# Security protocols verification in Abductive Logic Programming: a case study

Marco Alberti[1], Federico Chesani[2], Marco Gavanelli[1], Evelina Lamma[1], Paola Mello[2], and Paolo Torroni[2]

[1] ENDIF, Università di Ferrara - Via Saragat, 1 - 44100 Ferrara (Italy).
Email: {malberti|mgavanelli|elamma}@ing.unife.it
[2] DEIS, Università di Bologna - Viale Risorgimento 2 - 40126 Bologna (Italy).
Email: {fchesani|pmello|ptorroni}@deis.unibo.it

**Abstract.** In this paper we present by a case study an approach to the verification of security protocols based on Abductive Logic Programming.
We start from the perspective of open multi-agent systems, where the internal architecture of the individual system's components may not be completely specified, but it is important to infer and prove properties about the overall system behaviour. We take a formal approach based on Computational Logic, to address verification at two orthogonal levels: 'static' verification of protocol properties (which can guarantee, at design time, that some properties are a logical consequence of the protocol), and 'dynamic' verification of compliance of agent communication (which checks, at runtime, that the agents do actually follow the protocol).
We adopt as a running example the well-known Needham-Schroeder protocol. We first show how the protocol can be specified in our previously developed *SOCS-SI* framework, and then demonstrate the two types of verification.

## 1 Introduction

The recent and fast growth of network infrastructures, such as the Internet, is allowing for a new range of scenarios and styles of business-making and transaction-management. In this context, the use of security protocols has become common practice in a community of users who often operate in the hope (and sometimes in the trust) that they can rely on a technology which protects their private information and makes their communications secure and reliable. A large number of formal methods and tools have been developed to analyse security protocols, achieving notable results in determining their strengths (by showing their security properties) and their weaknesses (by identifying attacks on them).

The need for well defined protocols is even more apparent in the context of multi-agent systems, where agents are abstraction for autonomous computational entities which can act on behalf of their users. By well defined, we mean that protocols should be specified so as to guarantee that, provided that the

agents follow them, the resulting interaction will exhibit some desirable properties. In order to achieve reliability and users' trust, formal proofs of such properties need to be provided. We call the generation of such formal proofs *static verification of protocol properties*. A tool performing this activity automatically, rather than by hand, is an obvious request.

Open agent societies are defined as dynamic groups of agents, where new agents can join the society at any time, without disclosing their internals or specifications, nor providing any formal credential of being "well behaved". Open agent societies are a useful setting for heterogenous agent to interact; but, since no assumptions can be made about the agents and their behaviour, it cannot be assumed that the agents will follow the protocols. Therefore, at run-time, the resulting agent interaction may not exhibit the protocol properties that were verified statically at design time. However, one possible way to tackle (at least partly) this problem is to be able to guarantee that, if one agent misbehaves, its violation will be detected (and, possibly, sanctioned). Following Guerin and Pitt [GP02], we call this *on-the-fly verification of compliance*. This kind of verification should be performed by a trusted entity, external to the agents.

In previous work, and in the context of the EU-funded SOCS project [SOCa] we developed a Computational Logic-based framework, called *SOCS-SI* (where *SI* stands for *Social Infrastructure*), for the specification of agent interaction. In order to make *SOCS-SI* applicable to open agent societies, the specifications refer to the *observable* agent behaviour, rather than to the agents' internals or policies, and do not overconstrain the agent behaviour. We have shown in previous work that *SOCS-SI* is suitable for semantic specification of agent communication languages [ACG+03], and that it lends itself to the definition of a range of agent interaction protocols [AGL+03b]. A repository of protocols is available on the web [SOCb].

In this paper, we demonstrate by a case study on the well known Needham-Schroeder security protocol [NS78] how the *SOCS-SI* framework supports both static verification of protocol properties and on-the-fly verification of compliance. The two kinds of verifications are achieved by means of the operational counterpart of the *SOCS-SI* framework, consisting of two abductive proof procedures ($\mathcal{S}$CIFF and g-$\mathcal{S}$CIFF). Notably, the same specification of the protocol in our language is used for both kinds of verification: in this way, the protocol designer is relieved from a time consuming (and, what is worse, possibly erroneous) translation.

*SOCS-SI*, together with its proof procedures, can thus be seen as a tool which lets the protocol designer automatically verify: (*i*) at design time, that a protocol enjoys some desirable properties, and (*ii*) at runtime, that the agents follow the protocol, so making the interaction indeed exhibit the properties.

The paper is structured as follows. In Sect. 2, we describe an implementation of the well-known Needham Schroeder Public Key authentication protocol in our framework, and in Sect. 3 we show how we perform on-the-fly verification of compliance and static verification of properties of the protocol. Related work and conclusions follow.

## 2 Specifying the Needham-Schroeder Public Key encryption protocol

In this section, we show how our framework can be used to represent the well-known Needham-Schroeder security protocol [NS78].

(1) $A \rightarrow B : \langle N_A, A \rangle_{pub\_key(B)}$
(2) $B \rightarrow A : \langle N_A, N_B \rangle_{pub\_key(A)}$
(3) $A \rightarrow B : \langle N_B \rangle_{pub\_key(B)}$

**Fig. 1.** The Needham-Schroeder protocol (simplified version)

The protocol consists of seven steps, but, as other authors do, we focus on a simplified version consisting of three steps, where we assume that the agents know the public key of the other agents. A protocol run can be represented as in Figure 1, where $A \rightarrow B : \langle M \rangle_{PK}$ means that $A$ has sent to $B$ a message $M$, encrypted with the key $PK$.

(1) $a \rightarrow i : \langle N_a, a \rangle_{pub\_key(i)}$
(2) $i \rightarrow b : \langle N_a, a \rangle_{pub\_key(b)}$
(3) $b \rightarrow i : \langle N_a, N_b \rangle_{pub\_key(a)}$
(4) $i \rightarrow a : \langle N_a, N_b \rangle_{pub\_key(a)}$
(5) $a \rightarrow i : \langle N_b \rangle_{pub\_key(i)}$
(6) $i \rightarrow b : \langle N_b \rangle_{pub\_key(b)}$

**Fig. 2.** Lowe's attack on the Needham-Schroeder protocol

**Lowe's attack on the protocol.** Eighteen years after the publication of the Needham-Schroeder protocol, Lowe [Low96] proved it to be prone to a security attack. Lowe's attack on the protocol is presented in Figure 2, where a third agent $i$ (standing for *intruder*) manages to successfully authenticate itself as agent $a$ with a third agent $b$, by exploiting the information obtained in a legitimate dialogue with $a$.

### 2.1 The social model

In this section we give a brief summary of the *SOCS-SI* social framework developed within the EU-funded SOCS project [SOCa] [3] to specify interaction protocols for open societies of agents in a declarative way.

The agent interaction is observed and recorded by the social infrastructure in a set **HAP** (called *history*), of *events*. Events are represented as ground atoms

$$\mathbf{H}(Event[, Time])$$

[3] The reader can refer to [AGL+03a] for a more detailed description.

The term *Event* describes the event that has happened, according to application-specific conventions (e.g., a message sent or a payment issued); *Time* (optional) is a number, meant to represent the time at which the event has happened.

For example,

$$\mathbf{H}(send(a, b, content(key(k_b), agent(a), nonce(n_a))), 1)$$

could represent the fact that agent $a$ sent to agent $b$ a message consisting its own identifier ($a$) and a nonce ($n_a$), encrypted with the key $k_b$, at time 1.

While events represent the actual agent behaviour, the desired agent behaviour is represented by *expectations*. Expectations are "positive" when they refer to events that are expected to happen, and "negative" when they refer to events that are expected *not* to happen. The following syntax is adopted

$$\mathbf{E}(Event[, Time]) \qquad \mathbf{EN}(Event[, Time])$$

for, respectively, positive and negative expectations. Differently from events, expectations can contain variables (we follow the Prolog convention of representing variables with capitalized identifiers) and CLP [JM94] constraints can be imposed on the variables. This is because the desired agent behaviour may be under-specified (hence variables), yet subject to restriction (hence CLP constraints).

For instance,

$$e(send(a, b, content(key(k_b), nonce(n_b), empty(0))), T)$$

could represent the expectation for agent $a$ to send to agent $b$ a message consisting of a nonce ($n_b$) and an empty part ($empty(0)$), encrypted with a key $k_b$, at time $T$. A CLP constraint such as $T \leq 10$ can be imposed on the time variable, to express a deadline.

Explicit negation can be applied to expectations ($\neg\mathbf{E}$ and $\neg\mathbf{EN}$).

A protocol specification $\mathcal{S} = \langle KB_S, \mathcal{IC}_S \rangle$ is composed of:

- the *Social Knowledge Base* ($KB_S$) is a logic program whose clauses can have expectations and CLP constraints in their bodies. It can be used to express domain-specific knowledge (such as, for instance, deadlines);
- a set $\mathcal{IC}_S$ of *Social Integrity Constraints* (also SICs, for short, in the following): rules of the form $Body \rightarrow Head$. SICs are used to express how the actual agent behaviour generates expectations on their behaviour; examples can be found in the following sections.

In abductive frameworks [KKT93], *abducibles* represent hypotheses, and abduction is used to select a set of hypotheses consistent with some specification. In our (abductive) framework, expectations are abducibles, and the abductive semantics is used to select a desired behaviour consistent with the instance in question.

In particular, we say that a history **HAP** is *compliant* to a specification $\mathcal{S} = \langle KB_S, \mathcal{IC}_S \rangle$ iff there existss a set **EXP** of expectations that is

– $\mathcal{IC}_S$-*consistent*: it must entail $\mathcal{IC}_S$, for the given $\mathcal{S}_{\mathbf{HAP}}$;
– $\neg$-*consistent*: for any $p$, **EXP** cannot include $\{\mathbf{E}(P), \neg\mathbf{E}(p)\}$ or $\{\mathbf{EN}(p), \neg\mathbf{EN}(p)\}$;
– **E**-*consistent*: for any $p$, **EXP** cannot include $\{\mathbf{E}(p), \mathbf{EN}(p)\}$ (an event cannot be both expected to happen and expected not to happen);
– *fulfilled*: **EXP** cannot contain $\mathbf{EN}(p)$ if **HAP** contains $\mathbf{H}(p)$, and **EXP** cannot contain $\mathbf{E}(p)$ if **HAP** does not contain $\mathbf{H}(p)$.

In order to support goal-oriented societies, **EXP** is also required to entail, together with $KB_S$, a *goal* $\mathcal{G}$ which is defined as a conjunction of literals.

### 2.2 Representing the Needham-Schroeder protocol in the *SOCS-SI* social model

With the atom:

$$\mathbf{H}(send(X, Y, content(key(K), Term1, Term_2)), T_1)$$

we mean that a message is sent by an agent $X$ to an agent $Y$; the content of the message consists of the two terms $Term_1$ and $Term_2$ and has been encrypted with the key $K$. $T_1$ is the time at which $Y$ receives the message.

The interaction of Fig. 1, for instance, can be expressed as follows:

$$\mathbf{H}(send(a, b, content(key(k_b), agent(a), nonce(n_a))), 1)$$
$$\mathbf{H}(send(b, a, content(key(k_a), nonce(n_a), nonce(n_b))), 2)$$
$$\mathbf{H}(send(a, b, content(key(k_b), nonce(n_b), empty(0))), 3)$$

A first group of SICs, depicted in Fig. 3, defines the protocol itself, i.e, the expected sequence of messages.

```
    H( send( X, B, content( key( KB), agent( A), nonce( NA))), T1)
--->
    E( send( B, X, content( key( KA), nonce( NA), nonce( NB))), T2)
    /\ NA!=NB /\ T2 > T1.

    H( send( X, B, content( key( KB), agent( A), nonce( NA))), T1)
    /\ H( send( B, X, content( key( KA), nonce( NA), nonce( NB))), T2)
    /\ T2 > T1
--->
    E( send( X, B, content( key( KB), nonce( NB), empty( 0))), T3)
    /\  T3 > T2.
```

**Fig. 3.** Social Integrity Constraints defining the Needham-Schroeder protocol.

The first SIC of Fig. 3 states that, whenever an agent $B$ receives a message from agent $X$, and this message contains the name of some agent $A$ (possibly the name of $X$ himself), some nonce $N_A$, encrypted with $B$'s public key $K_B$, then a message is expected to be sent at a later time from $B$ to $X$, containing the original nonce $N_A$ and a new nonce $N_B$, encrypted with the public key of $A$.

The second SIC of Fig. 3 expresses that if two messages have been sent, with the characteristics that: *a*) the first message has been sent at the instant $T_1$, from $X$ to $B$, containing the name of some agent $A$ and some nonce $N_A$, encrypted with some public key $K_B$; and *b*) the second message has been sent at a later instant $T_2$, from $B$ to $X$, containing the original nonce $N_A$ and a new nonce $N_B$, encrypted with the public key of $A$; then a third message is expected to be sent from $X$ to $B$, containing $N_B$, and encrypted with the public key of $B$.

```
    H( send( X, Y, content( key( KY), Term1, Term2)), T0)
    /\ one_of(NX, Term1, Term2) /\ not isNonce( X, NX)
--->
    E( send( V, X, content( key( KX), Term3, Term4)), T1)
    /\ X!=V /\ isPublicKey( X, KX) /\ T1 < T0
    /\ one_of (nonce(NX), Term1, Term2)
\/
    E( send( V, X, content( key( KY), Term1, Term2)), T2)
    /\ T2 < T0
```

**Fig. 4.** Social Integrity Constraint expressing that an agent cannot guess a *nonce* generated by another agent (after Dolev-Yao [DY83]).

The second group of SICs consists of the one in Fig. 4, which expresses the condition that an agent is not able to guess another agent's *nonce*. The predicate *one_of(A, B, C)*, defined in the $KB_S$, is true when $A$ unifies with at least one of $B$ and $C$. The SIC says that, if agent $X$ sends to another agent $Y$ a message containing a nonce that $X$ did not create, then $X$ must have received $N_X$ previously in a message encrypted with $X$'s public key, or $X$ must be forwarding a message that it has received.

## 3 Verification of security protocols

In this section we show the application of the *SOCS-SI* social framework to on-the-fly verification of compliance and static verification of protocol properties, adopting the well-known Needham-Schroeder security protocol as a case study.

In our approach, both types of verification are applied to the same specification of the protocol, without the need for a translation: the protocol designer, in this way, can be sure that the protocol for which he or she has verified formal properties will be the same that the agents will be required to follow.

The two types of verification are achieved by means of two abductive proof procedures, $\mathcal{S}$CIFF and g-$\mathcal{S}$CIFF, which are closely related. In fact, the proof procedure used for the static verification of protocol properties (g-$\mathcal{S}$CIFF) is defined as an extension of the one used for on-the-fly verification of compliance ($\mathcal{S}$CIFF): for this reason, we first present on-the-fly verification, although, in the intended use of *SOCS-SI*, static verification would come first.

```
h(send( a, b, content( key( kb), agent( a), nonce( na))), 1).
h(send( b, a, content( key( ka), nonce( na), nonce( nb))), 2).
h(send( a, b, content( key( kb), nonce( nb), empty( 0))), 3).
```

**Fig. 5.** A compliant history.

```
h(send( a, b, content( key( kb), agent( a), nonce( na))), 1).
h(send( b, a, content( key( ka), nonce( na), nonce( nb))), 2).
```

**Fig. 6.** A non-compliant history (the third message is missing).

### 3.1 On-the-fly verification of compliance

In this section, we show examples where the $\mathcal{S}$CIFF proof procedure is used as a tool for verifying that the agent interaction is *compliant* (see Sect. 2.1) to a protocol.

$\mathcal{S}$CIFFverifies compliance by trying to generate a set **EXP** which fulfils the four conditions defined in Section 2.1.

The $\mathcal{S}$CIFF proof procedure [AGL$^+$04] is an extension of the IFF proof procedure[4] [FK97]. Operationally, if the agent interaction has been compliant to the protocol, $\mathcal{S}$CIFF reports success and the required set **EXP** of expectations (see sect. 2.1); otherwise, it reports failure. The proof procedure has been proven sound and complete with respect to the declarative semantics. A result of *termination* has also been proved, when the knowledge of the society is *acyclic*.

The following examples can be verified by means of $\mathcal{S}$CIFF. Fig. 5 shows an example of a history compliant to the SICs of Fig. 3 and Fig. 4.

Fig. 6 instead shows an example of a history that is not compliant to such SICs. The reason is that the protocol has not been completed. In fact, the two events in the history propagate the second integrity constraints of Fig. 3 and impose an expectation

```
e(send( a, b, content( key( kb), nonce( nb), empty( 0))), T3)
```

(with the CLP constraint `T3>2`), not fulfilled by any event in the history.

The history in Fig. 7, instead, while containing a complete protocol run, is found by $\mathcal{S}$CIFF to violate the integrity constraints because agent `a` has used a nonce (`nc`) that it cannot know, because is not one of `a`'s nonces (as defined in the $KB_S$) and `a` has not received it in any previous message. In terms of integrity constraints, the history satisfies those in Fig. 3, but it violates the one in Fig. 4.

Fig. 8 depicts Lowe's attack, which $\mathcal{S}$CIFF finds compliant both to the protocol and to the SICs in Fig. 4.

[4] Extended because, unlike IFF, it copes with (*i*) universally quantified variables in abducibles, (*ii*) dynamically incoming events, (*iii*) consistency, fulfillment and violations, and (*iv*) CLP-like constraints.

```
h(send( a, b, content( key( kb), agent( a), nonce( nc))), 1).
h(send( b, a, content( key( ka), nonce( nc), nonce( nb))), 2).
h(send( a, b, content( key( kb), nonce( nb), empty( 0))), 3).
```

**Fig. 7.** A non-compliant history (agent `a` has used a nonce that it cannot hold).

```
h(send( a, i, content( key( ki), agent( a), nonce( na))), 1).
h(send( i, b, content( key( kb), agent( a), nonce( na))), 2).
h(send( b, i, content( key( ka), nonce( na), nonce( nb))), 3).
h(send( i, a, content( key( ka), nonce( na), nonce( nb))), 4).
h(send( a, i, content( key( ki), nonce( nb), empty( 0))), 5).
h(send( i, b, content( key( kb), nonce( nb), empty( 0))), 6).
```

**Fig. 8.** Lowe's attack, recognized as a compliant history.

### 3.2 Static verification of protocol properties

In order to verify protocol properties, we have developed an extension of the $\mathcal{S}$CIFF proof procedure, called g-$\mathcal{S}$CIFF. Besides verifying whether a history is compliant to a protocol, g-$\mathcal{S}$CIFF is able to generate a compliant history, given a protocol. g-$\mathcal{S}$CIFF has been proved sound, which means that the histories that it generates (in case of success) are guaranteed to be compliant to the interaction protocols while entailing the goal. Note that the histories generated by g-$\mathcal{S}$CIFF are in general not only a collection of ground events, like the sets **HAP** given as an input to $\mathcal{S}$CIFF. They can, in fact, contain variables, which means that they represent *classes* of event histories.

The use of g-$\mathcal{S}$CIFF for verification of properties can be summarised as follows.

We express properties to be verified as formulae, defined as conjunctions of literals. If we want to verify if a formula $f$ is a property of a protocol $\mathcal{P}$, we express the protocol in our language and $\neg f$ as a g-$\mathcal{S}$CIFF goal. Two results are possible:

- g-$\mathcal{S}$CIFF returns success, generating a history **HAP**. Thanks to the soundness of g-$\mathcal{S}$CIFF, **HAP** entails $\neg f$ while being compliant to $\mathcal{P}$: $f$ is not a property of $\mathcal{P}$, **HAP** being a counterexample;
- g-$\mathcal{S}$CIFF returns failure: this suggests that $f$ is a property of $\mathcal{P}$[5].

In the following, we show the automatic generation of Lowe's attack by g-$\mathcal{S}$CIFF, obtained as a counterexample of a property of the Needham-Schroeder protocol. The property that we want to disprove is $\mathcal{P}_{trust}$ defined as $trust_B(X, A) \rightarrow X = A$, i.e., if $B$ trusts that he is communicating with $A$, then he is indeed communicating with $A$. The notion of $trust_B(X, A)$ is defined as follows:

**Definition 1** ($trust_B(X, A)$)**.**

[5] If we had a completeness result for g-$\mathcal{S}$CIFF, this would indeed be a proof and not only a suggestion.

*B trusts that the agent X he is communicating with is A, once two messages have been exchanged at times $T_1$ and $T_2$, $T_1 < T_2$, having the following sender, recipient, and content:*

$(T_1)\ B \rightarrow X : \{N_B, \dots\}_{pub\_key(A)}$
$(T_2)\ X \rightarrow B : \{N_B, \dots\}_{pub\_key(B)}$

*where $N_B$ is a nonce generated by B.*

In order to check whether $\mathcal{P}_{trust}$ is a property of the protocol, we ground $\mathcal{P}_{trust}$ and define its negation $\neg\mathcal{P}_{trust}$ as a goal, $g$, where we choose to assign to $A$, $B$, and $X$ the values $a$, $b$ and $i$:

$g \leftarrow isNonce(NA), NA \neq nb,$
$\quad \mathbf{E}(send(b, i, content(key(ka), nonce(NA), nonce(nb))), 3),$
$\quad \mathbf{E}(send(i, b, content(key(kb), nonce(nb), empty(0))), 6).$

This goal negates $\mathcal{P}_{trust}$, in that $b$ has sent to an agent one of its nonces, encrypted with $a$'s public key, and has received the nonce back unencrypted, so being entitled to believe the other agent to be $a$; whereas the other agent is, in fact, $i$.
Besides defining $g$ for three specific agents, we also assign definite time points (3 and 6) in order to improve the efficiency of the proof by exploiting constraint propagation.

Running the g-$\mathcal{S}$CIFF on $g$ results in a compliant history:
$\mathbf{HAP}_g = \{\ h(send(a, i, content(key(ki), agent(a), nonce(na))), 1),$
$h(send(i, b, content(key(kb), agent(a), nonce(na))), 2),$
$h(send(b, i, content(key(ka), nonce(na), nonce(nb))), 3),$
$h(send(i, a, content(key(ka), nonce(na), nonce(nb))), 4),$
$h(send(a, i, content(key(ki), nonce(nb), empty(0))), 5),$
$h(send(i, b, content(key(kb), nonce(nb), empty(0))), 6)\},$
that is, we generate Lowe's attack on the protocol. $\mathbf{HAP}_g$ represents a counterexample of the property $\mathcal{P}_{trust}$.

## 4 Related Work

The focus of our work is not on security protocols themselves, for which there exist many efficient specialised methods, but on a language for describing protocols, for verifying the compliance of interactions, and for proving general properties of the protocols. To the best of our knowledge, this is the only comprehensive approach able to address both the two types of verification, using the same protocol definition language. Security protocols and their proof of flawedness are, in our viewpoint, instances of the general concepts of agent protocols and their properties.

In recent years the provability of properties in particular has drawn considerable attention in several communities. Various techniques have been adopted to address automatic verification of security properties. In the following, we summarise and compare some of the logic-based approaches with ours.

Russo *et al.* [RMNK02] discuss the application of abductive reasoning for analysing safety properties of declarative specifications expressed in the Event Calculus. In their abductive approach, the problem of proving that, for some invariant $I$, a domain description $D$ entails $I$ ($D \models I$), is translated into an equivalent problem of showing that it is not possible to consistently extend $D$ with assertions that particular events have actually occurred (i.e., with a set of abductive hypotheses $\Delta$), in such a way that the extended description entails $\neg I$. In other words, there is no set $\Delta$ such that $D \cup \Delta \models \neg I$. They solve this latter problem by a complete abductive decision procedure, thus exploiting abduction in a refutation mode. Whenever the procedure finds a $\Delta$, the assertions in $\Delta$ act as a counterexample for the invariant. Our work is closely related: in fact, in both cases, goals represent negation of properties, and the proof procedure attempts to generate counterexamples by means of abduction. However, we rely on a different language (in particular, ours can also be used for checking compliance on the fly without changing the specification of the protocol, which is obviously a complex and time consuming task) and we deal with time by means of CLP constraints, whereas Russo *et al.* employ a temporal formalism based on Event Calculus.

In [BMV03] the authors present a new approach, On-the-Fly Model Checker, to model check security protocols, using two concepts quite related to our approach. Firstly, they introduce the concept of lazy data types for representing a (possibly) infinite transition system; secondly, they use variables in the messages that an intruder can generate. In particular, the use of unbound variables reduces the state space generated by every possible message that an intruder can utter. Protocols are represented in the form of transition rules, triggered by the arrival of a message: proving properties consists of exploring the tree generated by the transition rules, and verifying that the property holds for each reachable state. They prove results of soundness and completeness, provided that the number of messages is bounded. Our approach is very similar, at least from the operational viewpoint. The main difference is that the purpose of our language is not limited to the analysis of security protocols. Moreover, we have introduced variables in all the messages, and not only in the messages uttered by the intruder; we can pose CLP constraints on these variables, where OFMC instead can generate only equality/inequality constraints. OFMC provides state-of-the-art performance for security protocol analysis; our approach instead suffers for its generality, and its performance is definitely worse than the OFMC.

A relevant work in computer science on verification of security protocols was done by Abadi and Blanchet [Bla03,AB05]. They adopt a verification technique based on logic programming in order to verify security properties of protocols, such as secrecy and authenticity in a fully automatic way, without bounding the number of sessions. In their approach, a protocol is represented in extensions of pi calculus with cryptographic primitives. The protocol represented in this extended calculus is then automatically translated into a set of Horn clauses [AB05]. To prove secrecy, in [Bla03,AB05] attacks are modelled by relations and secrecy can be inferred by non-derivability: if $attacker(M)$ is not derivable, then secrecy

of $M$ is guaranteed. More importantly, the derivability of $attacker(M)$ can be used, instead, to reconstruct an attack. This approach was later extended in [Bla02] in order to prove authenticity. By first order logic, having variables in the representation, they overcome the limitation of bounding the number of sessions. We achieve the same generality of [Bla03,AB05], since in their approach Horn clause verification technique is not specific to any formalism for representing the protocol, but a proper translator from the protocol language to Horn clause has to be defined. In our approach, we preferred to directly define a rewriting proof procedure ($\mathcal{S}$CIFF) for the protocol representation language. Furthermore, by exploiting abduction and CLP constraints, also in the implementation of g-$\mathcal{S}$CIFF transitions themselves, in our approach we are able to generate proper traces where terms are constrained when needed along the derivation avoiding to impose further parameters to names as done in [AB05]. CLP constraints can do this more easily.

Armando *et al.* [ACL04] compile a security program into a logic program with choice lp-rules with answer set semantics. They search for attacks of length $k$, for increasing values of $k$, and they are able to derive the flaws of various flawed security protocols. They model explicitly the capabilities of the intruder, while we take the opposite viewpoint: we explicitly state what the intruder cannot do (like decrypting a message without having the key, or guessing the key or the nonces of an agent), without implicitly limiting the abilities of the intruder.

Our social specifications can be seen as intensional formulations of the possible (i.e., compliant) traces of communication interactions. In this respect, our way of modeling protocols is very similar to the one of Paulson's inductive approach [Pau98]. In particular, our representation of the events is almost the same, but we explicitly mention time in order to express temporal constraints. In the inductive approach, the protocol steps are modeled as possible extensions of a trace with new events and represented by (forward) rules, similar to our SICs. However, in our system we have expectations, which allow us to cope with both compliance on the fly and verification of properties without changing the protocol specification. Moreover, SICs can be considered more expressive than inductive rules, since they deal with constraints (and constraint satisfaction in the proof), and disjunctions in the head. As far as verification, the inductive approach requires more human interaction and expertise, since it exploits a general purpose theorem prover, and has the disadvantage that it cannot generate counterexamples directly (as most theorem prover-based approaches). Instead, we use a specialized proof procedure based on abduction that can perform the proof without any human intervention, and can generate counterexamples.

Millen and Shmatikov [MS01] define a proof-procedure based on constraint solving for cryptographic protocol analysis. Their proof-procedure is sound and complete. Corin and Etalle [CE02] improve the system by Millen and Shmatikov by adding explicit checks and considering partial runs of the protocols; the resulting system is considerably faster. g-$\mathcal{S}$CIFF is based on constraint solving as well, but with a different flavour of constraint: while the approaches by Millen and Shmatikov and by Corin and Etalle are based on abstract algebra, our con-

straint solver comprises a CLP(FD) solver, and embeds constraint propagation techniques to speed-up the solving process.

In [Son99], Song presents Athena, an approach to automatic security protocol analysis. Athena is a very efficient technique for proving protocol properties: unlike other techniques, Athena copes well with state space explosion and is applicable with an unbounded number of peers participating in a protocol, thanks to the use of theorem proving and to a compact way to represent states. Athena is correct and complete (but termination is not guaranteed). Like Athena, the representation of states and protocols in g-SCIFF is non ground, and therefore general and compact. Unlike Athena's, the implementation of g-SCIFF is not optimised, and suffers from the presence of symmetrical states. On the other hand, a clear advantage of the SOCS approach is that protocols are written and analyzed in a formalism which is the same used for run-time verification of compliance.

## 5 Conclusion and future work

In this paper, we have shown how the *SOCS-SI* abductive framework can be applied to the specification and verification of security protocols, using, as a running example, the Needham-Schroeder Public Key authentication protocol.

The declarative framework is expressive enough to specify both which sequences of messages represent a legal protocol run, and constraints about the messages that a participant is able to synthesize.

We have implemented and experimented with the operational counterpart of the *SOCS-SI* framework. Two kinds of verification are possible on a protocol specified in the *SOCS-SI* formalism: on-the-fly verification of compliance at runtime (by means of the sound and complete $\mathcal{S}$CIFF proof procedure), and static verification of protocol properties at design time (by means of the sound g-$\mathcal{S}$CIFF proof procedure). In this way, our approach tackles both the case of agents misbehaving (which, in an open society, cannot be excluded) and the case of a flawed protocol (which can make the interaction exhibit an undesirable feature even if the participants follow the protocol correctly). The refutation of a protocol property is given by means of a counterexample. Both types of verification are automatic, and require no human intervention.

We believe that one of the main contributions of this work consists of providing a unique framework to both the two types of verification. The language used for protocol definition is the same in both the cases, thus lowering the chances of errors introduced in describing the protocols with different notations. The protocol designer can benefit of our approach during the design phase, by proving properties, and during the execution phase, where the interaction can be proved to be compliant with the protocol, and thus to exhibit the protocol properties.

Future work will be aimed to prove a result of completeness for g-$\mathcal{S}$CIFF, and to extend the experimentation on proving protocol properties to a number of security and e-commerce protocols, such as NetBill [CTS95] and SPLICE/AS [YOM91].

## Acknowledgments

This work has been supported by the European Commission within the SOCS project (IST-2001-32530), funded within the Global Computing Programme and by the MIUR COFIN 2003 projects *La Gestione e la negoziazione automatica dei diritti sulle opere dell'ingegno digitali: aspetti giuridici e informatici* and *Sviluppo e verifica di sistemi multiagente basati sulla logica.*

## References

[AB05] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1):102–146, 2005.

[ACG+03] M. Alberti, A. Ciampolini, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. A social ACL semantics by deontic constraints. In V. Mařík, J. Müller, and M. Pěchouček, editors, *Multi-Agent Systems and Applications III. Proceedings of the 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003*, volume 2691 of *Lecture Notes in Artificial Intelligence*, pages 204–213, Prague, Czech Republic, June 16–18 2003. Springer-Verlag.

[ACL04] Alessandro Armando, Luca Compagna, and Yuliya Lierler. Automatic compilation of protocol insecurity problems into logic programming. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Artificial Intelligence*, pages 617–627. Springer-Verlag, 2004.

[AGL+03a] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. An Abductive Interpretation for Open Societies. In A. Cappelli and F. Turini, editors, *AI*IA 2003: Advances in Artificial Intelligence, Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence, Pisa*, volume 2829 of *Lecture Notes in Artificial Intelligence*, pages 287–299. Springer-Verlag, September 23–26 2003.

[AGL+03b] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and verification of agent interactions using social integrity constraints. *Electronic Notes in Theoretical Computer Science*, 85(2), 2003.

[AGL+04] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Abduction with hypothesis confirmation. Number 390 in Quaderno del Dipartimento di Matematica, Research Report. Università di Parma, November 2004.

[Bla02] Bruno Blanchet. From secrecy to authenticity in security protocols. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 342–359, London, UK, 2002. Springer-Verlag.

[Bla03] Bruno Blanchet. Automatic verification of cryptographic protocols: a logic programming approach. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 1–3, New York, NY, USA, 2003. ACM Press.

[BMV03] David A. Basin, Sebastian Mödersheim, and Luca Viganò. An on-the-fly model-checker for security protocol analysis. In Einar Snekkenes and Dieter Gollmann, editors, *ESORICS*, volume 2808 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2003.

