Annals of Mathematics and Artificial Intelligence 25 (1999) 165–199

Building a knowledge base: an example

Michael Gelfond [a] and Alfredo Gabaldon [b]

[a] *Department of Computer Science, University of Texas at El Paso, El Paso, TX 79968, USA*
E-mail: mgelfond@cs.utep.edu
[b] *Department of Computer Science, University of Toronto, Toronto, Canada M5S 3G4*
E-mail: alfredo@cs.toronto.edu

The main goal of this paper is to illustrate applications of some recent developments in the theory of logic programming to knowledge representation and reasoning in common sense domains. We are especially interested in better understanding the process of development of such representations together with their specifications. We build on the previous work of Gelfond and Przymusinska in which the authors suggest that, at least in some cases, a formal specification of the domain can be obtained from specifications of its parts by applying certain operators on specifications called *specification constructors* and that a better understanding of these operators can substantially facilitate the programming process by providing the programmer with a useful heuristic guidance. We discuss some of these specification constructors and their realization theorems which allow us to transform specifications built by applying these constructors to declarative logic programs. Proofs of two such theorems, previously announced in a paper by Gelfond and Gabaldon, appear here for the first time. The method of specifying knowledge representation problems via specification constructors and of using these specifications for the development of their logic programming representations is illustrated by design of a simple, but fairly powerful program representing simple hierarchical domains.

1. Introduction

"The only effective way to raise the confidence level of a program significantly is to give a proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand. ...If one first asks oneself what the structure of a convincing proof would be and, having found this, then construct a program satisfying this proof's requirements, then these correctness concerns turn out to be a very effective heuristic guidance."

E. Dijkstra, The Humble Programmer

The main goal of this paper is to illustrate applications of some recent developments in the theory of logic programming to knowledge representation and reasoning in common sense domains. The paper is written in the framework of the declarative logic programming paradigm which strives to reduce a substantial part of the pro-

gramming process to the description of objects comprising the domain of interest and relations between these objects. After such a description is produced by a programmer, it can be queried to establish truth or falsity of statements about the domain or to find objects satisfying various properties. The software development process in this paradigm starts with a natural language description of the domain which, after a necessary analysis and elaboration, is described in mathematical terms. The main purpose of this initial mathematical specification is to be understood by the people involved in the development of the system and to serve as a contract between the system specifiers and implementors. This specification is then gradually transformed by implementors into an executable program satisfying the specification. (The process will of course normally require more than one iteration.) There are many languages which can be used during various stages of this transformation. This paper adopts the view from [26] where the authors argue that for a rather broad class of knowledge representation problems the process of representing knowledge can be divided into three (mutually dependent) parts:

- Elaboration of a natural language description of the domain can be done in a language of functional specifications – possibly incomplete definitions of new relations between objects of the domain of discourse given in terms of the old, known relations.[1] These specifications can be defined by a specifier directly in a simple set-theoretic language, or they can be built from previously defined specifications with the help of specification constructors – simple functions from specifications to specifications. This method is applicable when our knowledge representation problem can be parameterized with respect to some possible inputs. (A good discussion on the importance of explicitly specifying input and output of knowledge representation problems in AI can be found in [30].)
- On the second stage implementors of the system are given the functional specification f constructed during the first stage of the process. Now they are confronted with the task of representing f in a logical language with a precisely described entailment relation. Gelfond and Przymusinska [26] advocate the use of a language A-Prolog of logic programs with two types of negation and the answer set semantics. The choice is determined by the ability of A-Prolog to represent default assumptions, i.e., statements of the form "Elements of the class A normally (typically, as a rule) have property P", epistemic statements "P is unknown", "P is possible", and other types of statements needed for describing common sense domains. Other important factors are the simplicity of the semantics, strong expressive power of the language, existence of a mathematical theory providing a basis for proving properties of programs in A-Prolog, and the availability of query answering systems which can be used for rapid prototyping.[2] There are other reasons but we will not

[1] In precise terms, a functional specification consists of two signatures σ_{i} and σ_{o} and a (possibly partial) function f which maps sets of ground literals over σ_{i} into sets of ground literals over σ_{o}.

[2] There is a variety of query-answering systems which can be used to compute answers to queries in A-Prolog. A simple meta-interpreter build on top of Prolog or XSB [10] proved to be sufficient for

discuss them in this paper. The alternative approach which uses logic programs with well-founded semantics and its extensions can be found in [1].
At the end of the second stage the implementor will have a logic program π which, taken in conjunction with a collection X of facts about known relations of f, will entail exactly those facts about the new relations which belong to $f(X)$. Programs of this sort are called lp-functions. In [26] the authors suggest that the construction of π from f can be substantially facilitated by mathematical results relating specification constructors to some operations on logic programs. Such results are called realization theorems. They can provide an implementor with a useful heuristic guidance and the means to establish correctness of his programs. Several examples of such theorems and their applications will be given in the following sections.

- At the last stage of the process, the lp-function π representing functional specification f will be transferred into an efficient logic program Π computing (or approximating) the entailment relation of π. Unlike π, the construction of Π will depend significantly on the choice of query answering system used by the implementors. For instance, the use of standard Prolog interpreter may require modification of π which includes the cut operator of Prolog to avoid loops or to improve efficiency. The use of XSB will avoid some of these problems but will require the choice of predicates to be tabled, etc.

Space limitations preclude us from giving any serious comparison of our approach with other methodologies of representing knowledge. Moreover, we believe that such a comparison can only be done when all of these methodologies are more fully developed. Still a short remark is in order. At the moment, the specification language most frequently used for the first formal refinement of a problem is probably the language of first-order logic (FOL). As others before us we conjecture that FOL is not fully adequate for our purpose. Its expressive power is insufficient to define even fairly simple f-specifications such as transitive closure of database relations. It also does not seem to be the best language for representing defaults, epistemic statements, and other types of "common sense" knowledge. These observations are well known and led to various extensions of FOL which allow nonmonotonic reasoning. Our preference for a less powerful A-Prolog is similar to our preference of Modula 2 over ADA. Since, as was shown in [7,22], programs of A-Prolog can be viewed as theories of Reiter's default logic [40], we hope that those who prefer power over simplicity can still benefit from this work. The next question to answer is "Why not to use the language of logic programming during the first stage of the development process?" What are the advantages of first formulating the problem in terms of func-

the purposes of this paper. Description of such a meta-interpreter build on top of SLDNF of Prolog can be found in section 4. For more complicated programs, especially for those which have multiple stable models, this interpreter will go into a loop or answer *unknown* to too many interesting queries. Roughly speaking this happens because Prolog and XSB do not allow reasoning by cases. This type of reasoning is incorporated in a more powerful system, called SLG [9], which was successfully used to answer queries in the presence of multiple stable models. A different type of engine, [11,17,36], are based on algorithms for computing stable models of programs without function symbols.

tional specifications? There are two reasons why we do that. The first advantage of the language of f-specifications over A-Prolog is its simplicity. The construction of f requires knowledge of a simple set-theoretic notation together with definitions of a (hopefully small) collection of specification constructors. The specifier involved at the first stage of the process does not need to know anything about the semantics of A-Prolog. Another possible advantage of translating a natural language description of a problem into an f-specification f is the ability to use the structure of f and the corresponding realization theorems for reducing the construction of π_f to the construction of simpler programs. One of the main goals of this paper is to demonstrate, by way of example, how this can be achieved. Finally, we want to mention that in this paper we are mainly interested in the first two stages of the development process. In this sense our approach is complementary to the work on program development in Prolog (see, for instance, [14]). (See, however, proposition 13.)

The paper is organized as follows. In the next two sections we review the answer set semantics of logic programs and the notions of f-specification and lp-function. In the next section we give a first example of applying our methodology to solving a knowledge representation problem associated with a simple type of taxonomic hierarchy. (This and other problems related to inheritance reasoning are commonly used to test strengths and weaknesses of various nonmonotonic formalisms.) We start with specifying the problem in terms of functional specifications, representing it by a declarative logic program, and showing that this program can be correctly executed by the Prolog interpreter. In the process we introduce a specification constructor, called *incremental extension*, and give a realization theorem for it. The knowledge representation problem discussed in this section assumes completeness of knowledge. In the next section we consider the situation when this assumption is relaxed and our knowledge of some relations from the domain of the problem becomes incomplete. The new specification is obtained from the original one by applying a specification constructor called *input opening*. We discuss the definition of this constructor, its decomposition into two simpler constructors, *interpolation* and *domain completion*, and the corresponding realization theorems. (These theorems were stated in [20] without proofs, and hence their proofs are included in this paper.) Finally, in the last section, we discuss one more knowledge representation problem defined in terms of another specification constructor, called *input extension*. Somewhat surprisingly, the resulting class of programs formalizes inheritance reasoning with incomplete information which was not previously considered.

2. Logic programs

In this section we give a brief introduction to the answer set semantics for logic programs with two kinds of negation [23].[3] For a more detailed discussion see [5,31].

[3] For programs without classical negation the answer set semantics coincides with the stable model semantics of [21].

The language of a logic program, like a typed first-order language, is determined by its signature σ, consisting of types (*types*(σ)), object constants for each type τ (*obj*(τ, σ)), and typed function and predicate constants (*func*(σ) and *pred*(σ), respectively). Signature σ_1 is a sub-signature of signature σ_2 if *types*(σ_1) $\subseteq$ *types*(σ_2), *func*(σ_1) $\subseteq$ *func*(σ_2), *pred*(σ_1) $\subseteq$ *pred*(σ_2), and for each type τ which belongs to both signatures, *obj*(τ, σ_1) = *obj*(τ, σ_2); $\sigma = \sigma_1 \cup \sigma_2$ if elements of σ are unions of the corresponding elements of σ_1 and σ_2. Terms are built as in the corresponding typed first-order language; positive literals (or atoms) have the form $p(t_1, \ldots, t_n)$, where t's are terms of proper types and p is a predicate symbol of arity n; negative literals are of the form $\neg p(t_1, \ldots, t_n)$. (Logical connective $\neg$ in the context of logic programming is called "classical", or "explicit", or "strong" negation.) Literals of the form $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ are called contrary. By $\overline{l}$ we denote a literal contrary to l. Literals and terms not containing variables are called ground. The sets of all ground terms, atoms and literals over σ will be denoted by *terms*(σ), *atoms*(σ) and *lit*(σ), respectively. For a set P of predicate symbols from σ, *atoms*(P, σ) (*lit*(P, σ)) will denote the sets of ground atoms (literals) of σ formed with predicate symbols from P. Consistent sets of ground literals over signature σ are called *states* of σ and denoted by *states*(σ).

A rule is an expression of the form

$$l_0 \leftarrow l_1, \ldots, l_m, \mathit{not}\, l_{m+1}, \ldots, \mathit{not}\, l_n, \tag{1}$$

where $n \geqslant 0$, l_i's are literals and *not* is a logical connective called *negation as failure* [12,39].

A pair $\{\sigma, \pi\}$ where σ is a signature and π is a collection of rules over σ is called a *logic program*. (We often denote such pairs by their second element π. The corresponding signature will be denoted by $\sigma(\pi)$.)

If r is a rule of type (1) then *head*(r) = $\{l_0\}$, *pos*(r) = $\{l_1, \ldots, l_m\}$, *neg*(r) = $\{l_{m+1}, \ldots, l_n\}$. For any program π, *head*(π) = $\bigcup_{r \in \pi}$ *head*(r). Similarly, for *pos* and *neg*. Sometimes, $\mathit{not}\, l_i, \ldots, \mathit{not}\, l_{i+k}$ will be denoted by *not*($\{l_i, \ldots, l_{i+k}\}$).

Unless otherwise stated, we assume that l's in rule (1) are ground. Rules with variables will be used as a shorthand for the sets of their ground instantiations. Variables will be denoted by capital letters.

The answer set semantics of a logic program π assigns to π a collection of *answer sets* – sets of ground literals over signature $\sigma(\pi)$ of π corresponding to beliefs which can be built by a rational reasoner on the basis of rules of π. Under this semantics the rule (1) can be viewed as a constraint on such beliefs and is read as "if literals $l_1, \ldots, l_m$ are believed to be true and there is no reason to believe that literals $l_{m+1}, \ldots, l_n$ are true then the reasoner must believe l_0". We say that literal $l \in$ *lit*($\sigma(\pi)$) is *true* in an answer set S of π if $l \in S$; l is *false* in S if $\overline{l} \in S$; π entails l ($\pi \models l$) if l is true in all answer sets of π. We say that π's answer to a query $l \in$ *lit*($\sigma(\pi)$) is *yes* if $\pi \models l$, *no* if $\pi \models \overline{l}$, and *unknown* otherwise.

