-shop scheduling", Technical Report, Department of Computing, Imperial College, London, 1994.

[28] Van Hentenryck, P.: *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

[29] Van Hentenryck, P., Saraswat, V. A. and Deville, Y.: "Design, Implementation, and Evaluation of the Constraint Language `cc(FD)`", Podelski, A. (ed.): *Constraint Programming: Basic and Trends*, LNCS 910, Springer Verlag, 1995, 293–316.

[30] Wetzel, G.: *Abductive and Constraint Logic Programming*, Ph.D. Thesis, Imperial College, London, 1997.

[31] Wetzel, G.: "A Unifying Framework for Abductive and Constraint Logic Programming", Bry, F. et al. (eds.), *12th Workshop on Logic Programming (WLP'97)*, München 1997, 58–68.

[32] Wetzel, G.: "Using Integrity Constraints as Deletion Rules", Bonner, A. et al. (eds.), *DYNAMICS'97 — Proc. of the Workshop on (Trans)Actions and Change in Logic Programming and Deductive Databases*, Port Jefferson, NY, 1997, 147–161.

[33] Wetzel, G., Kowalski, R. A. and Toni, F.: "PROCALOG — Programming with Constraints and Abducibles in Logic", Maher, M. (ed.), *Proc. of the Joint Int. Conf. and Symp. on Logic Programming*, MIT Press, 1996, 535.

[34] Wetzel, G., Kowalski, R. A. and Toni, F.: "A Theorem-Proving Approach to CLP", Krall, A. and Geske, U. (eds.), *Workshop Logische Programmierung*, Vienna, 1995 63–72.