[CE02] Ricardo Corin and Sandro Etalle. An improved constraint-based system for the verification of security protocols. In Manuel V. Hermenegildo and German Puebla, editors, *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 326–341, Berlin, Germany, 2002. Springer.

[CTS95] B. Cox, J.C. Tygar, and M. Sirbu. Netbill security and transaction protocol. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, New York, July 1995.

[DY83] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[FK97] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, November 1997.

[GP02] F. Guerin and J. Pitt. Proving properties of open agent systems. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II*, pages 557–558, Bologna, Italy, July 15–19 2002. ACM Press.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[KKT93] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[Low96] G. Lowe. Breaking and fixing the Needham-Shroeder public-key protocol using CSP and FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems: Second International Workshop, TACAS'96*, volume 1055 of *Lecture Notes in Artificial Intelligence*, pages 147–166. Springer-Verlag, 1996.

[MS01] Jonathan K. Millen and Vitaly Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 166–175. ACM press, 2001.

[NS78] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[RMNK02] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for analysing event-based requirements specifications. In P.J. Stuckey, editor, *Logic Programming, 18th International Conference, ICLP 2002*, volume 2401 of *Lecture Notes in Computer Science*, pages 22–37, Berlin Heidelberg, 2002. Springer-Verlag.

[SOCa] Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. IST-2001-32530. Home Page: `http://lia.deis.unibo.it/Research/SOCS/`.

[SOCb] The SOCS Protocol Repository. Available at `http://lia.deis.unibo.it/research/socs/partners/societies/protocols.html`.

[Son99] Dawn Xiaodong Song. Athena: a new efficient automatic checker for security protocol analysis. In *CSFW '99: Proceedings of the 1999 IEEE Com-*

*puter Security Foundations Workshop*, page 192, Washington, DC, USA, 1999. IEEE Computer Society.

[YOM91] Suguru Yamaguchi, Kiyohiko Okayama, and Hideo Miyahara. The design and implementation of an authentication system for the wide area distributed environment. *IEICE Transactions on Information and Systems*, E74(11):3902–3909, November 1991.

# Implementation of Dynamic Logic Programs

F. Banti[1], J. J. Alferes[1], and A. Brogi[2]

[1] CENTRIA, Universidade Nova de Lisboa, Portugal
[2] Dipartimento di Informatica, Università di Pisa, Italy

**Abstract.** Theoretical research has spent some years facing the problem of how to represent and provide semantics to updates of logic programs. This problem is relevant for addressing highly dynamic domains with logic programming techniques. Two of the most recent results are the definition of the refined stable and the well founded semantics for dynamic logic programs that extend stable model and well founded semantic to the dynamic case. We present implementations of these semantics based on program transformations and rely on softwares for the computation of the stable model and the well founded semantics.

## 1 Introduction

In recent years considerable effort was devoted to explore the problem of how to update knowledge bases represented by logic programs (LPs) with new rules. This allows, for instance, to better use LPs for representing and reasoning with knowledge that evolves in time, as required in several fields of application. The LP updates framework has been used, for instance, as the base of the MINERVA agent architecture [15] and of the action description language EAPs [5].

Different semantics have been proposed [1, 2, 6, 7, 9, 16, 18, 19, 23] that assign meaning to arbitrary finite sequences $P_1, \ldots, P_m$ of logic programs. Such sequences are called *dynamic logic programs* (DyLPs), each program in them representing a supervenient state of the world. The different states can be seen as representing different time points, in which case $P_1$ is an initial knowledge base, and the other $P_i s$ are subsequent updates of the knowledge base. The different states can also be seen as knowledge coming from different sources that are (totally) ordered according to some precedence, or as different hierarchical instances where the subsequent program represent more specific information. The role of the semantics of DyLPs is to employ the mutual relationships among different states to precisely determine the meaning of the combined program comprised of all individual programs at each state. Intuitively, one can add at the end of the sequence, newer rules or rules with precedence (arising from newly acquired, more specific or preferred knowledge) leaving to the semantics the task of ensuring that these added rules are in force, and that previous or less specific rules are still valid (by inertia) only as far as possible, i.e. that they are kept as long as they are not rejected. A rule is rejected whenever it is in conflict with a newly added one (*causal rejection of rules*). Most of the semantics defined for DyLPs [1, 2, 6, 7, 9, 16] are based on such a concept of causal rejection.

We provide an intuitive example to explain the intuition behind such DyLPs.

*Example 1.* Sara, Cristina and Bob, are deciding what they will do on Saturday. Sara decides she is going to a museum, Cristina wants to go shopping and Bob decides to go fishing in case Sara goes to the museum. Later on they update their plans: Cristina decides not to go shopping, Sara decides she will not go to the museum if it snows and Bob decides he will not go fishing if it is a rainy day. Moreover we know from the forecast that Saturday can be either a sunny day or a raining day. We represent the situation with the DyLP $P_1, P2$, where (using the abbreviation $msm$ for museum):

$$
\begin{array}{lll}
P_1 : msm(s). & P_2 : not\ fish(b) \leftarrow rain. & sunny \leftarrow not\ rain. \\
\quad shop(c). & \quad not\ shop(c). & rain \leftarrow not\ sunny. \\
\quad fish(b) \leftarrow msm(s). & \quad not\ msm(s) \leftarrow snow. &
\end{array}
$$

The intended meaning of $P_1, P_2$ is that it does not snow on Saturday, but we do not know if it does rain or not, we know Sara goes to the museum on Saturday, we do not know whether Bob goes fishing and, finally, Cristina does not go shopping. It can be checked that, according to all existing stable models based semantics for updates of [1, 2, 6, 7, 9, 16] $P_1, P_2$ has two stable models, namely $M_1 = \{msm(s), fish(b), not\ shop(c), sunny, not\ rain, nafsnow\}$ and $M_2 = \{msm(s), rain, not\ sunny, not\ shop(c), not\ fish(b), not\ snow\}$, thus matching the intuition.

With the exception of the semantics proposed in [6], these semantics are extensions of the stable model semantics [11] to DyLPs and are proved to coincide on large classes of programs [9, 14, 12]. In [1, **?**] the authors provides theoretical results which strongly suggest that the refined semantics [1] should be regarded as the proper stable model-like semantics for DyLPs based on causal rejection.

As discussed in [6], though a stable model-like semantics is the most suitable option for several application domains [1] other domains exist, whose specificities require a different approach. In particular, domains with huge amount of distributed and heterogenous data require an approach to automated reasoning capable to quickly process knowledge and to deal with inconsistent information even at the cost of losing some inference power. Such areas demand a different choice of basic semantics, such as the well founded semantics [10]. For such reasons in [6] we defined a well founded paraconsistent semantics for DyLPs (WFDy) The WFDy semantics is shown to be a skeptical approximation of the refined semantic defined in [1], moreover it is always defined, even when the considered program is inconsistent and its computational complexity is polynomial wrt. the number of rules of the program. Referring to example 1, the well founded model of $P_1, P_2$ is, $W = \{msm(s), not\ shop(c), not\ snow\}$ which corresponds to the intersection of $M_1$ and $M_2$.

For these reasons we believe that the refined and the well founded semantics for DyLPs are useful paradigms in the knowledge representation field and hence implementations for computing both semantics are in order.

[1] In particular, the stable model semantics has been shown to be a useful paradigm for modelling NP-complete problems.

Proper implementations for such semantics should satisfies some requirements. First of all, the algorithms used must be sound and complete wrt. the semantics they intend to compute. Then, the implementations should be reasonably efficient and they should handle also programs with variables in order to be flexible enough for practical purposes. Finally, the implementations should preferably match the syntax and the features shown by available implementations of the stable model and well founded semantics, in particular the DLV[8] and smodels systems [20] and the XSB-Prolog language [22].

One more important point regards the possibility to query the knowledge base at different time points. Both the considered semantics can be defined wrt. a given program $P_i$ in the considered DyLP. When this is done, the programs in the sequence that follows $P_i$ are not considered in the computation of the semantics. Considering the various updates as different time points, it is possible to ask to the program what is true or false at a given time. A proper implementation for semantics of DyLPs should allow this kind of queries.

Hereafter we describe the implementations of the two semantics. Such implementations are based on theoretical results on the two semantics. We show, for both semantics, different program transformations from DyLPs to normal logic programs and provide theorems that show semantical equivalence between the transformed programs and the original DyLPs. These transformations are themselves relevant. They provide further results on the computational complexity of the considered semantics. Moreover they give new insights on how the rejection mechanism works and how it creates new dependencies among rules.

Regarding the refined semantics, the considered DyLP is transformed into a normal LP and then the stable models of the transformed program is computed by using either DLV or smodels. These systems have obtained in the last years a high degree of optimizations that allows to use the stable model semantics in real problems with hundreds of thousands of rules. Regarding the WFDy semantics, the knowledge base can be consulted by querying the transformed program via XSB-Prolog.

Both the implementations support the possibility of querying the considered DyLP at any given state. The correctness of this approach is guaranteed by the equivalence between the original programs and the transformed ones. By using existing softwares we profit of all the optimizations incorporated in such products. The main point about complexity issues is then the size of the transformed programs. In sections 3 and 4 we provide upper bounds to the size of the transformed programs for the implementations. The transformations can be applied on programs with variables thus obtaining transformed programs with variables. Then the grounding algorithms of DLV and smodels and the computed answer substitution techniques of XSB-Prolog are used for dealing with programs with variables. The implementations provide the same syntax and the features of the DLV, smodels, and XSB-Prolog systems. Finally, the implementations are written in Prolog and are publicly available at http://centria.di.fct.unl.pt/∼banti/implementation.htm

There are in literature other implementations of DyLPs based on program transformations (see for instance [2, 4, 13]). However, such program transformations are not sound and complete wrt. neither the refined nor the well founded semantics for DyLPs. Moreover, the existing transformations, unlike the presented ones, have the drawback of being dependent on the number of updates in the considered sequence, and thus posing serious problems of efficiency whenever a program is updated several times. In order to overcome these limitations, the program transformations and the related implementations presented herein had to be developed in a substantially different way from the existing ones.

The rest of the paper is structured as follows. Section 2 establishes notation and provides some background and the formal definition of the refined and well founded semantics for DyLPs. Section 3 illustrates the transformation used in the implementation of the refined semantics, describes notable properties of such a transformation. Section 4 gives a similar view of the transformation used for the WFDy semantics. Then section 5 briefly illustrates the two implementations. Section 6 examines other implementations of DyLPs, draws conclusions and mentions some future developments.

## 2 Background: Concepts and notation

In this section we briefly recall the syntax of DyLPs, and the refined and well founded semantics defined, respectively, in [1] and [6].

To represent negative information in logic programs and their updates, DyLP uses generalized logic programs (GLPs) [17], which allow for default negation *not* $A$ not only in the premises of rules but also in their heads. A language $\mathcal{L}$ is any set literals of the form $A$ or *not* $A$ such that $A \in \mathcal{L}$ iff *not* $A \in \mathcal{L}$. A GLP defined over a propositional language $\mathcal{L}$ is a (possibly infinite) set of ground rules of the form $L_0 \leftarrow L_1, \ldots, L_n$, where each $L_i$ is a literal in $\mathcal{L}$, i.e., either a propositional atom $A$ in $\mathcal{L}$ or the default negation *not* $A$ of a propositional atom $A$ in $\mathcal{L}$. We say that $A$ is the *default complement* of *not* $A$ and viceversa. With a slight abuse of notation, we denote by *not* L the default complement of $L$ (hence if $L$ is the atom $A$, *not* $L$ is $A$ and viceversa). Given a rule $\tau$ as above, by $hd(\tau)$ we mean $L_0$ and by $B(\tau)$ we mean $\{L_1, \ldots, L_n\}$.

In the sequel an *interpretation* is simply a set of literals of $\mathcal{L}$. A literal $L$ is *true* (resp. *false*) in $I$ iff $L \in I$ (resp. *not* $L \in I$) and *undefined* in $I$ iff $\{L, not\ L\} \cap I = \{\}$. A conjunction (or set) of literals $C$ is true (resp. false) in $I$ iff $C \subseteq I$ (resp. $\exists\ L \in C$ such that $L$ is false in $I$). We say that $I$ is *consistent* iff $\forall\ A \in \mathcal{L}$ *at most* one of $A$ and *not* $A$ belongs to $I$, otherwise we say $I$ is *paraconsistent*. We say that $I$ is *2-valued* iff for each atom $A \in \mathcal{L}$ *exactly* one of $A$ and *not* $A$ belongs to $I$.

A *dynamic logic program* with length $n$ over a language $\mathcal{L}$ is a finite sequence $P_1, \ldots, P_n$ (also denoted $\mathcal{P}$ , where the $P_i$s are GLPs indexed by $1, \ldots, n$), where all the $P_i$s are defined over $\mathcal{L}$. Intuitively such a sequence may be viewed as the result of, starting with program $P_1$, updating it with program $P_2$, ..., and updating it with program $P_n$. For this reason we call the singles $P_i s$ *updates*.

Let $P_j$ and $P_i$ be two updates of $\mathcal{P}$ . We say that $P_j$ is *more recent* than $P_i$ iff $P_j$ follows $P_i$ in the sequence (or, in practice, if $i < j$). We use $\rho(\mathcal{P})$ to denote the multiset of all rules appearing in the programs $P_1, ..., P_s$.

The *refined stable model semantics* for DyLPs is defined in [1] by assigning to each DyLP a set of stable models. The basic idea of the semantics is that, if a later rule $\tau$ has a true body, then former rules in conflict with $\tau$ should be *rejected* . Moreover, any atom $A$ for which there is no rule with true body in any update, is considered false by default. The semantics is then defined by a fixpoint equation that, given an interpretation $I$, tests whether $I$ has exactly the consequences obtained after removing from the multiset $\rho(\mathcal{P})$ all the rules rejected given $I$, and imposing all the default assumptions given $I$. Formally, let:

$$Default(\mathcal{P}, I) = \{not\ A \mid \nexists\ A \leftarrow body \in \rho(\mathcal{P}) \wedge body \subseteq I\}$$
$$Rej^S(\mathcal{P}, I) = \{\tau \mid \tau \in P_i | \ \exists\ \eta \in P_j\ i \leq j,\ \tau \bowtie \eta \wedge\ \ B(\eta) \subseteq I\}$$

where $\tau \bowtie \eta$ means that $\tau$ and $\eta$ are conflicting rules, i.e. the head of $\tau$ is the default complement of the head of $\eta$.

**Definition 1.** *Let $\mathcal{P}$ be any DyLP of length $n$, $i \leq n$ over language $\mathcal{L}$ and $M$ a two valued interpretation and let $\mathcal{P}^i$ be the prefix of $\mathcal{P}$ with length $i$. Then $M$ is a refined stable model of $\mathcal{P}$ , at state $i$, iff $M$ is a fixpoint of $\Gamma^S_{\mathcal{P}_i}$:*

$$\Gamma^S_{\mathcal{P}_i}(M) = least\left(\rho\left(\mathcal{P}^i\right) \setminus Rej^S(\mathcal{P}^i, M) \cup Default(\mathcal{P}^i, M)\right)$$

*where $least(P)$ denotes the least Herbrand model of the definite program obtained by considering each negative literal not $A$ in $P$ as a new atom*[2]*.*

The definition of dynamic stable models of DyLPs [2] is as the one above, but where the $i \leq j$ in the rejection operator is replaced by $i < j$. I.e., if we denote this other rejection operator by $Rej(\mathcal{P}, I)$, and define $\Gamma_{\mathcal{P}}(I)$ by replacing in $\Gamma^S_{\mathcal{P}}$ $Rej^S$ by $Rej$, then the stable models of $\mathcal{P}$ are the interpretations $I$ such that $I = \Gamma_{\mathcal{P}}(I)$.

The *well founded semantics for DyLPs* is defined through the two operators $\Gamma$ and $\Gamma^S$. We use the notation $\Gamma\Gamma^S$ to denote the operator obtained by first applying $\Gamma^S$ and then $\Gamma$. The well founded model of a program is defined as the least fix point of such operator. Formally:

**Definition 2.** *The well founded model $WFDy(\mathcal{P})$ of a DyLP $\mathcal{P}$ at state $i$ is the (set inclusion) least fixpoint of $\Gamma_{\mathcal{P}^i}\Gamma^S_{\mathcal{P}_i}$ where $\mathcal{P}^i$ is the prefix of $\mathcal{P}$ with length $i$.*

Since the operators $\Gamma$ and $\Gamma^S$ are anti-monotonous (see [6]) the composite operator $\Gamma\Gamma^S$ is monotonous and, as it follows from the classical results of Tarski (see [21]), it always has a least fixpoint. In other words, $WFDy$ is uniquely defined for every DyLP. Moreover, $WFDy(\mathcal{P})$ can be obtained by (transfinitely) iterating $\Gamma\Gamma^S$, starting from the empty interpretation. As already mentioned in

[2] Whenever clear from the context, hereafter we omit the $\mathcal{P}$ in any of the above defined operators.

the introduction, the refined and well founded semantics for DyLPs are strongly related. In particular, they share analogous connections to the ones shared by the stable model and the well founded semantics of normal logic programs, as we see from the following proposition.

**Proposition 1.** *Let $M$ be any refined stable model of $\mathcal{P}$ . The well founded model $WFDy(\mathcal{P})$ is a subset of $M$. Moreover, if $WFDy(\mathcal{P})$ is a 2-valued interpretation, it coincides with the unique refined stable model of $\mathcal{P}$ .*

This property does not hold if, instead of the refined semantics, we consider any of the other semantics based on causal rejection [2, 7, 9, 16].

*Example 2.* Let $\mathcal{P}: P_1, P_2$ be the as follows: $P_1 : a \leftarrow b.\ b.\ c. \quad P_2 : not\ a \leftarrow c.$ The well founded model of $\mathcal{P}$ : is $M = \{b, c, not\ a\}$. Moreover, $M$ is a two valued interpretation and hence, by proposition 1 we conclude that $M$ is also the unique refined model.

## 3 A program transformation for the refined semantics.

As stated in the introduction, the core of the implementation of the refined semantics for DyLPs is a program transformation. This transformation turns a DyLP $\mathcal{P}$ in the language $\mathcal{L}$ into a normal logic program $\mathcal{P}^R$ in an extended language called the *refined transformational equivalent of $\mathcal{P}$* . We provide herein a formal procedure to obtain the transformational equivalent of a given DyLP.

Let $\mathcal{L}$ be a language. By $\mathcal{L}^R$ we denote the language whose elements are either atoms of $\mathcal{L}$, or atoms of one of the following forms: $u$, $A^-$, $rej(A, i)$, $rej(A^-, i)$, where $i$ is a natural number, $A$ is any atom of $\mathcal{L}$ and no one of the atoms above belongs to $\mathcal{L}$. Intuitively, $A^-$ stands for "$A$ is false",while $rej(A, i)$ (resp. $rej(A^-, i)$ ), stands for: "all the rules with head $A$ (resp. *not* $A$) in the update $Pi$ are rejected". For every literal $L$, if $L$ is an atom $A$, then $\overline{L}$ denotes $A$ itself, while if $L$ is a negative literal *not* $A$ then $\overline{L}$ denotes $A^-$. Finally, $u$ is a new atom not belonging to $\mathcal{L}$ which is used for expressing integrity constraints of the form $u \leftarrow not\ u, L_1, \ldots, L_k$.

**Definition 3.** *Let $\mathcal{P}$ be a Dynamic Logic Program whose language is $\mathcal{L}$. By the refined transformational equivalent of $\mathcal{P}$ , denoted $\mathcal{P}^R$ , we mean the normal program $P_1^R \cup \ldots \cup P_n^R$ in the extended language $\mathcal{L}^R$, where each $P_i^R$ exactly consists of the following rules.*

**Default assumptions** *For each atom $A$ of $\mathcal{L}$ appearing in $P_i$, and not appearing in any other $P_j$, $j \leq i$ a rule:*

$$A^- \leftarrow not\ rej(A^-, 0)$$

**Rewritten rules** *For each rule $L \leftarrow body$ in $P_i$, a rule:*

$$\overline{L} \leftarrow \overline{body},\ not\ rej(\overline{L}, i)$$

**Rejection rules** *For each rule $L \leftarrow body$ in $P_i$, a rule:*

$$rej(\overline{not\ L}, j) \leftarrow \overline{body}$$

*where $j \leq i$ is either the largest index such that $P_j$ has a rule with head not $L$ or. If no such $P_j$ exists, and $L$ is a positive literals, then $j = 0$, otherwise this rule is not part of $P_i^R$. Moreover, for each rule $L \leftarrow body$ in $P_i$, a rule:*

$$rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$$

*where $j < i$ is the largest index such that $P_j$ also contains a rule $L \leftarrow body$. If no such $P_j$ exists, and $L$ is a negative literal, then $j = 0$, otherwise this rule is not part of $P_i^R$.*

**Totality constraints** *For each pair of conflicting rules in $P_i$, with head $A$ and not $A$, the constraint:*

$$u \leftarrow not\ u,\ not\ A,\ not\ A^-$$

Let us briefly explain the intuition for each of these rules and their role. The *default assumptions* specify that a literal of the form $A^-$ is true (i.e. $A$ is false) unless this initial assumption is rejected by some later rule. The *rewritten rules* are basically the original rules of the sequence of programs with an extra condition in their body. This condition specifies that in order to derive conclusions, the considered rule must not be rejected. Note that, both in the head and in the body of a rule, the negative literals of the form *not* $A$ are replaced by the corresponding atoms of the form $A^-$. The role of *rejection rules* is to specify whether all the rules with a given head in a given state are rejected or not. Such a rule may have two possible forms. Let $L \leftarrow body$ be a rule in $P_i$. The rule of the form $rej(\overline{L}, j) \leftarrow \overline{body}$ specifies that all the rules with head *not* $L$ in the most recent update $P_j$ with $j \leq i$ must be rejected. Then the rules of the form $rej(\overline{L}, j) \leftarrow rej(\overline{L}, k)$ "propagate" the rejection to the updates below $P_j$. Finally, *totality constraints* assure that, for each literal $A$, at least one of the atoms $A$, $A^-$ belongs to the model. This is done to ensure that the models of transformed program corresponds to two valued interpretations of the original $DyLP$.

The role of the atoms of the extended language $\mathcal{L}^R$ that do not belong to the original language $\mathcal{L}$ is merely auxiliary, as we see from the following theorem. Let $\mathcal{P}$ be any Dynamic Logic Program and $P_i$ and update of $\mathcal{P}$ . We use $\rho(\mathcal{P})^{Ri}$ to denote the set of all rules appearing in the programs $P_0^R, ..., P_i^R$.

**Theorem 1.** [3] *Let $\mathcal{P}$ be any Dynamic Logic Program in the language $\mathcal{L}$, $P_i$ and update of $\mathcal{P}$ , and let $\rho(\mathcal{P})^{Ri}$ be as above. Let $M$ be any interpretation over $\mathcal{L}$. Then $M$ is a refined stable model of $\mathcal{P}$ at $P_i$ iff there exits a two valued interpretation $M^R$ such that $M^R$ is a stable model of $\rho(\mathcal{P})^{Ri}$ and $M \equiv_{\mathcal{L}} M^R$. Moreover, $M$ and $M^R$ satisfies the following conditions.*

$$\begin{array}{ll} A \in M \Leftrightarrow A \in M^R & not\ A \in M \Leftrightarrow A^- \in M^R \\ not\ A \in Default(\mathcal{P}^i, M)) \Leftrightarrow & rej(A^-, 0) \notin M^R \\ \tau \in rej^S(M, \mathcal{P}^i) \wedge \tau \in P_i \Leftrightarrow & rej(\overline{hd(\tau)}, i) \in M^R \end{array}$$

[3] A complete proof of the theorems is available at http://centria.di.fct.unl.pt/∼banti/FedericoBantiHomepage/biblio.htm

We present an example of the computation of the refined transformational equivalent of a DyLP.

We present an example of the computation of the refined transformational equivalent of a DyLP.

*Example 3.* Let $\mathcal{P}: P_1, P_2$ be the as in example 2. The transformational equivalent of $\mathcal{P}$ is the following sequence $P_1^R$, $P_1^R$, $P_2^R$:

$$\begin{array}{lll} P_1^R: & a^- \leftarrow not\ rej(a^-,0). & b^- \leftarrow not\ rej(b^-,0). \\ & a \leftarrow b,\ not\ rej(a,1). & rej(a^-,0) \leftarrow b. \\ P_2^R & c^- \leftarrow not\ rej(c^-,0). & \\ & rej(b^-,0). & rej(c^-,0). \\ & b \leftarrow not\ rej(b,2). & c \leftarrow not\ rej(c,2). \\ P_3^R: & a^- \leftarrow c,\ not\ rej(a,3). & rej(a,1) \leftarrow c. \end{array}$$

For computing the refined semantics of $\mathcal{P}$ at $P_2$ we just have to compute the stable model semantics of the program $P_1^R \cup P_2^R$. This program has a single stable model $M^R$ consisting of the following set[4].

$$M^R = \{a,\ b,\ c,\ rej(a^-,0),\ rej(b^-,0),\ rej(c^-,0)\}$$

We conclude that, $\mathcal{P}$ has $M = \{a,\ b,\ c\}$ as the unique refined model. To compute the refined semantics of $\mathcal{P}$ we have to compute, instead, the stable model semantics of the program $P^R = P_1^R \cup P_2^R \cup P_3^R$. Let us briefly examine the transformed program $P^R$ and see how it clarifies the meaning of the related DyLP. Since there exist no rules with head $rej(b,2)$ and $rej(c,2)$, we immediately infer $b$ and $c$. Then we also infer $rej(a,0)$, $rej(b,0)$ and $rej(c,0)$, and so that all the default assumptions are rejected. The last rule of $P_3^R$ implies $rej(a,1)$, thus the rule $a \leftarrow b$ in $P_1$ is rejected and we do not infer $a$. In fact, we infer $a^-$ by the first rule of $P_3^R$. Hence, the program has still a single stable model:

$$\{b,\ c,\ a^-, rej(a^-,1), rej(a^-,0),\ rej(b^-,0),\ rej(c^-,0)\}$$

which means $\mathcal{P}$ as the unique refined model $\{b,\ c\}$.

To provide a more meaningful example, we show the refined transformational equivalent of the DyLP $P1, P_2$ appearing in example 1

$$\begin{array}{lll} P_1^R: & msm(s)^- \leftarrow rej(msm(s)^-,0). & shop(c)^- \leftarrow not\ rej(shop(c)^-,0). \\ & fish(b)^- \leftarrow rej(fish(b)^-,0). & \\ & msm(s) \leftarrow, not\ rej(msm(s),1). & rej(msm(s)^-,0). \\ & shop(c) \leftarrow, not\ rej(shop(c),1). & rej(shop(c)^-,0). \\ & fish(b) \leftarrow msm(s), not\ rej(fish(b),1). & rej(fish(b)^-,0) \leftarrow msm(s). \\ P_2^R: & rain^- \leftarrow not\ rej(rain^-,0). & sunny^- \leftarrow not\ rej(sunny^-,0). \\ & snow^- \leftarrow not\ rej(snow^-,0). & \\ & fish(b)^- \leftarrow rain, not\ rej(fish(b)^-,2). & rej(fish(b),1) \leftarrow rain. \\ & sunny \leftarrow rain^-, not\ rej(sunny,2). & rej(sunny^-,0) \leftarrow rain^-. \\ & rain \leftarrow sunny^-, not\ rej(sunny,2). & rej(rain^-,0) \leftarrow sunny^-. \\ & shop(c)^- \leftarrow not\ rej(shop(c)^-,2). & rej(shop(c),1). \\ & msm(s)^- \leftarrow snow, not\ rej(msm(s)^-,2). & rej(msm(s)^-) \leftarrow snow. \end{array}$$

[4] As usual in the stable model semantics, hereafter we omit the negative literals

The program $P_1^R \cup P_2^R$ has two stable models, namely

$$M_1^R = \{snow^-, rain^-, sunny, fish(b), shop(c)^-, msm(s), rej(shop(c), 1), \\ rej(sunny^-, 0), rej(fish(b)^-, 0), rej(shop(c)^-, 0), rej(msm(s)^-, 0)\}$$
$$M_2^R = \{snow^-, rain, sunny^-, fish(b)^-, shop(c)^-, msm(s), rej(shop(c), 1), \\ rej(fish(b), 1),\ rej(rain^-, 0), rej(fish(b)^-, 0), rej(shop(c)^-, 0), rej(msm(s)^-, 0)\}$$

Which, restricting to the original language of the program, are equal to $M_1$ and $M_2$ of example 1.

To compute the refined semantics of a given DyLP $P_1, \ldots, P_n$ at a given state, it is hence sufficient to compute its refined transformational equivalent $P_0^R, \ldots P_n^R$, then to compute the stable model semantics of the normal logic program $\rho(\mathcal{P})^{Ri}$ and, finally, to consider only those literals that belong to the original language of the program. The efficiency of the implementation relies on large part on the size of the transformed program compared to the size of the original one. We present here a theoretical result that provides an upper bound for the number of clauses of the refined transformational equivalent of a DyLP.

**Theorem 2.** [3] *Let $\mathcal{P} : P_1, \ldots P_m$ be any finite ground DyLP in the language $\mathcal{L}$ and let $\rho(\mathcal{P})^{Rn}$ be the set of all the rules appearing in the refined transformational equivalent of $\mathcal{P}$. Moreover, let $m$ be the number of clauses in $\rho(\mathcal{P})$ and $l$ be the cardinality of $\mathcal{L}$*[5]*. Then, the program $\rho(\mathcal{P})^{Rn}$ consists of at most $2m + l$ rules.*

The problem of satisfiability under the stable model semantics (i.e. to find a stable model of a given program) is known to be NP-Complete, while the inference problem (i.e. to determine if a given proposition is true in all the stable models of a program) is co-NP-Complete [11]. Hence, from theorems 1 and 2, it immediately follows that such problems are still NP-Complete and co-NP-Complete also under the refined semantics for DyLPs. The size of the refined transformational equivalent of a DyLP depends linearly and solely on the size of the program and of the language, hence, it has an upper bound which does not depend from the number of updates performed. Hence, we gain the possibility to perform several updates of our knowledge base and query the knowledge base at different time points without losing too much on efficiency.

## 4 Transformational well founded semantics

As for the refined semantics, the core of the implementation of the well founded semantics for DyLPs is a program transformation that turns a given DyLP $\mathcal{P}$ in the language $\mathcal{L}$ into a normal logic program $\mathcal{P}^{TW}$ in an extended language $\mathcal{L}^W$ called the *well founded transformational equivalent of $\mathcal{P}$* .

[5] Since $\mathcal{L}$ contains the positive and the negative literals , $l$ is equal to two times the number of predicates appearing in $\mathcal{P}$ .

Let $\mathcal{L}$ be a language and. By $\mathcal{L}^W$ we denote the language whose atoms are either atoms of $\mathcal{L}$, or are atoms of one of the following forms: $A^S$ $A^{-S}$, $rej(A,i)$, $rej(A^S,i)$, $rej(A^-,i)$, and $rej(A^{-S},i)$, where $i$ is a natural number, $A$ is any atom of $\mathcal{L}$ and no one of the atoms above belongs to $\mathcal{L}$.