To give a definition of answer sets of logic programs, let us first consider programs without negation as failure.

- The *answer set* of a program π not containing negation as failure *not* is the smallest (in the sense of set-theoretic inclusion) subset S of *lit*($\sigma(\pi)$) such that

  (i) for any rule $l_0 \leftarrow l_1, \ldots, l_m$ from π, if $l_1, \ldots, l_m \in S$, then $l_0 \in S$,

  (ii) if S contains a pair of contrary literals, then $S =$ *lit*($\sigma(\pi)$).

It can easily be shown that every program π that does not contain negation as failure has a unique answer set, which will be denoted by *ans*(π).

- Now let π be an arbitrary logic program without variables. For any set S of literals, let π^S be the logic program obtained from π by deleting

  (i) each rule that has an occurrence of *not* l in its body with $l \in S$, and

  (ii) all occurrences of *not* l in the bodies of the remaining rules.

Clearly, π^S does not contain *not*, and hence its answer set is already defined. If this answer set coincides with S, then we say that S is an *answer set* of π. In other words, the answer sets of π are characterized by the equation

$$S = \mathit{ans}\big(\pi^S\big). \tag{2}$$

A logic program is said to be *consistent* if it has a consistent answer set.[4] It is not difficult to show that any answer set of a consistent program is consistent.

We conclude our short introduction by presenting a theorem useful for computing the set of consequences of logic programs. This process can be sometimes simplified by "splitting" the program into parts. We say that a set U of literals *splits* a program π if, for every rule r of π, *pos*(r) $\cup$ *neg*(r) $\subseteq U$ whenever *head*(r) $\in U$. If U splits π then the set of rules in π whose heads belong to U will be called the base of π (relative to U). We denote the base of π by *base*(π, U). The rest of the program (called the top of π) will be denoted by *top*(π, U). Whenever possible the reference to U will be omitted.

Consider for instance a program π_1 consisting of the rules

$$q(a) \leftarrow \mathit{not}\ q(b), \qquad q(b) \leftarrow \mathit{not}\ q(a), \qquad r(a) \leftarrow q(a), \qquad r(a) \leftarrow q(b).$$

Then $U = \{q(a), q(b)\}$ is a splitting set of π_1, *base*(π_1, U) consists of the first two rules while *top*(π_1, U) consists of the last two.

Let U be a splitting set of a program π. For any set $V \subseteq U$, by *red*(π, V) we denote the program obtained from *top*(π, U) by

(1) deleting each rule r such that *pos*(r) $\cap\ (U \setminus V) \neq \emptyset$ or *neg*(r) $\cap\ V \neq \emptyset$,

(2) replacing the body of each remaining rule r by *pos*(r) $\setminus\ U \cup$ *not*(*neg*(r) $\setminus\ U$).

Theorem 1 (Splitting set theorem, [32]). Let U be a splitting set of a program π. A consistent set of literals is an answer set of π iff it can be represented in the form $S_1 \cup S_2$, where S_1 is an answer set of *base*(π, U) and S_2 is an answer set of *red*(π, S_1).

[4] A set of literals is consistent if it does not contain contrary literals.

The theorem suggests the following approach to computing consistent answer sets of a program π which is split by some set U. First find all the answer sets of the program *base*(π, U). For each such set S_1, compute the program *red*(π, S_1), and find all its answer sets. For each such set S_2 form a union $S_1 \cup S_2$. The consistent unions found in this way are the consistent answer sets of π. Consider, for instance, program π_1 above. Then the program

$$q(a) \leftarrow \mathit{not}\ q(b), \qquad q(b) \leftarrow \mathit{not}\ q(a)$$

is the base of π_1 with respect to U. It has answer sets $S_1 = \{q(a)\}$ and $S_2 = \{q(b)\}$. The corresponding reducts are

$$\mathit{red}(\pi_1, S_1) = \mathit{red}(\pi_1, S_2) = r(a) \leftarrow$$

and, by the splitting set theorem, the answer sets of π_1 are $\{q(a), r(a)\}$ and $\{q(b), r(a)\}$. In the context of logic programming, the splitting set theorem and its generalizations first appeared in [32]. Similar results for autoepistemic logic can be found in [25].

3. F-specifications and lp-functions

In some cases a knowledge representation problem consists in representing a (possibly partial) definition of new relations between objects of the problem domain which is given in terms of previously known relations between these objects. Such a definition can be mathematically described as a function f which maps states of an input signature σ_{i} into the states of an output signature σ_{o}. The states of σ_i are called *input states*. They contain all the currently available information about the old, given relations. Similarly, the states of σ_{o} are called *output states*; $f(X)$ contains all we may know about defined relations given an input state X. We follow [25] and call a triple consisting of such a function f together with input and output signatures $\sigma_{\mathrm{i}}(f)$ and $\sigma_{\mathrm{o}}(f)$ a *functional specification*; *dom*(f) will denote the domain of f.[5]

The following example shows how a simple knowledge representation problem can be refined in terms of f-specifications.

Example 2. Consider the following informal specification of a knowledge representation problem:

- Formalize the assertion "x has a PhD iff x is a faculty". Given a complete list of faculty your formalization should entail all the available information about PhD degrees of people from this list. Use unary predicate constants *faculty* and *phd*.

[5] This view is common in databases where one of the most important knowledge representation problems consists in defining new relations (views) in terms of basic relations stored in the database tables. Unlike our case however databases normally assume the close world assumption and hence only need to represent positive information. As a result, views can be viewed as functions from sets of atoms to sets of atoms.

Despite its simplicity, the above specification is slightly ambiguous. Its possible refinements may depend on the form of representation of a "complete list of faculty". We look at two different representations of this list defined by two f-specifications f_0 and g_0. The specifications have the same signatures:

- input signature $\sigma_{\rm i}$ consisting of individual names, say, *mike*, *john* and *mary* and a unary relation symbol *faculty*;
- output signature $\sigma_{\rm o}$ consisting of the same individual names and a unary predicate symbol *phd*.

Function f_0 is defined on complete states of σ_i.[6] Obviously, for any $X \in dom(f_0)$

$$f_0(X) = \{phd(c):\ faculty(c) \in X\} \cup \{\neg phd(c):\ \neg faculty(c) \in X\}.$$

Similarly, function g_0 is defined on sets of ground atoms of $\sigma_{\rm i}$:

$$g_0(X) = \{phd(c):\ faculty(c) \in X\} \cup \{\neg phd(c):\ faculty(c) \notin X\}.$$

Making an early distinction between these two specifications of our informal problem helps to eliminate possible difficulties which may appear later in the development process.

Example 3. Now let us look at a slightly different informal specification. Assume that the language of our domain is determined, as in example 2, by signatures $\sigma_{\rm i}$ and $\sigma_{\rm o}$ but, unlike that example, the available information about faculty can be incomplete. For instance, we may know that Mike is a faculty, John is not, but have no relevant information about Mary. This can be represented by an incomplete set of literals $\{faculty(mike), \neg faculty(john)\}$. In general, the input of the corresponding f-specification $\tilde{f}_0$ consists of (possibly incomplete) states of $\sigma_{\rm i}$ and $\tilde{f}_0$ can be defined as follows:

$$\tilde{f}_0(X) = \bigcap_{\widehat{X} \in c(X)} f_0(\widehat{X})$$

where $c(X)$ is the set of all complete supersets of X from $dom(f_0)$. Notice that the new specification is defined in terms of f_0 and can be viewed as an interpolation of function f_0 from complete to all input states (see section 5 for more details). It is easy to see that $\tilde{f}_0$ computes all possible information about PhD degrees which can be obtained from X and f_0.

To further refine the above f-specifications we will use the notion of lp-function from [6].

Definition 4. A four-tuple $\pi = \{\pi, \sigma_{\rm i}(\pi), \sigma_{\rm o}(\pi), dom(\pi)\}$ where

(1) π is a logic program (with some signature $\sigma(\pi)$),

[6] A set X of literals over signature σ is called *complete* if for any $l \in lit(\sigma)$, $l \in X$ or $\overline{l} \in X$.

(2) $\sigma_{\rm i}(\pi), \sigma_{\rm o}(\pi)$ are sub-signatures of $\sigma(\pi)$ called input and output signatures of π, respectively,

(3) $dom(\pi) \subseteq states(\sigma_{\rm i}(\pi))$

is called *lp-function* if for any $X \in dom(\pi)$ program $\pi \cup X$ is consistent, i.e., has a consistent answer set.

For any $X \in dom(\pi)$,

$$\pi(X) \stackrel{\rm def}{=} \{l\colon\ l \in lit(\sigma_{\rm o}(\pi)), \pi \cup X \models l\}.$$

We say that an lp-function π *represents* an f-specification f if π and f have the same input and output signatures and domain and for any $X \in dom(f)$, $f(X) = \pi(X)$.

Let E be an inference engine which takes as an input a logic program π and a ground literal l. We will say that an lp-function π is *computable* by E if for any query $l \in lit(\sigma_{\rm o})$ and any $X \in dom(\pi)$, $E(\pi \cup X, l)$ returns *yes* if $l \in \pi(X)$, *no* if $\overline{l} \in \pi(X)$, and *unknown* otherwise.

Example 5. It is easy to show that f-specification f_0 from example 2 can be represented by lp-function π_{f_0} with signature σ consisting of types, objects, and predicate symbols of $\sigma_{\rm i}(f_0)$ and predicate symbol *phd* and the rules:

$$\left.\begin{array}{l} phd(P) \leftarrow faculty(P), \\ \neg phd(P) \leftarrow \neg faculty(P). \end{array}\right\} \pi_{f_0}$$

The input and output signatures of π_{f_0} are the same as those of f_0 and $dom(\pi_{f_0}) = dom(f_0)$. An alternative representation $\pi'(f_0)$ can also be given by replacing the rules of π_{f_0} by the rules

$$\left.\begin{array}{l} phd(P) \leftarrow faculty(P), \\ \neg phd(P) \leftarrow not\ faculty(P). \end{array}\right\} \pi'_{f_0}$$

It is also easy to see that the restriction π_{g_0} of π'_{f_0} on the $dom(g_0)$ is an lp-function representing g_0.

Now let us consider f-specification $\tilde{f}_0$ from example 3. One can show that the lp-function $\pi_{\tilde{f}_0}$ whose rules are those of π_{f_0} and signatures and domain are those of $\tilde{f}_0$, represents $\tilde{f}_0$. This means that π_{f_0} is elaboration tolerant with respect to allowing new, incomplete inputs, i.e., the change of specification from f_0 to $\tilde{f}_0$ does not require any changes in π_{f_0}. This is certainly not the case for the second representation of f_0. In this sense, π_{f_0} is superior to π'_{f_0}.

Since all the lp-functions above are (efficiently) computable by a slight modification of the Prolog interpreter the refinement process stops here.[7]

[7] Of course, by considering only ground queries we limit ourselves to a very restricted notion of computability of lp-function f by inference engine E. It can of course be extended to allow queries from $\sigma_{\rm o}(f)$ with variables but then the function π_{g_0} will stop being computable due to floundering. This again can be remedied by introducing types like *person*, *degree*, etc., and modifying our program

To complete this section we give several definitions and results which proved to be useful for reasoning about lp-functions.

An lp-function π is called *monotonic* if for any X, Y from *dom*(π) such that $X \subseteq Y$, $\pi(X) \subseteq \pi(Y)$.

The absolute value of a literal l (symbolically, $|l|$) is l if l is positive, and $\overline{l}$ otherwise.

Definition 6. Let π be a logic program with signature σ. A *signing* of π is a set $S \subseteq$ *atoms*(σ) such that

(1) for any rule (1) from π, either $|l_0|, \ldots, |l_m| \in S$, $|l_{m+1}|, \ldots, |l_n| \notin S$ or $|l_0|, \ldots, |l_m| \notin S$, $|l_{m+1}|, \ldots, |l_n| \in S$,

(2) for any atom $l \in S$, $\neg l$ does not appear in π.

If a program has a signing, we say that it is signed [29,44]. The notion of signing for finite logic programs without classical negation was introduced by Kunen [29], who used it as a tool in his proof that, for a certain class of programs, two different semantics of logic programs coincide. Turner in [44] extends the definition to the class of logic programs with two kinds of negation. Obviously, programs without negation as failure are signed with an empty signing.

Signed programs enjoy several important properties which make them attractive from the standpoint of knowledge representation. In particular:

1. Signed programs without explicit negation are consistent. This is a special case of a more general theorem by Fages [19].
2. If π is signed and consistent then the set of consequences of the program under the answer set semantics coincides with its set of consequences under the well-founded semantics [37,46]. Notice this result shows that interpreters, such as XSB, which compute the well founded semantics of logic programs, can also be used to compute the consequences of such programs under the answer set semantics.