**Definition 4.** *Let $\mathcal{P}$ be a Dynamic Logic Program on the language $\mathcal{L}$. By the well founded transformational equivalent of $\mathcal{P}$ , denoted $\mathcal{P}^{TW}$ , we mean the normal program $P_1^W \cup \ldots \cup P_n^W$ in the extended language $\mathcal{L}^W$, where each $P_i$ exactly consists of the following rules:*

**Default assumptions** *For each atom $A$ of $\mathcal{L}$ appearing in $P_i$, and not appearing in any other $P_j$, $j \leq i$ the rules:*

$$A^- \leftarrow not\ rej(A^{-S},0) \qquad A^{-S} \leftarrow not\ rej(A^-,0)$$

**Rewritten rules** *For each rule $L \leftarrow body$ in $P_i$, the rules:*

$$\overline{L} \leftarrow \overline{body},\ not\ rej(\overline{L}^S,i) \qquad \overline{L}^S \leftarrow \overline{body}^S,\ not\ rej(\overline{L},i)$$

**Rejection rules** *For each rule $L \leftarrow body$ in $P_i$, a rule:*

$$rej(\overline{not\ L},j) \leftarrow \overline{body}$$

*where $j < i$ is the largest index such that $P_j$ has a rule with head not $L$. If no such $P_j$ exists, and $L$ is a positive literals, then $j = 0$, otherwise this rule is not part of $P_i^W$.*
*Moreover, for each rule $L \leftarrow body$ in $P_i$, a rule:*

$$rej(\overline{not\ L}^S,k) \leftarrow \overline{body}^S.$$

*where $k \leq i$ is the largest index such that $P_k$ has a rule with head not $L$. If no such $P_j$ exists, and $L$ is a positive literals, then $j = 0$, otherwise this rule is not part of $P_i^W$.*
*Finally, for each rule $L \leftarrow body$ in $P_i$, the rules:*

$$rej(\overline{L}^S,j) \leftarrow rej(\overline{L}^S,i) \qquad rej(\overline{L},j) \leftarrow rej(\overline{L},i)$$

*where $j < i$ is the largest index such that $P_j$ also contains a rule $L \leftarrow body$. If no such $P_j$ exists, and $L$ is a negative literal, then $j = 0$, otherwise these rules are not part of $P_i^W$.*

As the reader can see, the program transformation above resemble the one of definition 3. The main difference is that the transformation of definition 4 duplicates the language and the rules. This is done for simulating the alternate application of two different operators, $\Gamma$ and $\Gamma^S$ used in the definition of $WFDy$. The difference between these two operators relies on the rejection strategies: the $\Gamma^S$ operator, in fact, allows rejection of rules in the same state, while the $\Gamma$ operator does not. In the transformation above this difference is captured by the definition of the rejection rules. Let $L \leftarrow body$ be a rule in the update $P_i$. In the rules of the form $rej(\overline{L},j) \leftarrow \overline{body}S$ and $rej(\overline{L}^S,j) \leftarrow \overline{body}^S$, $j$ is *less* than $i$, in the first case, and *less or equal* than $i$ in the second one. A second difference

is the absence of the *totality constraints*. This is not surprising since the well founded model is not, in the general case, a two valued interpretation. However, the introduction of any rule of the form $u \leftarrow not\ u,\ body$ would not change the semantics. As for the program transformation of definition 3 the atoms of the extended language $\mathcal{L}^W$ that do not belong to the original language $\mathcal{L}$ are merely auxiliary.

Let $\mathcal{P}$ be any Dynamic Logic Program and $P_i$ and update of $\mathcal{P}$ . We use $\rho(\mathcal{P})^{Ri}$ to denote the set of all rules appearing in the programs $P_0^W, ..., P_i^W$.

**Theorem 3.** [3] *Let $\mathcal{P}$ be any Dynamic Logic Program in the language $\mathcal{L}$, $P_i$ and update of $\mathcal{P}$ , and let $\rho(\mathcal{P})^{Ri}$ be as above. Moreover, Let $W_i$ be the well founded model of the normal logic program $\rho(\mathcal{P})^{Ri}$ and $WFDy(\mathcal{P}^i)$ be the well found model of $\mathcal{P}$ at $P_i$. Then $WFDy(\mathcal{P}^i) = \{A \mid \ \in W_i\} \cup \{not\ A \mid A^- \in W_i\}$*

To compute the well founded semantics of a given DyLP $P_1, \ldots, P_n$ at a given state, it is hence sufficient to compute its well founded transformational equivalent $P_0^W, \ldots P_n^W$, then to compute the well founded model of the normal logic program $\rho(\mathcal{P})^{Rn}$ and, finally, to consider only those literals that belong to the original language of the program.

We present here a result analogous that of theorem 2 that provides an upper bound to the size of the well founded transformational equivalent.

**Theorem 4.** [3] *Let $\mathcal{P} : P_1, \ldots P_m$ be any finite ground DyLP in the language $\mathcal{L}$ and let $\rho(\mathcal{P})^{Rn}$ be the set of all the rules appearing in the transformational equivalent of $\mathcal{P}$. Moreover, let $m$ be the number of clauses in $\rho(\mathcal{P})$ and $l$ be the cardinality of $\mathcal{L}$. Then, the program $\rho(\mathcal{P})^{Rn}$ consists of at most $5m + l$ rules.*

The problem of computing the well-founded model of a normal logic program has a polynomial complexity [10]. Hence, from theorems 3 and 4, it immediately follows that such a problem is polynomial also under the well founded semantics for DyLPs.

## 5 System illustration

In this section we illustrate the implementations for the refined and well-founded semantics for DyLPs[6], their specific features and way of usage. The code of the implementation is written in XSB-Prolog [22] (though the implementation of the refined semantics runs under most of the existing versions of Prolog). The implementation for the refined semantics also needs either the smodels [20] or the DLV [8] systems. In both implementations commands are passed through the shell of Prolog.

The input system of the two implementations is the same. The various updates can be specified directly to the system or they can be edited in a file and then passed to the system. In the first case, the rules of the updates are specified in a list and passed as the argument of the command **updateL/1**. In the second

[6] Freely downloadable at http://centria.di.fct.unl.pt/∼banti/implementation.htm

case, the edited file is passed as the argument of the command **updateF/1**. In both cases, the result is to update the current knowledge base with the specified updates. In the same list or file it is possible to delimit the end of an update and the beginning of the next one by inserting the fact *update*. In both cases the various rules are processed one by one through the predicate **processClause/1** that edit the rules of the transformed equivalent of the program into a temporary file. When the commands **updateF** and **updateL** are iterated several time, the new rules are added to the file. It is possible to save the result of the running session into a file through the command **saveF/1** and then to load it during a new session with **loadF/1**.

In both implementations, updates are indexed with ascending positive integers. In the refined implementation we can specify at which update the refined models must be computed. This is done by invoking the command **process/2** where the first argument is an output file and the second one is the number of the update at which the semantics is computed. If the second arguments is *now* the semantics is computed at the last update. The obtained file can be processed with either DLV or smodels.

To consult the program under the well founded implementations is different and somehow more direct that to consult it under the refined one. In this implementation, the file containing the transformational equivalent is immediately compiled by the underlying XSB system and it can be queried as any Prolog program. The user has to specify at which update any predicate in the query is processed. This is achieved by adding an extra argument to any predicate whose value is either the index of the an update or *now*, which means that the predicate is queried at the last update. The implementation uses tabling techniques (see [22]) for computing the well founded semantics of the considered program. An answer *yes* means that the predicate is either true or undefined. Moreover, meta level predicates, called *epistemic predicates* are defined that allow the user to ask about the real truth value of each predicate. A predicate $A$ can be **true** ($A$ belongs to the semantics), **false** (*not* $A$ belongs to the semantics) **consistent** ($A$ belongs to the semantics and *not* $A$ does not) **undefined** ($A$ and *not* $A$ does not belong to the semantics) and **inconsistent** (both $A$ and *not* $A$ belongs to the semantics). For more details on the implementations and their usage see the source code and the relative documentation at the above mentioned URL.

## 6 Comparisons and concluding remarks

The transformation defined in section 3 presents some similarities with the one presented in [2]. In the two transformations new atoms for representing negative conclusions are used and such new atoms substitute the original negative literals in the head and in the body of the rules. Moreover both transformations use new atoms to represent rejection of rules.

A first and fundamental difference between the two transformations is that they are not semantically equivalent. The transformation defined in [2] is defined for implementing the *dynamic stable model semantics of DyLPs* defined in [2].

Such semantics is not equivalent to the refined one. The refined semantics was introduced for solving some counterintuitive behaviours of the previously existing semantics for DyLPs as detailed in [1]. In particular, it is proved in [1] that every refined stable model is also a dynamic stable model but the opposite is not always true. Hence the implementation based on such program transformation[7] is not equivalent to the one discussed here which implements the refined semantics. Moreover, the size of the transformation defined in [2] depends linearly on the number of updates in the considered DyLP, while the semantics presented herein is independent from such parameter as discussed in section 3.

Regarding the well founded semantics, there exists in the literature other examples of program transformations for the computation of a well founded-like semantics for DyLPs [2, 4, 13]. However, as shown in [6], such programs transformations, (and hence the related implementations) do not compute the well founded semantics of DyLPs.

Dynamic Logic Programs are a framework for representing knowledge that evolves with time. The purpose of this paper was to illustrate the principles and the properties of the implementations of the refined and well founded semantics for DyLPs. The core of the implementations are program transformations that transform a DyLP into a normal logic program. This allows the usage of software like DLV, smodels (for the refined semantics implementation) and XSB-Prolog (for the well founded one) for computing the models (in refined semantics implementation) and querying (in the well founded implementation) the considered DyLP. Moreover, we have shown that the size of the transformed programs used in the implementation is linearly bound by the size of the original program, hence the computational cost of the two semantics is near to the computational cost of, respectively, the stable model and the well founded semantics for normal logic programs. Moreover, both implementations allow the user to ask to the knowledge base what was true or false at any given time.

The close relationships between the two semantics rise the question whether an approach to the implementation based on a *single* program transformation would have been possible instead. Indeed, the answer is positive. The program transformation of definition 4 already computes the WFDy semantics of a DyLP. By adding proper integrity constraints of the form $u \leftarrow not\ u,\ body$ to the transformed program it is possible to obtain a program transformation that computes the refined semantics. Moreover, as noted in section 4, the addition of such constraints would not change the well founded model of the program, and hence the new program transformation would still compute the $WFDy$ semantics of the considered program.

We opted instead for presenting two separate transformations mainly for a matter of optimization. In fact, the size of the transformed program after the transformation of definition 3 is less than half the size that the unique transformation would require. Moreover, the transformed program can be used also for "reading" a DyLP as a normal logic program, a task for which a program

[7] Available at http://centria.di.fct.unl.pt/∼jja/updates/

transformation with many more auxiliary rules and predicates would not be suitable.

As mentioned in the introduction there are several works on possible usage of DyLPs. Other possible usages can be found in any application areas where evolution and reactivity are a primary issue. We believe it is the time for the research on DyLPs to realize practical applications of the framework, to provide implementations for such applications and face real world problems. In this perspective the paper presented here is a still preliminary, but fundamental step.

## References


1. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. Semantics for dynamic logic programming: a principled based approach. In *7th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, volume 1730 of *LNAI*. Springer, 2004.
2. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, 2000. A preliminary version appeared in KR'98.
3. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, September/October 2000.
4. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.
5. J.J. Alferes, F. Banti, and A. Brogi. From logic programs updates to action description updates. In *CLIMA V*, 2004.
6. F. Banti, J. J. Alferes, and A. Brogi. The well founded semantics for dynamic logic programs. In Christian Lemaître, editor, *Proceedings of the 9th Ibero-American Conference on Artificial Intelligence (IBERAMIA-9)*, LNAI, 2004.
7. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming (ICLP-99)*, Cambridge, November 1999. MIT Press.
8. DLV. The DLV project - a disjunctive datalog system (and more), 2000. Available at `http://www.dbai.tuwien.ac.at/proj/dlv/`.
9. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2002.
10. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
12. M. Homola. Dynamic logic programming: Various semantics are equal on acyclic programs. In J. Leite and P. Torroni, editors, *5th Int. Ws. On Computational Logic In Multi-Agent Systems (CLIMA V)*. Pre-Proceedings, 2004. ISBN: 972-9119-37-6.
13. J. A. Leite. Logic program updates. Master's thesis, Dept. de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, November 1997.
14. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
15. J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva - a dynamic logic programming agent architecture. In J. J. Meyer and M. Tambe, editors, *Intelligent Agents VIII — Agent Theories, Architectures, and Languages*, volume 2333 of *LNAI*, pages 141–157. Springer-Verlag, 2002.

16. J. A. Leite and L. M. Pereira. Iterated logic program updates. In J. Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP-98)*, pages 265–278, Cambridge, 1998. MIT Press.
17. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3th International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*. Morgan-Kaufmann, 1992.
18. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-99)*, volume 1730 of *LNAI*, pages 147–161, Berlin, 1999. Springer.
19. J. Sefranek. A kripkean semantics for dynamic logic programming. In *Logic for Programming and Automated Reasoning(LPAR'2000)*. Springer Verlag, LNAI, 2000.
20. SMODELS. The SMODELS system, 2000. Available at `http://www.tcs.hut.fi/Software/smodels/`.
21. A. Tarski. A lattice–theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
22. XSB-Prolog. The XSB logic programming system, version 2.6, 2003. xsb.sourceforge.net.
23. Y. Zhang and N. Y. Foo. Updating logic programs. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, 1998.

# Social Behavior of Agents and Stable models

Francesco Buccafurri and Gianluca Caminiti

DIMET, Università degli Studi Mediterranea di Reggio Calabria,
via Graziella loc. Feo di Vito, I-89060 Reggio Calabria, Italy,
{bucca, gianluca.caminiti}@unirc.it

**Abstract.** As in human world many of our goals could not be achieved without interacting with other people, in case many agents are part of the same environment one agent should be aware that he is not alone and he cannot assume other agents sharing his own goals. Moreover, he may be required to interact with other agents and to reason about their mental state in order to find out potential friends to join with (or opponents to fight against). In this paper we focus on a language derived from logic programming which both supports the representation of mental states of agent communities and provides each agent with the capability of reasoning about other agents' mental states and acting accordingly. The proposed semantics is shown to be translatable into stable model semantics of logic programs with aggregates.

## 1 Introduction

Beside *autonomy*, agents [16, 15] may be required to have *social ability*, which is the capability of interacting with other self-interested agents and, as a consequence, producing *beliefs*, *desires* and *intentions* (BDI) [2, 3] which may be dependent on such interactions.

Social ability means not only using a common language for agent communication. In this respect, KQML [13] and FIPA ACL [7], both based on the *speech act theory* by Cohen and Levesque [6], represent the main efforts done in the last years. Another important issue is reasoning about the content of such a communication [15, 14, 11]. In this paper we focus on a language derived from logic programming which both supports the representation of mental states of agent communities and provides each agent with the capability of reasoning about other agents' mental states and acting accordingly.

Consider the following example: There are four agents which have been invited to the same wedding party. Some agents are less autonomous than the others, i.e. they may decide either to join the party or not to go at all, possibly depending on the other agents' choice. Moreover some agents may tolerate some options. These are the desires of the agents:

**Agent**$_1$ will go to the party only if at least the half of the total number of agents (not including himself) goes there.

**Agent**$_2$ possibly does not go to the party, but he tolerates such an option. In case he goes, then he possibly drives the car.

**Agent**$_3$ would like to join the party together with **Agent**$_2$, but he is not so much safe with **Agent**$_2$'s driving skill. Thus he decides to go to the party only if **Agent**$_2$ both goes there and does not want to drive the car.
**Agent**$_4$ does not go to the party.

It is possible to represent the above desires using logic programming with negation as failure (*not*) where each agent is represented by a single program and requested/desired items (representing the mental state of the agent) are modelled as atoms occurring inside rule heads. In particular, mandatory items are modelled as facts. Moreover, it is possible to represent tolerated items, i.e. items which are not requested, but possibly accepted. To this aim we use the predicate *okay*(), previously introduced in [5].

However, representing the requests/acceptances of single agents in a community is not enough. A social language should provide also a machinery to handle compromises among those agents. Thus, we introduce a new construct providing one agent with the ability to reason about other agents' mental state and then to act accordingly. Program rules may have the form:

$$head \leftarrow [selection\_condition]\{body\}, \tag{1}$$

where *selection_condition* predicates about some social condition concerning either the cardinality of communities or particular individuals satisfying *body*.

For instance, consider the following rule, belonging to a program representing a given agent **A**: $\mathtt{a} \leftarrow [\, l, h \,]\, \{\mathtt{b},\ not\ \mathtt{c}\}$. This rule means that **A** will require `a` only if $n$ agents (other than **A**) exist such that they require or tolerate `b`, do not require or tolerate `c` and it holds that $0 \leq l \leq n \leq h \leq n_{agent} - 1$, where $n_{agent}$ is the total number of agents[1].

This enriched language is referred to as *SOcial Logic Programming* (SOLP). The wedding party example above may be represented by the four SOLP programs shown in Table 1, where the program $\mathcal{P}_4$ is empty since the corresponding agent has not any desire to express.

The intended models must represent the mental states of each agent inside the community. For instance, the agents' choices w.r.t. the party can be:
$\{\}$, $\{\mathtt{go\_wedding}_{\mathcal{P}_1}, \mathtt{go\_wedding}_{\mathcal{P}_2}, \mathtt{drive}_{\mathcal{P}_2}\}$, and $\{\mathtt{go\_wedding}_{\mathcal{P}_1}, \mathtt{go\_wedding}_{\mathcal{P}_2}, \mathtt{go\_wedding}_{\mathcal{P}_3}\}$, where the subscript $_{\mathcal{P}_i}$ $(1 \leq i \leq n_{agent})$ references, for each atom in a model, the program (resp. agent) that atom is entailed by. The models respectively mean that either $(i)$ no agent will go to the party, $(ii)$ only **Agent**$_1$ and **Agent**$_2$ will go and also **Agent**$_2$ will drive the car, or $(iii)$ all agents but **Agent**$_4$ will go to the party.

Indeed, **Agent**$_4$ anyway does not go. On the one hand, if **Agent**$_2$ does not go to the party, then **Agent**$_3$ will do the same. Now, let $n'$ be the number of agents which are going to the party, it is $n' = 0$. **Agent**$_1$ requires that at least $\nu = \frac{n_{agent}}{2} - 1$ agents (other than himself) go to the party, but since it is $\nu = \frac{4}{2} - 1 = 1$ and $n' < \nu$, then **Agent**$_1$ does not go to the party (case $(i)$).

[1] By default, $l = 0$ and $h = n_{agent} - 1$.

| $\mathcal{P}_1$ (**Agent**$_1$) : | $\mathcal{P}_2$ (**Agent**$_2$) : |
|---|---|
| `go_wedding` $\leftarrow [\frac{n_{agent}}{2} - 1,\ ]\{$`go_wedding`$\}$ | *okay*(`go_wedding`) $\leftarrow$ <br> *okay*(`drive`) $\leftarrow$ `go_wedding` |
| $\mathcal{P}_3$ (**Agent**$_3$) : | $\mathcal{P}_4$ (**Agent**$_4$) : |
| `go_wedding` $\leftarrow$ [**Agent**$_2$]{`go_wedding`, *not* `drive`} | *empty program* |

**Table 1.** The wedding party example

On the other hand, if **Agent**$_2$ goes to the party, it is possible that he wants either to drive the car or not. If he wants to drive, then **Agent**$_3$ will not join the party. Now, it is $n' = 1 = \nu$, then **Agent**$_1$ will go to the party (case $(ii)$). Otherwise, if **Agent**$_2$ does not want to drive the car, then all conditions required by **Agent**$_3$ are satisfied, thus he will go to the party. Now, it is $n' = 2 > \nu$ and then **Agent**$_1$ will join the party too (case $(iii)$).

The intended models are referred to as *social models*, since they express the results of the interactions among agents.

Our work is strongly related to [5], where the *Joint Fixpoint Semantics* (JFP), that is a semantics providing a way to reach a compromise (in terms of a common agreement) among many agents, is proposed. Therein, each model contains atoms representing items being common to all the agents. Our paper extends such a semantics, providing *feature-selective atom subset community*, i.e. given a set $S$ of SOLP programs representing a community of agents, a program $\mathcal{P} \in S$ and a rule $r \in \mathcal{P}$ of the form $head \leftarrow [selection\_condition]\{body\}$, then $head$ will belong to an intended model if all properties enclosed in $\{body\}$ are entailed by either $(i)$ any subset $S' \subseteq (S \setminus \{\mathcal{P}\})$ of programs with a given cardinality (specified by $[selection\_condition]$) or $(ii)$ some particular program different from $\mathcal{P}$.

An example of case $(i)$ is shown in Table 1 by the program $\mathcal{P}_1$: An intended model $M$ will include the atom `go_wedding`$_{\mathcal{P}_1}$ if a set of programs $S' \subseteq \{\mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4\}$ exists such that $\forall \mathcal{P}_i \in S'$, `go_wedding`$_{\mathcal{P}_i} \in M$ and $|S'| \geq \frac{n_{agent}}{2} - 1$. An example of case $(ii)$ is represented by the program $\mathcal{P}_3$, which requests the atom `go_wedding`$_{\mathcal{P}_3}$ to be part of an intended model $M$ if `go_wedding`$_{\mathcal{P}_2}$ belongs to $M$, but the atom `drive`$_{\mathcal{P}_2}$ does not.

Importantly, social constraints can be nested. Consider for example the program: `download`$(X) \leftarrow [min,\ ]\{$`shared`$(X), [1,\ ]\{not$ `incomplete`$(X)\}\}$, `file`$(X)$. This program represents a Peer-to-Peer file-sharing system where a user can share his collection of files with other users on the Internet. In order to get better performances, a file is split into several parts being downloaded separately (possibly

each part from a different user)[2]. Thus, the program describes the behavior of an agent (acting on behalf of a given user) that wants to download any file $X$ being shared by at least a number *min* of users such that at least one of them owns a complete version of $X$.

We show also that, given a set of SOLP programs in input, a source-to-source transformation is possible which provides as output a single DLP$^{\mathcal{A}}$ [8] program whose stable models are in one-to-one correspondence with the intended ones. We recall that DLP$^{\mathcal{A}}$ is basically disjunctive logic programming with aggregate functions, supported by the DLV system [9]. The translation to DLP$^{\mathcal{A}}$ give us the ability of exploiting DLV (widely accepted as the state-of-the-art system implementing disjunctive logic programming)[3]. Observe that since our language includes neither disjunction nor classical negation (even though the extensions to these cases could be considered), both disjunction and classical negation of DLP$^{\mathcal{A}}$ are never enabled by our translation. Moreover, Section 5 shows that our kind of social reasoning is not trivial, since even in the case of positive programs, the semantics of SOLP has a computational complexity which is NP complete.

The paper is organized as follows: in Sections 2 and 3 we respectively define the notion of SOLP programs and define their semantics (*Social Semantics*). In Section 4 we illustrate how a set of SOLP programs, each representing a different agent, is translated into a single DLP$^{\mathcal{A}}$ logic program whose stable models describe the mental states of the whole agent community and then we show that such a translation is correct. In Section 5 we prove that the Social Semantics extends the JFP Semantics [5] and we study the complexity of the problem of searching for a social model. In Section 6 we describe how this novel approach may be used for knowledge representation and finally, we draw our conclusions. For space restrictions, proofs of theorems and lemmata are omitted. They can be found in [4].

## 2 Social Logic Programs: Basic Definitions

In this section we introduce the notion of SOLP program.

A *term* is either a variable or a constant. An *atom* or *positive literal* is an expression $p(t_1, \cdots, t_n)$, where $p$ is a *predicate* of arity $n$ and $t_1, \cdots, t_n$ are terms. A *negative literal* is the *negation as failure (NAF) not* $a$ of a given atom $a$.

**Definition 1.** A ($n$-)*social selection constraint* $s$, said also ($n$-)SSC, is an expression of the form $cond(s)$ $property(s)$, such that:

(1) $cond(s)$ is an expression $[\alpha]$ where $\alpha$ is either $(i)$ a pair of integers $l, h$ such that $0 \leq l \leq h \leq n-1$, or $(ii)$ an integer belonging to $\{1, \cdots, n\}$ said *program identifier*[4].

[2] Among others, KaZaA, EDonkey, WinMX and BitTorrent are the most popular Internet P2P file-sharing systems exploiting such a feature.

[3] Of course, our approach may easily be adapted to other systems supporting cardinality constraints, such as Smodels.

[4] We will show, at the end of the section, that the program identifier uniquely identifies a program (i.e., an agent).

(2) $property(s) = content(s) \cup skel(s)$, where $content(s)$ is a non-empty set of literals and $skel(s)$ is a (possibly empty) set of SSCs.

Concerning item (1) of the above definition, in case $(i)$, $cond(s)$ is said *cardinal selection condition*, while, in case $(ii)$, $cond(s)$ is said *member selection condition*.

$n$-social selection constraints operate over a collection of $n$ programs (we will formally define later in this section which kind of program are allowed). Thus, with a little abuse of notation, we often denote a member selection condition by $[\mathcal{P}_j]$ instead of $[j]$.

Concerning item (2) of Definition 1, if $skel(s) = \emptyset$ then $s$ is said *simple*. For a simple SSC $s$ such that $content(s)$ is singleton, the enclosing braces can be omitted. Finally, given a SSC $s$, the formula *not* $s$ is said the NAF of $s$.

In our initial wedding party example, $[\frac{n_{agent}}{2} - 1, ]\{\texttt{go\_wedding}\}$ and $[\mathbf{Agent}_2]$ $\{\texttt{go\_wedding}, not\, \texttt{drive}\}$ are two simple SSCs. On the contrary, the SSC occurring in the example regarding a Peer-to-Peer system (see Page 3) is not simple.

As a further example, if $s = [l, h]\{a, b, c, [l_1, h_1]\{d, [l_2, h_2]e\}, [l_3, h_3]f\}$, then $s$ is not simple, $content(s) = \{a, b, c\}$ and $skel(s) = \{[l_1, h_1]\{d, [l_2, h_2]e\}, [l_3, h_3]f\}$.

Now we define a function which returns, for a given SSC $s$, its nesting depth.

Given a SSC $s$, we define the function *depth* as follows:

$$\begin{cases} depth(s) = depth(s') + 1, & \text{if } \exists s' \mid s \in skel(s') \\ depth(s) = 0, & \text{otherwise.} \end{cases}$$

Given two SSCs $s$ and $s'$ such that $cond(s) = [l, h]$ and $cond(s') = [l', h']$, i.e. they are cardinal selection conditions, we say that $cond(s') \subseteq cond(s)$ if $h' \leq h$.

A SSC $s$ is *well-formed* if either $(i)$ $s$ is simple, or $(ii)$ $s$ is not simple, $cond(s)$ is a cardinal selection condition and $\forall s' \in skel(s)$ it holds that:

($a$) If $cond(s')$ is a cardinal selection condition, then $s'$ is *well-formed* and $cond(s') \subseteq cond(s)$;
($b$) If $cond(s')$ is a member selection condition, then $s'$ is simple.

From now on, we consider only well-formed SSCs.

*Example 1.* The SSC $s = [1, 8]\{a, [3, 6]\{b, [\mathbf{Agent_X}]\{c, d\}\}\}$ is well-formed, while $s_1 = [4, 7]\{a, [3, 9]b\}$ is not a well-formed SSC, because $[3, 9] \not\subseteq [4, 7]$.

We introduce now the notion of rule. Our definition generalizes the notion of classical logic rule.

**Definition 2.** A ($n$-)*social rule* $r$ is a is a formula $a \leftarrow b_1 \wedge \cdots \wedge b_m \wedge s_1 \wedge \cdots \wedge s_k$ ($m \geq 0$, $k \geq 0$), where $a$ is an atom, each $b_i$ ($1 \leq i \leq m$) is a literal and each $s_j$ ($1 \leq j \leq k$) is either a $n$-SSC or the NAF of a $n$-SSC. The atom $a$ is said the *head* of $r$, while the conjunction $b_1 \wedge \cdots \wedge b_m \wedge s_1 \wedge \cdots \wedge s_k$ is said the *body* of $r$. In case $a$ is of the form $okay(p)$, where $p$ is an atom, then $r$ it is said ($n$-)*tolerance (social) rule* and $p$ is said the *head* of $r$. In case $k = 0$, a social non-tolerance rule is said *classical rule*.

Given a rule $r$, we denote by $head(r)$ (resp. $body(r)$) the head (resp. the body) of $r$. Moreover, $r$ is said a *fact* in case the body is empty, while $r$ is said an *integrity constraint* if the head is missing.

**Definition 3.** A SOLP *collection* is a set $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ of SOLP programs, where each SOLP *program* is a set of $n$-social rules. The cardinal $i$ $(1 \leq i \leq n)$ is called *program identifier* of the program $\mathcal{P}_i$.

A SOLP program is *positive* if no NAF symbol *not* occurs in it. For the sake of presentation we only refer, in the following sections, to *ground* (i.e., variable-free) SOLP programs – the extension to the general case is straightforward.

## 3 Semantics of SOLP programs

In this section we introduce the *Social Semantics*, i.e. the semantics of a collection of SOLP programs. We assume the reader is familiar with the basic concepts of logic programming [1, 12].

We start by introducing the notion of interpretation for a single SOLP program (note that this is the same as for classical programs). An *interpretation* for a ground SOLP program $\mathcal{P}$ is a subset of $Var(\mathcal{P})$, where $Var(\mathcal{P})$ is the set of atoms appearing in $\mathcal{P}$. A positive literal $a$ (resp. a negative literal *not* $a$) is *true* w.r.t. an interpretation $I$ if $a \in I$ (resp. $a \notin I$); otherwise it is *false*. A rule is *satisfied* (or is *true*) w.r.t. $I$ if its head is true or its body is false w.r.t. $I$.

Before defining the intended models of our semantics, we need some preliminary definitions. Let $\mathcal{P}$ be a SOLP program. We define the *autonomous reduction* of $\mathcal{P}$, denoted by $A\mathcal{P}$, the program obtained from $\mathcal{P}$ by removing all the SSCs from the rules in $\mathcal{P}$. Thus, if the program $\mathcal{P}$ represents the social behavior of an agent, then $A\mathcal{P}$ represents the behavior of the same agent in case he decides to operate independently of the other agents.

Given a SOLP program $\mathcal{P}$ and an interpretation $I \subseteq Var(A\mathcal{P})$, let $\mathrm{CL}(A\mathcal{P})$ (resp. $\mathrm{TR}(A\mathcal{P})$) be the set of classical (resp. tolerance) rules in $A\mathcal{P}$. The *autonomous immediate consequence operator* $AT_{\mathcal{P}}$ is the function from $2^{Var(A\mathcal{P})}$ to $2^{Var(A\mathcal{P})}$ defined as follows:

$$AT_{\mathcal{P}}(I) = \{head(r) \mid \forall r \in CL(A\mathcal{P}), body(r) \text{ is true w.r.t. } I\} \cup$$
$$\{head(r) \mid \forall r \in TR(A\mathcal{P}), body(r) \wedge head(r) \text{ is true w.r.t. } I\}.$$

**Definition 4.** An interpretation $I$ for a SOLP program $\mathcal{P}$ is an *autonomous fixpoint* of $\mathcal{P}$ if $I$ is a fixpoint of the associated transformation $AT_{\mathcal{P}}$, i.e. if $AT_{\mathcal{P}}(I) = I$. The set of all autonomous fixpoints of $\mathcal{P}$ is denoted by $AFP(\mathcal{P})$.

Thus, the autonomous fixpoints of a given SOLP program $\mathcal{P}$ represent the mental states of the corresponding agent, whenever every social constraint in $\mathcal{P}$ is discarded.

**Definition 5.** Let $\mathcal{P}$ be a SOLP program and $L$ be a set of literals. The *labeled version* of $L$ w.r.t. $\mathcal{P}$ is the set $L_{\mathcal{P}} = \{a_{\mathcal{P}} \mid a \in L\}$.

Let $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ be a SOLP collection. A *social interpretation* for $C$ is a set $\bar{I} = I^1_{\mathcal{P}_1} \cup \cdots \cup I^n_{\mathcal{P}_n}$, where $I^j$ is an interpretation for $\mathcal{P}_j$ $(1 \leq j \leq n)$.

*Example 2.* If $C = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\}$, $I^1 = \{a, b, c\}$, $I^2 = \{a, d, e\}$ and $I^3 = \{b, c, d\}$, where $I^j$ is an interpretation for $\mathcal{P}_j$ $(1 \leq j \leq 3)$, then $\bar{I} = \{a_{\mathcal{P}_1}, b_{\mathcal{P}_1}, c_{\mathcal{P}_1}, a_{\mathcal{P}_2}, d_{\mathcal{P}_2}, e_{\mathcal{P}_2}, b_{\mathcal{P}_3}, c_{\mathcal{P}_3}, d_{\mathcal{P}_3}\}$ is a social interpretation for $C$.

Let $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ be a SOLP collection and $\mathcal{P} \in C$. Given a social interpretation $\bar{I}$ for $C$, a positive literal $a \in Var(\mathcal{P})$ (resp. a negative literal *not* $a$) is *true* w.r.t. $\bar{I}$ if $a_{\mathcal{P}} \in \bar{I}$ (resp. $a_{\mathcal{P}} \notin \bar{I}$); otherwise it is *false*.

Before giving the definition of truth for a SSC, we introduce a way to reference any SSC $s$ (and also every SSC nested in $s$) occurring in a given rule $r$ of a SOLP program $\mathcal{P}$.

Given a SOLP program $\mathcal{P}$, a social rule $r \in \mathcal{P}$ and an integer $n \geq 0$, we define the set $MSSC^{\langle \mathcal{P}, r, n\rangle} = \{s \mid s \text{ is a SSC occurring in } r \in \mathcal{P} \wedge depth(s) = n\}$. Observe that $MSSC^{\langle \mathcal{P}, r, 0\rangle}$ denotes the set of SSCs as they appear in the rule $r$ of the SOLP program $\mathcal{P}$.

*Example 3.* Let $a \leftarrow [1,8]\{a, [3,6]\{b, [\mathbf{Agent_X}]\{c, d\}\}\}, [2,3]\{e, f\}$ be a rule $r$ in a SOLP program $\mathcal{P}$. Then:

$MSSC^{\langle \mathcal{P}, r, 0\rangle} = \{\ [1,8]\{a, [3,6]\{b, [\mathbf{Agent_X}]\{c, d\}\}\}, [2,3]\{e, f\}\ \},$
$MSSC^{\langle \mathcal{P}, r, 1\rangle} = \{\ [3,6]\{b, [\mathbf{Agent_X}]\{c, d\}\}\ \},$
$MSSC^{\langle \mathcal{P}, r, 2\rangle} = \{\ [\mathbf{Agent_X}]\{c, d\}\ \},$
$MSSC^{\langle \mathcal{P}, r, 3\rangle} = \emptyset.$

Given a SOLP program $\mathcal{P}$, we define the set $MSSC^{\mathcal{P}} = \bigcup_{r \in \mathcal{P}} MSSC^{\langle \mathcal{P}, r, 0\rangle}$. Thus $MSSC^{\mathcal{P}}$ is the set of all the SSCs (with depth 0) occurring in $\mathcal{P}$.

Now we provide the definition of truth of a SSC w.r.t. a given social interpretation and, subsequently, the definition of truth of a social rule.

**Definition 6.** Let $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ be a SOLP collection, $C' \subseteq C$ and $\mathcal{P}_j \in C'$. Given a social interpretation $\bar{I}$ for $C'$ and a $n$-SSC $s \in MSSC^{\mathcal{P}_j}$, we say that $s$ is *true* in $C'$ w.r.t. $\bar{I}$ if it holds that either:

(1) $cond(s) = [k] \wedge$
$\exists \mathcal{P}_k \in C' \mid \forall a_{\mathcal{P}_k} \in (content(s))_{\mathcal{P}_k} \quad a$ is true w.r.t. $\bar{I}$, or

(2) $cond(s) = [l, h] \wedge$
$\exists D \subseteq C' \setminus \{\mathcal{P}_j\} \mid l \leq |D| \leq h \wedge$
$\forall a_{\mathcal{P}} \in \bigcup_{\mathcal{P} \in D} (content(s))_{\mathcal{P}} \quad a$ is true w.r.t. $\bar{I} \wedge$
$\forall s' \in skel(s)\ \exists D' \subseteq D \mid s'$ is true in $D'$ w.r.t. $\bar{I}$,

where $l$, $h$ and $k$ are integers (observe that $k$ is a program identifier). If $C' = C$, then we simply say that $s$ is *true* w.r.t. $\bar{I}$. A $n$-SSC not true (in $C'$) w.r.t. $\bar{I}$ is said *false* (in $C'$) w.r.t. $\bar{I}$.

Finally, the NAF of a $n$-SSC $s$, *not* $s$, is said *true* (resp. *false*) (in $C'$) w.r.t. $\bar{I}$ if $s$ is false (resp. true) (in $C'$) w.r.t. $\bar{I}$.

Thus, given a SSC $s$ included in $\mathcal{P}_j$, $s$ is true w.r.t. a social interpretation $\bar{I}$ if a single SOLP program corresponding to a program identifier $k$ (resp. a set of

SOLP programs) exists (resp. not including $\mathcal{P}_j$) such that all the elements in *property*$(s)$ are true w.r.t. $\bar{I}$. Observe that the truth of *property*$(s)$ w.r.t. $\bar{I}$ is possibly defined recursively, since $s$ may contain nested SSCs.

Once the notion of truth of SSCs has been defined, we are able to define the notion of satisfaction of a social rule w.r.t. a social interpretation.

Let $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ be a SOLP collection and $\mathcal{P} \in C$. Given a social interpretation $\bar{I}$ for $C$, a social rule in $\mathcal{P}$ is *satisfied* (or is *true*) w.r.t. $\bar{I}$ if its head is true w.r.t. $\bar{I}$ or its body is false w.r.t. $\bar{I}$.

Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, we define the set of *candidate social interpretations* for $\mathcal{P}_1, \cdots, \mathcal{P}_n$ as

$$\mathcal{U}(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \left\{F^1_{\mathcal{P}_1} \cup \cdots \cup F^n_{\mathcal{P}_n} \mid F^i \in AFP(\mathcal{P}_i), 1 \leq i \leq n\right\}.$$

where, recall, $AFP(\mathcal{P}_i)$ is the set of autonomous fixpoints of the SOLP program $\mathcal{P}_i$, introduced in Definition 4 and by $F^i_{\mathcal{P}}$ $(1 \leq i \leq n)$ we denote the labeled version of $F^i$ w.r.t. $\mathcal{P}$ (see Definition 5). $\mathcal{U}(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ represents all the configurations obtained by combining the autonomous (i.e. without considering the social constraints) mental states of the agents corresponding to the programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$. Each candidate social interpretation is a candidate intended model.

The intended models are then obtained by enabling the social constraints.

Now, we are ready to give the definition of intended model w.r.t. the Social Semantics.

**Definition 7.** Given a SOLP collection $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, a candidate social interpretation $\bar{I}$ for $C$ is a *social model* of $C$ if $\forall r \in \bigcup_{1 \leq i \leq n} \mathcal{P}_i$, $r$ is true w.r.t. $\bar{I}$.

**Definition 8.** Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, the *Social Semantics* (often referred to as *S-Semantics*) of $\mathcal{P}_1, \cdots, \mathcal{P}_n$ is the set

$$\mathcal{SOS}(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \{\bar{M} \mid \bar{M} \in \mathcal{U}(\mathcal{P}_1, \cdots, \mathcal{P}_n) \wedge \bar{M} \text{ is a social model of } \mathcal{P}_1, \cdots, \mathcal{P}_n\},$$

Thus $\mathcal{SOS}(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is the set of all social models of $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

Given a SOLP collection $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ and a SOLP program $\mathcal{P} \in C$, we define the *S-Semantics* of $\mathcal{P}$ as

$$\mathcal{S}(\mathcal{P}) = \{F \mid F \in AFP(\mathcal{P}) \wedge \exists \bar{M} \in \mathcal{SOS}(\mathcal{P}_1, \cdots, \mathcal{P}_n) \mid F_{\mathcal{P}} \subseteq \bar{M}\},$$

Hence $\mathcal{S}(\mathcal{P})$ represents the autonomous mental states of an agent, corresponding to a SOLP program $\mathcal{P}$, which are also included in some social model of $\mathcal{P}_1, \cdots, \mathcal{P}_n$, and then fulfill all social requirements.

## 4 Translation

In this section we give the translation from SOLP under the Social Semantics to DLP$^{\mathcal{A}}$ [8] under Stable Model Semantics. We assume that the reader is familiar with the Stable Model Semantics [10]. Given a classical logic program $\mathcal{P}$, we denote by $SM(\mathcal{P})$ the set of all the stable models of $\mathcal{P}$.

Given a SOLP program $\mathcal{P}$, we define the set $USSC^{\mathcal{P}} = \bigcup_{r \in \mathcal{P}} \bigcup_{n \geq 0} MSSC^{\langle \mathcal{P}, r, n \rangle}$. Thus, $USSC^{\mathcal{P}}$ includes all the SSCs (at any nesting depth) in $\mathcal{P}$.

Given a SOLP program $\mathcal{P}$, we define the functions $\rho$ and $g$, each establishing a one-to-one correspondence between each element in $USSC^{\mathcal{P}}$ and a set of atoms $L$ such that both $(i)$ $L \cap Var(\mathcal{P}) = \emptyset$ and $(ii)$ $\forall s, t \in USSC^{\mathcal{P}}, \rho(s) \neq g(t)$. Thus, given a SSC $s$ included in a SOLP program $\mathcal{P}$, $\rho(s)$ is a unique positive literal identifying $s$ and we denote by $(\rho(s))_{\mathcal{P}}$ the labeled version of $\rho(s)$ w.r.t. $\mathcal{P}$. Similar considerations hold for $g(s)$. Moreover, with a little abuse of notation we write $(g(s))_{\mathcal{P}}(x)$ denoting the labeled version of the predicate $(g(s))(x)$ w.r.t. $\mathcal{P}$.

Now we introduce the translation of a single SSC and then we extend such a translation to all SSCs included in a social rule, a SOLP program and a SOLP collection, respectively.

**Definition 9.** Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_j, \cdots, \mathcal{P}_n\}$ and $s \in USSC^{\mathcal{P}_j}$, we define the *translation of* $s$ as the DLP$^{\mathcal{A}}$ program $\Psi^{\mathcal{P}_j}(s) = GUESS^{\mathcal{P}_j}(s) \cup CHECK^{\mathcal{P}_j}(s)$, where $GUESS^{\mathcal{P}_j}(s) =$

$$= \begin{cases} \{(g(s))_{\mathcal{P}_j}(k) \leftarrow \bigwedge_{b \in content(s)} b_{\mathcal{P}_k}\}, & \text{if } cond(s) = [k], \\ \{(g(s))_{\mathcal{P}_j}(i) \leftarrow \bigwedge_{b \in content(s)} b_{\mathcal{P}_i} \wedge \\ \quad \bigwedge_{s' \in skel(s)} (g(s'))_{\mathcal{P}_j}(i) \mid \\ \quad 1 \leq i \neq j \leq n\} \cup \\ \quad \{GUESS^{\mathcal{P}_j}(s') \mid s' \in skel(s)\}, & \text{if } cond(s) = [l, h], \end{cases}$$

and $CHECK^{\mathcal{P}_j}(s) =$

$$= \begin{cases} \{(\rho(s))_{\mathcal{P}_j} \leftarrow (g(s))_{\mathcal{P}_j}(k)\}, & \text{if } cond(s) = [k], \\ \{(\rho(s))_{\mathcal{P}_j} \leftarrow l \leq \#\texttt{count}\{K : (g(s))_{\mathcal{P}_j}(K), K \neq j\} \leq h\} \cup \\ \quad \{CHECK^{\mathcal{P}_j}(s') \mid s' \in skel(s)\}, & \text{if } cond(s) = [l, h], \end{cases}$$

where `#count` is an aggregate function which returns the cardinality of a set of literals satisfying some conditions [8]. Observe that the above translation produces a safe aggregate-stratified DLP$^{\mathcal{A}}$ program and thus the computational complexity remains the same as for standard DLP [8, 9].

Given a SOLP program $\mathcal{P}_j$ and a social rule $r \in \mathcal{P}_j$, we define the *SSC translation of* $r$ as the DLP$^{\mathcal{A}}$ program $T^{\mathcal{P}_j}(r) = \bigcup_{s \in MSSC^{\langle \mathcal{P}, r, 0 \rangle}} \Psi^{\mathcal{P}_j}(s)$. Observe that, for any classical rule $r \in \mathcal{P}_j$, it holds that $T^{\mathcal{P}_j}(r) = \emptyset$.

Given a SOLP program $\mathcal{P}_j$, the *SSC translation of* $\mathcal{P}_j$ is the DLP$^{\mathcal{A}}$ program $W^{\mathcal{P}_j} = \bigcup_{r \in \mathcal{P}_j} T^{\mathcal{P}_j}(r)$.

**Definition 10.** Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, we define the *SSC translation of the collection* as the DLP$^{\mathcal{A}}$ program $C(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \bigcup_{1 \leq i \leq n} W^{\mathcal{P}_i}$.

Thus $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is a DLP$^{\mathcal{A}}$ program representing the translation of all the SSCs included in $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

We have defined above the translation for the SSCs included in a SOLP program. Now we give the translation for the whole SOLP program, but first we need a preliminary processing of all tolerance rules. The latter is done by means of the transformation defined as follows:

Given a SOLP program $\mathcal{P}$, $\hat{\mathcal{P}} = \mathcal{P} \setminus TR(\mathcal{P}) \cup \{head(r) \leftarrow head(r) \wedge body(r) \mid r \in TR(\mathcal{P})\}$. Thus $\hat{\mathcal{P}}$ is obtained from $\mathcal{P}$ by replacing each tolerance rule $okay(p) \leftarrow body$ with the rule $p \leftarrow p, body$.

**Definition 11.** Let $\mathcal{P}$ be a SOLP program. We define the program $\Gamma'(\hat{\mathcal{P}})$ over the set of atoms $Var(\Gamma'(\hat{\mathcal{P}})) = \{a_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup \{a'_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup \{sa_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup \{fail_{\mathcal{P}}\}$ as $\Gamma'(\hat{\mathcal{P}}) = S'_1(\hat{\mathcal{P}}) \cup S'_2(\hat{\mathcal{P}}) \cup S'_3(\hat{\mathcal{P}})$, where $S'_1(\hat{\mathcal{P}})$, $S'_2(\hat{\mathcal{P}})$ and $S'_3(\hat{\mathcal{P}})$ are defined as follows:

$S'_1(\hat{\mathcal{P}}) = \{a_{\mathcal{P}} \leftarrow not\ a'_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup \{a'_{\mathcal{P}} \leftarrow not\ a_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\}$,

$S'_2(\hat{\mathcal{P}}) = \{sa_{\mathcal{P}} \leftarrow b^1_{\mathcal{P}}, \cdots, b^n_{\mathcal{P}}, (\rho(s_1))_{\mathcal{P}}, \cdots (\rho(s_m))_{\mathcal{P}} \mid a \leftarrow b_1, \cdots b_n, s_1, \cdots, s_m \in \mathcal{P}\}$,

$S'_3(\hat{\mathcal{P}}) = \{fail_{\mathcal{P}} \leftarrow not\ fail_{\mathcal{P}}, sa_{\mathcal{P}}, not\ a_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup$
$\{fail_{\mathcal{P}} \leftarrow not\ fail_{\mathcal{P}}, a_{\mathcal{P}}, not\ sa_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\}$.

**Definition 12.** Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, we define the set $P'_u = \bigcup_{1 \leq i \leq n} \Gamma'(\hat{\mathcal{P}}_i)$.

Thus $P'_u$ is a classical logic program representing the translation of the SOLP collection. This program is used in conjunction with the DLP$^{\mathcal{A}}$ program $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ in order to enable the social constraints.

In the next theorem we state that a one-to-one correspondence exists between the social models in $SOS(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ and the stable models of the DLP$^{\mathcal{A}}$ program $P'_u \cup \bar{C}$. First, we need the following definition and results:

Let $\mathcal{P}$ be a SOLP program and $M \subseteq Var(\mathcal{P})$. We denote by $[M]_{\mathcal{P}}$ the set $\{a_{\mathcal{P}} \mid a \in M\} \cup \{a'_{\mathcal{P}} \mid a \in Var(\mathcal{P}) \setminus M\} \cup \{sa_{\mathcal{P}} \mid a \in M\}$.

**Lemma 1.** Given a SOLP collection $SP = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, a social interpretation $\bar{I}$ for $SP$, a SOLP program $\mathcal{P}_j \in SP$ and a SSC $s \in MSSC^{\mathcal{P}_j}$, assume $\bar{C} = C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ and $Q = \{a \leftarrow \mid a \in \bar{I}\}$. Then:

$$s \text{ is true w.r.t. } \bar{I} \text{ iff } \exists M \in SM(\bar{C} \cup Q) \mid (\rho(s))_{\mathcal{P}_j} \in M.$$

**Definition 13.** Given a SOLP collection $SP = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, a social interpretation $\bar{I}$ for $SP$, a SOLP program $\mathcal{P}_j \in SP$ and a SSC $s \in MSSC^{\mathcal{P}_j}$, assume $\bar{C} = C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ and $Q = \{a \leftarrow \mid a \in \bar{I}\}$. We define the set

$$SAT^{\mathcal{P}_j}_{\bar{I}}(s) = \{\ h \mid h = head(r), r \in \Psi^{\mathcal{P}_j}(s) \wedge \exists M \in SM(\bar{C} \cup Q) \mid h \in M\}.$$

Thus, by virtue of Lemma 1, if $s$ is true w.r.t. $\bar{I}$, then $SAT^{\mathcal{P}_j}_{\bar{I}}(s)$ includes the literal $(\rho(s))_{\mathcal{P}_j}$ and those heads of rules in $\Psi^{\mathcal{P}_j}(s)$ corresponding (by means of the functions $\rho$ and $g$) to both $s$ and the SSCs which are nested in $s$.

**Theorem 1.** Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, and $\bar{C} = C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$. Then $A = B$, where:

$A = \quad SM(P'_u \cup \bar{C})$ and
$B = \{\ \bar{F} \cup \bar{G} \cup \bar{H} \mid$
$\bar{F} = \bigcup_{1 \leq i \leq n} F^i_{\mathcal{P}_i} \wedge F^i \in AFP(\mathcal{P}_i) \wedge \bar{F} \in SOS(\mathcal{P}_1, \cdots, \mathcal{P}_n)$
$\bar{G} = \bigcup_{1 \leq i \leq n} G^i_{\mathcal{P}_i} \wedge G^i_{\mathcal{P}_i} = [F^i]_{\mathcal{P}_i} \setminus F^i_{\mathcal{P}_i}$
$\bar{H} = \bigcup_{1 \leq i \leq n} H^i_{\mathcal{P}_i} \wedge H^i_{\mathcal{P}_i} = \bigcup_{s \in MSSC^{\mathcal{P}_i}} SAT^{\mathcal{P}_i}_{\bar{F}}(s)\}$.

Thus each stable model $x \in A$ may be partitioned in three sets: $\bar{F}$ (the social model of the SOLP collection, which corresponds to $x$), $\bar{G}$ and $\bar{H}$ (both including overhead literals needed by the translation).

## 5 Social Models, Joint Fixpoints and Complexity

In this section we show that the Social Semantics extends the JFP semantics [5]. Basically, COLP programs are logic programs containing also *tolerance* rules, that are rules of the form $okay(p) \leftarrow body(r)$. The semantics of a collection of COLP programs is defined over classical programs obtained by the COLP programs by translating each rule of the form $okay(p) \leftarrow body(r)$ into the rule $p \leftarrow p, body(r)$. The semantics of a collection $\mathcal{P}_1, \cdots, \mathcal{P}_n$ of COLP programs is defined in [5] in terms of joint (i.e., common) fixpoints (of the *immediate consequence operator*) of the logic programs obtained from $\mathcal{P}_1, \cdots, \mathcal{P}_n$ by transforming *tolerance* rules occurring in them (as shown above). Recall that the *immediate consequence operator* $T_{\mathcal{P}}$ is a function from $2^{Var(\mathcal{P})}$ to $2^{Var(\mathcal{P})}$ defined as follows. For each interpretation $I \subseteq Var(\mathcal{P})$, $T_{\mathcal{P}}(I)$ is the set of all heads of rules in $\mathcal{P}$ whose bodies are true w.r.t. $I$.

First, we define a translation from COLP programs [5] to SOLP programs:

**Definition 14.** Given a COLP program $\mathcal{P}$ and an integer $n \geq 1$, the *SOLP translation* of $\mathcal{P}$ is a SOLP program $\sigma^n(\mathcal{P}) = \{\sigma_{rule}(r) \mid r \in \mathcal{P}\}$, where

$$\sigma_{rule}(r) = \begin{cases} head(r) \leftarrow [n-1, n-1]head(r), body(r) & \text{if } r \text{ is a classical rule,} \\ okay(p) \leftarrow [n-1, n-1]p, body(r) & \text{if } head(r) = okay(p). \end{cases}$$

The next theorem states that the JFP semantics is a special case of the Social Semantics. $JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ denotes the set of the joint fixpoints of $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

**Theorem 2.** Given $n \geq 1$, let $\mathcal{P}_1, \cdots, \mathcal{P}_n$ be COLP programs and $Q_1, \cdots, Q_n$ be SOLP programs such that $Q_i = \sigma^n(\mathcal{P}_i)$. Then:

$$\mathcal{SOS}(Q_1, \cdots, Q_n) = \left\{ \bigcup_{1 \leq i \leq n} F_{Q_i} \mid F \in JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n) \right\}.$$

Now we introduce a relevant decision problem w.r.t. the Social Semantics and discuss its complexity. Observe that the analysis is done in case of positive programs. Indeed, the case of non positive programs is straightforward: Since it is NP complete to determine whether a *single* non-positive program has a fixpoint, it is easy to see that the same holds for non-positive SOLP programs and autonomous fixpoints. Thus, checking whether a SOLP collection containing at least one non-positive SOLP program has a social model is trivially NP hard. Moreover, since this problem is easily seen to be in NP, it is NP complete.

**PROBLEM $SOS_n$ (social model existence):**

**Instance:** A SOLP collection $\mathcal{P}_1, \ldots, \mathcal{P}_n$
**Question:** Is $\mathcal{SOS}(\mathcal{P}_1, \ldots \mathcal{P}_n) \neq \emptyset$, i.e., does the SOLP collection $\mathcal{P}_1, \ldots, \mathcal{P}_n$ have any social model?

**Theorem 3.** The problem $SOS_n$ is NP complete.

## 6 Knowledge Representation with SOLP programs

In this section, we provide two real-life examples showing the capability of our language of representing common knowledge.

*Example 4.* **Seating.** We must arrange a seating for a number $n_{agent}$ of agents, with $m$ tables and a maximum of $c$ chairs per table. Agents who like (resp. dislike) each other should (resp. should not) sit at the same table. Moreover, an agent can express some requirements w.r.t. the number and the identity of other agents sitting at the same table. Assume that the $i$-th agent is represented by a predicate $agent(i)$ $(1 \leq i \leq n_{agent})$ and his knowledge base is included in a single SOLP program. The predicate $like(i)$ (resp. $dislike(i)$) means that $Agent_i$ is desired (resp. not tolerated) at the same table, $table(T)$ represents a table $(1 \leq T \leq m)$ and $at(T)$ expresses the desire to sit at table $T$. For instance, the program $\mathcal{P}_1$ (which is associated to $Agent_1$) could be written as follows:

$$
\begin{array}{rl}
r_1: & agent(1) \leftarrow \\
r_2: & \leftarrow at(T1), at(T2), T1 <> T2 \\
r_3: & at(T) \leftarrow [, c-1]\{at(T), agent(P)\}, like(P), table(T) \\
r_4: & \leftarrow at(T), [1,]\{at(T), agent(P)\}, dislike(P) \\
r_5: & \leftarrow like(P), dislike(P) \\
r_6: & like(2) \leftarrow \\
r_7: & dislike(3) \leftarrow \\
r_8: & okay(like(4)) \leftarrow \\
r_9: & \leftarrow at(T), [3,]\{at(T)\}
\end{array}
$$

where rules from $r_1$ to $r_5$ are common to all the programs (of course, the argument of $agent()$ in $r_1$ is suited to the enclosing program) and rules from $r_6$ to $r_9$ express agent's own requirements. In particular, while the rule $r_2$ states that any agent cannot be seated at more than one table, the rules $r_3$ and $r_4$ mean that an agent wants to share the table with no more than $c-1$ agents he likes and with no agent he dislikes, respectively. The rule $r_5$ provides consistency for *like* and *dislike*. The rule $r_8$ is used to declare that $Agent_1$ tolerates $Agent_4$, i.e. $Agent_4$ possibly shares a table with $Agent_1$, and finally the rule $r_9$ means that $Agent_1$ does not want to share a table with 3 agents or more.

Observe that while the rule $r_3$ generates possible seating arrangements, the rules $r_2$, $r_4$ and $r_9$ discard those which are not allowed.

*Example 5.* **Room arrangement.** Consider a house having $m$ rooms. We have to distribute some objects (i.e. furniture and appliances) over the rooms without exceeding the maximum number of objects, say $c$, allowed per each room. Further constraints may be set w.r.t. the color and/or the type of objects in the same room, etc. Each object is represented by a single SOLP program encoding both the properties of the object and the constraints we want to meet. As an example consider the following program, representing an object *cupboard*:

$$
\begin{array}{ll}
r_1: & name(cupboard) \leftarrow \\
r_2: & type(furniture) \leftarrow \\
r_3: & color(yellow) \leftarrow \\
r_4: & \leftarrow at(R1), at(R2), R1 <> R2 \\
r_5: & at(R) \leftarrow [,2]\{at(R), type(appliance), not\ color(yellow), \\
 & \qquad [1,1]\{name(fridge)\}\}, room(R) \\
r_6: & at(R) \leftarrow [,c-3]\{at(R), type(furniture)\}, room(R) \\
r_7: & \leftarrow at(R), [1,]\{at(R), color(green)\}
\end{array}
$$

where the properties of the object are encoded as predicates representing the *name* (fridge, cupboard, table, etc.), the *type* (furniture or appliance), the *color* and so on. In particular, the rule $r_4$ states that an object may not be in more than one room, while by means of the rule $r_5$, we allow no more than 2 appliances to share the room with the cupboard, provided that they are not yellow and also that one of them is a fridge. The rule $r_6$, means that we want the cupboard to be in the same room with any other pieces of furniture, but no more than $c-3$, in order not to exceed the maximum number of objects. Finally, the rule $r_7$ states that the cupboard cannot share the room with any green object.

## 7 Conclusions

In this work we have proposed a new language, *SOcial Logic Programming* (SOLP), which enables social behavior among a community of agents represented by logic programs, extending COLP [5]. Thus, the intended models of the *Social Semantics* represent those mental states satisfying social requirements imposed by the agents. Moreover, we have given a translation from SOLP to logic programming with aggregates and discussed the computational complexity of SOLP, which has been proved to be NP complete.

## References


1. C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.
2. M. Bratman. *Intention, Plans, and Practical Reason.* Harvard Univ. Press, 1987.
3. M. Bratman, D. J. Israel, and M. E. Pollack. Plans and Resource-Bounded Practical Reason. AI center technical note SRI-AI 425R, SRI International, 1988.
4. F Buccafurri and G. Caminiti. A Social-Oriented Semantics for Multi-Agent Systems. Technical Report - TR Lab. Ing. Inf. 05/01, DIMET - University of Reggio Calabria, 2005. Available from the authors.

5. F. Buccafurri and G. Gottlob. *Multiagent Compromises, Joint Fixpoints, and Stable Models*, volume 2407 of *LNCS and LNAI*. Springer, 2002.
6. P. R. Cohen and H. Levesque. Rational interaction as the basis for communication. In *Intentions in Communication*. MIT Press, 1990.
7. R. S. Cost, T. Finin, and Y. Labrou. Coordinating Agents Using ACL Conversations. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 183–196, 2001.
8. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *IJCAI-03, Proc. of the 18th Int. Joint Conf. on Artificial Intelligence*, pages 847–852, 2003.
9. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem-solving in DLV. In *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer, 2000.
10. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *5th Conf. on Logic Programming*, pages 1070–1080. MIT Press, 1988.
11. V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent software agents. *TPLP*, 4(4):429–494, 2004.
12. J. Minker and D. Seipel. *Disjunctive Logic Programming: A Survey and Assessment*, volume 2407 of *LNCS and LNAI*. Springer, 2002.
13. R. S. Patil, R. E. Fikes, P. F. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The DARPA knowledge sharing effort: progress report. In *Principles of KR and Reasoning: Proc. of the 3rd Int. Conf.* Kaufmann, 1992.
14. W. van der Hoek and W. Wooldrige. Towards a logic of rational agency. *Logic Journal of the IGPL*, 11(2):135–159, 2003.
15. M. Wooldridge. *Reasoning about Rational Agents*. Intelligent Robots and Autonomous Agents. MIT Press, Cambridge, Massachusetts, 2000.
16. M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 2(10):115–152, 1995.

# SAT as an effective solving technology for constraint problems

**Preliminary report**

Marco Cadoli, Toni Mancini, Fabio Patrizi

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
cadoli|tmancini|patrizi@dis.uniroma1.it

**Abstract.** In this paper we make a preliminary investigatation on the use of SAT technology for solving constraint problems. In particular, we solve many instances of several common benchmark problems for CP with different SAT solvers, by exploiting the declarative modelling language NPSpec, and Spec2Sat, an application that allows to compile NPSpec specifications into SAT instances. Furthermore, we start investigating whether some reformulation techniques already used in CP are effective when using SAT as solving engine. We present preliminary but encouraging experimental results in this direction, showing that this approach can be appealing.

## 1 Introduction

Several benchmark problems have been proposed in the literature for testing the performance of Constraint Programming tools (cf., e.g., [21, 15]), spanning several areas, from combinatorics, to planning, scheduling, or problems on graphs.

There is also a great variety in the kind of solvers that are used for Constraint Programming: those based on backtracking (cf., e.g., [28]), mathematical programming (cf., e.g., [13]), answer sets and stable model semantics (cf., e.g., [10, 20]), which are typically complete, or those that rely on local-search techniques (cf., e.g., [18, 29]) which are intrinsically incomplete. In this paper we focus on a different kind of tools, i.e., solvers for Propositional Satisfiability (SAT), showing how they can be transparently and effectively used for solving constraint problems.

The intuition behind the usage of a SAT solver, is that every CSP can be reduced in polynomial time to an instance of SAT, since the complexity of solving a CSP is in NP, and SAT is one of the prototypical NP-complete problems. Actually, the latter aspect has led to a great interest and to a huge amount of research in the field of SAT solving (cf., e.g., the proceedings of the last SAT conferences), leading to the current availability of very efficient solvers that can deal with very large formulae. State-of-the-art SAT solvers include complete ones, such as zChaff [19], and incomplete ones, such as walksat [25],

and BG-WALKSAT [30]. For an up-to-date list, we refer the reader to the URLs `www.satlib.org` and `www.satlive.org`.

The availability of fast solvers for SAT has led a great interest in the CP research community, and many papers show how to translate (compile) into SAT instances of various problems, like, e.g., scheduling, planning, or combinatorial ones (cf., e.g., [8, 17, 11, 16]). However, the complexity of the translation task is a major obstacle, since the compilation strongly depends on the constraints of the problem to be solved. Nowadays, this task is typically made by problem-dependent programs hence, in practice, preventing SAT to be one of the actual solving technologies for Constraint Programming.

The availability of specification languages that compile problem instances into SAT formulae, e.g., the language NPSPEC and the SPEC2SAT system [2, 7], is an important step ahead, providing the user with the possibility of easily building specifications for new constraint problems in a purely declarative way, maintaining a strong independence from the instances.

In this paper, we present some preliminary experiments showing that SAT technology can be *effectively* used for solving CSPs, by using NPSPEC on several common benchmark problems for CP, experiencing different SAT solvers. The problems we focus on are a significant subset of those present in the benchmark repository CSPLib [15], very well-known in the CP research community. Problems in CSPLib are usually described only in natural language, and no formal specification is given for most of them. Hence, as a side-effect, our work also proposes declarative specifications (in the language NPSPEC) for such problems. For the future, we plan to strengthen our evaluation by adding more problems from CSPLib, run the same problems using other systems, including OPL [28] smodels [20], and dlv [10], and possibly adopting more SAT solvers.

In general, given a specification of a problem, several techniques have been proposed to reformulate it, in order to improve the solver efficiency, while maintaining equivalence (or at least, the possibility to efficiently reconstruct valid solutions to the original problem from solutions to the reformulated one). Such techniques include, e.g., adding new constraints to the model, such as symmetry-breaking ones, or the somewhat opposite strategy of ignoring some of the constraints that are guaranteed to be reinforced in a later stage (the so called *safe-delay constraints* [4]). We started experiencing the application of the above techniques while performing our experiments, and present some results. It is worth noting that these techniques are applied at the symbolic level of the specification, and hence independently on the instance. This possibility further increases the level of declarativeness of the modelling stage, which is fundamental in order to effectively take advantage of SAT technology.

The paper is organized as follows: in Section 2 we briefly illustrate the language NPSPEC and the SPEC2SAT program that, given a NPSPEC specification and an instance, compiles it into a SAT instance. In Section 3 we present the chosen benchmark problems, while in Section 4 we present and comment our experimental results. Finally, Section 5 concludes the paper.

## 2 The NPSpec language and Spec2Sat

NPSpec and Spec2Sat have been extensively described in [7]. Hence, in what follows, we just recall the syntax and the informal semantics of the modelling language, and the general architecture of the compiler.

The Home Page of the NPSpec project (`www.dis.uniroma1.it/~cadoli/research/projects/NP-SPEC/`) contains all the specifications proposed in this paper, as well as the program itself.

### 2.1 The NPSpec language

An NPSpec program consists of a `DATABASE` section and a `SPECIFICATION` section. The former includes the definition of the problem instance, in terms of extensional relations, and integer intervals and constants. The latter section instead, consists of the problem specification, that is divided into two parts: the declaration of a *search space*, and the definition of constraints that a point in the search space has to satisfy in order to be a solution to the problem instance. The declaration of the constraints is given by a stratified DATALOG program [1], which can include the six predefined relational operators and negative literals.