We will however be especially interested in the following monotonicity theorem.

Theorem 7 (Monotonicity theorem, Turner). If an lp-function π has a signing S such that $S \cap (\mathit{lit}(\sigma_i) \cup \mathit{lit}(\sigma_o)) = \emptyset$ then π is monotonic.

This result is a special case of a substantially more general result from [45].

Example 8. Consider an lp-function π_2 with input signature $\sigma_{\rm i} = \{\{a, b, c\}, \{p, r\}\}$, output signature $\sigma_{\rm o} = \{\{a, b, c\}, \{q\}\}$, *dom*($\pi_2$) = *states*($\sigma_i$) and logic program π_2 consisting of the following rules:

$$\left.\begin{array}{l} q(X) \leftarrow p(X), \mathit{not}\ \mathit{ab}(X), \\ \neg q(X) \leftarrow r(X), \\ \mathit{ab}(X) \leftarrow \mathit{not}\ \neg r(X). \end{array}\right\} \pi_2$$

accordingly or by using a query answering system with constructive negation [8,38]. For the sake of readability we will only consider ground queries.

It is easy to see that lp-function π_2 has a signing consisting of atoms formed by predicate *ab*. The signing satisfies the condition of theorem 7, and hence π_2 is monotonic. It is worth noticing that π_2 considered as a logic program is nonmonotonic. The addition of extra rules (or facts) about ab can force us to withdraw previous conclusions about q. Monotonicity is however preserved for inputs from $\sigma_{\rm i}$.

4. An example: *is*-nets on closed domains

In this section, we illustrate the notions of functional specification and lp-function by using them to solve a knowledge representation problem associated with a simple type of taxonomic hierarchy, called *is-nets*. The problem of specifying and representing is-nets is commonly used to test strengths and weaknesses of various nonmonotonic formalisms. Logic programming approaches to this problem (which assume completeness of its domain) can be found in [1,33]. Modifications of this example will be used throughout the rest of the paper.

An is-net N can be viewed as a combination of graphs $N_{\rm s}$ and $N_{\rm d}$ where $N_{\rm s}$ describes the (proper) subclass relation between classes and $N_{\rm d}$ consists of positive and negative defeasible links connecting classes with properties. The latter represent defaults: "elements of class c normally satisfy (do not satisfy) property p". As usual we assume that N_s is acyclic and that a class c and a property p can be connected by at most one link. We use (possibly indexed) letters o, c, p, and d to denote objects, classes, properties and defaults respectively. Figure 1(a) gives a pictorial representation of a net. Here $c_1, \ldots, c_5$ are classes and p is a property. Links from c_5 to p and from c_4 to p represent positive and negative defaults while the other links represent the subclass relationship.

There are many knowledge representation problems which can be associated with a net N. We start with the simplest one, when N is viewed as an informal specification of a function f_N which takes as an input the links of N together with a collection I of positive links from objects to classes (see figure 1(b)). We denote the resulting graph

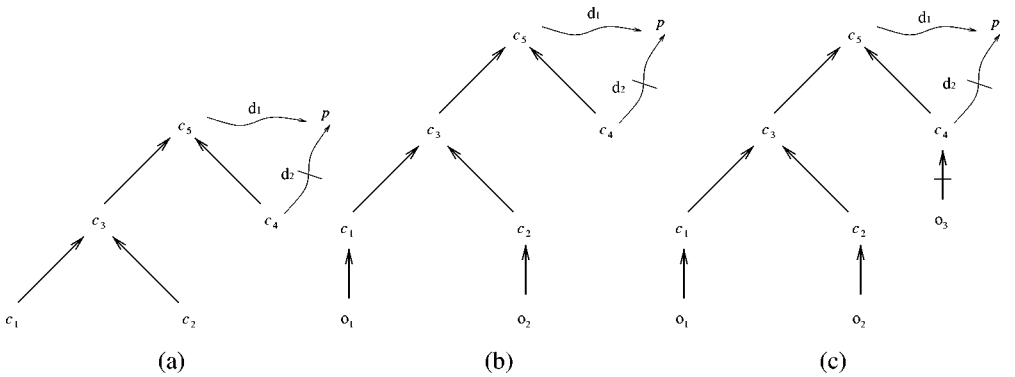

Figure 1. A simple taxonomic hierarchy.

by $N(I)$ and assume that for any object o and class c from $N(I)$, o is an element of c iff $N(I)$ contains a path from o to c. Similarly, c_1 is a (proper) subclass of c_2 iff there is a path from c_1 to c_2. Given $N(I)$, the function f_N computes all possible conclusions about relations *subclass*, *is* and *has* which a rational agent can obtain from this net.[8] Our ability to represent an input of f_N by positive links only is justified by our assumption that, at least with respect to objects included in $N(I)$, the net $N(I)$ contains complete information about relations *is* and *subclass*. Accordingly, we call f_N a *closed domain* specification of N.

In what follows we will be interested in applying our methodology for providing a rigorous specification of this function and for finding its logic programming representation. Later we consider more complex knowledge representation problems which can also be associated with N.

4.1. *Specifying the problem*

We start by playing the role of a specifier and give a precise definition of function f_N using the language of f-specifications. To this goal, we first fix three collections of object constants for objects, classes and properties of the nets. The corresponding types will be denoted by $\tau_{\rm o}, \tau_{\rm c}$, and $\tau_{\rm p}$. Constants of the fourth type ($\tau_{\rm d}$) will be used to name the nets default links. Next we identify graphs $N_{\rm s}$ and $N_{\rm d}$ with the collection of literals of the form *subclass*$_0(c_1, c_2)$, *default*$(d, c, p, +)$, and *default*$(d, c, p, -)$ which correspond to the links of these graphs. (The last parameter in *default* is used to distinguish positive and negative links.) A set I of links from objects to classes will be identified with the collection of the corresponding statements of the form *is*$_0(o, c)$. Function f_N will be constructed as a combination of three simpler functions $f_{\rm s}$, $f_{\rm i}$ and $f_{\rm d}$:

(a) An f-specification $f_{\rm s}$ will take as an input a net $N_{\rm s}$ – a collection of atoms *subclass*$_0(c_i, c_j)$ where c_i, c_j are classes connected by a link of $N_{\rm s}$ and return the complete set of literals, formed by the predicate symbol *subclass*, defining the transitive closure of the input; both, input and output signatures of $f_{\rm s}$ have the same object constants of the type $\tau_{\rm c}$, and predicate symbols *subclass*$_0$ and *subclass*, respectively.

(b) An f-specification $f_{\rm i}$ takes as an input a complete set of *subclass*-literals (representing transitive and asymmetric relations *subclass*) together with the set I of atoms *is*$_0(o, c)$ representing links from objects to classes; $f_{\rm i}$ returns the complete set of *is*-literals representing the membership relation defined by the input. Input and output signatures of $f_{\rm i}$ have object constants of two types, $\tau_{\rm o}$ and $\tau_{\rm c}$; *pred*$(\sigma_{\rm i}(f_{\rm i})) = \{$*is*$_0$, *subclass*$\}$ while *pred*$(\sigma_{\rm o}(f_{\rm i})) = \{$*is*$\}$.

(c) We also need a more complex f-specification $f_{\rm d}$ defined as follows:

[8] *is*(o, c) stands for "object o is an element of class c"; *has*(o, p) means that "object o has property p". Both predicates are typed.

- The input signature $\sigma_{\rm i}(f_{\rm d})$ of $f_{\rm d}$ consists of object constants of three types, $\tau_{\rm o}, \tau_{\rm c}$, and $\tau_{\rm d}$ and predicate symbols *is*, *subclass*, and *default*. The output signature $\sigma_{\rm o}(f_{\rm d})$ consists of object constants of the types $\tau_{\rm o}$ and $\tau_{\rm p}$ and predicate symbol *has*.
- *dom*$(f_{\rm d})$ consists of states of $\sigma_{\rm i}(f_{\rm d})$ which are complete with respect to *is* and *subclass* and satisfy the constraints:[9]

$$\begin{array}{l} \leftarrow \mathit{is}(o, c_1), \\ \quad \mathit{subclass}(c_1, c_2), \\ \quad \neg\mathit{is}(o, c_2). \\ \leftarrow \mathit{subclass}(c, c). \\ \leftarrow \mathit{default}(d, c, p, +), \\ \quad \mathit{default}(d, c, p, -). \end{array}$$

- For any $X \in \mathit{dom}(f_{\rm d})$, $\mathit{has}(o, p) \in f_{\rm d}(X)$ iff there are d_1 and c_1 such that

  (i) $\mathit{default}(d_1, c_1, p, +) \in X$,

  (ii) $\mathit{is}(o, c_1) \in X$,

  (iii) for any $\mathit{default}(d_2, c_2, p, -) \in X$,

$$\neg\mathit{is}(o, c_2) \in X \text{ or } \mathit{subclass}(c_1, c_2) \in X.$$

  This condition corresponds to the Inheritance Principle [43] which says that more specific defaults override less specific ones.

  Similarly for $\neg\mathit{has}(o, p)$.

(d) To complete the construction of f_N we will need the following notation: for any $\alpha \subseteq \mathit{pred}(\sigma)$ and $X \subset \mathit{lit}(\sigma)$ by $[X]_\alpha$ we denote the set of literals from X formed by predicate symbols from α.
Now the f-specification f_N will be defined as follows:

- the input signature of f_N consists of object constants of the types $\tau_{\rm o}, \tau_{\rm c}, \tau_{\rm d}$ and predicate symbols $\mathit{is}_0, \mathit{subclass}_0, \mathit{default}$;
- the output signature of f_N consists of objects constants of the types $\tau_{\rm o}, \tau_{\rm c}, \tau_{\rm p}$ and predicate symbols *subclass*, *is*, *has*;
- $\mathit{dom}(f_N) = \{X\colon\ X \subseteq \mathit{atoms}(\sigma_i(f_N))\}$;[10]

[9] A constraint is a rule of the form $\leftarrow \Delta$ where Δ is a list of literals from some signature σ. A set $X \in \mathit{states}(\sigma)$ satisfies the constraint $\leftarrow \Delta$ if $\Delta \not\subseteq X$. X satisfies a collection C of constraints if it satisfies every constraint in C.

[10] The domain of f_N can be alternatively represented by complete sets of literals of $\sigma_{\rm i}(f_N)$. The resulting specification will be isomorphic to f_N and used when convenient.

- f_N is defined by combining functions $f_{\rm s}, f_{\rm i}$ and $f_{\rm d}$ as follows:

$$\begin{aligned} Y_0(X) &= f_{\rm s}\big([X]_{\mathit{subclass}_0}\big),\\ Y_1(X) &= f_{\rm i}\big(Y_0(X) \cup [X]_{\mathit{is}_0}\big),\\ Y_2(X) &= f_{\rm d}\big(Y_0(X) \cup Y_1(X) \cup [X]_{\mathit{default}}\big),\\ f_N(X) &= Y_0(X) \cup Y_1(X) \cup Y_2(X). \end{aligned}$$

Note that this definition does not require any sophisticated mathematics. In particular, it presupposes no knowledge of logic programming.

It is worth noticing that we can view f_N as being constructed from f-specifications $f_{\rm d}, f_{\rm s}$ and $f_{\rm i}$ by a specification constructor called *incremental extension* [26]. Since this fact will be useful later in our search for an lp-function representing f_N we recall the corresponding definition.[11] For simplicity we only consider signatures which agree on their common types, i.e., if $\tau \in \sigma_1 \cap \sigma_2$ then $\mathit{obj}(\tau, \sigma_1) = \mathit{obj}(\tau, \sigma_2)$.

Definition 9. The f-specification $f \circ g$ is called the *incremental extension* of g by f if it satisfies the following conditions:

(1) $\sigma_{\rm i}(f \circ g) = \sigma_{\rm i}(g) \cup \sigma_{\rm i}(f)$,

(2) $\sigma_{\rm o}(f \circ g) = \sigma_{\rm o}(g) \cup \sigma_{\rm o}(f)$,

(3) $X \in \mathit{dom}(f \circ g)$ if
$[X]_{\mathit{pred}(\sigma_{\rm i}(g))} \in \mathit{dom}(g)$ and $[X]_{\mathit{pred}(\sigma_{\rm i}(f))\setminus\mathit{pred}(\sigma_{\rm o}(g))} \cup g([X]_{\mathit{pred}(\sigma_{\rm i}(g))}) \in \mathit{dom}(f)$,

(4) $f \circ g(X) = f([X]_{\mathit{pred}(\sigma_{\rm i}(f))\setminus\mathit{pred}(\sigma_{\rm o}(g))} \cup g([X]_{\mathit{pred}(\sigma_{\rm i}(g))})) \cup g([X]_{\mathit{pred}(\sigma_{\rm i}(g))})$.

It is easy to see that the definition of f_N above can be rewritten as

$$f_N = f_{\rm d} \circ (f_{\rm i} \circ f_{\rm s}). \tag{3}$$

4.2. Building the declarative representation

Now let us assume the role of an implementor, who just received the above definition of f_N and is confronted with the task of building a computable lp-function representing f_N. (Recall that we will only be interested in getting answers to ground queries formed by predicates *subclass*, *is*, and *has*.) According to our methodology, we will first ignore the computability requirement and concentrate on correctness of our representation viewed as a declarative logic program.

[11] The definition we give in this paper actually extends that in [26].

We start with a representation for f_s. Since N_s is acyclic, it is easy to show that f_s can be represented by the program π_s:

$$\left.\begin{array}{rl} \mathit{subclass}(C_1, C_2) \leftarrow & \mathit{subclass}_0(C_1, C_2). \\ \mathit{subclass}(C_1, C_2) \leftarrow & \mathit{subclass}_0(C_1, C_3), \\ & \mathit{subclass}(C_3, C_2). \\ \neg\mathit{subclass}(C_1, C_2) \leftarrow & \mathit{not}\ \mathit{subclass}(C_1, C_2). \end{array}\right\} \pi_s$$

It is equally easy to show that f_i can be represented by an lp-function

$$\left.\begin{array}{rl} \mathit{is}(O, C) \leftarrow & \mathit{is}_0(O, C). \\ \mathit{is}(O, C_2) \leftarrow & \mathit{is}_0(O, C_1), \\ & \mathit{subclass}(C_1, C_2). \\ \neg\mathit{is}(O, C) \leftarrow & \mathit{not}\ \mathit{is}(O, C). \end{array}\right\} \pi_i$$

In our next step we need to find a representation $\pi_{f_i \circ f_s}$ of the specification $f_i \circ f_s$. For simple lp-functions like π_i and π_s we can directly check that this can be done by combining the rules of π_i and π_s. As was shown in [25], however, this is not always the case, i.e., there are f-specifications f, g represented by lp-functions π_f and π_g such that $\pi_f \cup \pi_g$ does not represent $f \circ g$. The following realization theorem (which is a slight modification of a similar theorem from [25]) is useful for eliminating this possibility.

Theorem 10 (Realization theorem for incremental extension). Let f and g be f-specifications represented by lp-functions π_f and π_g, respectively, and let $\pi_{f \circ g} = \pi_f \cup \pi_g$.[12] A 4-tuple $\pi_{f \circ g} = \{\pi_{f \circ g}, \sigma_i(f \circ g), \sigma_o(f \circ g), \mathit{dom}(f \circ g)\}$ is an lp-function representing $f \circ g$ if π_f and π_g satisfy the following conditions:

(1) for any $X \in \mathit{dom}(\pi_g)$ and any answer sets A_1 and A_2 of $\pi_g \cup X$, $A_1 \cap \mathit{lit}(\sigma_o(\pi_g)) = A_2 \cap \mathit{lit}(\sigma_o(\pi_g))$,

(2) $\mathit{lit}(\sigma(\pi_f)) \cap \mathit{lit}(\sigma(\pi_g)) \subseteq \mathit{lit}(\sigma_o(\pi_g))$,

(3) $\mathit{head}(\pi_f) \cap \mathit{lit}(\sigma(\pi_g)) = \emptyset$.

It is easy to see that functions π_i and π_s defined above satisfy the conditions of the theorem, which justifies our construction.

[12] Recall that $\pi = \pi_1 \cup \pi_2$ if $\sigma(\pi) = \sigma(\pi_1) \cup \sigma(\pi_2)$ and the rules of $\pi_{f \circ g}$ consist of rules of π_f and π_g. Note that this definition is applicable to programs with and without variables.

To continue our search for an lp-function representing the specification f_N we need to find a representation of $f_{\rm d}$. To this goal let us consider a program $\pi_{\rm d}$:

$$\left.\begin{array}{rl}
\mathit{has}(X,P) \leftarrow & \mathit{default}(D,C,P,+),\\
& \mathit{is}(X,C),\\
& \mathit{not\ exceptional}(X,D,+).\\
\neg\mathit{has}(X,P) \leftarrow & \mathit{default}(D,C,P,-),\\
& \mathit{is}(X,C),\\
& \mathit{not\ exceptional}(X,D,-).\\
\mathit{exception}(E,D_1,+) \leftarrow & \mathit{default}(D_1,C,P,+),\\
& \mathit{default}(D_2,E,P,-),\\
& \mathit{not\ subclass}(C,E).\\
\mathit{exception}(E,D_1,-) \leftarrow & \mathit{default}(D_1,C,P,-),\\
& \mathit{default}(D_2,E,P,+),\\
& \mathit{not\ subclass}(C,E).\\
\mathit{exceptional}(X,D,S) \leftarrow & \mathit{exception}(E,D,S),\\
& \mathit{is}(X,E).
\end{array}\right\}\ \pi_{\rm d}$$

The signature of this program is obtained from $\sigma_{f_{\rm d}}$ by adding predicate constants *exception* and *exceptional*. The statement *exception*$(e,d,+)$ reads as "the positive default d is not applicable to elements of the class e"; the statement *exceptional*$(o,d,+)$ states that "the positive default d is not applicable to the object o of the domain"; similarly for negative defaults. The program uses a standard logic programming representation of defaults and their exceptions (see, for instance, [5]). We would like $\pi_{\rm d}$ to be viewed as an lp-function whose input and output signatures are the same as those of $f_{\rm d}$ and whose domain is *dom*$(f_{\rm d})$, i.e., we need the following:

Proposition 11. For any $X \in$ *dom*$(f_{\rm d})$, program $\pi_{\rm d} \cup X$ is consistent.

Proof (sketch). A program $\pi_{\rm d} \cup X$ can be viewed as a logic program Π without classical negation in a language which contains new predicate symbols $\neg$*is*, $\neg$*subclass*, $\neg$*has*. This program is stratified [2] and has therefore a unique answer set A [21]. As shown in [22], A is also the unique answer set of $\pi_{\rm d} \cup X$ iff A does not contain contrary literals. Since literal l belongs to an answer set A of a program Π (not containing classical negation) iff there is a rule in Π whose head is l and whose body is satisfied by A [34], we need to show that there are no rules r_1 and r_2 with the heads *has*(o,p), $\neg$*has*(o,p) whose bodies are satisfied by A. Suppose that there are c_1 and c_2 such that *default*$(d_1,c_1,p,+) \in A$, *default*$(d_2,c_2,p,-) \in A$, *is*$(x,c_1) \in A$ and *is*$(x,c_2) \in A$. Since a class c and property p can be connected by at most one link, $c_1 \neq c_2$. Since the net N_s is acyclic, *subclass*$(c_1,c_2) \notin A$ or *subclass*$(c_2,c_1) \notin A$. Therefore, *exceptional*$(x,d_1,+)$ or *exceptional*$(x,d_2,-)$ is also in A, i.e., at least one of the rules r_1, r_2 has a body which is not satisfied by A. □

Now we can show correctness of our construction.

Proposition 12. Lp-function π_d represents f_d, i.e., for any $X \in dom(f_{\rm d})$, $\pi_{\rm d}(X) = f_{\rm d}(X)$.

Proof (sketch). In the previous proof we showed that $\pi_{\rm d} \cup X$ has a unique consistent answer set. Let us denote it by A.

Since a literal l belongs to a consistent answer set A of the program iff there is a rule with l in the head whose body is satisfied by A, we have that *has*$(o, c_1) \in A$ iff there are literals *default*$(d_1, c_1, p, +)$, *is*$(o, c_1) \in A$ and *exceptional*$(o, d_1, +) \notin A$. This happens iff *default*$(d_1, c_1, p, +) \in N_{\rm d}$, *is*$(o, c_1) \in X$ and for any *default*$(d_2, c_2, p, -) \in N_{\rm d}$, *is*$(o, c_2) \notin X$ or *subclass*$(c_1, c_2) \in N_{\rm s}$. Since X is complete, *is*$(o, c_2) \notin X$ iff $\neg$*is*$(o, c_2) \in X$ and therefore, *has*$(o, p) \in \pi_{\rm d}(X)$ iff *has*$(o, p) \in f(X)$. A similar argument works for negative literals. □

To complete our search for a representation of f_N we use the construction from theorem 10. It is easy to see that programs $\pi_{f_{\rm i} \circ f_{\rm s}}$ and $\pi_{\rm d}$ satisfy the conditions of the theorem and hence $\pi_{f_N} = \{\pi_{f_N}, \sigma_{\rm i}(f_N), \sigma_{\rm o}(f_N), dom(f_N)\}$ where

$$\pi_{f_N} = \pi_{\rm s} \cup \pi_{\rm i} \cup \pi_{\rm d} \tag{4}$$

represents f_N.

4.3. Answering queries

Now we are ready for the implementation stage of our design. This stage crucially depends on the choice of inference engine we use to answer queries to π_{f_N}. Since Prolog is the most popular logic programming language to date, we base this engine on the Prolog interpreter. To do that we must first replace all occurrences of negative literals $\neg p(t_1, \ldots, t_n)$ in π_{f_N} by *np*$(t_1, \ldots, t_n)$. The resulting program can be viewed as a Prolog program with variables. We consider an inference engine $\mathcal{A}$ defined as a simple meta-interpreter which takes a query q and a program π as an input. If q is an atom, $\mathcal{A}$ makes two calls to the Prolog interpreter; one with parameters q and π and another with parameters nq, π. If the first query is answered *yes*, the meta-interpreter $\mathcal{A}$ returns *yes*; if *yes* is the answer to the second query, $\mathcal{A}$ returns *no*; otherwise the answer is unknown. Similarly, if q is a negative literal. The meta-interpreter is defined for programs which do not entail q and $\neg q$ (for any query q).

Proposition 13. The lp-function π_{f_N} is computable by $\mathcal{A}$.

Proof (sketch). Let us start by listing the questions which need to be addressed to prove this proposition. First, it is well known that, for some programs, the Prolog interpreter may produce unsound results. This may happen because of the absence of the occur-check which, in some cases, is necessary for soundness of the SLDNF resolution, or because the interpreter may flounder, i.e., may select for resolution a goal of the form *not* q where q contains an uninstantiated variable. Second, the interpreter

may fail to terminate. Even if we show that for any $X \in dom(f_N)$ and ground query q, the interpreter which takes $\pi_{f_N} \cup X$ and q as an input terminates, does not flounder, and does not require the occur-check, the soundness of our result is guaranteed only with respect to the *unsorted* grounding of π_{f_N}, i.e., the grounding of π_{f_N} by terms of signature σ_u obtained from signature $\sigma(f_N)$ by removing types and type information. In what follows we briefly discuss how these questions can be addressed.

(a) We show that for any $X \in dom(\pi_{f_N})$ and query $q \in lit(\sigma_o(f_N))$ the program $\pi_{f_N} \cup X \cup \{\leftarrow q\}$ is occur-check free, i.e., the unification algorithm used by Prolog interpreters never selects a step requiring the occur-check. To prove this we need a notion of *mode* [13] – a function assigning $+$ or $-$ to each parameter of a predicate symbol p from the language of program π. If the parameter in a position i of p is assigned $+$, then i is called an input position. Otherwise it is called an output position. In our further discussion we need the following mode m_0 of the program π_{f_N}:

$$
\begin{aligned}
&\mathit{has}(+,+)\\
&\neg\mathit{has}(+,+)\\
&\mathit{default}(-,-,-,-)\\
&\mathit{is}_0(-,-)\\
&\mathit{is}(+,+)\\
&\neg\mathit{is}(+,+)\\
&\mathit{subclass}_0(-,-)\\
&\mathit{subclass}(+,+)\\
&\neg\mathit{subclass}(+,+)\\
&\mathit{exceptional}(+,+,+)\\
&\mathit{exception}(-,-,-)
\end{aligned}
$$

A rule r of a program π is called *well-moded* with respect to a mode m if:

(i) Every variable occurring in an input position of a literal l from the body of r occurs either in an input position of the head or in an output position of some literal from the body that precedes l.

(ii) Every variable occurring in an output position of the head occurs in an input position of the head, or in an output position of a literal from the body.

A program π is well-moded with respect to m if all its rules are. It is not difficult to check that the program $\pi_{f_N} \cup X$ is well-moded with respect to mode m_0.
Apt and Pellegrini in [4] showed that if a program π is well-moded with respect to some mode m and there is no rule in π whose head contains more than one occurrence of the same variable in its output positions then π is occur-check free with respect to any ground query. The mode m_0 satisfies these conditions and therefore answering query q to $\pi_{f_N} \cup X$ does not require the occur-check.