The full syntax of NPSpec is given in [7], hence here we just recall it with an example. In particular, we show an NPSpec program for the *Hamiltonian path* NP-complete problem [14, Prob. GT39, p. 199], i.e., the problem where the input is a graph and the question is whether a traversal exists that touches each node exactly once.

```
DATABASE
  n = 6;  // no. of nodes
  edge = {(1,2),(3,1),(2,3),(6,2),(5,6), (4,5),(3,5),(1,4),(4,1)};
SPECIFICATION
  Permutation({1..n},path).                          // H1
  fail <-- path(X,P), path(Y,P+1), NOT edge(X,Y). // H2
```

The following comments are in order:

- The input graph is defined in the `DATABASE` section, which is generally provided in a separate file.
- In the search space declaration (metarule `H1`) the user declares the predicate symbol `path` to be a "guessed" one, implicitly of arity 2. All other predicate symbols are, by default, not guessed. Being guessed means that we admit all extensions for the predicate, subject to the other constraints.
- `path` is declared to be a permutation of the finite domain {`1..n`}. This means that its extension must represent a permutation of order 6. As an example, $\{(1,5),(2,3),(3,6),(4,2),(5,1),(6,4)\}$ is a valid extension.
- Comments can be inserted using the symbol "`//`".
- Rule `H2` is the constraint that permutations must obey in order to be Hamiltonian paths: a permutation `fail`s, i.e., it is not valid, if two nodes `X` and `Y` which are adjacent in the permutation are not connected by an edge. `X` and `Y` are adjacent because they hold places `P` and `P+1` of the permutation, respectively.

Running this program on the NPSpec compiler produces the following output:

```
path: (1, 1) (2, 5) (3, 6) (4, 2) (5, 3) (6, 4)
```

which means "1 is the first node in the path, 4 is the second node in the path, ..., 3 is the sixth node in the path", and is indeed an Hamiltonian path.

The search space declaration, which corresponds to the definition of the domain of the guessed predicates, is, in general, a sequence of declarations of the form:

1. `Subset(<domain>, <pred_id>).`
2. `Permutation(<domain>, <pred_id>).`
3. `Partition(<domain>, <pred_id>, n).`
4. `IntFunc(<domain>, <pred_id>, min..max).`

We do not formally give further details of the NPSpec syntax, but, in the following sections, we present and comment several other examples. We just remark that the declarative style of programming in NPSpec is very similar to that of DATALOG, and it is therefore easy to extend programs for incorporating further constraints. As an example, the program for the Hamiltonian path can be extended to the Hamiltonian cycle problem [14, Prob. GT37, p. 199] by adding the following rule

```
fail <-- path(X,n), path(Y,1), NOT edge(X,Y).     // H3
```

Moreover, undirected graphs can be handled by including a further literal `NOT edge(Y,X)` in the body of both rules `H2` and `H3`.

Concerning syntax, we remark that NPSpec offers also useful SQL-style aggregates, such as `SUM`, `COUNT`, `MIN`, and `MAX`. Several examples of Section 3 use such operators.

### 2.2 The NPSpec to SAT compiler

Spec2Sat is an application that allows the compilation of a NPSpec specification (when given together with input data) into a SAT instance. We do not give here the technical details of the compilation task, that can be found in [7], but just briefly describe the general architecture of the application, shown in Figure 1.

The module Parser receives text files containing the specification $S$ in NPSpec and the instance data $I$, parses them, and builds its internal representation. The module Spec2Sat compiles $S \cup I$ into a CNF formula $T$ in DIMACS format, and builds an object representing a dictionary which makes a 1-1 correspondence between ground atoms of the Herbrand base of $S \cup I$ and propositional variables of the vocabulary. The file in DIMACS format is given as an input to a SAT solver (the choice of the SAT solver is completely independent from the application, as long as it accepts the standard DIMACS format as input, and can be chosen by the user), which delivers either a model of $T$, if satisfiable, or

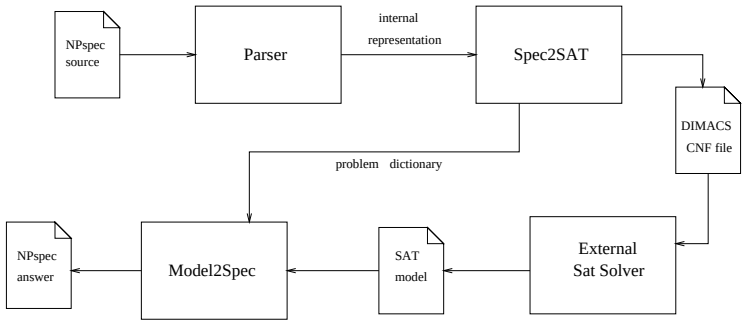



**Fig. 1.** Architecture of the NPSpec compilation and execution environment.

the indication that it is unsatisfiable. At this point, the Model2Spec module performs, using the dictionary, a backward translation of the model (if found) into the original language of the specification. Appropriate interfaces for different SAT solvers and the Model2Spec module have, of course, been written, in order to translate the (proprietary) format for the output, into Model2Spec.

## 3 Benchmark problems

As mentioned in Section 1, in this paper we show our preliminary experiments in solving typical Constraint Programming benchmark problems using SAT technology, by taking advantage of NPSpec. In this section, we list the problems used for our experiments, that are a significant subset of those present in the well-known library CSPLib [15]. For the experiments of this paper we chose seven problems which cover five out of the seven classes of problems offered by the CSPLib. More specifications can be found on-line. As a side-effect, we obtain declarative specifications (in the language NPSpec) for such problems. The number in parentheses close to each problem is its identification number in the library:

*Golomb ruler (problem nr. 006).* A Golomb ruler of length $L$ with $m$ marks is defined as a set of $m$ integers $0 = a_1 < a_2 < ... < a_m = L$ such that the $m(m-1)/2$ differences $a_j - a_i, 1 \leq i < j \leq m$ are all distinct. In our formulation, given $L$ and $m$, we are interested in finding a Golomb ruler of any length *not greater than* $L$.

An NPSpec specification for this problem is as follows:

```
IntFunc({1..N_MARKS}, ruler, 0..LENGTH).           // G1

fail <-- NOT ruler(1,0).                           // G2
fail <-- ruler(I,V_I), ruler(J,V_J), J > I,        // G3
         V_I >= V_J.
fail <-- ruler(I,V_I), ruler(J,V_J), ruler(K,V_K), // G4
```

```
        ruler(L,V_L), I < J, K < L,
        I != K, V_J - V_I == V_L - V_K.
fail <-- ruler(I,V_I), ruler(J,V_J), ruler(K,V_K), // G5
        ruler(L,V_L), I < J, K < L,
        J != L, V_J - V_I == V_L - V_K.
```

The search space is defined (metarule `G1`) as the set of all total integer functions from {`1..N_MARKS`} to the integer range {`0..LENGTH`}, hence assigning a position on the ruler to each mark. Rule `G2` forces the first mark to be at the beginning of the ruler (position 0). Rule `G3` forces marks to be in ascending order, i.e., the second mark is on the right of the first one, the third on the right of the second, and so on. Rules `G4` and `G5` force distances between two marks to be all different.

*All-interval series (problem nr. 007).* Given the twelve standard pitch-classes (c, c#, d, ...), represented by numbers 0,1,...,11, find a series in which each pitch-class occurs exactly once and in which the musical intervals between neighboring notes cover the full set of intervals from the minor second (1 semitone) to the major seventh (11 semitones). Here, we generalize the problem by replacing the twelve standard pitch-classes with an arbitrary set `pitch` of $N$ pitch-classes (all-interval series problem of size $N$).

An ad-hoc encoding of this problem into SAT has been made in [16]. In NPSpec, the All-interval series problem can be specified as follows:

```
Permutation(pitch, series).             // A1
Permutation(interval, neighbor).        // A2

fail <-- series(P,X), series(Q, X + 1), // A3
         NOT neighbor(abs(P - Q), X).
```

By metarule `A1`, guessed predicate `series` assigns a different order number to every pitch in the series. Metarule `A2` guesses a second guessed predicate, `neighbor`, to be an ordering of all possible intervals (`interval` is defined in the instance file to be the integer range $[1, N-1]$): a tuple $\langle intv, idx \rangle$ in `neighbor` means that pitches at positions $idx$ and $idx+1$ in the series are divided by interval $intv$. The final rule `A3` actually forces `series` to respect this constraint: for each pair of adjacent pitches `P` and `Q` (at positions `X` and `X+1`), the interval dividing them must have order `X` in the permutation `neighbor`.

*Social golfer (problem nr. 010).* Given `N_PLAYERS` = `N_GROUPS` $*$ `GROUP_SIZE` golf players, this problem amounts to find an arrangement for all of them into `N_GROUPS` groups of size `GROUP_SIZE` over `N_WEEKS` weeks, in such a way that no player plays in the same group as any other in more than one week.

```
IntFunc({1..N_PLAYERS}><{1..N_WEEKS}, play, 1..N_GROUPS).

fail <-- play(P1,W1,G1), play(P2,W1,G1), P1 != P2,
         play(P1,W2,G2), play(P2,W2,G2), W1 != W2.
fail <-- COUNT(play(*,W,G),X), X != GROUP_SIZE.
```

*Schur's lemma (problem nr. 015).* The problem is to put `N_BALLS` balls labelled $\{1, ..., \texttt{N\_BALLS}\}$ into `N_BOXES` boxes so that for any triple of balls $(x, y, z)$ with $x + y = z$, not all are in the same box.

An NPSpec specification for this problem is as follows:

```
Partition({1..N_BALLS}, putIn, N_BOXES).          // S1

fail <-- putIn(X,Box), putIn(Y,Box), putIn(Z,Box), // S2
         X + Y == Z.
```

Metarule `S1` declares the search space to be the set of all partitions of the set of `N_BALLS` balls into `N_BOXES` boxes, while rule `S2` expresses the constraint.

*Ramsey problem (problem nr. 017).* The Ramsey problem is to color the edges of a complete graph with $n$ nodes using at most $k$ colors, in such a way that there is no monochromatic triangle in the graph. A specification is as follows:

```
Partition({1..N_EDGES}, coloring, N_COLORS).     // R1

fail <-- edge(X,A,B), edge(Y,B,C), edge(Z,A,C),  // R2
         coloring(X,Col), coloring(Y,Col),
         coloring(Z,Col).
```

*Magic square (problem nr. 019).* A magic square of order $N$ is an $N$ by $N$ matrix containing the numbers 1 to $N^2$, with numbers on each row, column and main diagonal having the same sum.

An NPSpec specification for this problem is as follows:

```
  Permutation({1..N}><{1..N}, square).                              // M1
  Permutation({1..N^2}, magic_square).                              // M2

  Subset({1..N^2}, diag).                                           // M3
  Partition({1..N^2}, column, N).                                   // M4
  Partition({1..N^2}, row, N).                                      // M5

  // Channeling constraints for "square"
  square(X,Y,I) <-- I == (X -1)*N + Y.                              // M6
  fail <-- square(X,Y,I), I != (X -1)*N + Y.                        // M7

  // Channeling constraints for "diag"
  fail <-- NOT diag(X), magic_square(X,I), square(R,C,I), R == C. // M8
  fail <-- diag(X), magic_square(X,I), square(R,C,I), R != C.     // M9

  // Channeling constraints for "column"
  fail <-- NOT column(V,C-1), magic_square(V,I), square(_,C,I).   // M10

  // Channeling constraints for "row"
  fail <-- NOT row(V,R-1), magic_square(V,I), square(R,_,I).      // M11
```

```
fail <-- SUM(column(*,I),C:0..MAX_SUM),                          // M12
         SUM(diag(*),D:0..MAX_SUM), C != D.
fail <-- SUM(row(*,I),C:0..MAX_SUM),                             // M13
         SUM(diag(*),D:0..MAX_SUM), C != D.
```

Metarule `M1`, together with rules `M6` and `M7`, defines `square` to be an enumeration of all the entries of the magic square (each of them having coordinates in $[1, N] \times [1, N]$). Hence, each entry is assigned an index in $[1, N^2]$. Metarule `M2` then guesses a value in $[1, N^2]$ for each entry. Such values are all different.

The other guessed predicates (rules `M3`, ..., `M5`) are redundant, and represent a more convenient representation of the values in the main diagonal, in each row and column respectively. Rules `M8`, ..., `M11` define them starting from `magic_square`, acting as channeling constraints.

Finally, rules `M12` and `M13` force entries to be such that the sum of values in the main diagonal, every row and every column are all equal. `MAX_SUM` is an instance-dependent value representing the maximum value for the sums.

*Langford's number (problem nr. 024).* The $L(k, n)$ problem is to arrange $k$ sets of numbers 1 to $n$, so that each appearance of the number $m$ is $m$ numbers on from the last. As an example, the $L(3, 9)$ problem is to arrange 3 sets of the numbers 1 to 9 so that the first two 1's and the second two 1's appear one number apart, the first two 2's and the second two 2's appear two numbers apart, etc. A specification is as follows:

```
IntFunc({1..NUMBERS*SETS}, sequence, 1..NUMBERS). // L1

fail <-- COUNT(sequence(*,_),X), X != SETS.       // L2
fail <-- sequence(I,N), NOT sequence(I+N+1,N),    // L3
         sequence(J,N), J > I.
```

Metarule `L1` declares the guessed predicate `sequence` to be a function from `{1..NUMBERS*SETS}` to `1..NUMBERS`. Rule `L2` forces each number to appear exactly `SETS` times in the sequence, while `L3` checks whether the distances between two occurrences of the same number are correct.

*Balanced academic curriculum problem (problem nr. 030).* The Balanced academic curriculum problem (BACP) amounts to design a balanced academic curriculum by assigning periods to courses in a way that the academic load of each period is balanced, i.e., as similar as possible. The curriculum must obey several administrative and academic regulations, such that the respect of prerequisites among courses, minimum and maximum number of courses per periods, as well as minimum and maximum academic load in each period.

```
Partition(load, curriculum, PERIODS).

fail <-- SUM(curriculum(?,*,P), X:0..MAX_LOAD),
         period(P,M,_,_,_), X < M.
fail <-- SUM(curriculum(?,*,P), X:0..MAX_LOAD),
```

```
         period(P,_,M,_,_), X > M.
fail <-- COUNT(curriculum(?,*,P), X:0..MAX_LOAD),
         period(P,_,_,M,_), X < M.
fail <-- COUNT(curriculum(?,*,P), X:0..MAX_LOAD),
         period(P,_,_,_,M), X > M.
fail <-- prerequisite(Pre, Post), curriculum(Pre,_,P),
         curriculum(Post,_,Q), Q <= P.
```

## 4 Experiments

We chose a non-trivial set of instances for each problem defined above, by using publicly available benchmarks when possible (e.g., for BACP), and compiled all such instances into SAT ones. Then, we ran different SAT solvers on those instances, and measured their solving times. In this preliminary stage, we used two recent SAT solvers, very different in nature:

- zChaff, one of the fastest, complete solvers today available;
- BG-WALKSAT, a sound but incomplete one, based on local search. BG-WALKSAT is a recent extension of the well-known WALKSAT, where the search is guided by *backbones* of the formula.

Furthermore, as already mentioned in Section 1, we started investigating whether the application of different reformulation techniques was suitable for improving solvers' performances. In particular, we applied two different, and in some sense, complementary, techniques: adding symmetry-breaking constraints and neglecting *safe-delay* constraints. These techniques are briefly described in the following:

*Symmetry-breaking.* The presence of symmetries in CSPs has been widely recognized to be one of the major obstacles for their efficient resolution. To this end, different approaches have been followed in the literature in order to deal with them. The most well known one is to add additional constraints to the CSP model, that filter out many (hopefully all but one) of the symmetrical points in the search space. These are called symmetry-breaking constraints, cf., e.g., [23, 9, 24, 26, 27, 12].

We used this approach with a major difference, adding symmetry-breaking constraints at the level of the problem specification. Hence, we broke "structural" symmetries of the problems, i.e., those symmetries that depend on the problem structure, and not on the particular instance considered. Breaking symmetries at the specification level has been proved to be effective for different classes of solvers, on different problems [3], and comes natural when using a purely declarative modelling language such as NPSpec.

*Safe-delay constraints.* Given a problem specification, a safe-delay constraint is a constraint whose evaluation can be safely ignored in a first step, hence simplifying the problem, and efficiently reinforced in a second step, when a solution to the relaxed problem has been found [4]. The importance of safe-delay constraints

is that their reinforcement can always be done in polynomial time, without further search. Highlighting (and delaying) safe-delay constraints can be very useful when solving constraint problems, at least for three reasons: *(i)* For every instance, the set of solutions is enlarged (since some constraints are removed), and this can be beneficial for some classes of solvers. *(ii)* The instantiation stage can be done more efficiently, since a fewer number of constraints have to be grounded: this is the case also when using SAT technology, since delaying constraints reduces the number of clauses generated during instantiation. *(iii)* The reinforcement of delayed constraints (which is guaranteed to be polynomial time) is often very efficient, e.g., linear of logarithmic time in the size of the input, (we show some examples in Section 4). It is worth noting that also the deletion of safe-delay constraints is done by reformulating the declarative specification of the problem, hence independently on the instance.

For the various problems of Section 3, we used the following instances:

- Golomb ruler: lengths up to 15, with up to 9 marks when using zChaff, and lengths up to 12, with up to 5 marks when using bg-walksat;
- All-interval series: pitch classes up to 18 (zChaff) and up to 40 (bg-walksat);
- Social golfer: up to 8 players, 6 weeks;
- Schur's lemma: up to 50 balls and 10 boxes;
- Ramsey problem: up to 20 nodes and 7 colors (zChaff), and 12 nodes and 5 colors (bg-walksat);
- Magic squares: sizes up to 5, when using zChaff, and 4 when using bg-walksat;
- Langford's number: up to 4 sets and 19 numbers (zChaff), and 4 sets and 10 numbers (bg-walksat);
- BACP: 2 benchmark instances, taken from CSPLib, solved with zChaff.

Results of our experiments are shown in Table 1 (both compilation and solving processes had a timeout of 1 hour). In particular, Table 1(a) shows compilation and solving times for zChaff, while Table 1(b) shows times when using bg-walksat on the same problems.

For each solver and problem, we report the number of instances run, and the number of those solved successfully in 1 hour. As for the incomplete bg-walksat instead, we report its success ratio, i.e., the percentage of the instances for which this solver gave the correct answer. Then, we list the overall times for compiling and solving the whole set of instances for each problem.

Moreover, we investigate the effectiveness of the reformulation techniques discussed above. In particular, we ignored safe-delay constraints on the following problems:

- Golomb ruler: rule `G3` of the problem specification can be ignored, hence enlarging the set of solutions by all their permutations. However, in this case, a simple modification of the other constraints is required [4]): in particular, the *absolute values* of distances between marks have to be different.

| Problem name | zChaff | | | | | |
|---|---|---|---|---|---|---|
| | Instances | | | SAT compil | SAT solving | Total |
| | nr. | solved | unsolved | time (sec) | time (sec) | time (sec) |
| Golomb Ruler | 34 | 34 | 0 | 39412.96 | 2.46 | 39415.42 |
| with safe delay | 34 | 34 | 0 | 26654.29 | 27.66 | 26681.95 |
| All-Interval Series | 14 | 13 | 1 | 6.29 | 6600.70 | 6606.99 |
| Social Golfer | 168 | 110 | 58 | 64467.93 | 212527.78 | 276995.71 |
| with symm breaking | 168 | 162 | 6 | 62774.72 | 3782.73 | 66567.45 |
| Schur's Lemma | 164 | 164 | 0 | 2412.57 | 0.08 | 2412.65 |
| with safe delay | 164 | 164 | 0 | 2510.13 | 0.12 | 2510.12 |
| with symm breaking | 164 | 164 | 0 | 2537.14 | 0.08 | 2537.22 |
| Ramsey problem | 85 | 82 | 3 | 155.24 | 10803.04 | 10958.28 |
| with safe delay | 85 | 82 | 3 | 153.95 | 10802.61 | 10956.56 |
| with symm breaking | 85 | 82 | 3 | 9099.76 | 484.017 | 9583.78 |
| Magic Square | 3 | 3 | 0 | 281.16 | 128.59 | 409.75 |
| with symm breaking | 3 | 3 | 0 | 282.03 | 38.25 | 320.28 |
| Langford's number | 43 | 41 | 2 | 1982.14 | 18109.22 | 20091.36 |
| BACP | 2 | 2 | 0 | 2798.85 | 2.20 | 2801.05 |

(a)

| Problem name | bg-walksat | | | | |
|---|---|---|---|---|---|
| | Instances | | SAT compil | SAT solving | Total |
| | nr. | success ratio | time (sec) | time (sec) | time (sec) |
| Golomb Ruler | 20 | 100% | 15274.17 | 3528.55 | 18802.72 |
| with safe delay | 20 | 60% | 7617.11 | 6315.08 | 13932.19 |
| All-Interval Series | 36 | 17% | 171.21 | 702.98 | 874.19 |
| Social Golfer | 137 | 43% | 16453.92 | 3633.48 | 20087.40 |
| with symm breaking | 137 | 46% | 17132.50 | 3792.73 | 20925.23 |
| Schur's Lemma | 164 | 100% | 2412.57 | 4.32 | 2416.89 |
| with safe delay | 164 | 99% | 2510.00 | 4.18 | 2514.18 |
| with symm breaking | 164 | 100% | 2537.14 | 7.03 | 2544.17 |
| Ramsey problem | 85 | 94% | 155.24 | 8.47 | 163.71 |
| with safe delay | 85 | 100% | 153.95 | 7.49 | 161.44 |
| with symm breaking | 85 | 94% | 154.64 | 8.05 | 162.69 |
| Magic Square | 3 | 33% | 281.16 | 31.97 | 313.13 |
| with symm breaking | 3 | 33% | 282.03 | 32.07 | 314.10 |
| Langford's number | 36 | 67% | 813.64 | 355.53 | 1169.17 |

(b)

**Table 1.** Results of the experiments using zChaff (a) and bg-walksat (b) for solving different CSPs.

- Schur's lemma: we let balls to be put in more than one box at the same time. If such a solution exists, a valid solution of the original problem can be derived by arbitrarily choosing a single box for each ball.
- Ramsey problem: we let multiple colors to be assigned to the same node. If such a coloring exists, then it suffices to arbitrarily choose an arbitrary color for each node having multiple ones.

We note that, in the last two cases, ignoring safe-delay constraints reduces to guess multi-valued functions for the guessed predicates. This task can be accomplished by the current implementation of Spec2Sat by defining a guessed predicate as a `multivalued` partition or integer function. We also observe that for all three problems, the second stage, i.e., recovering a solution to the original problem from a solution to the simplified one, can be performed very efficiently: in $m \log m$ for Golomb ruler (by ordering marks), and in linear time for both

Ramsey and Schur's lemma problems (we remind that the reinforcement of safe-delay constraints is always guaranteed to be polynomial).

As for symmetry-breaking instead, we broke some of the symmetries in the Social golfer, Schur's lemma, Magic square, and Ramsey problems. In particular, as for Social golfer, we fixed the scheduling for the first two weeks, and made the first player play always in the first group (these symmetry-breaking constraints are shown in [27]). As for Schur's lemma and Magic square, we simply fixed the choice for the first edge and the first square, respectively. As for Ramsey instead, a slighly more complex symmetry-breaking constraint is added, by fixing a suitable ordering on the colors and on the edges.

Some comments on the results in Table 1 are in order. First of all, both SAT solvers behave well in many cases, being able to solve instances of reasonable size. However, it is not the case that one solver is always better than the other: zChaff seems much faster than bg-walksat for solving Golomb ruler or BACP instances (bg-walksat was not able to run on instances of the latter problem, due to their large size), while the latter is better for All-intervals series, Ramsey, and Social golfer, even if its success ratio (i.e., the ratio of satisfiable instances for which this incomplete solver was actually able to find a solution) is not always very high.

Interestingly, applying the two reformulation techniques sometimes greatly helps zChaff. As an example, by ignoring safe-delay constraints on Golomb Ruler, the overall compilation time falls down of about 13000 seconds, while the solving time increases only of about 25 seconds. A similar behavior happens also when solving this problem with bg-walksat.

zChaff is also positively affected by symmetry-breaking. As for Magic square, Ramsey, and Social golfer, the speed-up is impressive. It is interesting to observe that the compilation times do not grow significantly, since the symmetry-breaking constraints we chose are quite simple (apart for Ramsey, where a more complex symmetry-breaking constraint was used). It is worth noting that adding more complex constraints to these problems led to poorer performances for both Spec2Sat and zChaff. Another interesting aspect is that the local search solver bg-walksat does not take benefit from symmetry-breaking, hence confirming the intuition that a reduction of the solution density is an obstacle for local search (cf., e.g., [22, 3]).

## 5 Conclusions and future work

In this paper, we discussed how the availability of specification languages for constraint problems that automatically compile instances into SAT, can make SAT solving technology an effective tool for Constraint Programing. We experienced NPSpec and Spec2Sat on several well-known benchmark problems for CP, and demonstrated how they can be easily formulated in NPSpec, and solved by exploiting state-of-the-art SAT solvers. Additionally, we showed how applying reformulation techniques such as ignoring safe-delay constraints or adding

symmetry-breaking constraints to the problem specifications can be very effective in improving solvers' performances, and/or easing the compilation task.

It is worth noting that such techniques have already been used in ad-hoc SAT encodings of particular CP problems. As an example, the standard DIMACS SAT encoding of Graph $k$-coloring, actually omits the "at-most-one color" constraint of the problem, that is actually safe-delay [4]. The same happens in the SAT encoding for Job shop scheduling given in [8], where propositional variables represent the encoding of *earliest starting times* and *latest ending times* for all tasks, rather than their exact scheduled times. As for symmetry-breaking, ad-hoc SAT encodings of various problems usually take care of breaking (some of the) symmetries in the generated SAT instances.

The main difference of our work with respect to the others, is that the user can perform such optimizations at the symbolic level of the specification, by relying on a purely declarative language such as NPSpec. Right now, this is done manually by the NPSpec modeller, however, in some previous work, we showed how recognizing whether a reformulation technique can be applied, can be in principle autonomously made by system, since it reduces to check properties of first-order logic formulae [4, 3, 6]. Of course, since the goodness of a particular optimization technique is expected to depend both on the problem and on the solver (cf. results in Table 1), a much wider experimentation is needed, using a larger number of solvers on a wider set of problems. A deeper investigation of the applicability of other techniques to the SAT case (cf., e.g., [5]) is also planned.

Another important point that lacks in the current preliminary version of this work is the comparison of SAT with respect to CP solvers for the investigated problems. To this end, we plan to make a much wider experimentation that involves also state-of-the-art CP solvers, e.g., among the others, the well known OPL language [28], by Ilog.

## References


1. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 89–142. Morgan Kaufmann, Los Altos, 1988.
2. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *Computer Languages*, 26:165–195, 2000.
3. M. Cadoli and T. Mancini. Detecting and breaking symmetries on specifications. In *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems, in conjunction with the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, Kinsale, Ireland, 2003.
4. M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. In *Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2004)*, pages 388–398, Whistler, BC, Canada, 2004. AAAI Press/The MIT Press.
5. M. Cadoli and T. Mancini. Exploiting functional dependencies in declarative problem specifications. In *Proceedings of the Ninth European Conference on Logics in*

*Artificial Intelligence (JELIA 2004)*, volume 3229 of *Lecture Notes in Artificial Intelligence*, Lisbon, Portugal, 2004. Springer.

6. M. Cadoli and T. Mancini. Using a theorem prover for reasoning on constraint problems. In *Proceedings of the Third International Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation, in conjunction with the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, Toronto, Canada, 2004.
7. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162:89–120, 2005.
8. J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 1092–1097, Seattle, WA, USA, 1994. AAAI Press/The MIT Press.
9. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159, Cambridge, MA, USA, 1996. Morgan Kaufmann, Los Altos.
10. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In *Proceedings of the Sixth International Conference on the Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417, Trento, Italy, 1998. Morgan Kaufmann, Los Altos.
11. M. Ernst, T. Millstein, and D. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 1169–1177, Nagoya, Japan, 1997. Morgan Kaufmann, Los Altos.
12. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *Lecture Notes in Computer Science*, page 462 ff., Ithaca, NY, USA, 2002. Springer.
13. R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming*. International Thomson Publishing, 1993.
14. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, CA, USA, 1979.
15. I. P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, APES-09-1999, 1999. Available from `http://www.csplib.org`.
16. H. H. Hoos. *Stochastic Local Search - Methods, Models, Applications*. PhD thesis, TU Darmstadt, 1998.
17. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, pages 1194–1201, Portland, OR, USA, 1996. AAAI Press/The MIT Press.
18. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5(1):43–84, 2000.
19. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the Thirtyeighth Conference on Design Automation (DAC 2001)*, pages 530–535, Las Vegas, NV, USA, 2001. ACM Press.
20. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

21. OR Library. Available at `www.ms.ic.ac.uk/info.html`.
22. S. Prestwich. Supersymmetric modeling for local search. In *Proceedings of the Second International Workshop on Symmetry in Constraint Satisfaction Problems, in conjunction with the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, Ithaca, NY, USA, 2002.
23. J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In H. J. Komorowski and Z. W. Ras, editors, *Proceedings of the Seventh International Symposium on Methodologies for Intelligent Systems (ISMIS'93)*, volume 689 of *Lecture Notes in Computer Science*, pages 350–361, Trondheim, Norway, 1993. Springer.
24. E. Rothberg. Using cuts to remove symmetry. In *Proceedings of the Seventeenth International Symposium on Mathematical Programming (ISMP 2000)*, Atlanta, GA, USA, 2000.
25. B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In M. Trick and D. S. Johnson, editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, Providence, RI, USA, 1993.
26. H. D. Sherali and J. Cole Smith. Improving discrete model representations via symmetry considerations. *Management Science*, 47:1396–1407, 2001.
27. B. Smith. Reducing symmetry in a combinatorial design problem. In *Proceedings of the Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2001)*, pages 351–360, Ashford, Kent, UK, 2001.
28. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
29. P. Van Hentenryck and L. Michel. Control abstractions for local search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *Lecture Notes in Computer Science*, pages 65–80, Kinsale, Ireland, 2003. Springer.
30. W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 1179–1186, Acapulco, Mexico, 2003. Morgan Kaufmann, Los Altos.

# Enhancing computational power: DALI child agents generation*

Stefania Costantini Arianna Tocchio

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
{stefcost,tocchio}@di.univaq.it

**Abstract.** In this paper we introduce a novel feature of the DALI language: a DALI agent is now able to activate child agents and to feed them either with a goal to be reached or a result to be obtained. Each child agent is independent and can communicate with its father or with other agents. When the child finally reaches the given goal, it notifies the father. At any point, the latter may possibly decide to stop it. Then: (i) each child is aware of the identity of its father; (ii) each child will notify the father about its achievements; (iii) a child can be stopped by the father; (iv) the father may set a limited amount of time for children's activities completion. We introduce the mechanisms for children generation and the corresponding operational semantics, and then present an example.

## 1 Introduction

Intelligent agents by using their potentialities are able at least to some extent to overcome problems such as limited computational resources, non-deterministic environment, and insufficient knowledge. When a problem is not naturally multi-agent based, a sole agent is capable of solving it by taking enough computational resources and information about its environment.

In a lot of problem domains however, the context naturally requires several agents to take a role in problem-solving or, more generally, requires the adoption of a multi-agent strategy. A multi-agent system is composed of multiple interacting agents which are typically capable of cooperating to solve problems that are beyond the capabilities of any individual agent. Building a cooperation strategy is not easy: an agent, contrary to an object, can renounce to cooperate or, as emphasized in [5], can reveal itself an unreliable collaborator.

So, when an agent accepts the aid of another one, it implicitly assumes a certain risk degree on its future activity. Can an agent minimize this risk? In some cases the response to this query is 'yes'. Some kind of problems requiring a certain degree of computational power that a single agent cannot provide can be faced not by invoking the collaboration of external agents, but by generating child agents.

* We acknowledge support by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

The difference is relevant: a child agent is reliable and cannot refuse to give assistance. In fact, the basic premise of coordination is that if an agent cannot solve an assigned problem using local resources/expertise, it will decompose the problem into sub-problems and try to find other willing agents with the necessary resources/expertise to solve these sub-problems. By using child agents, the sub-problems assignment is solved by a simple message exchange between father and children without adopting a contracting mechanism. Moreover, the possibility to assign complex tasks to one or more child agents allows the father to keep its energies for more strategic activities. In particular a father agent, by delegating a time-expensive jobs to a child, can maintain a high reactivity degree and respond timely to the changes in the environment. This is a not negligible detail. A limit of this approach is that a child agent cannot resolve tasks that require a knowledge degree that the father agent does not posses, unless the child acquires knowledge autonomously from other external sources.