(b) To prove that a query $q \in lit(\sigma_o(f_N))$ to $\pi_{f_N} \cup X$ does not flounder we use another theorem from [4,42]: if program π is well-moded with respect to some mode assignment m and all predicate symbols occurring in π under *not* are moded completely by input then a ground query to π does not flounder. Again, the mode m_0 satisfies this property. From (a) and (b) we can conclude that if the Prolog interpreter terminates on query q to program π then there is an SLDNF resolution derivation of q from π.

(c) The termination of $\pi_{f_N} \cup X$ follows from [3] where the authors gave a notion of *acceptable* program and proved that for such programs the Prolog interpreter terminates on ground queries. It is easy to check that $\pi_{f_N} \cup X$ is acceptable.

(d) Let $gr(\pi, \sigma)$ denote the set of all ground instantiations of π by ground terms of σ. From soundness of SLDNF resolution with respect to stable model semantics of logic programs we can now conclude that, given a query $q \in lit(\sigma_o(f_N))$ and the program $\pi_{f_N} \cup X$, the Prolog interpreter answers *yes* iff $gr(\pi_{f_N}, \sigma_u(f_N)) \cup X \models q$. We need to show that $gr(\pi_{f_N}, \sigma_u(f_N)) \cup X \models q$ iff $gr(\pi_{f_N}, \sigma(f_N)) \cup X \models q$. In [35] McCain and Turner showed that to guarantee this property we need to check that π_{f_N} is *stable* and *predicate-order-consistent*. The notion of stability is based on the program having a mode satisfying certain properties. It is easy to check that the mode m_0 defined above satisfies these conditions. The predicate-order-consistency of this program follows immediately from the absence of function symbols and the absence of negative edges from a predicate symbol to itself in the dependency graph of π_{f_N}. □

5. Opening closed domain specifications

In this section we introduce another knowledge representation problem associated with is-nets and demonstrate how a specification constructor, called *input opening*, can be used to specify and represent this problem.

5.1. Specifying the problem

So far we have assumed that net $N(I)$ contains complete information about relations *default*, *subclass* and *is*. In the process of development and modification of the system this assumption may become too strong and the specifier may decide to limit it to some of the relations or even to completely remove it from the specification. In this section we will consider a case when information about defaults and subclasses is still complete but information about the membership relation can be incomplete. Pictorially, the input to a net will be represented by positive and negative links from objects to classes (see figure 1(c)). These links will be recorded by literals of the form $is(o, c)$ and $\neg is(o, c)$. (Notice that here we use *is* instead of is_0 which was used in the previous specification. This change is not essential and is done only to simplify the presentation.) Obviously, not any collection I of such links is consistent with our

interpretation of the net N. To ensure consistency I must not contain contrary literals, and the *is* and *subclass* relations defined by $N(I)$ must satisfy the constraint:

$$\begin{aligned} \leftarrow\ & \mathit{is}(o, c_1), \\ & \mathit{subclass}(c_1, c_2), \\ & \neg \mathit{is}(o, c_2). \end{aligned} \tag{5}$$

Now the net N can be viewed as an informal specification of the function f°_{N} which takes as an input the links of N together with $I \in \mathit{lit}(\mathit{is})$ and returns all conclusions about relations *subclass*, is and *has* which a rational agent can obtain from N and I. (Such a function f°_{N} will be called the *open domain* specification of N.) The problem is to precisely define the set of all such conclusions. To do that we use the representation of N and functions f_{s} and f_{d} from the previous section. Function f_{s} remains unchanged; f_{d} however needs to be modified since we do not necessarily have enough information to provide it with input complete with respect to *is*. To modify f_{d} the specifier may use a specification constructor called *input opening* which removes the closed world assumption from (some or all) input predicates of a specification. To define this constructor we first need the following terminology.

Let D be a collection of states over some signature σ and let $\alpha \subseteq \mathit{pred}(\sigma)$. Predicates from α are called *open*; the other predicates from $\mathit{pred}(\sigma)$ are called *closed*. A state X of σ is called an α-*state* if for any $l \in \mathit{lit}(\mathit{pred}(\sigma \setminus \alpha))$, $l \in X$ or $\overline{l} \in X$. The set of all α-states of σ will be denoted by α-$\mathit{states}(\sigma)$. A set $X \in \mathit{states}(\sigma)$ is called *D-consistent* if there is $\widehat{X} \in D$ such that $X \subseteq \widehat{X}$; $\widehat{X}$ is called a *D-cover* of X.

For instance, in our example the only open predicate of $\sigma_{\mathrm{i}}(f_{\mathrm{d}})$ will be *is*; D will be the set of all possible complete inputs X to f_{d} such that

(i) $[X]_{\mathit{subclass,default}} = N \cup \overline{N}$, where $N \subseteq \mathit{atoms}(\mathit{default}, \mathit{subclass})$ which represents the net N and

$$\overline{N} = \big\{\neg l\colon\ l \in \mathit{atoms}(\mathit{default}, \mathit{subclass}) \wedge l \notin N\big\},$$

(ii) X satisfies the constraint (5).

It is easy to see that for the net N from figure 1(c) the set $N \cup \overline{N} \cup \{\mathit{is}(o_1, c_1), \mathit{is}(o_2, c_2)\}$ is D-consistent while $N \cup \overline{N} \cup \{\mathit{is}(o_1, c_1), \mathit{is}(o_2, c_2), \neg \mathit{is}(o_2, c_3)\}$ is not.

The set of all D-covers of a set X is denoted by $c(D, X)$. The set of all D-consistent α-states of σ is called the α-*interior* of D and is denoted by D^{α}. An f-specification f defined on a collection of complete states of $\sigma_i(f)$ is called *closed domain specification*.

Definition 14 (Input opening). Let f be a closed domain specification with domain D and $\alpha \subseteq \mathit{pred}(\sigma(f))$. An f-specification f^{α} is called the *input opening* of f with respect to α if

$$\sigma_{\mathrm{i}}\big(f^{\alpha}\big) = \sigma_{\mathrm{i}}(f), \qquad \sigma_{\mathrm{o}}\big(f^{\alpha}\big) = \sigma_{\mathrm{i}}(f) \cup \sigma_{\mathrm{o}}(f), \tag{6}$$

$$dom\big(f^{\alpha}\big) = D^{\alpha}, \tag{7}$$

$$f^{\alpha}(X) = \bigcap_{\widehat{X} \in c(D,X)} f\big(\widehat{X}\big) \cup \bigcap_{\widehat{X} \in c(D,X)} \widehat{X}. \tag{8}$$

The open domain f-specification f°_{N} of a net N can be defined as

$$f^{\circ}_{N} = f^{\alpha}_{\mathrm{d}} \circ f_{\mathrm{s}}. \tag{9}$$

As was shown in [20] the input opening f^{α} of f can be further decomposed using two simpler specification constructors called *interpolation* and *domain completion*. To define these constructors we need the following definitions.

Definition 15. A set $X \in \mathit{states}(\sigma)$ is called *maximally informative* with respect to a set $D \subseteq \mathit{states}(\sigma)$ if X is D-consistent and

$$X = \bigcap_{\widehat{X} \in c(D,X)} \widehat{X}. \tag{10}$$

Consider, for instance, the net N from figure 1(b). The set

$$N_{\mathrm{s}} \cup \overline{N}_{\mathrm{s}} \cup \big\{\mathit{is}(o_1, c_1),\ \mathit{is}(o_1, c_3),\ \mathit{is}(o_1, c_5),\ \mathit{is}(o_2, c_2),\ \mathit{is}(o_2, c_3),\ \mathit{is}(o_2, c_5)\big\}$$

is maximally informative with respect to the set of all complete input states of N, while the set

$$N_{\mathrm{s}} \cup \overline{N}_{\mathrm{s}} \cup \big\{\mathit{is}(o_1, c_1),\ \mathit{is}(o_2, c_2)\big\}$$

is not.

Definition 16 (Interpolation). Let f be a closed domain f-specification with domain D and $\alpha \subseteq \mathit{pred}(\sigma_i(f))$. By $\widetilde{D}$ we denote the set of all α-states of $\sigma_{\mathrm{i}}(f)$ which are maximally informative with respect to D. F-specification $\tilde{f}^{\alpha}$ with the same signatures as f and the domain $\widetilde{D}$ is called the *interpolation* of f if

$$\tilde{f}^{\alpha}(X) = \bigcap_{\widehat{X} \in c(D,X)} f\big(\widehat{X}\big). \tag{11}$$

This is a slight generalization of the notion of interpolation introduced in [6], where the authors only considered interpolations of functions defined by general logic programs.

Definition 17 (Domain completion). Let D be a collection of complete states over signature σ and $\alpha \subseteq \mathit{pred}(\sigma)$. The *domain completion* of D with respect to α is a function $\tilde{f}^{\alpha}_{D}$ which maps D-consistent α-states of σ into their maximally informative supersets.

The following proposition follows almost immediately from the definitions.

Proposition 18. For any closed domain f-specification f with domain D,

$$f^{\alpha} = \tilde{f}^{\alpha} \circ \tilde{f}^{\alpha}_{D}. \tag{12}$$

Proof. First note that by definition of f^{α} and $\tilde{f}^{\alpha}_{D}$, $X \in dom(f^{\alpha})$ iff $X \in D^{\alpha}$. Let $X \in D^{\alpha}$ and $Y = \tilde{f}^{\alpha}_{D}(X)$. By definition of $\tilde{f}^{\alpha}_{D}$ and $\tilde{f}^{\alpha}$, $Y \in dom(\tilde{f}^{\alpha})$. By definition of incremental extension, $\tilde{f}^{\alpha} \circ \tilde{f}^{\alpha}_{D}(X) = Y \cup \tilde{f}^{\alpha}(Y)$. Moreover, by definition of $\tilde{f}^{\alpha}$, $\tilde{f}^{\alpha}(Y) = \bigcap_{\widehat{X} \in c(D,Y)} f(\widehat{X})$. Clearly, $c(D, Y) = c(D, X)$ and hence $\tilde{f}^{\alpha}(Y) = \bigcap_{\widehat{X} \in c(D,X)} f(\widehat{X})$. Finally, by definition of domain completion, $f^{\alpha}(X) = \tilde{f}^{\alpha} \circ \tilde{f}^{\alpha}_{D}(X)$. □

From proposition 18 and equation (9) we have that the open domain specification f°_{N} can be given by equation

$$f^{\circ}_{N} = \tilde{g}^{\alpha} \circ \tilde{g}^{\alpha}_{D} \circ f_{\mathrm{s}}, \tag{13}$$

where $g = f_{\mathrm{d}}$, $\alpha = \{is\}$ and $D = dom(f_d)$.

5.2. Realization theorems for domain completion and interpolation

Equation (13) above suggests that a representation for f°_{N} can be constructed from lp-functions representing $\tilde{g}^{\alpha}$ and $\tilde{g}^{\alpha}_{D}$. In the construction of these functions we will be aided by realization theorems for domain completions and interpolations.

5.3. Domain completion

Let C be a collection of constraints over signature σ, $\alpha \subseteq pred(\sigma)$, and D be a collection of complete states of σ satisfying C. We will be interested in representing the domain completion of D with respect to α. Such a representation is especially simple for constraints which are *binary* with respect to α, i.e., constraints whose bodies contain exactly two literals from $lit(\alpha)$. By the disjunctive image $d(c)$ of a binary constraint

$$c = \leftarrow l_1, l_2, \Gamma \tag{14}$$

where $l_1, l_2 \in lit(\alpha)$ we mean the disjunction $\overline{l_1} \vee \overline{l_2}$; by $d(C)$ we denote the set of disjunctive images of constraints from C. A set C of binary constraints is said to be *indefinite* if there is no $l \in lit(\sigma)$ such that $d(C) \models l$ or $d(C) \models \overline{l}$.

Notice that the set of constraints $\{\leftarrow p, q; \leftarrow \neg p, q\}$ (where $p, q \in \alpha$) is not indefinite.

Let $\tilde{\pi}_D$ be a program obtained from C by replacing each constraint $\leftarrow l_1, l_2, \Gamma$ by the rules

$$\left.\begin{array}{l} \overline{l_1} \leftarrow l_2, \Gamma, \\ \overline{l_2} \leftarrow l_1, \Gamma. \end{array}\right\} \tilde{\pi}_D$$

Theorem 19 (Realization theorem for domain completion). Let C be a collection of binary constraints over signature σ, $\alpha \subseteq pred(\sigma)$ and D a collection of complete states of σ satisfying C. If C is indefinite then a four-tuple $\{\tilde{\pi}_D, \sigma, \sigma, D^\alpha\}$ is an lp-function which represents the domain completion $\tilde{f}_D^\alpha$ of D.

In the proof of this theorem, which appears below, we use the following lemmas.

Lemma 20. Let $X \in D^\alpha$. Then for any $\widehat{X} \in c(D, X)$ there is a set $A \subseteq \widehat{X}$ which is the answer set of $\tilde{\pi}_D \cup X$.

Proof. Since $X \in D^\alpha$ we have that $c(D, X)$ is not empty. Let $\widehat{X} \in c(D, X)$. By definition of $c(D, X)$, $\widehat{X}$ is complete and satisfies the constraints from C. This implies that $\widehat{X}$ is closed under the rules of $\tilde{\pi}_D$. Consider a (possibly infinite) sequence X_i of sets of literals such that

- $X_0 = \widehat{X}$,
- X_{i+1} is a proper subset of X_i, which contains X and is closed under the rules of $\tilde{\pi}_D$.

Let $A = \bigcap X_i$. A contains X and is closed under the rules of $\tilde{\pi}_D$. (Suppose $l_1 \leftarrow l_2, \Gamma$ is a rule from $\tilde{\pi}_D$ and $\Gamma, l_2 \in A$. Then, for every i, $l_2, \Gamma \in X_i$. Since X_i is closed under the rules, l_1 is also in X_i and hence in A). Obviously, there is no smaller set of literals closed under $\tilde{\pi}_D \cup X$, i.e., A is one of its answer sets. Since our program does not contain negation as failure, A is its only answer set. □

The following corollary follows immediately from this lemma.

Corollary 21. For all $X \in D^\alpha$,

- $\tilde{\pi}_D \cup X$ is consistent,
- $\tilde{\pi}_D(X) \subseteq \bigcap_{\widehat{X} \in c(D,X)} \widehat{X}$.

Lemma 22. If $X \in D^\alpha$, then for every $\widehat{X} \in c(D, X)$, $\tilde{\pi}_D(\widehat{X}) = \widehat{X}$.

Proof. By definition of $c(D, X)$, $\widehat{X}$ is consistent. Thus to prove the lemma it only remains to show that $\widehat{X}$ is closed under the rules of $\tilde{\pi}_D$. Let $l \leftarrow l', \Gamma$ be any rule in $\tilde{\pi}_D$ and assume that $\Gamma, l' \in \widehat{X}$. By definition of $\tilde{\pi}_D$ there is a constraint $\leftarrow \overline{l}, l', \Gamma \in C$. Hence $\overline{l} \notin \widehat{X}$. By definition of D, $\widehat{X}$ is complete and therefore $l \in \widehat{X}$. □

Lemma 23. For all $X \in D^\alpha$, $\tilde{\pi}_D(X) \supseteq \bigcap_{\widehat{X} \in c(D,X)} \widehat{X}$.

Proof. Consider an arbitrary $X \in D^{\alpha}$ and suppose that the lemma is false, i.e., that there is a literal l such that

$$l \in \bigcap_{\widehat{X} \in c(D,X)} \widehat{X} \text{ but } l \notin \tilde{\pi}_D(X). \tag{15}$$

By lemma 22, the above is equivalent to

$$l \in \bigcap_{\widehat{X} \in c(D,X)} \tilde{\pi}_D\big(\widehat{X}\big) \text{ but } l \notin \tilde{\pi}_D(X). \tag{16}$$

We will first show that, under this assumption,

$$\overline{l} \notin \tilde{\pi}_D(X). \tag{17}$$

Suppose it does, i.e., $\overline{l} \in \tilde{\pi}_D(X)$. Then, since $X \in D^{\alpha}$, there is $\widehat{X} \in D$ containing X. Since $\tilde{\pi}_D$ does not contain negation as failure, we have that

$$\tilde{\pi}_D(X) \subseteq \tilde{\pi}_D\big(\widehat{X}\big).$$

This implies that $\overline{l} \in \tilde{\pi}_D(\widehat{X})$ which contradicts assumption (16). Hence (17) holds.

Now we will construct a set $Z \in c(D, X)$ containing $\overline{l}$. By corollary 21, $\tilde{\pi}_D(X)$ is consistent. Since it contains no negation as failure it has exactly one answer set, say, A. Let

$$\begin{aligned} C' &= \{\leftarrow l_1, l_2\colon \leftarrow l_1, l_2,\ \Gamma \in C,\ \Gamma \subseteq A\},\\ P_1 &= \big\{l_i\colon\ l_i \in \tilde{\pi}_D(X) \text{ or } \overline{l_i} \in \tilde{\pi}_D(X)\big\}, \quad \text{and}\\ P_2 &= \mathit{lit}(\alpha) \setminus P_1. \end{aligned}$$

Let C_1 be a collection of constraints from C' such that $\overline{l_1}$ or $\overline{l_2}$ is in A, and let $C_2 = C' \setminus C_1$.

Notice that no literal l_i occurring in constraints from C_2 belongs to P_1. (If it were the case then we would have that $\overline{l_i} \in A$ or $l_i \in A$. The former is impossible since it would place the constraint containing this literal into C_1. The latter is also impossible since in this case the second literal from this constraint would be falsified by A, again placing it in C_1.) By construction, P_2 is closed under $\neg$, i.e., for any $l_i \in \mathit{lit}(\sigma)$ if $l_i \in P_2$ then so is $\overline{l_i}$. This implies that if l_i belongs to a constraint of C_2 then $l_i, \overline{l_i} \in P_2$ and, therefore, interpretations of C_2 can be viewed as complete and consistent sets of literals from P_2. From (16) and (17) we have that $l \in P_2$. Since C is indefinite so is C_2 and therefore there is an interpretation B of C_2 which makes $d(C_2)$ true and l false. Now let

$$Z = A \cup B.$$

To show that Z is complete consider two cases: If $l_i \notin \mathit{lit}(\alpha)$ then, since $X \in D^{\alpha}$, we have that $l_i \in X$ or $\overline{l_i} \in X$ and hence either l_i or $\overline{l_i}$ is in A. Otherwise, $l_i \in P_1$ or $l_i \in P_2$. If $l_i \in P_1$ then, by definition of P_1, $l_i \in A$ or $\overline{l_i} \in A$. For $l_i \in P_2$ completeness follows from the construction of B.

Consistency follows from consistency of A and B and the fact that if $l_i \in B$ then $\overline{l_i} \notin A$.

It is also easy to check that, by construction, Z satisfies the constraints C and hence, $Z \in c(D, X)$. This contradicts our assumption that $l \in \bigcap_{\widehat{X} \in c(D,X)} \widehat{X}$ and therefore the lemma holds. □

Proof of theorem 19. We need to show that for all $X \in D^\alpha$, $\tilde{f}^\alpha_D(X) = \tilde{\pi}_D(X)$. By definition of domain completion, this is equivalent to

$$\tilde{\pi}_D(X) = \bigcap_{\widehat{X} \in c(D,X)} \widehat{X}$$

which follows immediately from corollary 21 and lemma 23. □

5.4. Interpolation

Now we discuss a realization theorem for interpolation of f-specifications. Let σ be a signature, $\alpha \subseteq pred(\sigma)$, and D be a collection of complete α-states of σ. Function f defined on the interior D^α of D is called *separable* if

$$\bigcap_{\widehat{X} \in c(D,X)} f(\widehat{X}) \subseteq f(X)$$

or, equivalently, if for any $X \in \mathit{dom}(f)$ and any output literal l such that $l \notin f(X)$ there is $\widehat{X} \in c(D, X)$ such that $l \notin f(\widehat{X})$.

The following propositions give simple sufficient conditions for separability which may help to better understand this notion.

Proposition 24. Let D be the set of complete states over some signature σ_i, $\alpha = \mathit{pred}(\sigma_i)$, and let π be an lp-function defined on $D^\alpha = \mathit{states}(\sigma_i)$, such that

1. The sets of input and output predicates of π are disjoint and input literals do not belong to the heads of π.
2. For any $l \in \sigma_i$, $l \notin \mathit{lit}(\pi)$ or $\overline{l} \notin \mathit{lit}(\pi)$. (Here by $\mathit{lit}(\pi)$ we mean the collection of all literals which occur in the rules of the ground instantiation of π.)

Then π is separable.

Proof (sketch). Let $X \in \mathit{dom}(\pi)$ and consider the set

$$X^* = X \cup \{l\colon\ \overline{l} \notin X,\ l \in \mathit{lit}(\pi)\} \cup \{l\colon\ l \in \mathit{atoms}(\sigma_i),\ \neg l \notin X,\ l \notin \mathit{lit}(\pi),\ \neg l \notin \mathit{lit}(\pi)\}.$$

By condition (2), X^* is consistent. Since it is also complete by construction we have that $X^* \in \mathit{dom}(\pi)$. By condition (1) $\mathit{lit}(\sigma_i)$ is a splitting set of programs $\pi \cup X$ and $\pi \cup X^*$. By the splitting set theorem and condition (1) we have that for any output

literal l, $l \in \pi(X)$ iff *red*$(\pi, X) \models l$ and $l \in \pi(X^*)$ iff *red*$(\pi, X^*) \models l$. But by construction of X^* and the definition of *red*, *red*$(\pi, X) =$ *red*(π, X^*), hence $l \in \pi(X)$ iff $l \in \pi(X^*)$ and therefore, π is separable. □

The next example shows that the last condition is essential.

Example 25. Let σ_i be a signature consisting of object constant a and predicate constant p, and consider $D = \{\{p(a)\}, \{\neg p(a)\}\}$ and a function f_1 defined on D^α by the program

$$q(a) \leftarrow p(a)$$
$$q(a) \leftarrow \neg p(a)$$

where $\sigma_o(f_1)$ consists of a and q. Let $X = \emptyset$. Obviously, $f_1(X) = \emptyset$ while $\bigcap_{\widehat{X} \in c(D,X)} f_1(\widehat{X}) = \{q(a)\}$ and hence f_1 is not separable.

In some cases, to establish separability of an lp-function π it is useful to represent π as the union of its independent components and to reduce the question of separability of π to separability of these components. This can be done in the following way: Let π be an lp-function with input signature σ_i and output signature σ_o. We assume that the input literals of π do not belong to the heads of rules of π. We say that π is decomposable into *independent components* $\pi_0, \ldots, \pi_n$ if $\pi = \pi_0 \cup \cdots \cup \pi_n$ and *lit*$(\pi_m) \cap$ *lit*$(\pi_k) \subseteq$ *lit*(σ_i) for any $k \neq m$.

Proposition 26. Let $\pi_0, \ldots, \pi_n$ be independent components of lp-function π. Then:

(i) for any $0 \leqslant k \leqslant n$, four-tuple $\{\pi_k, \sigma_i, \sigma_o,$ *dom*$(\pi)\}$ is an lp-function,

(ii) if π_k is separable for any $0 \leqslant k \leqslant n$ then so is π.

Proof. The proposition follows immediately from the splitting set theorem and definition of separability. □

Example 27. Consider σ, α and D from example 25 and let function f_2 with the output signature consisting of a, q_1 and q_2 be defined by a program

(1) $q_1(a) \leftarrow p(a)$,

(2) $q_2(a) \leftarrow \neg p(a)$.

Obviously, π can be decomposed into independent components π_1 and π_2 consisting of rules (1) and (2), respectively. By proposition 24, π_1 and π_2 are separable, and hence, by proposition 26, so is π.

The following simple observation proves to be useful for constructing interpolations.

Theorem 28 (Realization theorem for interpolation). Let f be a closed domain specification with domain D represented by an lp-function π and let $\tilde{\pi}$ be the result of replacing some occurrences of input literals l in $pos(\pi)$ by $not\,\bar{l}$. If $\{\tilde{\pi}, \sigma_{\rm i}(f), \sigma_{\rm o}(f), dom(\tilde{f})\}$ is a separable and monotonic lp-function then it represents the interpolation $\tilde{f}$ of f.