According to the above considerations, we have introduced in the DALI framework the ability to generate children. An important motivation for this improvement has been the need for our agents to face not-trivial planning problems by means of the invocation of a performant planner, such as for instance an Answer Set solver [7]. The idea of Answer Set Programming [20] is to represent a given computational problem by means of a logic program whose answer sets correspond to solutions and then use an answer set solver, e.g., SMODELS or DLV, to find an answer set for this program. Answer Set Programming has proved to be a strong formalism for planning [12], and thus appears suitable for an integration with DALI. As a planning process can require a significant amount of time to find a solution, the possibility for an agent to assign this time-expensive activity to its children can constitute a real advantage.

Another motivation for generating children is, more generally, that of splitting an agent goal into subgoals to be delegated to children. This possibly with the aim of obtaining different results by means of different strategies, and then comparing the various alternatives and choosing the best ones. The father provides the child with all the information useful to find the solution and, optionally, with an amount of time within which to resolve the assigned problem.

In this paper, we present the details on the child generation capability of DALI agents while the current work to integrate DALI and Answer Set Programming will be presented in forthcoming papers. This paper is organized as follows: in Section 2 we introduce the main functionalities of the DALI language; in Section 3 we explain briefly the DALI communication architecture; in Section 4 we present the Operational Semantics of our language; Section 5 is reserved to outline the child generation mechanism of DALI agents, Section 6 presents the related operational semantics laws, Section 7 shows an example of application. Finally, we conclude this paper with some remarks and discussion of related work.

## 2 The DALI language

DALI [3] is an Active Logic Programming language designed in the line of [10] for executable specification of logical agents. The reactive and proactive behavior of the

DALI agent is triggered by several kinds of events: external events, internal, present and past events. All the events and actions are timestamped, so as to record when they occurred.

An external event is a particular stimulus perceived by the agent from the environment. In fact, if we define $S = \{s_1 : t_0, ..., s_n : t_k\}$ as the set of external stimuli $s_k$ that the agent received from the world during the interval $(t_0, t_k)$, where the set of "external events" $E$ is a subset of $S$. In particular, we can define the set of external events as follows:

**Definition 1 (Set of External Events).** *We define the set of external events perceived by the agent from time $t_1$ to time $t_n$ as a set $E = \{e_1 : t_1, ..., e_n : t_n\}$ where $E \subseteq S$.*

A single external event $e_i$ is an atom indicated with a particular postfix in order to be distinguished from other DALI language events. More precisely:

**Definition 2 (External Event).** *An external event is syntactically indicated by postfix $E$ and it is defined as:*
$ExtEvent ::= << Atom_E >> | seq << Atom_E >>$ *where an* Atom *is a predicate symbol applied to a sequence of* terms *and a* term *is either a constant or a variable or a function symbol applied in turn to a sequence of terms.*

External events allow an agent to react through a particular kind of rules, reactive rules, aimed at interacting with the external environment. When an event comes into the agent from its "external world", the agent can perceive it and decide to react. The reaction is defined by a reactive rule which has in its head that external event. The special token $:>$, used instead of $:-$, indicates that reactive rules performs forward reasoning.

**Definition 3 (Reactive rule).** *A reactive rule has the form: $ExtEvent_E :> Body$ or $ExtEvent_{1E}, ..., ExtEvent_{nE} :> Body$*
*where $Body ::= seq << Obj >>$ and*
$Obj ::= << Action_A >> | << Goals_G >> | << Atom >> | ...$

The agent remembers to have reacted by converting the external event into a *past event* (time-stamped). Operationally, if an incoming external event is recognized, i.e., corresponds to the head of a reactive rule, it is added into a list called $EV$ and consumed according to the arrival order, unless priorities are specified.

The internal events define a kind of "individuality" of a DALI agent, making it proactive independently of the environment, of the user and of the other agents, and allowing it to manipulate and revise its knowledge. More precisely:

**Definition 4 (Internal Event).** *An internal event is syntactically indicated by postfix $I$:* $InternalEvent ::= << Atom_I >>$
*The structure of an internal event is composed by two rules. The first one contains the conditions (knowledge, past events, procedures, etc.) that must be true so that the reaction (in the second rule) may happen:*
$IntEvent : -Conditions$
$IntEvent_I :> Body$

*where* $Conditions ::= seq << Obj\_cond >>$ *and*
$Obj\_cond ::= << PastEvent_P >> | << Atom >> | << Belief >> |...$
*Moreover,*
$Body ::= seq << Obj\_body >>$ *and*
$Obj\_body ::= << Action_A >> | << Goals_G >> | << Atom >> |...$

Internal events are automatically attempted with a default frequency customizable by means of directives in the initialization file. The user's directives can tune several parameters: at which frequency the agent must attempt the internal events; how many times an agent must react to the internal event (forever, once, twice,...) and when (forever, when triggering conditions occur, ...); how long the event must be attempted (until some time, until some terminating conditions, forever).

When an agent perceives an event from the "external world", it does not necessarily react to it immediately: it has the possibility of reasoning about the event, before (or instead of) triggering a reaction. Reasoning also allows a proactive behavior. In this situation, the event is called present event and is formalized as follows:

**Definition 5 (Present Event).** *A present event is syntactically indicated by postfix* $N$*:*
$PresentEvent ::= << Atom_N >> | seq << Atom_N >>$
*The syntax of a present event usage is:*
$InternalEvent : -PresentEvent_N$
$InternalEvent_I :> Body$
*where* $Body ::= seq << Object >>$ *and*
$Object ::= << Action_A >> | << Goals_G >> | << Atom >> |...$

Actions are the agent's way of affecting the environment, possibly in reaction to either an external or internal event. An action in DALI can be also a message sent by an agent to another one.

**Definition 6 (Action).** *An action is syntactically indicated by postfix* $A$*:*
$Action ::= << Atom_A >> | message_A << Atom, Atom >>$
*Actions take place in the body of rules:*
$Head : -Body$
*where* $Body ::= seq << Object >>$ *and*
$Object ::= << Action_A >> | << Goals_G >> | << Atom >> |...$

In DALI, actions may have or not preconditions: in the former case, the actions are defined by actions rules, in the latter case they are just action atoms. An action rule is just a plain rule, but in order to emphasize that it is related to an action, we have introduced the new token $:<$, thus adopting the following syntax:

**Definition 7 (Action rule).** *An action rule has the form:*
$Action :< Preconditions$
*where* $Preconditions ::= seq << Object >>$ *and*
$Object ::= << PastEvent_P >> | << Atom >> | << Belief >> |...$

Similarly to external and internal events, actions are recorded as past actions.

A DALI agent is able to build a plan in order to reach an objective, by using internal events of a particular kind, called *planning goals*. A goal has postfix $G$, and like an internal event is defined by two rules. The first one is attempted when the goal is invoked and activates its subgoals, if any. The second one contains a reaction related to the reached subgoal. The relevant difference between an internal event and a planning goal is that while the former starts being attempted when the agent is born, the latter is attempted when invoked by a rule. A DALI agent is also able to verify if a goal was reached by using a special kind of atom with a postfix $T$. When the interpreter meets the construct $goal_T$, it checks if a past event $goal_P$ or a fact corresponding to this predicate exists.

Past events represent the agent's "memory", that makes it capable to perform future activities while having experience of previous events, and of its own previous conclusions. Past events are kept for a certain default amount of time, that can be modified by the user through a suitable directive in the initialization file. A past event is formalized as follows:

**Definition 8 (Past Event).** *A past event is syntactically indicated by the postfix* $P$*:*
$PastEvent ::= << Atom_P >>$

## 3 DALI Communication Architecture

The DALI communication architecture consists of four levels. The first and last levels implement the DALI/FIPA communication protocol and a filter on communication, i.e. a set of rules that decide whether or not receive (*told* check level) or send a message (*tell* check level). The DALI communication filter is specified by means of meta-level rules defining the distinguished predicates *tell* and *told*. Whenever a message is received, with content part *primitive(Content,Sender)* the DALI interpreter automatically looks for a corresponding *told* rule. If such a rule is found, the interpreter attempts to prove $told(Sender, primitive(Content))$. If this goal succeeds, then the message is accepted, and $primitive(Content))$ is added to the set of the external events incoming into the receiver agent. Otherwise, the message is discarded. Symmetrically, the messages that an agent sends are subjected to a check via *tell* rules. The second level includes a meta-reasoning layer, that tries to understand message contents, possibly based on ontologies and/or on forms of commonsense reasoning. The third level consists of the DALI interpreter.

## 4 Operational Semantics

The operational semantics of DALI system [4] is defined by adopting an approach which is a novelty in the agent world. The novelty in particular is that we use a formal dialogue game in order to define the *full* operational semantics of the DALI interpreter.

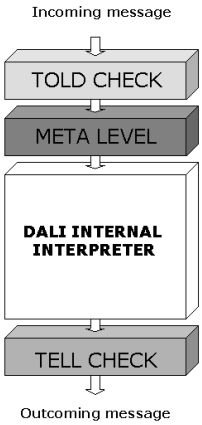



**Fig. 1.** DALI communication architecture

Recently, formal dialogue games, which have been studied in philosophy since the time of Aristotle, have found application as the basis for interaction protocols between autonomous agents [13] [14]. Dialogue games are formal interactions between two or more participants, in which participants "move" by uttering statements according to pre-defined rules.

Dialogue game protocols have been proposed for agent team formation, persuasion, negotiation over scarce resources, consumer purchase interactions and joint deliberation over a course of action is some situation ([11],[17],[18],[19]) but, to the best of our knowledge, they have not been used up to now to give a formal description of an agent language. In our formalization we assume that the DALI interpreter plays a game and thus makes "moves" not only towards other agents, but also towards itself. By adopting this approach we explain the behavior of each layer of the architecture and their interactions. We define a formal dialogue game framework that focuses on the rules of dialogue, regardless the meaning the agent may place on the locutions uttered. Dialogue games has been applied successfully in negotiation contexts because in these cases is possible to individuate easily players and moves.

The first question that we faced in order to formalize the operational semantics of DALI architecture has been in fact: which are the players and which moves can they make? We considered that the DALI architecture is composed by layers and each layer adopts a specific behavior. A layer can be viewed as a *dark box* whose behavior is determined only by moves of other correlated layers and by its policy. By adopting this view point, our players are the layers and moves are defined through laws and transitions rules.

A strategy for a player is a set of rules that describe exactly how that player should choose, depending on how the other player has chosen at earlier moves. The rules of the operational semantic show how the states of an agent change according to the applica-

tion of the transition rules. We define a rule as a combination of states and laws. Each law links the rule to the interpreter behavior and is based on the DALI architecture. Our work demonstrates how solutions from game theory together with computing theories can be used to publicly specify rules and prove desirable properties for agent systems. In order to make it clear what we intend for state, law and transition rule, we adopt the following definitions.

**Definition 9 (State of a DALI agent).** *Let $Ag_x$ be the name of a DALI agent. We define the internal state $IS_{Ag_x}$ of a DALI agent as the tuple $< E, N, I, A, G, T, P >$ composed by its sets of events, actions and goals.*

**Definition 10 (Law).** *We define a law $L_x$ as a framework composed by the following elements:*

- ***name:*** *the name of law;*
- ***locution:*** *the arguments that the law takes;*
- ***preconditions:*** *the preconditions to apply the law;*
- ***meaning:*** *the meaning of the law;*
- ***response:*** *the effects of the applied law;*

**Definition 11 (Transition rule).** *A transition rule is described by two pairs and some laws. If the transition is internal to the same agent, a transition rule corresponds to :*
$< Ag_x, < P, IS, Mode >> \overset{L_i,...,L_j}{\longrightarrow} < Ag_x, < NewP, NewIS, NewMode >>$
*Starting from the first pair and by applying the current laws, we obtain the second pair where some parameters have changed. Each pair is defined as $< Ag_x, S_{Ag_x} >$, where $Ag_x$ is the name of the agent and the operational state $S_{Ag_x}$ is the triple $< P_{Ag_x}, IS_{Ag_x}, Mode_{Ag_x} >$. The first argument is the logic program (written in DALI) of the agent, the second one is the internal state, the third one is a particular attribute describing what the interpreter is doing. $NewP$, $NewIS$ and $NewMode$ indicate, respectively, $P$, $IS$ and $Mode$ updated after applying $L_i, ..., L_j$ laws .*

*A transition rule can also describe how an agent can influence an other one. In this case, we will have:*
$< Ag_x, < P_{Ag_x}, IS_{Ag_x}, Mode_{Ag_x} >> \overset{L_i,...,L_j}{\longrightarrow} < Ag_y, < P_{Ag_y}, IS_{Ag_y}, Mode_{Ag_y} >>$
$where x \neq y$

The operational semantics viewed with the eyes of game theory transforms TOLD filter into TOLD player, META level into META player, and so on until TELL filter that becomes TELL player. Also the DALI internal interpreter becomes a player that plays with the other structural player and with itself. What will we expect from these players? Their behavior is surely cooperative because only if all levels work together, a DALI agent will satisfy the user expectations. The players are not malicious because our game is innocent and does not involve any competition strategy. So, we expect each player to follow deterministically the laws and rules and produces a set of moves admissible. These moves will influence the other players and will determine the global game.

When does a player win? The game that an agent plays with itself and with the other agents is innocent, so we do not intend define rigorously the concept of winner.

Our winner is the player which play with success a specific game. More precisely, we intend, after defining the general operational semantics, to prove some relevant properties of DALI language. For us, each property that must be demonstrated is a particular game that a player must face through defined the laws and rules. A player wins if plays successfully a game/property proposed. Next sections will describe the ability of DALI agents to generate children.

## 5 Child generation capability

A DALI agent is able to activate child agents and to feed them either with a goal to be reached or a result to be obtained. Each child agent is independent and can communicate with its father or with other agents. When the child finally reaches the given goal, it notifies the father. At any point, the latter may possibly decide to stop it. This will mostly happen either after obtaining results, or when the time amount that the father means to allocate to the child's task has expired. Then: (i) each child is aware of the identity of its father; (ii) each child will notify the father about its achievements; (iii) a child can be stopped by the father; (iv) the father may set a limited amount of time for children's activities completion.

Apart from that, a child agent is a DALI one, equipped with its own knowledge base, directives and communication filter, and can in turn create children. This feature is relevant for DALI multi-agent system scalability. From a cognitive point of view, it allows the father for instance to: compute and then compare various alternative plans (or intentions in the BDI view); perform hypothetical reasoning; create its own local social setting in the form of a society of agents, each one with its role and commitment. The resulting architecture, useful to DALI agents to generate children, is divisible in three modules, each of which offers specific functionalities. The first module allows a father agent to create children, the second one establishes a connection between father and child, the third one determines the child life time.

### 5.1 Create children

This first module allows each DALI agent to activate, through a specific action, one or more children. The new generated agent can include, according to the fatherly will, either the knowledge base of the father or a different knowledge base KB specified at the generation moment. If the child incorporates the father logic program and knowledge, the action able to create it will be:

– $create_A(Num\_Children)$**,** where $Num\_Children$ specifies how many agents the father intends to generate.

The KB specification implies that the child agent will have the knowledge and logic program contained in the specified file:

– $create_A(NumFigli, KB)$**,** where $Num\_Children$ has the same meaning specified above and KB specifies the file name containing the knowledge base.

For instance, an agent $party$ who plays the role of a party organizer, can generate the following children:

$create_A(1, c : /kb/cake.txt, ontology\_1).$
$create_A(1, c : /kb/fizz.txt, ontology\_2).$

Children will be named by default $party\_child1$ and $party\_child2$ ($child1$ and $child2$ for the father). The files $cake.txt$ and $fizz.txt$ contain all the activation data (including knowledge bases and, optionally, ontologies) for the two agents.

After this generation process, a child agent will have all potentialities to be able itself to generate further children. In other words, each child agent can become a father one. A particular mechanism avoids child agents to be given the same name. The last step of this module is to check if the activation succeeds: to this aim, the father agent sends a specific message to each child.

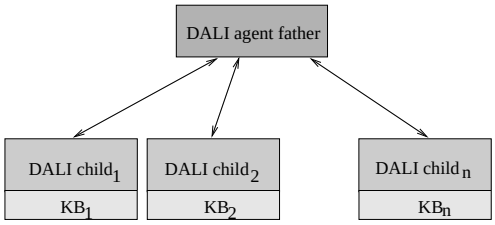



**Fig. 2.** DALI agent and children

### 5.2 Connect module

This module provides two functionalities: the first one establishes the connection between father and child agents; the second one allows the father to assign a subgoal to its child that, when it reaches its task, advises the father on its success. As soon as the child agent becomes active, it receives by the father the message: $born(Father\_name)$. Its child keeps in its memory the father name and sends to it the message: $hello\_dad(Child\_name)$ Starting from the moment in which this happens, two agents can communicate between them. When the father reaches the internal conclusion that it is necessary to assign a goal to a child, it sends one of the following messages:

– $solve\_goal(Goal, Ev, Time)$*:* the child has a time limit to resolve its task. The $Ev$ parameter is necessary because the father must trigger specific reactive rules (in the child program) to activate the resolution process;
– $solve\_goal(Goal, Ev)$*:* the child agent does not have a fixed amount of time to return the solution to the father;

The child, as soon as its goal is reached, tells the father through a *confirm* message.

### 5.3 Lifetime module

This third module kills the child agent when its allocated time has expired. DALI child agents have a specific internal event that checks from time to time if the current agent elapsed time has exceeded the value specified at the generation act. In this case, not only the agent is killed but also its data are erased.

## 6 Operational semantics of children generation

In this Section we show the operational semantics rules that cope with children generation. In particular, the laws are L19-L24 in the context of the 119 overall transition rules [21].

– **L19: initialize_child(.)** law:
***Locution:*** $initialize\_child(Logic\_program/KB, Ontology)$
***Preconditions:*** The agent reaches the conclusion (by an internal event) that it needs a child.
***Meaning:*** This law allows an agent to generate a child agent. If either Logic_program or Ontology are empty, the generated child will inherit the parameters of the father, else it takes the specified value.
***Response:*** The agent has a child agent.

– **L20:** The **active_child** law:
***Locution:*** $active\_child$
***Preconditions:*** The child agent has been initialized.
***Meaning:*** This law activates a child agent. After the activation, the child agent enters the "wait" mode and is ready to receive communication acts from the father. Father and child can communicate by using the usual DALI primitives.
***Response:*** The child agent is active.

– **L21:** The **expired_time_child** law:
***Locution:*** $expired\_time\_child$
***Preconditions:*** The time assigned from the father to child is expired.
***Meaning:*** This law checks the time assigned to the child agent.
***Response:*** The father informs the child that the time is finished and asks for the results.

– **L22:** The **obtain_result** law:
***Locution:*** $obtain\_result$
***Preconditions:*** The time assigned to the child has expired.
***Meaning:*** The child agent has reached the requested result and it sends it to the father.
***Response:*** The father obtains the result.

– **L23:** The **not_obtain_result** law:
***Locution:*** $not\_obtain\_result$
***Preconditions:*** The time assigned to child has expired.
***Meaning:*** The child agent has not achieved the requested result.
***Response:*** The father does not obtain the result.

– **L24:** The **kill_child** law:
***Locution:*** $kill\_child$
***Preconditions:*** The child agent terminates its job.
***Meaning:*** The father resets the internal state of the agent and removes it from the environment.
***Response:*** The child is dead.

## 7 An example: organizing a party

In this section we show an example in which an agent, having had a promotion, organizes a party in order to offer a cake and a fizz bottle to its friends. To this aim, it identifies two subgoals: to prepare the cake and to buy the bottle. Then, it creates two children in order to assign them the two tasks. We suppose that the internal event triggering the party organization is $organize\_party$:

$$
\begin{array}{l}
organize\_party : -promotion_P. \\
organize\_party_I :> \\
\qquad\qquad child\_name(F1, 1), \\
\qquad\qquad child\_name(F2, 2), \\
\qquad\qquad message_A(F1, confirm( \\
\qquad\qquad\qquad\qquad solve\_goal(cake\_ready, cake), user)), \\
\qquad\qquad message_A(F2, confirm( \\
\qquad\qquad\qquad\qquad solve\_goal(fizz\_ready, fizz, 120000), user)).
\end{array}
$$

where the $child\_name/2$ predicate is useful to obtain the child agents names. Via the messages $solve\_goal$, the children receive the goals assignment. When the father agent receives the communications from the children that their tasks have been accomplished, it starts the party.

$$
\begin{array}{l}
start\_party : -cake\_ready_P, fizz\_ready_P. \\
start\_party_I :> write('The\ party\ is\ starting...'), invite\_everyone_A.
\end{array}
$$

After the generation, the child agents tell the user about their birth by printing:

*Hello World..... My name is party_child1*
*My father is party*

while the father *party*, verified the success of the generation process, writes:

*My son is party_child1*
*My son is party_child2*

Once started, children will react to an event of the form $solve\_goal(G)$ coming from their father. In this case, for instance, the father will be able to ask children to prepare a cake and drinks respectively, by means of the messages:

$message_A(child1, confirm(solve\_goal(cake\_ready)).$
$message_A(child1, confirm(solve\_goal(buy\_drinks)).$

The father will be notified by the children when the goal will have been reached, and made aware of results. Notice that the second child has a time limit to give a solution. Below we show the logic programs of two children.

**The agent *party_child1*** The knowledge base of this agent consists in the *cake.txt* file and contains the following rules:

$cake_E :> preparing\_cake_G.$
$preparing\_cake : -haveFlour_P.$
$preparing\_cake_I :> cake\_ready_A.$

The agent triggers the goal $preparing\_cake_G$ while the *connect module* starts to verify if the assigned time is expired. In order to reach its goal, the agent is in need of flour. If the agent receives the flour, it prepares the cake and informs its father:

*make(cake_ready)*
*send_message_to(party_child1,send_message(cake,party_child1))*
*send_message_to(party_child1,agree(cake_ready,party_child1))*
*send_message_to(party_child1,*
*inform(agree(cake_ready),**values(yes)**,party_child1))*
*Reached Goal: cake_ready*
*send_message_to(party,confirm(cake_ready,party_child1)).*

**The agent *party_child2*** This agent has the following logic program:

$fizz_E :> buy\_fizz_G.$
$buy\_fizz : -haveMoney_P.$
$buy\_fizz_I :> fizz\_ready_A.$

In order to reach its goal, this child must have sufficient money. In this case, it buys the bottle and advices its father. The last exchanged messages are:

*Reached Goal: fizz_ready*
*send_message_to(party,confirm(fizz_ready,party_child2))*

***The party is starting*** After receiving the messages indicating that the subgoals have been reached, the father agent starts the party:

*The party is starting...*
*make(invite_everyone)*

## 8 Conclusions and Related Work

We conclude this discussion with some considerations on DALI agents generation capabilities. The father agent is not required to know the contents of children KB except concerning external events that trigger children activities. Each child is under every respect a DALI agent that can interact with the other entities in the environment and can increase its knowledge independently of the father. The latter can only kill the child when it is no more useful.

While the children work, the father can continue its activity without losing contact with the environment. The father can also assign to children intermediate sub-goals and reorganize the obtained results. Each child can create its own children, thus increasing the computational power of the system. Finally, the time limit allows a system to spare computational resources. The child generation capability that we have presented is the starting point to improve DALI agents computational power: in the future we will make it possible for the father agent to specialize its children by making them import specific library modules. Also, this mechanism can be a useful features in the context of more general coordination frameworks and strategies.

In fact, the DALI communication architecture together with the children generation mechanism constitute a basic support for cooperation that DALI provides. The communication architecture neatly separates an agent's core behavior from the agent's behavior related to communication. The same DALI agent program equipped with a different communication architecture actually results in a different agent, as its relationship with its environment is different, and affects its internal state in a different way. Sub-agents can be employed so as to perform in a distributed fashion different specific tasks. These features combined together allow significant forms of social knowledge to be represented and reasoned about, and to evolve in time based on the agent's beliefs, experience, and interactions with other agents [4].

Many current multi-agent teamwork coordination strategies are based on theoretical frameworks such as [2], [8], [9], and typically involve the recognition of agent mental states, possibly by relying on the BDI ("Belief, Desire, Intentions") model [1]: and agent's *beliefs* correspond to information the agent has about the world, which may be incomplete and incorrect; an agent's *desires* intuitively correspond to its objectives, or to the tasks allocated to it; as an agent will not, in general, be able to achieve all its desires, the desires upon which the agent commits are *intentions* that the agent will try to achieve. These coordination approaches, and also those mainly based on communication, are limited whenever communication is unreliable, or information on the source incomplete.

Mediated interaction and environment-based coordination focus on cognitive and social theories and explicitly take into account the role of the environment in coordi-

nation, such as [2], [8], [15]. In [16], it is emphasized that any real conceptual and engineering framework for this approach should: (i) do not rely on simple reactivity only; (ii) not restrict to solution tailored to specific coordination problems; (iii) provide methodologies and infrastructures to make the framework effective.

The support for coordination provided by the DALI language, simple as it is on the one hand addresses some the problems that arise in BDI-based and communication-based approach, due to its powerful communication filter. On the other hand, DALI addresses issues (i)-(iii) above as it is a general-purpose language with powerful proactive features, has a precise declarative and operational semantics, and is fully implemented. A future aim of this research is to further extend and refine DALI support to coordination, and to put it at work in complex application domain such as for instance peer-to-peer negotiation.

## References


1. M. E. Bratman, D. J. Israel and M. E. Pollack. Plans and Resource-bounded Practical Reasoning, *Computational Intelligence*, vol. 4, 1988, 349–355.
2. P. Cohen and H. Levesque. Teamwork, Nous, Special Issue on Cognitive Science and AI, vol. 25, no. 4, 1991, 487–512.
3. S. Costantini and A. Tocchio. A Logic Programming Language for Multi-agent Systems, In S. Flesca, S. Greco, N. Leone, G. Ianni (eds.), Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002, LNAI 2424, Springer-Verlag, 2002.
4. S. Costantini, A. Tocchio and A. Verticchio. A Game-Theoretic Operational Semantics for the DALI Communication Architecture, Proc. of WOA04, 2004.
5. S. Costantini, A. Tocchio and A. Verticchio. Communication and Trust in the DALI Logic Programming Agent-Oriented Language, In: M. Cadoli, M. Milano and A. Omicini (eds.), Italian Conference on Intelligent Systems AI*IA'04, 2004.
6. M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A Formal Computational Model, Journal of Logic and Computation 8(3), 1998, 233–260.
7. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming, In Proceedings of the Fifth Joint International Conference and Symposium. The MIT Press, 1988, 1070–1080.
8. B.J. Grosz and S. Kraus. Collaborative Plans for Complex Group Action, Artificial Intelligence, 86(2), 1996, 269–357.
9. D. Kinny, M. Ljungberg, A. Rao, E. Sonenberg, G. Tidhar and E. Werner. Planned Team Activity, In: Artificial Social Systems, 4th Europ. Worksh. on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'92), Selected Papers, S.Martino al Cimino, Italy, 1994, 227–256.
10. R. A. Kowalski. How to be Artificially Intelligent - the Logical Way, Draft, revised February 2004, Available on line, URL http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/rak.html.
11. K. Larson and T. Sandholm. An alternating offers bargaining model for computationally limited agents, In: First International Conference on Autonomous Agents and Multiagent Systems (AAMAS'02), 135–142.
12. V. Lifschitz. Answer Set Programming and Plan Generation Artif. Intelligence 138 (1–2), Elsevier Science Publishers, 2002, 39–54.
13. P. McBurney and S. Parsons. Dialogue Games Protocols for Agent Purchase Negotiations, In: M.-P. Huget (ed.), Communication in Multi-Agent Systems: Agent Communication Languages and Conversation Policies, LNAI 2650, Springer-Verlag, 2001.

14. P. McBurney, R. M. Van Eijk, S. Parsons and L. Amgoud, A Dialogue Game Protocol for Agent Purchase Negotiations, Autonomous Agents and Multi-Agent Systems 7(3), Kluwer Academic Publishers, 2003, 235–273.
15. B.A. Nardi. Context and Consciousness: Activity Theory and Human-Computer Interaction, MIT Press, 1996.
16. A. Omicini, A. Ricci, M. Viroli,C. Castelfranchi,L. Tummolini. Coordination Artifacts: Environment-based Coordination for Intelligent Agents, In: *Proc. of the Third Int. Joint Conf. on Autonomous Agents and Multiagent Systems(AAMAS'04)*, ACM Press, 2004.
17. David C. Parkes. Optimal Auction Design for Agents with hard Valuation Problems In: Agent-Mediated Electronic Commerce Workshop at the International Joint Conference on Artificial Intelligence, Stockholm, 1999.
18. T. Sandholm. Unenforced e-commerce Transactions, IEEE Internet Computing 1(6) (November-December 1997), 47-54.
19. T. Sandholm, S. Suri, A. Gilpin and D. Levine. CABOB: A Fast Optimal Algorithm for Combinatorial Auctions, In: Proc. IJCAI-0l, Seattle, WA, 2001, 1102–1108.
20. Web location of the most known ASP solvers.
    Cmodels: *http://www.cs.utexas.edu/users/yuliya/*
    Aspps: *http://www.cs.uky.edu/ai/aspps/*
    DLV: *http://www.dbai.tuwien.ac.at/proj/dlv/*
    NoMoRe: *http://www.cs.uni-potsdam.de/~linke/nomore/*
    Smodels: *http://www.tcs.hut.fi/Software/smodels/*
21. A. Tocchio. Multi-Agent sistems in computational logic. Ph.D. Thesis (draft).

# Heuristics, Optimizations, and Parallelism for Protein Structure Prediction in CLP($\mathcal{FD}$) *

Alessandro Dal Palù[1], Agostino Dovier[1], and Enrico Pontelli[2]

[1] Dip. di Matematica e Informatica, Univ. di Udine
(dalpalu|dovier)@dimi.uniud.it
[2] Department of Computer Science, New Mexico State University.
epontell@cs.nmsu.edu

**Abstract.** The paper describes a constraint-based solution to the protein folding problem on *face-centered cubic lattices*—a biologically meaningful approximation of the general protein folding problem. The paper improves the results presented in [14] and introduces new ideas for improving efficiency: *(i)* proper reorganization of the constraint structure; *(ii)* development of novel, both general and problem-specific, heuristics; *(iii)* exploitation of parallelism. Globally, we obtain a speed up in the order of 60 w.r.t. [14]. We show how these results can be employed to solve the folding problem for large proteins containing subsequences whose conformation is already known.

## 1 Introduction

Proteins are responsible for nearly every function required for life. The sequence of elements (amino acids) identifying a protein is known as the primary (1D) structure. A functional protein can be thought of as a properly folded chain of amino acids in 3-dimensional (3D) space. The 3D structure of a protein characterizes its function. A folded protein interacts three-dimensionally with other proteins (e.g., lock and key arrangements) and this interaction determines the functions of the organism. In fact, an organism is essentially determined by the three-dimensional interactions between proteins and substrates. Thus, without knowing the 3D structure of the proteins coded in a genome, we cannot completely understand the phenotype and functioning of living organisms. Understanding how protein folds has profound implications—e.g., towards the theoretical design of exact drugs, the improvement of proteins functionality, and the precise modeling of cells.

In recent decades, most scientists have agreed that the answer to the folding problem lies in the concept of the *energy state* of a protein. The predominant strategy in solving the protein folding problem has been to determine a state of the amino acid sequence in the 3D space with minimum energy. According to this theory, the 3D conformation that yields the lowest energy state represents the protein's natural shape (a.k.a. the *native conformation*). The energy of a conformation can be modeled using *energy functions*, that determine the energy level based on the interactions between any pairs of amino acids [7]. Thus, we can reduce the protein folding problem to an optimization problem, where the energy function has to be minimized under a collection of constraints (e.g., derived from known chemical and physical properties) [12].

We employ *Constraint Logic Programming (CLP)*, in particular, constraint logic programming over *finite domains* (CLP($\mathcal{FD}$)), to model and solve a tractable representation of the protein folding problem—i.e., protein folding in the context of face-centered cubic

* This paper has been submitted and accepted in PPDP 2005.

lattices [28]. The choice of CLP is natural for a variety of reasons (see, e.g., [2]). CLP is a paradigm that is highly suitable to address optimization problems; it provides declarative and high-level modeling, combined with effective built-in search and resolution strategies. Furthermore, the high-level modeling offered by CLP allows us to easily *add* new constraints and to plug-and-play different search strategies and heuristics.

A preliminary approach to the use of CLP($\mathcal{FD}$) for the protein folding problem has been presented in [14]. In this paper, we elaborate such proposal to consider the issue of *efficiency* and *scalability*. The ultimate objective of this paper is to demonstrate that

- the modeling and optimization capabilities of CLP($\mathcal{FD}$) are highly suitable to tackle the protein folding problem;
- the high-level modeling capabilities allow us to easily add or modify constraints as they become available, and to explore the use of different heuristics and search strategies;
- efficiency and scalability can be achieved for realistic problems.