Proof. Let $\tilde{\pi}$ be a separable and monotonic lp-function. To show that it represents $\tilde{f}$, we need to show that

$$\tilde{\pi}(X) = \bigcap_{\widehat{X} \in c(D,X)} \pi\big(\widehat{X}\big).$$

From the separability of $\tilde{\pi}$ we have one direction, namely that

$$\bigcap_{\widehat{X} \in c(D,X)} \pi\big(\widehat{X}\big) \subseteq \tilde{\pi}(X).$$

By monotonicity, if $X \subseteq \widehat{X}$, then $\tilde{\pi}(X) \subseteq \tilde{\pi}(\widehat{X})$. Hence,

$$\tilde{\pi}(X) \subseteq \bigcap_{\widehat{X} \in c(D,X)} \tilde{\pi}\big(\widehat{X}\big).$$

Clearly, for any complete set $\widehat{X}$, $\tilde{\pi}(\widehat{X}) = \pi(\widehat{X})$. Therefore,

$$\tilde{\pi}(X) \subseteq \bigcap_{\widehat{X} \in c(D,X)} \pi\big(\widehat{X}\big). \qquad \square$$

Corollary 29. Let f, π and $\tilde{\pi}$ be as in theorem 28. If $\tilde{\pi} = \{\tilde{\pi}, \sigma_{\rm i}(f), \sigma_{\rm o}(f), dom(\tilde{f})\}$ is a separable lp-function with a signing S such that $S \cap (lit(\sigma_{\rm i}) \cup lit(\sigma_{\rm o})) = \emptyset$ then it represents the interpolation $\tilde{f}$ of f.

The corollary follows immediately from theorems 7 and 28.

5.5. *The declarative representation of open nets*

In this section we will go back to the task of building a computable lp-function π° representing the open domain specification f_N°. Recall that f_N° is defined via incremental extension and three f-specifications: $\tilde{g}^\alpha$, $\tilde{g}_D^\alpha$ and $f_{\rm s}$ where $g = f_{\rm d}$ and $\alpha = \{is\}$ (13). We start with finding a representation $\tilde{\pi}$ of $\tilde{g}^\alpha$. Theorem 28 will provide an important heuristic guidance for this construction. In particular, we will look for a transformation from this theorem which will turn the lp-function π (section 4.2) representing g into a monotonic lp-function. The only transformation achieving this goal is that of replacing the rule defining predicate *exceptional* in π by the rule

$$exceptional(X, D, S) \leftarrow exception(E, D, S), \\ not\ \neg is(X, E). \qquad (18)$$

Let us denote the resulting program by $\tilde{\pi}$ and use corollary 29 to prove the following proposition.

Proposition 30. $\tilde{\pi}$ represents $\tilde{g}^{\alpha}$.

Proof (sketch). We need to show that $\tilde{\pi}$ is an lp-function and that it is separable and monotonic.

1. (Consistency.) Let D be the domain of $g = f_{\mathrm{d}}$ and $\widetilde{D}$ be the domain of $\tilde{g}$. (Recall that *dom*(g) consists of complete states of $\sigma_{\mathrm{i}}(g)$ which satisfy the constraints (5) and *dom*($\tilde{g}$) is the set of all $\{is\}$-states of $\sigma_{\mathrm{i}}(g)$ which are maximally informative with respect to D). Let $X \in$ *dom*($\tilde{\pi}$). To show that $\pi_0 = \tilde{\pi} \cup X$ is consistent we use an argument similar to that which we used to prove consistency of $\pi \cup X$ in proposition 11. We use renaming to reduce $\tilde{\pi}$ to a program without $\neg$; use its stratifiability to entail existence of a unique answer set and show that this answer set is consistent.

2. (Separability.) Let $X \in \widetilde{D}$ and consider an output literal $l \notin \tilde{\pi}(X)$. Let us first assume that $l =$ *has*(x, p). We need to construct a cover $\widehat{X}$ of X such that $l \notin \tilde{\pi}(\widehat{X})$. We say that a class e is p-negative if there is a negative link from e to p in N; a p-negative class e can defeat *has*(x, p) if $\neg$*is*$(x, e) \notin X$. Let E_0 be the set of all classes which can defeat *has*(x, p) and consider

$$E = E_0 \cup \{c\colon\ \exists e\ (e \in E_0 \text{ and } \mathit{subclass}(e, c) \in N)\},$$
$$U = X \cup \{\mathit{is}(x, e)\colon\ e \in E\}, \quad \text{and}$$
$$\widehat{X} = U \cup \{\neg \mathit{is}(o, c)\colon\ \mathit{is}(o, c) \notin U\}.$$

We show that

(a) $\widehat{X}$ is a cover of X,

(b) $l \notin \tilde{\pi}(\widehat{X})$.

(a) By construction, $\widehat{X}$ is complete and consistent. We need to show that it satisfies the constraints (5).

Consider arbitrary o, c_1 and c_2 such that *subclass*$(c_1, c_2) \in N$ and *is*$(o, c_1) \in \widehat{X}$. First let us consider the case when *is*$(o, c_1) \in X$. Then any cover of X must contain *is*(o, c_2). Since X is maximally informative, *is*$(o, c_2) \in X$, and hence by consistency of $\widehat{X}$ the constraints (5) are satisfied. Suppose now that *is*$(o, c_1) \notin X$. If o is not x, then $\neg$*is*$(o, c_1) \in \widehat{X}$ and constraints (5) are satisfied by consistency of $\widehat{X}$. If o is x, then $c_1, c_2 \in E$ and, by construction, *is*$(x, c_2) \in \widehat{X}$. Consistency of $\widehat{X}$ again guarantees that constraints (5) are satisfied.

(b) Since $\widehat{X}$ is complete and consistent and *lit*(*is*) is a splitting set of $\tilde{\pi}$, we have that *red*$(\pi, \widehat{X})$ = *red*$(\tilde{\pi}, \widehat{X})$ and hence $\pi \cup \widehat{X}$ is consistent iff $\tilde{\pi} \cup \widehat{X}$ is consistent. Consistency of $\pi \cup \widehat{X}$ was established before and hence $\tilde{\pi} \cup \widehat{X}$ is also consistent and has a consistent answer set S. As shown in [34], *has*$(x, p) \in S$ iff there is class c such that *default*$(d, c, p, +) \in N$, *is*$(x, c) \in \widehat{X}$, *exceptional*$(x, d, +) \notin S$. Let c satisfy this condition and consider two cases:

(i) $\mathit{is}(x, c) \in X$.
Then, since $l \notin \tilde{\pi}(X)$, we have that for every class e such that $\mathit{exception}(e, d, +) \in S$, $\neg\mathit{is}(x, e) \notin X$. By construction of $\widehat{X}$ this implies that $\neg\mathit{is}(x, e) \notin S$, $\mathit{exceptional}(x, d, +) \in S$, and therefore $\mathit{has}(x, p) \notin \tilde{\pi}(\widehat{X})$.

(ii) $\mathit{is}(x, c) \notin X$.
Then $\mathit{is}(x, c) \in U$. By construction of U we have that there is a class e such that $\mathit{default}(d_2, e, p, -) \in N$, $\neg\mathit{is}(x, e) \notin X$ and either $e = c$ or $\mathit{subclass}(e, c) \in N$. By construction we have that $\mathit{is}(x, e) \in U$. Since our graph is acyclic, we have that $\mathit{exception}(e, d, +) \in S$, $\mathit{exceptional}(x, d, +) \in S$ and therefore $\mathit{has}(x, p) \notin \tilde{\pi}(\widehat{X})$. A similar argument works for $l = \neg\mathit{has}(x, p)$.

3. (Monotonicity.) Let $X_1, X_2 \in \mathit{dom}(\tilde{\pi})$ and $X_1 \subset X_2$. To show that

(i) $\tilde{\pi}(X_1) \subset \tilde{\pi}(X_2)$

we first use the splitting set theorem to eliminate from $\pi_1 = \tilde{\pi} \cup X_1$ and $\pi_2 = \tilde{\pi} \cup X_2$ all the occurrences of literals from $\mathit{lit}(\mathit{default})$. Since X_1 and X_2 are *is*-states of σ_{i} we have that $[X_1]_{\{\mathit{default}\}} = [X_2]_{\{\mathit{default}\}}$ and hence

(ii) $\mathit{top}(\pi_1, \mathit{lit}(\mathit{default})) = \mathit{top}(\pi_2, \mathit{lit}(\mathit{default}))$.

Let us denote this program by π'. By the splitting set theorem

(iii) $\pi'(X) = \tilde{\pi}(X)$ for any $X \in \mathit{dom}(\tilde{\pi})$.

Now observe that the set $S = \mathit{atoms}(\{\mathit{exception}, \mathit{exceptional}\})$ is a signing for π' and that $S \cap (\mathit{lit}(\sigma_{\mathrm{i}}(\pi) \cup \mathit{lit}(\sigma_{\mathrm{o}}(\pi))) = \emptyset$. The monotonicity condition follows from (iii) and theorem 7 and the conclusion of our proposition becomes an immediate consequence of theorem 28. □

To complete the construction of representation π° of f_N° we need to find a representation $\tilde{\pi}_D$ of the domain completion of $D = \mathit{dom}(g)$ with respect to $\alpha = \{\mathit{is}\}$. Recall that D is the collection of complete states of $\sigma_i(g)$ satisfying constraints (5). These constraints are binary with respect to α. It is also easy to see that they are indefinite and hence we can use the construction from theorem 19. The corresponding program $\tilde{\pi}_D$ consists of the rules

$$\mathit{is}(O, C_2) \leftarrow \mathit{is}(O, C_1), \mathit{subclass}(C_1, C_2), \tag{19}$$

$$\neg\mathit{is}(O, C_1) \leftarrow \neg\mathit{is}(O, C_2), \mathit{subclass}(C_1, C_2). \tag{20}$$

Proposition 31. $\tilde{\pi}_D$ represents $\tilde{g}_D^\alpha$.

Proof. Follows immediately from theorem 19. □

Finally, we can show that

Proposition 32. $\pi^\circ = \tilde{\pi} \cup \tilde{\pi}_D \cup \pi_s$ represents f_N°.

Proof. Follows immediately from the equation (13), propositions 30, 31, theorem 10, and the fact that $\tilde{\pi}$, $\tilde{\pi}_D$ and π_s have unique answer sets. □

As in the case of closed nets, proposition 32 shows the correctness of π° with respect to our specification if π° is viewed as a declarative program. However, due to the left recursion in rules (19) and (20), π° is not computable with respect to our interpreter $\mathcal{A}$. To achieve computability we need to change the order of literals in the bodies of these rules. The resulting program π° will look as follows:

$$\left.\begin{array}{rl}
\mathit{has}(X,P) \leftarrow & \mathit{default}(D,C,P,+),\\
& \mathit{is}(X,C),\\
& \mathit{not\ exceptional}(X,D,+).\\
\neg\mathit{has}(X,P) \leftarrow & \mathit{default}(D,C,P,-),\\
& \mathit{is}(X,C),\\
& \mathit{not\ exceptional}(X,D,-).\\
\mathit{exception}(E,D_1,+) \leftarrow & \mathit{default}(D_1,C,P,+),\\
& \mathit{default}(D_2,E,P,-),\\
& \mathit{not\ subclass}(C,E).\\
\mathit{exception}(E,D_1,-) \leftarrow & \mathit{default}(D_1,C,P,-),\\
& \mathit{default}(D_2,E,P,+),\\
& \mathit{not\ subclass}(C,E).\\
\mathit{exceptional}(X,D,S) \leftarrow & \mathit{exception}(E,D,S),\\
& \mathit{not}\ \neg\mathit{is}(X,E).\\
\mathit{is}(O,C_2) \leftarrow & \mathit{subclass}(C_1,C_2),\\
& \mathit{is}(O,C_1).\\
\neg\mathit{is}(O,C_1) \leftarrow & \mathit{subclass}(C_1,C_2),\\
& \neg\mathit{is}(O,C_2).\\
\mathit{subclass}(C_1,C_2) \leftarrow & \mathit{subclass}_0(C_1,C_2).\\
\mathit{subclass}(C_1,C_2) \leftarrow & \mathit{subclass}_0(C_1,C_3),\\
& \mathit{subclass}(C_3,C_2).\\
\neg\mathit{subclass}(C_1,C_2) \leftarrow & \mathit{not\ subclass}(C_1,C_2).
\end{array}\right\}\ \pi^\circ$$

Its computability can be shown by an argument similar to that in proposition 13. Notice that if we were to use an inference engine based on XSB which includes a form of loop checking, no changes in the rules (19) and (20) would be needed.

6. A simple generalization

In this section we generalize the knowledge representation problem associated with a net N by allowing strict (non-defeasible) links from objects to properties to belong to the net's input (see figure 2). We show that this generalization can be easily incorporated into the design. To do that we use another specification constructor from [26], called *input extension*.

Definition 33. Let f be a functional specification with disjoint sets of input and output predicates. An f-specification f^* with input signature $\sigma_{\rm i}(f)+\sigma_{\rm o}(f)$ and output signature $\sigma_{\rm o}(f)$ is called *input extension* of f if

(1) f^* is defined on elements of $dom(f)$ possibly expanded by consistent sets of literals from $\sigma_{\rm o}(f)$,

(2) for every $X \in dom(f)$, $f^*(X) = f(X)$,

(3) for any $Y \in dom(f^*)$ and any $l \in lit(\sigma_{\rm o}(f))$,

(i) if $l \in Y$ then $l \in f^*(Y)$,

(ii) if $l \notin Y$ and $\overline{l} \notin Y$ then $l \in f^*(Y)$ iff $l \in f(Y \cap lit(\sigma_i(f))$.

A new problem associated with a net N can be defined by f-specification

$$f^* = g^* \circ \tilde{g}^{\alpha}_D \circ f_{\rm s}, \tag{21}$$

where g^* is the input extension of $\tilde{g}^{\alpha}$.

To find an lp-function representing g^* we will use the following transformation from [26].

Definition 34. Let π be an lp-function. The result of replacing every rule

$$l_0 \leftarrow l_1, \ldots, l_m, not\ \ l_{m+1}, \ldots, not\ \ l_n$$

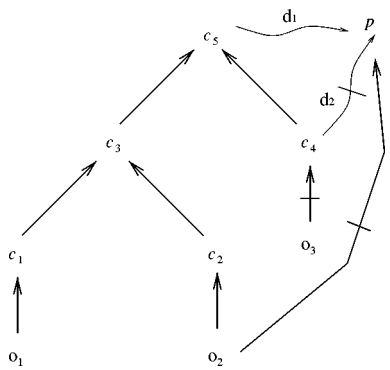

Figure 2. Hierarchy with links from objects to properties.

of π with $l_0 \in \mathit{lit}(\sigma_\circ(f))$ by the rule

$$l_0 \leftarrow l_1, \ldots, l_m, \mathit{not}\ \ l_{m+1}, \ldots, \mathit{not}\ \ l_n, \mathit{not}\ \ \bar{l}_0$$

is called the *guarded version* of π and is denoted by $\hat{\pi}$.

The following theorem is useful for constructing lp-functions representing input extensions.

Theorem 35 (Realization theorem for input extension [26]). Let f be a specification represented by lp-function π with signature σ. If the set $U = \mathit{lit}(\sigma) \backslash \mathit{lit}(\sigma_\circ)$ is a splitting set of π dividing π into two components $\pi_2 = \mathit{top}(\pi, U)$ and $\pi_1 = \mathit{base}(\pi, U)$, then lp-function $\pi^* = \pi_1 \cup \hat{\pi}_2$ represents the input extension f^* of f.

The representation π^* of input extension g^* of $\tilde{g}^\alpha$ is obtained by replacing the *has* rules in π° (section 5.5) by

$$\left.\begin{array}{rl} \mathit{has}(x,p) \leftarrow & \mathit{default}(d,c,p,+), \\ & \mathit{is}(x,c), \\ & \mathit{not\ exceptional}(x,d,+), \\ & \mathit{not}\ \neg\mathit{has}(x,p). \\ \neg\mathit{has}(x,p) \leftarrow & \mathit{default}(d,c,p,-), \\ & \mathit{is}(x,c), \\ & \mathit{not\ exceptional}(x,d,-) \\ & \mathit{not\ has}(x,p). \end{array}\right\}$$

The following proposition follows immediately from equation (21), the construction of π^*, and theorems 35 and 10.

Proposition 36.

(i) π^* represents g^*,

(ii) $\pi^* \cup \tilde{\pi}_D \cup \pi_s$ represents f_*.

This example again demonstrates that specification constructors and their realization theorems provided a useful heuristic guidance for specifying knowledge representation problems and for building program provably satisfying these specifications.

7. Conclusions

In this paper we discussed a systematic methodology of solving certain types of knowledge representation problems in logic programming. The methodology was illustrated by a detailed development of solutions of several knowledge representation

problems associated with simple taxonomic hierarchies. In each case we started with a functional specification of a problem constructed from specifications of simpler problems with the help of specification constructors, and used various realization theorems to guide the process of representing these specifications by declarative programs of A-Prolog. We also demonstrated how these programs can be transformed into executable programs of Prolog and XSB. The paper also contains previously unpublished proofs of several realization theorems.

Even though the formalization of inheritance reasoning was extensively studied in the AI literature [15,16,18,24,27,28,41,47], the emphasis on precise specifications of the problems allowed us to come up with a clearer picture of dependencies of various types of inheritance reasoning on the closed and open world assumptions about the problem domain and on the type of the allowed updates. In particular, we addressed the question of reasoning in open nets, which, to the best of our knowledge, was never investigated before. We are currently working on studying the applicability of our approach to more general forms of inheritance, to reasoning about actions and change, and to several other domains. The results so far seem to be rather promising.

Acknowledgements

Many people contributed to the development of ideas presented in this paper and we cannot name all of them here. We would like to mention however our collaborators who directly contributed to this approach – H. Przymusinska, C. Baral and O. Kosheleva. Our thanks to them and to all the others. The authors are grateful to the anonymous referees for their comments and suggestions for improving the paper. The first author acknowledges the support of NASA under grant NCCW-0089.

References

[1] J.J. Alferes and L.M. Pereira, *Reasoning with Logic Programming*, Lecture Notes in Artificial Intelligence (Springer, Berlin, 1996).

[2] K. Apt, H. Blair and A. Walker, Towards a theory of declarative knowledge, in: *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker (Morgan Kaufmann, San Mateo, CA, 1988) pp. 89–148.

[3] K. Apt and D. Pedreschi, Proving termination in general Prolog programs, in: *Proc. of the International Conference on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, Vol. 526 (Springer, Berlin, 1991) pp. 265–289.

[4] K. Apt and A. Pellegrini, On the occur-check free logic programs, ACM Trans. Programming Languages and Systems 16(3) (1994) 687–726.

[5] C. Baral and M. Gelfond, Logic programming and knowledge representation, J. Logic Programming 12 (1994) 1–80.

[6] C. Baral, M. Gelfond and O. Kosheleva, Expanding queries to incomplete databases by interpolating general logic programs, J. Logic Programming 35 (1998) 195–230.

[7] N. Bidoit and C. Froidevaux, Minimalism subsumes default logic and circumscription, in: *Proc. of LICS-87* (1987) pp. 89–97.

[8] D. Chan, Constructive negation based on the completed databases, in: *Proc. 5th International Conference and Symposium on Logic Programming* (1988) pp. 111–125.
[9] W. Chen, Extending Prolog with nonmonotonic reasoning, J. Logic Programming 27(2) (1996) 169–183.
[10] W. Chen, T. Swift and D. Warren, Efficient top-down computation of queries under the well-founded semantics, J. Logic Programming 24(3) (1995) 161–201.
[11] P. Cholewinski, W. Marek and M. Truszczynski, Default reasoning system DeReS, in: *Proc. of KR-96* (1996) pp. 518–528.
[12] K. Clark, Negation as failure, in: *Logic and Data Bases*, eds. H. Gallaire and J. Minker (Plenum Press, NY, 1978) pp. 293–322.
[13] P. Dembinski and J. Maluszynski, AND-parallelism with intelligent backtracking for annotated logic programs, in: *Logic Programming: Proc. of the International Symposium on Logic Programming*, eds. V. Saraswat and K. Ueda (1985) pp. 25–38.
[14] Y. Deville, *Logic Programming: Systematic Program Development* (Addison-Wesley, Reading, MA, 1990).
[15] P.M. Dung and T.C. Son, Nonmonotonic inheritance, argumentation, and logic programming, in: *Proceedings of LPNMR* (1995) pp. 316–329.
[16] P.M. Dung and T.C. Son, An argumentation-theoretic approach to reasoning with specificity, in: *Proceedings of KR '96* (1996) pp. 407–418.
[17] T. Eiter, N. Leone, C. Mateis, G. Pfeifer and F. Scarcello, A deductive system for non-monotonic reasoning, in: *Proc. 4th LPNMR*, Lecture Notes in Computer Science, Vol. 1265 (Springer, Berlin, 1997) pp. 363–374.
[18] D. Etherington and R. Reiter, On inheritance hierarchies with exceptions, in: *Proceedings of AAAI-83* (1983) pp. 104–108.
[19] F. Fages, Consistency of Clark's completion and existence of stable models, J. Methods Logic Comput. Sci. 1(1) (1994) 51–60.
[20] M. Gelfond and A. Gabaldon, From functional specifications to logic programs, in: *Logic Programming: Proc. of the 1997 International Symposium*, ed. J. Maluszynski (1997) pp. 355–371.
[21] M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, in: *Logic Programming: Proc. of the 5th International Conference and Symposium*, eds. R. Kowalski and K. Bowen (1988) pp. 1070–1080.
[22] M. Gelfond and V. Lifschitz, Logic programs with classical negation, in: *Logic Programming: Proc. of the 7th International Conference*, eds. D. Warren and P. Szeredi (1990) pp. 579–597.
[23] M. Gelfond and V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Generation Computing 9 (1991) 365–385.
[24] M. Gelfond and H. Przymusinska, Formalization of inheritance reasoning in autoepistemic logic, Fund. Inform. 13 (1990) 403–445.
[25] M. Gelfond and H. Przymusinska, On consistency and completeness of autoepistemic theories, Fund. Inform. 16(1) (1992) 59–92.
[26] M. Gelfond and H. Przymusinska, Towards a theory of elaboration tolerance: logic programming approach, J. Software Eng. Knowledge Eng. 6(1) (1996) 89–112.
[27] J.F. Horty, Some direct theories of non-monotonic inheritance, in: *Handbook of Logic in Artificial Intelligence and Logic Programming*, eds. D. Gabbay, C. Hogger and J.A. Robinson (1994).
[28] J.F. Horty, R.H. Thomason and D.S. Touretzky, A skeptical theory of inheritance in nonmonotonic semantic networks, Artificial Intelligence 42(2–3) (1990) 311–348.
[29] K. Kunen, Signed data dependencies in logic programs, J. Logic Programming 7(3) (1989) 231–245.
[30] V. Lifschitz, Restricted monotonicity, in: *Proc. of AAAI-93* (1993) pp. 432–437.
[31] V. Lifschitz, Foundations of logic programming, in: *Principles of Knowledge Representation*, ed. G. Brewka (CSLI Publications, 1996) pp. 69–128.

[32] V. Lifschitz and H. Turner, Splitting a logic program, in: *Proc. 11th ICLP*, ed. P. Van Hentenryck (1994) pp. 23–38.
[33] F. Lin, A study of nonmonotonic reasoning, Ph.D. thesis, Stanford University (1991).
[34] W. Marek and V.S. Subrahmanian, The relationship between logic program semantics and non–monotonic reasoning, in: *Proc. of the 6th International Conference on Logic Programming*, eds. G. Levi and M. Martelli (1989) pp. 600–617.
[35] N. McCain and H. Turner, Language independence and language tolerance in logic programs, in: *Proc. of the 11th International Conference on Logic Programming* (1994) pp. 38–57.
[36] L. Niemela and P. Simons, Efficient implementation of the well-founded and stable model semantics, in: *Proc. of JICSLP-96* (MIT Press, Cambridge, MA, 1996).
[37] L. Pereira and J. Alferes, Well-founded semantics for logic programs with explicit negation, in: *Proc. of the 10th European Conference on Artificial Intelligence* (1992) pp. 102–106.
[38] T. Przymusinki, On constructive negation in logic programming, in: *Proc. of the North American Conference of Logic Programming* (1989) pp. 16–20.
[39] R. Reiter, On closed world data bases, in: *Logic and Data Bases*, eds. H. Gallaire and J. Minker (Plenum Press, New York, 1978) pp. 119–140.
[40] R. Reiter, A logic for default reasoning, Artificial Intelligence 13(1–2) 81–132.
[41] L.A. Stein, Resolving ambiguity in non-monotonic inheritance hierarchies, Artificial Intelligence 55(2–3) (1992) 259–310.
[42] K. Stroetmann, A completeness result for SLDNF-resolution, J. Logic Programming 15(4) (1993) 337–355.
[43] D. Touretzky, *The Mathematics of Inheritance Systems* (Morgan Kaufmann, Los Altos, CA, 1986).
[44] H. Turner, A monotonicity theorem for extended logic programs, in: *Proc. 10th ICLP*, ed. D.S. Warren (1993) pp. 567–585.
[45] H. Turner, Signed logic programs, in: *Proc. of ILPS*, ed. M. Bruynooghe (1994) pp. 61–75.
[46] A. Van Gelder, K. Ross and J. Schlipf, The well-founded semantics for general logic programs, Journal of the ACM 38(3) (1991) 620–650.
[47] X. Wang, J.H. You and L.Y. Yuan, A default interpretation of defeasible network, in: *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)* (1997) pp. 156–161.