The first step in this process lies in a remodeling of the constraint problem, aim at making the modelization more suitable to the capabilities of current CLP($\mathcal{FD}$) solvers. We also introduce new high-level heuristics for guiding the exploration of the search space, leading to a more effective pruning and enhanced scalability. In particular, we introduce a heuristic called *Bounded Block Fails*. Where the built-in strategies are insufficient to achieve acceptable levels of performance, we introduce the use of *parallelism*, easily exploitable from the high-level search structure generated by the CLP execution. Using a novel structure, heuristics, and a 14 processors parallel machine, we obtain a speed-up in the order of 60 w.r.t. the performance of the code presented in [14]—running on a single processor of the same parallel machine. The investigation proposed here pushes the built-in capabilities of CLP solvers to, what we believe, are the limits for the problem at hand. The proposed results indicate also that *better performance could be accomplished*, but only at the price of building some of the proposed heuristics at a lower level—i.e., as an ad-hoc constraint solver, and bypassing the built-in strategies of CLP($\mathcal{FD}$). Finally, we demonstrate what, we believe, is one of the greatest application areas for our technology: folding large proteins containing subsequences whose native conformation is already known (e.g., by homology)—e.g., macro blocks linked by a neutral coil. This type of situation is very common; in our framework, the known structures can be directly added as constraints (with *no modifications* to the rest of the constraint model), allowing us to tackle large problems with excellent performance.

### 1.1 Related Works

The bibliography on the protein folding problem is extensive [8, 25]; the problem has been recognized as a fundamental challenge [22], and it has been addressed with a variety of approaches (e.g., comparative modeling through homology, fold recognition through threading, ab initio fold prediction).

An abstraction of the problem, that has been recently investigated, is the protein folding problem in the *HP* model, where amino acids are separated into two classes ($H$, hydrophobic, and $P$, hydrophilic). The goal is to search for a conformation produced by an *HP* sequence, such that most HH pairs are neighboring in a predefined lattice. The problem has been studied on 2D square lattices [13, 21], 2D triangular lattices [1], 3D square models [19], and face-centered cubic lattices (fcc) [23]. Backofen and Will have extensively studied this last problem [3–5]. The approach is suited for globular proteins, since the main force driving the folding process is the electrical potential generated by $H$s and $P$s, and the fcc lattices are one of the best and simplest approximation of the 3D space (Sect. 2.2). Compared to the work of Backofen and Will, our approach refines the energy contribution model, extending the interactions between classes $H$ and $P$ to interactions between each pair of amino

acids [7]. Moreover, we introduce the possibility to model secondary structure elements, that cannot be reproduced correctly using only a simple energy model as the one adopted by other researchers.

The use of constraint programming technology in the context of the protein folding problem has been fairly limited. Backofen and Will have made use of constraints over finite domains in the context of the *HP* problem [5]. Clark et al. employed Prolog to implement heuristics in pruning a exhaustive search for predicting $\alpha$-helix and $\beta$-sheet topology from secondary structure and topological folding rules [11]. Distributed search and continuous optimization have been used in ab initio structure prediction, based on selection of discrete torsion angles for combinatorial search of the space of possible protein foldings [18].

## 2 Problem modeling

### 2.1 The Protein Folding Problem

The *Primary* structure of a protein is a sequence of linked units (*amino acids* or *residues*) of a given length. The amino acids can be identified by an alphabet $\mathcal{A}$ of 20 different symbols, associated to specific chemical-physical properties. A protein has a high degree of freedom, and its 3D conformation is named *Tertiary* structure.

From the *energy* point of view, the molecule tends to reach a conformation with a minimal value of free energy (*Native* conformation). Native conformations are largely built from *Secondary Structure elements* (i.e., helices and sheets) often arranged in well-defined motifs. $\alpha$-helices are constituted by residues arranged in a regular right-handed helix with 3.6 residues per turn. $\beta$-sheets are constituted by extended strands. Each strand is made of contiguous residues, but strands participating in the same sheet are not necessarily contiguous in sequence. Algorithms, e.g., based on neural networks, that have been developed, are capable to predict the secondary structure with high accuracy (75% [8]).

Another important structural feature of proteins is the capability of cysteine residues of covalently bind through their sulphur atoms, thus forming disulfide bridges, which impose important constraints on the structure (also known as `ssbonds`). This kind of information is often available, either through experiments or predictions.

Several models have been proposed for reasoning about the 3D properties of proteins—i.e., dealing with the *Tertiary Structure*. Given a primary sequence $S = s_1 \cdots s_n$, with $s_i \in \mathcal{A}$, let us represent with $\omega(i)$ the *position* of a point representing the amino acid $s_i$ in space; $\omega(i)$ is a vector $\langle x_i, y_i, z_i \rangle \in \mathcal{D}$, for a given space domain $\mathcal{D}$. The values $x_i, y_i$, and $z_i$ can be real numbers—in models in which proteins are free of taking any positions in space—or integer numbers—in models where amino acids can assume only a finite number of positions within a suitable lattice structure. We call $\mathcal{D}$ the set of admissible points.

Given two points $\omega_1, \omega_2 \in \mathcal{D}$, we indicate with $\texttt{next}(\omega_1, \omega_2)$ the fact that the two points are admissible positions for two amino acids that are contiguous in the primary sequence. We assumed that consecutive amino acids are separated by a fixed distance.

We also employ the binary predicate `contact`, which is used to describe the fact that two amino acids are sufficiently close to be able to interact, and thus they contribute to the energy function: two non-consecutive amino acids $s_i$ and $s_j$ in the positions $\omega(i)$ and $\omega(j)$ are in contact (denoted by $\texttt{contact}(\omega(i), \omega(j))$) when their distance is less than a given threshold.

Given a primary sequence $S = s_1 \cdots s_n$, with $s_i \in \mathcal{A}$, a *folding* of $S$ is a function $\omega : \{1, \ldots, n\} \rightarrow \mathcal{D}$ such that:

1. $\texttt{next}(\omega(i), \omega(i+1))$ for $i = 1, \ldots, n-1$, and
2. $\omega(i) \neq \omega(j)$ for $i \neq j$ (namely, $\omega$ introduces no loops).

A simplified evaluation of the energy of a folding can be obtained by observing the *contacts* present in the folding. In particular, every time a contact between a pair of amino

acids is detected, a specific energy contribution is applied towards the global energy. These contributions can be obtained from tables developed using statistical methods applied to structures obtained from X-Rays and Nuclear Magnetic Resonance experiments [20, 7]; these tables associates an energy measure to each pair of non-consecutive amino acids when they are in contact. We denote with $\texttt{Pot}(s_i, s_j)$ the energy contribution associated to the amino acids $s_i$ and $s_j$ (the order does not matter).

The *protein structure prediction problem* can be modeled as the problem of finding the folding $\omega$ of $S$ such that the following energy cost function is minimized:

$$E(\omega, S) = \sum_{1 \le i < n} \sum_{i+2 \le j \le n} \texttt{contact}(\omega(i), \omega(j)) \cdot \texttt{Pot}(s_i, s_j).$$

With a slight abuse of notation predicate `contact` is here used as a Boolean function. This definition is sufficiently general to cover the case of several spatial models $\mathcal{D}$, such as the fcc lattice and the cubic lattice [12].

### 2.2 Lattice Models

Lattice models have long been used for protein structure prediction (see [26] for a survey). In [23] it is shown that the *Face-Centered Cubic Lattice* (fcc) model is a well-suited, realistic model for 3D conformations of proteins. The model is based on cubes of size 2, where the central point of each face is also admitted. The domain $\mathcal{D}$ consists of the set of triples $\langle x, y, z \rangle$, where $x, y, z \in \mathbb{N}$ such that $x + y + z$ is even (see Fig. 1). Points at Euclidean distance $\sqrt{2}$ are linked; their distance is called *lattice unit*. Observe that, for linked points $i$ and $j$, it holds that $|x_i - x_j| + |y_i - y_j| + |z_i - z_j| = 2$.

Each point is adjacent to 12 neighboring points. Thus, we define the predicate `next` as follows: $\texttt{next}(\omega(i), \omega(i+1))$ holds iff

- $|x_i - x_{i+1}| \in \{0, 1\}, |y_i - y_{i+1}| \in \{0, 1\}, |z_i - z_{i+1}| \in \{0, 1\}$,
- $|x_i - x_{i+1}| + |y_i - y_{i+1}| + |z_i - z_{i+1}| = 2$.

In fcc lattices, the angle between three adjacent residues may assume values $60°$, $90°$, $120°$, and $180°$. Volumetric constraints and energetic restraints in proteins make values $60°$ and $180°$ infeasible. Therefore, in our model, we retain only the $90°$ and $120°$ angles [28, 17]. No similar restriction exists on torsional angles among four adjacent residues. In detail, let $\boldsymbol{v}_{i-1,i} = \omega(i) - \omega(i-1)$ and $\boldsymbol{v}_{i,i+1} = \omega(i+1) - \omega(i)$. To impose that the angle between them can only be of $90°$ and $120°$, we use the scalar product between these two vectors: $\boldsymbol{v}_{i-1,i} \cdot \boldsymbol{v}_{i,i+1} = |\boldsymbol{v}_{i-1,i}||\boldsymbol{v}_{i,i+1}| \cos(\theta)$. Thus, since $|\boldsymbol{v}_{i-1,i}| = |\boldsymbol{v}_{i,i+1}| = \sqrt{2}$ we only need to impose that: $\boldsymbol{v}_{i-1,i} \cdot \boldsymbol{v}_{i,i+1} \in \{1, 0\}$.

A *contact* between two non-adjacent residues in fcc occurs when their separation is two lattice units—i.e., viewing the lattice as a graph whose edges connect adjacent points in the lattice, the positions of the residues are connected by a path of length 2.

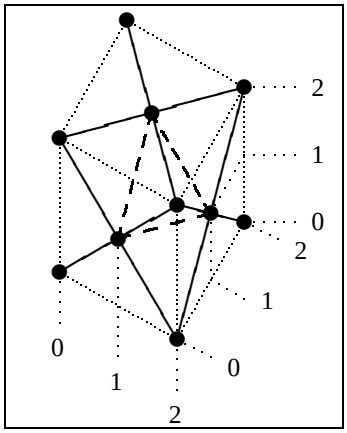


**Fig. 1:** A cube of the fcc lattice. Thick lines link connected points.

Physically, two amino acids in contact cannot be at the distance of a single lattice unit, because their volumes would overlap. Consequently, we impose the constraint that two non-consecutive residues $s_i$ and $s_j$ must be separated by more than one lattice unit, by adding the constraint (called `non_next`): $(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 > 2$. With these additional constraints, we can define: $\texttt{contact}(\omega(i), \omega(j))$ iff $|x_i - x_j| + |y_i - y_j| + |z_i - z_j| = 2$.

## 3 A CLP($\mathcal{FD}$) Implementation

The formalization of the protein structure prediction problem in fcc has been instantiated in a declarative program in CLP over finite domains—and it follows the basic structure outlined in [15, 14]. In this section, we describe the main predicates employed in such implementation. We wrote the system in CLP($\mathcal{FD}$) of SICStus PROLOG 3.12.0 [9] and the library `ic` of ECLiPSe 5.8 [10]. Complete code and other related material can be found in: `www.dimi.uniud.it/dovier/PF`.

The program has the classical *Constrain & Generate* [2] structure. The `constrain` predicate deterministically adds the constraints for the variables involved, while `labeling` searches for the solution in the search space (through chronological backtracking). The inputs are `Primary` (a list of amino acids), `Secondary` (a list of known secondary structure components), and `Matrix` (the matrix of energy contributions, see later in the Section). `Tertiary` is the output list of positions (a flat list of triples of integers) of the conformation and `Energy` is the output value of energy associated to such conformation. We do not discuss here the more technical parameters. The predicate `constrain` is defined here:

```
constrain(Primary, Secondary, Energy, Tertiary, Matrix,
          Compact, OrdTertiary):-
   length(Primary,N),
   generate_tertiary(N, Tertiary), domain_bounds(Tertiary,N),
   avoid_self_loops(Tertiary,N), next_constraints(Tertiary),
   distance_constraints(Tertiary), angles(Tertiary),
   compact_constraints(Tertiary,N,Compact),
   secondary_info(Secondary, Tertiary),
   avoid_symmetries(Secondary, Tertiary, SSP),
   define_variables_order(Secondary,Tertiary, SSP, OrdTertiary),
   energy_constraints(Primary, Tertiary, Secondary, Energy, Matrix).
```

Given the input list of amino acids `Primary` $= [s_1, \ldots, s_N]$, `generate_tertiary` creates the list `Tertiary` $= [X_1, Y_1, Z_1, \ldots, X_N, Y_N, Z_N]$ of $3N$ variables. The predicate `domain_bounds` specifies the domains for the $X_i, Y_i, Z_i$ variables (the range $0 \ldots 2 * N$), and adds the constraints forcing $X_i + Y_i + Z_i$ to be even. `avoid_self_loops` forces all triples to be distinct. `next_constraints` imposes that the points $[X_i, Y_i, Z_i]$ and $[X_{i+1}, Y_{i+1}, Z_{i+1}]$ are adjacent in the lattice. `distance_constraints` forces two non-consecutive points to be at a lattice distance greater than 1. `angles` defines the admissible angles formed by three consecutive amino acids. The predicate `compact_constraints` introduces a user-defined maximal distance between amino acids (called *compact factor*). `secondary_info` encodes the Secondary Structure information as constraints in the program. The secondary structure (`Secondary`) information is retrieved from the Protein Data Bank [6]; it is easy to modify the code to obtain such information from secondary structure prediction programs (e.g., [24]). `avoid_symmetries` removes equivalent admissible conformations modulo symmetries and/or rotations. `define_variables_order` makes use of the distribution of secondary structure elements to sort the variables in `Tertiary` for labeling purposes (see Section 4.1).

As described above, two amino acids $s_i, s_j$ are in contact if a path of length 2 lattice units links them. The contact information is maintained in a symmetric matrix $M$, such that $M[i, j]$ is the energy contribution provided by $s_i$ and $s_j$. The following code constrains the contact contribution to the matrix element:

```
table(si,sj,Pot), M[i,j] in {0,Pot},
2 #= abs(Xi-Xj) + abs(Yi-Yj) + abs(Zi-Zj) #<=> M[i,j] #= Pot.
```

where table reports the value $\texttt{Pot}(s_i, s_j)$ as computed in [7]. Values of Pot have been scaled w.r.t. [7] to have only integer values. The global energy is the sum of the elements in $M$. The optimal folding is reached when the global energy is minimal. During the *labeling* phase, the information stored in $M$ is used to control the minimization process and to prune the search tree.

Various heuristics have been developed, aimed at simplifying the computation of the contact matrix $M$ and at the search pruning based on the dynamic analysis of $M$.

To reduce the search space, it is possible to ignore the contact contributions for all pairs $s_i, s_j$ such that $j < i + \mathsf{min\_aa\_dist}$, where min_aa_dist is a parameter. In those cases, the constraint is not applied and $M[i, j] = 0$. When we increase the value min_aa_dist, we generate a simpler global constraint and, at the same time, we consider only contributions from distant amino acids (considered, from the biological perspective, more relevant). For more details about other heuristics we refer to [14, 15].

# 4 The new implementation

## 4.1 Constraints Redesign

In this section we present the improvements w.r.t. the code in [14], briefly described in the previous section, to seek better performance and scalability to larger proteins.

In our previous experiments, we observed that one of the causes behind the lack of scalability of the constraint system of Sect. 3 to large proteins is the limited propagation achieved in presence of non-linear constraints (e.g., constraints related to torsion angles, as they require vector scalar products). The first stage of redesign implied removing as many non-linear constraints as possible and enhancing the propagation structure.

**Angle-free Encoding** The constraint structure proposed in [14] to solve the problem, imposed an alternative local coordinate system, used for the description of the tertiary structure; this coordinate system is composed of a sequence of the torsion angles forming the protein. The secondary structure elements, due to their regular shape, were simply described by a regular sequence of angles. We noticed that, the resulting constraints caused a bottleneck during the search because of poor propagation.

In the current version, we opted to remove the torsion angles description of secondary structure elements and restrict our description of the tertiary structure only to Cartesian coordinates. We simplify and remodel the description of secondary structure elements by means of global coordinate constraints.

**Variable Selection Strategy** In the original constraint scheme in [14], the variable selection is dynamic. The strategy employed selects, for labeling, a variable which is adjacent to a ground subsequence of the primary sequence. Basically, the method tries to grow the ground part of protein until an acceptable solution is found.

The solution we propose here relies on precomputing the order of labeling of the variables, to avoid the run-time costs of the previous strategy. In the constrain phase, we select the longest secondary structure, and we assign to it ground values. The protein is then partitioned in five consecutive parts: *Left Tail*—containing no secondary structure elements; *Left Body*; *Longest Secondary Structure Element*—ground; *Right Body*; *Right Tail*—containing no secondary structure elements.

The exploration order used during labeling is the following: *(i)* Left Body first (in reverse order), *(ii)* Right Body, *(iii)* Left Tail (in reverse order), and *Right Tail*. The tails are free to assume every conformation—they do not contain any superimposed pattern. Since the main energy contribution is given by the protein body, the tails are instantiated at the end.

**Contact revisited** We introduce a modification in the computation of the energy function w.r.t. Sect. 3: for our computations we increased the parameter min_aa_dist to 3 from 2

(as in [14]). We noted that three consecutive amino acid, $s_i, s_{i+1}, s_{i+2}$ can form different angles ($90^\circ$ or $120^\circ$), that cause different energy contributions due to the lattice structure and our energy model. To overpass this asymmetry in local conformations we decided to skip those energy contributions. The intuition of our choice is that the "important" energy contributions that characterize a folding are those provided by pairs of amino acids that are "far apart" in the primary sequence. Indeed, we observed that the contributions deriving from local subsequences tend to mask the global energy evaluation and thus bias our search heuristic.

**Experimental Results** We compared the new ideas implementation to the previous [14]. The experiments highlight a marked improvement in performance; only in one case we observed a slow-down, while in many cases we achieved from $4$ to over $50$ fold speedup. On the other hand, although the results are good for relatively small proteins, the new method are not sufficient to produce significant speedups on bigger proteins. In the next section, we introduce a new search heuristic that, combined with the ideas introduced here, is capable to efficiently explore the search tree generated by larger proteins.

### 4.2 Bounded Block Fails heuristic

In this section, we present a novel heuristic to guide the exploration of the search tree, called *Bounded Block Fails (BBF)*. This technique is general and can be applied to every search with a fixed ordering of variables—though it is particularly effective when applied to the protein folding problem at hand.

The heuristic involves the concept of *block*. Let us assume that $V$ is a list $[V_1, \ldots, V_n]$ of variables and constants. A block $B_i$ is a sublist of $V$ of size $k$ composed of unbound variables. The concatenation of all the blocks $B_1 B_2 \ldots B_\ell$ gives the ordered list of unbound variables present in $V$, where $\ell \leq \lceil \frac{n}{k} \rceil$. The blocks are selected dynamically, and they could exclude some of the original variables, that have already been instantiated due to constraint propagation. The number of blocks, thus, could be less than $\lceil \frac{n}{k} \rceil$ and it could be not constant during the whole search. In Figure 2 we depict a simple example for $k = 3$: we consider a list of 9 variables. The dark boxes represent ground assignments.

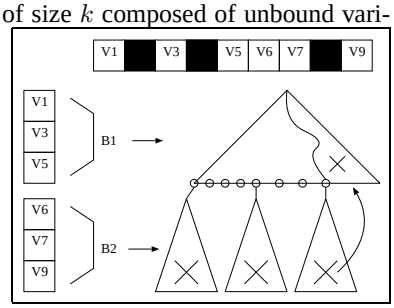



**Fig. 2:** The BBF heuristics

The heuristics consists of splitting the search among the $\ell$ blocks. Internally, each block $B_i$ is individually labeled according to the desired labeling strategy—in our case, the same heuristic employed in [14]. When a block $B_i$ has been completely labeled, the search moves to the successive block $B_{i+1}$, if any. If the labeling of the block $B_{i+1}$ fails, the search backtracks to the block $B_i$. Here there are two options: if the number of times that $B_{i+1}$ completely failed is below a certain threshold $t_i$, then the process continues, by generating one more solution to $B_i$ and re-entering $B_{i+1}$. Otherwise, if too many failures have occurred, then the Bounded Block Fail heuristic generates a failure for $B_i$ as well and backtracks to a previous block. Observe that the count of the number of failures includes both the regular search failures as well as those caused by the Bounded Block Failure strategy. The list $t_1, \ldots, t_\ell$ of thresholds determines the behavior of the heuristic. In the Figure, we assume $t_1 = 3$; the figure shows that, after the third failure of $B_2$, the search on $B_1$ fails as well.

The BBF heuristic is effective whenever the suboptimal solutions are spread sparsely in the search tree and/or for each admissible solution, there are many others with small differences in variables assignments and energy. In these cases, we can afford to skip solutions when generating block failure, because some others are going to be discovered following

other choices in some earlier blocks. For our problem, it is reasonable to keep high threshold values for the first blocks, while exploring only a small fraction of the search space present in the last blocks. Usually, many equivalent solutions can be found, just by changing the assignments in the last blocks, while the core of the problem lies in the first blocks.

In general, this technique can be effectively applied every time the variables are associated to some spatial properties and the corresponding physical object are related to each others, like in the case of a chain of amino acids. When assigning positions following the order of the amino acids on the chain, a failure in the current branch, means that the partial conformation does not allow to proceed without a collision. The BBF heuristic suggests to try to revise some earlier choices instead of exploring the whole space of possibilities depending on the block that collects failures. The high density and the great number of admissible solutions allow us to exclude some solutions, depending on the threshold values, and to still be able to find almost optimal solutions in shorter time.

### 4.3 Experimental Results

We run some specific tests to measure the benefits of BBF heuristic. On average the times are reduced by 2 times, while for *larger* proteins the benefits are more evident. Moreover the minima found are comparable to the ones obtained without activating the heuristic. Note that the current implementation of the BBF heuristic is performed in SICStus Prolog at a very high-level; e.g., when failing, many `fail` predicates are invoked, with a relatively high cost in the handling of the search tree. This makes the implementation rather inefficient. In spite of this, the BBF heuristics is designed to handle inputs with large sizes. For smaller proteins, the number of blocks tends to be small, and the heuristics accomplishes few block backtracks. In our experiments, we made use of blocks of size $3$ (corresponding to the three coordinates of a single amino acid); attempts to reduce this value (i.e., increase the number of blocks) actually produced degradation of performance, due to the excessively small size of the blocks (that defeats the original idea of BBF). To handle larger proteins, it is possible to use larger blocks (e.g. $k = 9$), that include more nodes within the corresponding subtree.

In Table 1, we report a comparison between the results obtained in the original constraint structure [14] and the corresponding ones obtained using the ideas reported in Section 4.1 and the BBF heuristic. In the column *CF* we indicate whether a specific compact factor was used (see Section 2)—when blank, we used a (high) default value (see [14] for more details). In the *AA* column, we indicate the number of amino acids in the protein, while in the *Core Zone* column we identify the subsequence of the protein without the tails—since the tails are less stable and less relevant for a quality test. The *RMSD* column reports the RMSD values (between brackets we indicate the RMSD for the Core Zone) of the computed solution w.r.t. the native structure deposited in the PDB (c.f. Section 4.3). At the bottom of the table (marked with (**)), we report two larger proteins, in order to demonstrate the power of the BBF heuristic, using larger size of the BBF blocks ($k = 9$). The thresholds are set to $t_1 = 4$ and $t_{\lceil n/k \rceil} = 2$. The foldings of these proteins were beyond the capabilities of the original constraint structure, while the computation time is extremely low using BBF; furthermore, the RMSD errors are also sufficiently low, thus making this folding a reasonable input to a molecular dynamics refinement step.

**RMSD: Discussion** The *root-mean-square deviation (RMSD)* is the common measure of the *structural diversity* of two proteins and it measures the average distance between the atoms of two optimally aligned sets of amino acids. In our case, the full-atom reference model with real coordinates is taken from the sequences in the Protein Data Bank [6]. In order to compare this model to our prediction, we extract the backbone of the original protein and obtain a chain of $\alpha$-carbons (the atoms that are considered by our model).

| | | | Results from [14] | | | New results with BBF | | | Core | |
|---|---|---|---|---|---|---|---|---|---|---|
| Protein | CF | AA | Time | Energy | RMSD | Time | Energy | RMSD | Zone | Speedup |
| 1LE0 | | 12 | 1.3s | -9040 | 2.8 (2.6) | 0.80s | -6251 | 3.9 (3.2) | 2-11 | 1.6 |
| 1KVG | | 12 | 7.3s | -14409 | 2.7 (2.4) | 4.12s | -12661 | 4.1 (4.1) | 2-15 | 1.8 |
| 1LE3 | | 16 | 2.3s | -13653 | 3.0 (2.7) | 1.53s | -11289 | 4.6 (3.4) | 3-11 | 1.5 |
| 1EDP | | 17 | 20.4s | -19389 | 4.3 (1.1) | 0.53s | -24492 | 4.0 (1.1) | 9-15 | 38.4 |
| 1PG1 | | 18 | 14.6s | -10126 | 6.0 (5.2) | 0.83s | -27016 | 4.2 (3.2) | 4-17 | 17.6 |
| 1E0N | | 27 | 7m 54s | -12029 | 5.2 (5.1) | 3m | -22308 | 6.4 (4.5) | 3-24 | 2.6 |
| 1ZDD | | 34 | 17m 25s | -22350 | 4.0 (4.0) | 1m 51s | -18455 | 5.3 (5.2) | 5-34 | 9.4 |
| 1VII | | 36 | 7h 42m | -26408 | 10.2 (7.8) | 3h 40m | -25914 | 6.0 (5.6) | 4-32 | 2.1 |
| 1VII | 0.3 | 36 | 3h58m | -28710 | 8.0 (7.4) | 22m 41s | -19181 | 8.2 (8.1) | 4-32 | 10.5 |
| 1E0M | | 37 | (*) 24h | -30163 | 8.9 (4.4) | 4h 35m | -26745 | 9.2 (6.3) | 8-29 | $\geq 5.2$ |
| 2GP8 | | 40 | 10h 39m | -29196 | 4.9 (3.5) | 1m 33s | -15187 | 6.6 (3.6) | 6-38 | 412.3 |
| 1ED0 | | 46 | 9h 38m | -38218 | 8.0 (7.2) | 1h 43m | -31565 | 6.9 (6.2) | 3-40 | 5.6 |
| 1ENH | | 54 | (*) 24h | -23373 | 9.9 (8.6) | 55m 6s | -28559 | 11.1 (9.5) | 8-52 | $\geq 26.1$ |
| 6PTI | 0.25 | 58 | (*) 48h | -42096 | 9.7 (9.4) | (*) 48h | -52258 | 8.0 (7.9) | 3-55 | 1 |
| 2IGD | 0.17 | 60 | 4h 59m | -40588 | 12.6 (11.5) | 2h 35m | -45462 | 10.6 (10.5) | 6-59 | 1.9 |
| 2ERA | 0.19 | 61 | (*) 1000s | -38138 | 11.6 (10.6) | (*) 1000s | -45006 | 11.9 (11.7) | 3-55 | 1 |
| 1SN1 | 0.25 | 63 | (*) 10h | -47121 | 8.6 (8.1) | (*) 10h | -47650 | 9.1 (9.2) | 2-51 | 1 |
| 1YPA | 0.17 | 63 | (*) 10h | -60244 | 12.9 (9.8) | (*) 10h | -45617 | 11.5 (10.5) | 12-52 | 1 |
| 1FVS | 0.15 | 72 | (**) | | | 11m 49s | -58587 | 13.1 (13.5) | 13-70 | - |
| 1B4R | 0.16 | 80 | (**) | | | 25m 47s | -78140 | 13.1 (13.1) | 2-79 | - |

**Table 1.** Computational results and comparisons.

The RMSD measure contains some intrinsic and unavoidable parts, that derive from the discretization of the backbone on the fcc lattice. To quantify how this problem reflects on the RMSD measure, we run some test of mapping on the lattice the $\alpha$-carbons of the backbone of some proteins in [6]. The placement fulfills our standard formalization of chain neighbors and allowed angles in the lattice. The RMSD errors due to the discretization range from 3.6Åand 6.6Å. These values are relatively high, even if the fcc lattice is considered one of the best known discrete approximations. These considerations stress the fact that our results are affected by this kind of systematic errors, which are inherent in the use of the fcc lattice and independent from the quality of our solution strategies. As shown in [14], it is possible to eliminate these errors by running some steps of molecular dynamics.

## 5 Comparison with other heuristics

In this section, we explore the impact of the different solvers on the performance of our protein folding problem solution. In particular, we tested the library CLP($\mathcal{FD}$) of SICStus Prolog 3.12.0 [9] and the libraries *ic* and *branch_and_bound* of ECLiPse 5.8 [10]. The tests have been performed using the optimized constraint system described in the previous sections. The aim is to compare the features of the two constraint solvers and, at the same time, understand which search strategy fits better to the problem.

As first test, we compared the efficiency of the built-in branch-and-bound solver. We coded the same program in SICStus and ECLiPSe—in the first case using the `minimize` option offered by the built-in `labeling` predicate, that starts a branch and bound search; for ECLiPSe we invoked a `complete` search of *ic* as parameter of the `bb_min` predicate in the *branch_and_bound* library. On average and uniformly, the SICStus solver performs 12.3 times faster than the ECLiPSe solver.

On the other hand, ECLiPSe offers a number of additional built-in search heuristics, that can be selected when solving a minimization problem. We tested the following built-in

strategies: the *Limited Discrepancy Search (LDS)*, the *Bounded Backtracking Search (BBS)*, and the *Depth Bounded Search (DBS)*—together with LDS.

In the Tables in Figure 3 we report the results obtained from benchmarks for different ECLiPSe heuristics. In these experiments we did *not* use the BBF strategy. Left Table compares SICStus to the LDS and the BBS strategies. Right Table focuses on the DBS strategy. Even if the SICStus solver is faster than ECLiPSe, it does not offer a comparable selection of built-in search strategies. Our pruning heuristic is implemented at a very high level, and thus not as efficient as possible. Nevertheless, it performs better in terms of time and best solution found. None of the tested ECLiPSe strategies is able to produce faster *or* better results than our heuristic in SICStus.

| | 1PG1 (54 Vars) | | 1E0N (81 Vars) | |
|---|---|---|---|---|
| Heuristic | Time | Energy | Time | Energy |
| SICStus | 0.73 | -27016 | 78.0 | -25493 |
| Lds(0) | 0.22 | -20767 | No | No |
| Lds(1) | 1.79 | -21412 | 11.17 | -13715 |
| Lds(2) | 8.8 | -27082 | 93.0 | -14958 |
| Bbs(10) | 0.3 | -20676 | 1.3 | -12017 |
| Bbs(100) | 2.17 | -22253 | 5.6 | -12017 |
| Bbs(1000) | 40.8 | -27853 | 75.0 | -17221 |

| 1E0N | Y=0 | | Y=1 | | Y=2 | |
|---|---|---|---|---|---|---|
| | Time | Energy | Time | Energy | Time | Energy |
| X=1 | No | No | 21.5 | -13715 | 166 | -15382 |
| X=2 | No | No | 30.3 | -12688 | 219 | -15382 |
| X=3 | No | No | 45.9 | -14052 | 326 | -18046 |

**Fig. 3.** ECLiPSe times (in seconds)

One of the problems we observed is that the ECLiPSe heuristics do not properly fit the problem. Conceptually, we would like to explore the space considering that:

- small changes in the protein folding are not relevant to the global energy—which is the opposite of what the LDS strategy does;
- once a solution is found, it is likely that another interesting solution lies in a complete different part of the search tree—which is in contrast to the BBS strategy, that performs a fixed number of backtracks, and keeps the search in the proximity of the solution found;
- the complete exploration of the first levels of the tree (as done in DBS) is time consuming, and a more selective heuristic should be employed.

Nevertheless, the experiments performed with ECLiPSe highlight that some of the search strategies (e.g., the BBS strategy) can produce solutions that are suboptimal in terms of global energy, but in a significantly shorter period of time. These considerations suggest also that a more efficient search could be performed on a specific solver in which we handle the search tree at low level and implement some new ad-hoc heuristics.

## 6 A Parallel Solution

In this section we provide an overview of a parallel scheme we designed to solve the protein folding problem. The overall parallelization scheme has been designed as a customization of general or-parallelism techniques [16] to the structure of the problem at hand. The parallel scheme builds on the constraint structure described in the previous sections. The search for solutions is divided among a number of processors, and the search tree is fragmented into subtrees (called *tasks*) and distributed for parallel exploration.

There are two main issues related to the task assignment. The first is that the tasks should be *uniformly* distributed among processors during the execution; this is easier if the number of tasks is large. The second issue relates to constraint propagation and pruning of the search tree through problem-dependent heuristics, that are more effective when applied to large

search trees—i.e., fewer tasks. Hence, the task scheduling strategy should strike a proper balance between these two conflicting requirements.

### 6.1 Overview of the System

The system is composed of three components: a *loader*, a *scheduler*, and a set of *clients*. Figure 4 shows the components and the main interactions. The loader is a C program, in charge of creating the communication channels (realized using shared memory segments) between the scheduler and the clients. In addition, the loader is in charge of launching both the scheduler and the clients, as parallel processes. The system we developed makes use of a centralized scheduling mechanism. The scheduler is also a C program, that handles the dynamic distribution of tasks to the clients, and implements strategies for load balancing. Each client is a CLP program, that explores the subtree (i.e., task) assigned to the client by the scheduler. When the task is exhausted, the client will notify the scheduler that it is ready to receive an additional task.

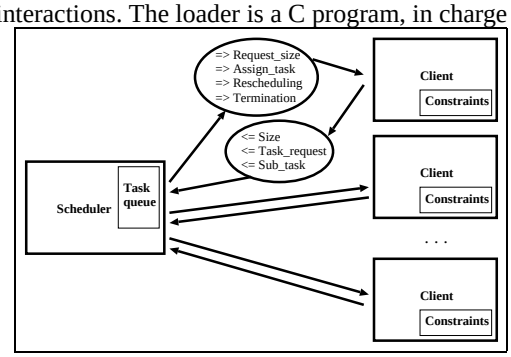



**Fig. 4:** The parallel system

### 6.2 Scheduling and Communication

The centralized scheduler implements a *direct scheduling* strategy. It relies on a static partitioning of the search tree, performed according to user defined parameters. During direct scheduling, tasks are assigned to clients upon request.

The scheduler determines the initial pool of tasks to be assigned (*task queue* in Fig. 4)—according to a static expansion of a user-specified number of levels (`Levels`) of the search tree. Since the scheduler is a C program, it does not have access to the collection of constraints; thus, the initial pool of tasks is generated by the clients, and retrieved by the scheduler during the initialization phase. Here, the first client is in charge of precomputing the expansion of `Levels` levels of the search tree and of returning the result of the expansion to the scheduler. The task queue is initialized with a set of subtrees of the search tree, all with roots at the same depth in the tree (`Level`). Each task is described by the list of nodes in the branch that connects the root of the search tree to the root of the task subtree. The scheduler assigns a task to a client whenever the client sends a `task_request` message. The task is assigned to the client by communicating the list of nodes describing it. The message that brings the task to the client is called `Assign_task`.

Due to the irregular structure of the search subtrees—because of the pruning performed by the constraint propagation process and by the heuristics—it is necessary to provide load balancing mechanisms. These mechanisms are employed when the scheduler has an empty task queue, and there is a mix of active and idle clients in the system. The purpose of load balancing is to dynamically generate new, smaller tasks, by further partitioning some of the tasks that are still active. Such smaller tasks can then be reassigned to the idle clients. The load balancing is implemented by a *rescheduling* procedure, activated by the scheduler every time there is at least one idle client and the task queue is empty. In this case, the scheduler selects, with a `Rescheduling` message, the client that has the estimated highest load of work. Due to lack of space, we do not provide technical details on this procedure.

The scheduler takes also care of detecting global termination. Termination occurs when the task queue is empty and task requests have been submitted from all clients. In such a situation, the scheduler returns a `Termination` message to each client.

### 6.3 Structure of the Client

Each client is a CLP program that implements the process of solving the constraints on a given subset of the search space—i.e., a subset of the domains of the variables in the problem. When launched, a client imports protein data, defines variable domains and applies constraints. After the initial loading, the first client communicates back to the scheduler the list of partial assignments (`Tasks`) obtained from the expansion of the search tree for `Levels` variables. After that, each client starts a loop that $(i)$ delivers a `Task_request` to the scheduler, $(ii)$ waits for the assignment of a task (`Assign_task`) and $(iii)$ executes the task. The processing of a task is based on the CLP scheme described earlier. During each task execution, the client checks for eventual requests for `Rescheduling`—realized by breaking down the labeling process into labeling of smaller lists. If the request is received, the client stops the execution and communicates all the subtasks left to the scheduler. The client stops its execution when it receives the special termination signal.

### 6.4 Implementation Details

The parallel system has been implemented using a combination of C programming and Constraint Logic Programming (specifically SICStus Prolog 3.12.0 [27]). The activation of the processes implementing the scheduler and the clients is accomplished by standard `fork` calls of C issued by the loader. Separate processes are created for the scheduler and for the different clients.

The communication is realized using shared memory. The clients are written in Prolog, and encapsulate some low level C routines to access the shared memory, for a fast interaction between processes. The message exchange between clients and the scheduler is realized using shared memory queues.

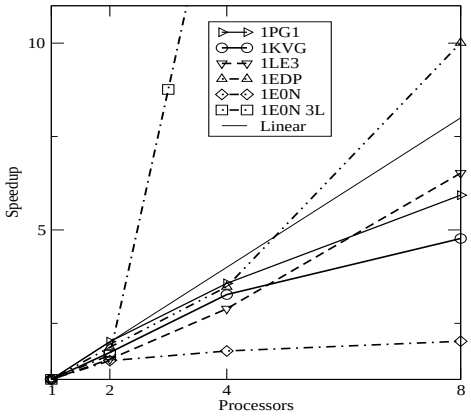
During the execution of Prolog predicates, we also relied on low-level C routines to maintain an efficient representation of the current state of execution of the task. This allows us to maintain a simplified representation of the state of the execution across the branches of the subtree (which are explored via backtracking by the Prolog system) and it simplifies communication with the scheduler during load balancing.

### 6.5 Experimental results

In this section, we report the experimental results obtained from the execution of our parallel system. The experiments have been performed using a HP RP8400 NUMA architecture, with 14 PA-RISC processors, 8GB RAM, and running HP-UX 11.1.0.

Figure 6.5 plots and shows the times, in seconds, of the parallel execution of the program, using the BBF strategy, no rescheduling and up to $8$ parallel clients. We have parallelized the code on numbers of processors that are powers of 2. For each protein, the first $4$ `Levels` of the tree are fully expanded. For the protein 1E0N we also included the results obtained using a value of `Levels` equal to $3$—as this is sufficient to generate a sufficient number of tasks. The last column reports the execution time, on one CPU of the parallel machine, of the code from [14].

The performance results show that the scalability factors are strongly dependent on the specific problem. We discuss here some of the reasons that generate this behavior. First of all, the subdivision of the search tree into tasks does not imply an equal partition of the workload. In fact, since constraints and heuristics are applied to the trees during the search, their effect




| | Processors | | | | [14] |
|---|---|---|---|---|---|
| Protein | 1 | 2 | 4 | 8 | 1 |
| 1PG1 | 12.51 | 6.21 | 3.5 | 2.11 | 48.7 |
| 1KVG | 13.03 | 7.56 | 3.99 | 2.73 | 24.9 |
| 1LE3 | 16.36 | 10.63 | 5.66 | 2.51 | 8.2 |
| 1EDP | 11.91 | 6.4 | 3.42 | 1.19 | 67.8 |
| 1E0N | 239.9 | 160 | 136.2 | 118.5 | 598.4 |
| 1E0N 3 (`Levels`) | 2249 | 1367 | 141.72 | 82 | 598.4 |

**Fig. 5.** Parallel Execution Time without rescheduling (in seconds)

on a local portion of the search tree can result in a different pruning compared to the same subtree during a sequential search. This effect can have appreciated side-effects: for instance, the superlinear speed-up obtained in the cases of 1EDP and 1E0N (3 levels).

Let us observe that, although the sequential speed up for some proteins is of the order of 40 (e.g., for 1EDP—see Table 1) and the parallel speed up is for some proteins of the order of 10 with eight processors (e.g., for 1EDP), the combined speed up is only of the order of 60 (instead of 400). The reason is the overheads associated to the management of parallel tasks—e.g., task exchange, communication costs, and the cost of resetting the constraint store when a new task is acquired. The latter cost, that seems to be the most significant one, could be removed by introducing a more ad-hoc handling of constraints—i.e., bypassing some of the constraint-handling functionalities of SICStus.

Note that clients share their intermediate results by placing them in shared memory, where it becomes accessible to every other client. In particular, the agents can share their current best solution, effectively parallelizing the branch-and-bound process.

Let us assume that the tasks $T_1, \ldots T_n$ are explored in that order in the equivalent sequential algorithm. In the parallel case, every task can take advantage of an earlier-found new bound for the search. For example, $T_i$ and $T_j$, with $i < j$, are explored in parallel, and a new best solution is found in $T_j$. This allows a client to prune $T_i$ as well, resulting in a speedup in the search. On the other hand, though, the symmetric case can arise: $T_j$ can be explored extensively because of a late discover of a bound in $T_i$. In the sequential case $T_j$ would have been pruned from the very beginning, because $T_i$ would have been completed before even starting the exploration of $T_j$. Thus, the partitioning can dramatically affect the search (see, e.g., protein 1E0N). We plan to investigate in the future the design of problem-dependent partitionings, in order to take advantage of the parallel sharing of information.

Let us conclude by observing that, in our experiments, the rescheduling procedure has been rarely effective in improving performance. This is due to the poor interaction between the fairly "sequential" way of rescheduling tasks and the more sophisticated exploration imposed by our heuristic strategies. Work is in progress to adapt rescheduling to better match our search strategies.

### 6.6 Scalability on Macro Blocks

When dealing with large proteins, it is common to encounter situations where the conformations of various subsequences are already known (e.g., by homology). Thus, the problem of predicting the structure of the whole protein is conceptually equivalent to predicting the placement of few rigid macro blocks, that are linked together by some coils. These ideas could be exploited by our prediction tool.

We defined the following example, to demonstrate that the current model can be feasibly used to attack larger proteins. We use the sequence XYZ, where X and Z are known protein sequences and Y is a fixed-length linking coil of non-interacting amino acids. The idea is to predict the structure of the sequence XYZ, using as input some strong constraints (i.e. Euclidean distances between every pair of amino acids) derived from X and Z.

In our tests, the execution times grow according to the length of the coil Y. As expected, for proteins of this size but with partially known structure, the times are significantly lower than a prediction that uses only the secondary structure information. This allows our system to handle larger protein complexes.

## 7 Conclusion and Future Work

In this paper, we provided a formalization of the protein folding problem on face-centered cubic lattice structures. The formalization has been transformed in a constraint system, and solved using constraint solving over finite domains. We analyzed different ways of organizing the constraint structure, and different heuristics and search strategies to solve them. We presented and tested a new search heuristic (Bounded Block Fails), well suited for this problem. We also provided a way to parallelize the process of exploring the search space, allowing concurrent constraint solvers to cooperate in the search of an optimal folding.

The results collected from the different approaches to the problem (sequential search strategies, parallel implementations, different implementations of solvers) converge on the need of a new dedicated and efficient solver. The analysis in Section 5 suggests that the only way to significantly improve our framework is to access the search tree at a lower level, and to implement new heuristics more suitable to the problem. Since this is not allowed by the current implementations of SICStus and ECLiPSe, we plan to develop our own ad-hoc constraint solver. The new solver will be dedicated to problem on lattices (and thus more efficient) and will implement ad-hoc search strategies. The new solver, applied to the protein folding problem of fcc, is expected to allow us to tackle larger problems—the final goal is to manage proteins composed of 500 amino acids. As shown in Table 1, currently we can solve (without using scalability—Section 6.6) proteins of length in the order of 80 amino acids.

We also plan to continue the development of the parallel solution; in particular, we intend to develop rescheduling strategies that better match the heuristics employed by the sequential system, allowing for a more effective load balancing and scalability.

**Acknowledgments** We thank Federico Fogolari, for his guidance and comments. The research has been partially supported by NSF grants HRD-0420407 and EIA-0220590, by the MIUR Project *Sybilla*, and by GNCS 2005.

## References

1. R. Agarwala et al. Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the HP model. *J. of Computational Biology*, pages 275–296, 1997.
2. K. R. Apt. *Principles of constraint programming*. Cambridge University press, 2003.

3. R. Backofen. The protein structure prediction problem: A constraint optimization approach using a new lower bound. *Constraints*, 6(2–3):223–255, 2001.
4. R. Backofen and S. Will. Fast, constraint-based threading of HP sequences to hydrophobic cores. *Int. Conf. on Principle and Practice of Constraint Programming*, 494–508, 2001. Springer Verlag.
5. R. Backofen and S. Will. A Constraint-Based Approach to Structure Prediction for Simplified Protein Models that Outperforms Other Existing Methods. *ICLP*, 2003, Springer Verlag.
6. H. M. Berman et al. The protein data bank. *Nucleic Acids Research*, 28:235–242, 2000. `http://www.rcsb.org/pdb/`.
7. M. Berrera, H. Molinari, and F. Fogolari. Amino acid empirical contact energy definitions for fold recognition in the space of contact maps. *BMC Bioinformatics*, 4(8), 2003.
8. R. Bonneau and D. Baker. Ab initio protein structure prediction: progress and prospects. *Annu. Rev. Biophys. Biomol. Struct.*, 30:173–89, 2001.
9. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. *PLILP*, Springer Verlag, 1997.
10. A. M. Cheadle et al. ECLiPSe: An Introduction. Technical Report IC-Parc 03–1, IC-Parc, 2003.
11. D. Clark, J. Shirazi, and C. Rawlings. Protein topology prediction through constraint-based search and the evaluation of topological folding rules. *Protein Engineering*, 4:752–760, 1991.
12. P. Clote and R. Backofen. *Computational Molecular Biology: An Introduction*. John Wiley & Sons, 2001.
13. P. Crescenzi et al. On the complexity of protein folding. In *Proc. of STOC*, pages 597–603, 1998.
14. A. Dal Palù, A. Dovier, and F. Fogolari. Constraint logic programming approach to protein structure prediction. *BMC Bioinformatics*, 5(186), 2004.
15. A. Dal Palù, A. Dovier, and F. Fogolari. Protein folding in $CLP(\mathcal{FD})$ with empirical contact energies. In *Recent Advances in Constraints*, Springer Verlag, 2004.
16. G. Gupta, E. Pontelli, M. Carlsson, M. Hermegildo, K. Ali. Parallel Execution of Prolog: a Survey. *ACM TOPLAS*, 23(4):472–602, 2001.
17. F. Fogolari et al. Modeling of polypeptide chains as C-$\alpha$ chains, C-$\alpha$ chains with C-$\beta$, and C-$\alpha$ chains with ellipsoidal lateral chains. *Biophysical Journal*, 70:1183–1197, 1996.
18. S. Forman. *Torsion Angle Selection and Emergent Non-local Secondary Structure in Protein Structure Prediction*. PhD thesis, U. of Iowa, 2001.
19. W. Hart and S. Israil. Fast protein folding in the hydrophobic-hytrophilic model within three-eighths of optimal. *J. of Computational Biology*, pages 53–96, 1996.
20. S. Miyazawa and R. L. Jernigan. Residue-residue potentials with a favorable contact pair term and an unfavorable high packing density term, for simulation and threading. *J. of Molecular Biology*, 256(3):623–644, 1996.
21. A. Newman. A New Algorithm for Protein Folding in the HP Model. In *Symposium on Discrete Algorithms* . Springer Verlag, 2002.
22. Committee on Mathematical Challenges from Computational Chemistry. National Research Council, 1995.
23. G. Raghunathan and R. L. Jernigan. Ideal architecture of residue packing and its observation in protein structures. *Protein Science*, 6:2072–2083, 1997.
24. B. Rost and C. Sander. Prediction of protein secondary structure at better than 70% accuracy. *J. Mol. Biol.*, 232:584–599, 1993.
25. J. Skolnick and A. Kolinski. Computational studies of protein folding. *Computing in Science and Engineering*, 3(5):40–50, 2001.
26. J. Skolnick and A. Kolinski. Reduced models of proteins and their applications. *Polymer*, 45:511–524, 2004.
27. Swedish Institute for Computer Science. Sicstus Prolog. `www.sics.se/sicstus/`.
28. L. Toma and S. Toma. Folding simulation of protein models on the structure-based cubo-octahedral lattice with contact interactions algorithm. *Protein Science*, 8:196–202, 1999.
29. M.L.G. William and D. Harvey. Limited discrepancy search. *IJCAI*, pages 607–615, Morgan Kaufmann, 1995.

# A comparison of CLP(FD) and ASP solutions to NP-complete problems*

Agostino Dovier[1], Andrea Formisano[2], and Enrico Pontelli[3]

[1] Univ. di Udine, Dip. di Matematica e Informatica. dovier@dimi.uniud.it
[2] Univ. di L'Aquila, Dip. di Informatica. formisano@di.univaq.it
[3] New Mexico State University, Dept. Computer Science. epontell@cs.nmsu.edu

**Abstract.** This paper presents experimental comparisons between declarative encodings of various computationally hard problems in both Answer Set Programming (ASP) and Constraint Logic Programming (CLP) over finite domains. The objective is to identify how the solvers in the two domains respond to different problems, highlighting strengths and weaknesses of their implementations and suggesting criteria for choosing one approach versus the other. Ultimately, the work in this paper is expected to lay the ground for transfer of concepts between the two domains (e.g., suggesting ways to use CLP in the execution of ASP).

## 1 Introduction

The objective of this work is to experimentally compare the use of two distinct logic-based paradigms in solving computationally hard problems. The two paradigms considered are *Answer Set Programming (ASP)* [2] and *Constraint Logic Programming over Finite Domains (CLP(FD))* [16]. The motivation for this investigation arises from the successful use of both paradigms in dealing with various classes of combinatorial problems, and the need to better understand their respective strengths and weaknesses. Ultimately, we hope this work will provide indication for integration and cooperation between the two paradigms (e.g., along the lines of [6]).

It is well-known [14, 2] that, given a propositional normal logic program $P$, deciding whether or not it admits an *answer set* [9] is a NP-complete problem. As a consequence, any NP-complete problem can be encoded as a propositional normal logic program under answer set semantics. Answer-set solvers [19] are programs designed for computing the answer sets of normal logic programs; these tools can be seen as theorem provers, or model builders, enhanced with several built-in heuristics to guide the exploration of the solution space. Most ASP solvers rely on variations of the Davis-Putnam-Longeman-Loveland procedure in their computations. Such solvers are often equipped with a front-end that transforms a collection of non-propositional normal clauses (with limited use of function symbols) in a *finite* set of ground instances of such clauses. Some solvers provide classes of *optimization statements*, used to select answer sets that maximize or minimize an objective function dependent on the content of the answer set.

An alternative framework, frequently adopted to handle NP-complete problems, is *Constraint Logic Programming over Finite Domains* [10, 16]. In this context, a finite

* This work is partially supported by GNCS2005 project on constraints and their applications.

domain of objects (typically integers) is associated to each variable in the problem specification, and the typical constraints are literals of the forms $s = t$, $s \neq t$, $s < t$, $s \leq t$, where $s$ and $t$ are arithmetic expressions. Encodings of NP-complete problems and of search strategies are very natural and declarative in this framework. Indeed, a large literature has been developed presenting applications of CLP(FD) to a variety of search and optimization problems [16].

In this paper, we report the outcomes of experiments aimed at comparing these two declarative approaches in solving combinatorial problems. We address a set of computationally hard problems—in particular, we mostly consider decision problems known to be NP-complete. We formalize each problem, both in CLP(FD) and in ASP, by taking advantage of the specific features available in each logical frameworks, attempting to encode the various problems in the most declarative possible way. In particular, we adopt a *constraint-and-generate* strategy for the CLP code, while in ASP we exploit the usual *generate-and-test* approach. Wherever possible, we make use of solutions to these problems that have been presented and widely accepted in the literature.

With this work we intend to develop a bridge between these two logic-based frameworks, in order to emphasize the strengths of each approach and in favor of potential cross-fertilizations. This study also complements the system benchmarking studies, that have recently appeared for both CLP(FD) systems [8, 17] and ASP solvers [1, 13, 11].

## 2 The Experimental Framework

In order to conduct our experiments, we selected one CLP(FD) implementation and two ASP-solvers. The CLP programs have been designed for execution by SICStus Prolog 3.11.2 (using the library `clpfd`)—though the code is general enough to be used on different platforms (e.g., ECLiPSe). The choice of SICStus has been suggested by its good performance (better than ECLiPSe on some of the benchmarks at hand). The ASP programs have been designed to be processed by *lparse*, the grounding preprocessor adopted by both the SModels (version 2.28) and the CModels (version 3.03) systems [19]. The CModels system makes use of a SAT solver to compute answer sets—in our experiments we selected the default underlying SAT solver, namely mChaff.

We focused on well-known computationally-hard problems. Among them: Graph $k$-coloring (Section 3), Hamiltonian circuit (Section 4), Schur numbers (Section 5), protein structure prediction on a 2D lattice [3] (Section 6), planning in a block world (Section 7), and generalized Knapsack (Section 8). Observe that, while some of the programs have been drawn from the best proposals appeared in the literature, others are novel solutions, developed in this project (e.g., the ASP implementation of the PF problem and the planning implementation in CLP(FD)).

In the remaining sections of this paper, we describe the solutions to the various problems and report the results from the experiments. All the timing results, expressed in seconds, have been obtained by measuring only the time needed for computing the first solution, if any (CPU usage time)—thus, we ignore the time spent in reading the input, as well as the time spent to ground the program, in the case of the ASP solvers. We used the `runtime` option to measure the time in CLP(FD), that does not account for the time spent for garbage collection and for system calls. All tests have been performed on a PC (P4 processor 2.8 GHz, and 512 MB RAM memory) running Linux kernel 2.6.3. Complete codes and results are reported in `www.di.univaq.it/~formisano/CLPASP`.

## 3 $k$-Coloring

The $k$-coloring problem computes the coloring of a graph using $k$ colors. The main source of case studies adopted in our experiments is the repository of "Graph Coloring and its Generalizations" [20], which provides a rich collection of instances, mainly aimed at benchmarking algorithms and approaches to graph problems. Let us describe the two formalizations of $k$-coloring.

**CLP(FD):** In this formulation, we assume that the input graph is represented by a single fact of the form `graph([1,2,3],[[1,2],[1,3],[2,3]])`, where the first argument represents the list of nodes (a list of integers), while the second argument is the set of edges. This is a possible constrain-and-generate CLP(FD)-encoding of $k$-coloring:

```
coloring(K, Output) :- graph(Nodes, Edges),
      create_output(Nodes, Colors, Output), domain(Colors, 1, K),
      different(Output, Edges), labeling([ff], Colors).
create_output([],[],[]).
create_output([N | Nodes], [C|Colors], [N-C|Output]) :-
      create_output(Nodes, Colors, Output).
different(_, []).
different(Output, [[A,B]|R]) :- member(A-CA, Output),
      member(B-CB, Output), CA #\= CB, different(Output, R).
```

In this program, `Output` is intended to be a list of pairs of variables `N-C` where, for each node `N` we introduce a color variable `C` in the range `1...K`. The predicate `different` imposes disequality constraints between variables related to adjacent nodes. We used the `ff` options of `labeling` since it offered the best results for this problem.

**ASP:** Regarding the ASP encoding of $k$-coloring we adopt a different representation for graphs. Nodes are represented, as before, by natural numbers. Edges are rendered by facts, as in the following instance:

```
node(1..138). edge(1,36). edge(2,45). ... edge(138,7). edge(138,36).
```

A natural ASP encoding of the $k$-coloring problem is:

```
(1)      col(1..k).
(2)      :- edge(X,Y), col(C), color(X,C), color(Y,C).
(3)      1{color(X,C): col(C)}1 :- node(X).
```

Rule `(1)` states that there are `k` colors (the parameter is a constant to be initialized in the grounding stage). The ASP-constraint `(2)` asserts that two adjacent nodes cannot have the same color, while `(3)` states that each node has exactly one color. Note that, by using domain restricted variables, the single ASP-constraint `(2)` states the property that two adjacent nodes cannot have the same color for all edges $\langle X, Y\rangle$. The same property is described by the predicate `different` in the CLP(FD) code, but in that case a recursive definition is required. This fact shows a common situation that will be observed again in the following sections: ASP often permits a significantly more compact encoding of the problem w.r.t. CLP(FD).

**Results:** We tested the above programs on more than one hundred instances drawn from [20]. Such instances belong to various classes of graphs which come from different sources in the literature. Table 1 shows an excerpt of the results we obtained for $k$-coloring with $k = 3, 4, 5$. The columns report the time (in seconds) using the

| Instance | | 3-colorability | | | | 4-colorability | | | | 5-colorability | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Graph | V×E | | SModels | CModels | CLP(FD) | | SModels | CModels | CLP(FD) | | SModels | CModels | CLP(FD) |
| 1-FullIns_5 | 282 × 3247 | N | 1.06 | 0.15 | 0.10 | N | – | 0.23 | 2.90 | N | – | 107.78 | – |
| 4-FullIns_4 | 690 × 6650 | N | 0.94 | 0.29 | 0.46 | N | 2.20 | 0.35 | 1.98 | N | 10.02 | 0.42 | – |
| 5-FullIns_4 | 1085 × 11395 | N | 1.72 | 0.47 | 1.26 | N | 4.67 | 0.57 | 3.58 | N | 23.79 | 0.70 | – |
| 3-FullIns_5 | 2030 × 33751 | N | 5.92 | 1.23 | 7.24 | N | 21.31 | 1.51 | 13.69 | N | – | 1.96 | – |
| 4-FullIns_5 | 4146 × 77305 | N | 15.11 | 2.69 | 33.44 | N | 69.30 | 3.37 | 42.53 | N | 414.93 | 4.19 | – |
| 3-Insertions_3 | 56 × 110 | N | 4.28 | 4.16 | 1281.18 | Y | 0.03 | 0.04 | <0.01 | Y | 0.04 | 0.04 | <0.01 |
| 4-Insertions_3 | 79 × 156 | N | 328.25 | 1772.14 | – | Y | 0.05 | 0.04 | <0.01 | Y | 0.06 | 0.05 | <0.01 |
| 2-Insertions_4 | 149 × 541 | N | 1.20 | 0.15 | 2.04 | ? | – | – | – | Y | 0.25 | 0.07 | 0.01 |
| 4-Insertions_4 | 475 × 1795 | N | – | 1443.33 | – | ? | – | – | – | Y | 3.402 | 0.32 | – |
| 2-Insertions_5 | 597 × 3936 | N | 45.08 | 0.50 | 6.97 | ? | – | – | – | ? | – | – | – |
| DSJR500.1 | 500 × 3555 | N | 0.53 | 0.18 | 0.18 | N | 2.78 | 0.21 | 0.18 | N | – | 0.26 | 0.19 |
| DSJC500.1 | 500 × 12458 | N | 2.19 | 0.45 | 0.64 | N | 12.30 | 0.57 | 0.76 | N | 6328.45 | 6.21 | 46.55 |
| DSJR500.5 | 500 × 58862 | N | 25.76 | 1.81 | 2.97 | N | 175.63 | 2.26 | 2.98 | N | 971.46 | 2.71 | 3.09 |
| DSJC500.5 | 500 × 62624 | N | 28.29 | 1.92 | 3.15 | N | 376.35 | 2.36 | 3.19 | N | 2707.64 | 2.84 | 3.47 |
| DSJR500.1c | 500 × 121275 | N | 84.19 | 3.66 | 6.07 | N | 1083.17 | 4.54 | 6.18 | N | 9881.35 | 5.50 | 6.19 |
| DSJC500.9 | 500 × 224874 | N | 74.44 | 3.39 | 5.67 | N | 543.02 | 4.29 | 5.67 | N | 3146.96 | 5.09 | 5.77 |
| DSJC1000.1 | 1000 × 49629 | N | 12.99 | 1.61 | 5.01 | N | 241.43 | 2.02 | 5.06 | N | – | 3.61 | – |
| flat300_20_0 | 300 × 21375 | N | 6.39 | 0.68 | 0.63 | N | 86.91 | 0.84 | 0.64 | N | 1555.37 | 1.08 | 0.69 |
| flat300_26_0 | 300 × 21633 | N | 6.45 | 0.70 | 0.65 | N | 131.91 | 0.87 | 0.67 | N | 3711.80 | 1.13 | 0.69 |
| flat300_28_0 | 300 × 21695 | N | 6.51 | 0.70 | 0.65 | N | 34.76 | 0.86 | 0.69 | N | 322.99 | 1.02 | 0.67 |
| fpsol2.i.1 | 496 × 11654 | N | 2.75 | 0.41 | 0.77 | N | 24.98 | 0.52 | 0.77 | N | 205.12 | 0.61 | 0.84 |
| fpsol2.i.2 | 451 × 8691 | N | 1.92 | 0.33 | 0.53 | N | 16.66 | 0.40 | 0.54 | N | 279.96 | 0.52 | 0.55 |
| fpsol2.i.3 | 425 × 8688 | N | 1.91 | 0.32 | 0.5 | N | 16.63 | 0.40 | 0.51 | N | 277.91 | 0.49 | 0.51 |
| gen200_p0.9_44 | 200 × 17910 | N | 5.53 | 0.57 | 0.36 | N | 30.87 | 0.70 | 0.36 | N | 306.81 | 0.84 | 0.38 |
| gen200_p0.9_55 | 200 × 17910 | N | 5.54 | 0.57 | 0.36 | N | 39.56 | 0.71 | 0.36 | N | 287.14 | 0.85 | 0.38 |
| gen400_p0.9_55 | 400 × 71820 | N | 38.91 | 2.19 | 2.88 | N | 656.07 | 2.68 | 2.89 | N | 4892.74 | 3.24 | 2.93 |
| gen400_p0.9_65 | 400 × 71820 | N | 39.02 | 2.16 | 2.88 | N | 275.33 | 2.67 | 2.87 | N | 1563.52 | 3.22 | 2.92 |
| gen400_p0.9_75 | 400 × 71820 | N | 38.87 | 2.17 | 2.88 | N | 270.12 | 2.70 | 2.89 | N | 1608.19 | 3.22 | 2.94 |
| inithx.i.1 | 864 × 18707 | N | 4.92 | 0.65 | 2.28 | N | 57.15 | 0.81 | 2.29 | N | 415.41 | 1.00 | 2.32 |
| inithx.i.2 | 645 × 13979 | N | 3.50 | 0.50 | 1.28 | N | 34.19 | 0.63 | 1.28 | N | 268.87 | 0.83 | 1.31 |
| inithx.i.3 | 621 × 13969 | N | 3.50 | 0.50 | 1.22 | N | 34.14 | 0.64 | 1.24 | N | 268.36 | 0.80 | 1.26 |
| le450_5a | 450 × 5714 | N | 0.85 | 0.24 | 0.26 | N | 9.06 | 0.29 | 0.28 | Y | 190.38 | 12.30 | 5.29 |
| le450_5b | 450 × 5734 | N | 0.85 | 0.24 | 0.29 | N | 7.77 | 0.29 | 0.3 | Y | 5146.86 | 0.98 | 0.48 |
| le450_5c | 450 × 9803 | N | 1.64 | 0.37 | 0.44 | N | 9.98 | 0.46 | 0.44 | Y | 217.77 | 0.70 | 0.03 |
| le450_5d | 450 × 9757 | N | 1.64 | 0.36 | 0.44 | N | 10.29 | 0.44 | 0.47 | Y | 530.63 | 0.60 | 0.08 |
| mulsol.i.1 | 197 × 3925 | N | 0.58 | 0.17 | 0.10 | N | 2.80 | 0.20 | 0.10 | N | 19.07 | 0.24 | 0.11 |
| mulsol.i.2 | 188 × 3885 | N | 0.57 | 0.17 | 0.10 | N | 2.83 | 0.20 | 0.09 | N | 31.25 | 0.24 | 0.11 |
| mulsol.i.3 | 184 × 3916 | N | 0.58 | 0.16 | 0.09 | N | 2.86 | 0.20 | 0.10 | N | 31.93 | 0.25 | 0.10 |
| mulsol.i.4 | 185 × 3946 | N | 0.58 | 0.17 | 0.10 | N | 2.92 | 0.20 | 0.10 | N | 32.83 | 0.25 | 0.11 |
| mulsol.i.5 | 186 × 3973 | N | 0.59 | 0.17 | 0.10 | N | 2.93 | 0.20 | 0.09 | N | 33.02 | 0.26 | 0.11 |
| wap05a | 905 × 43081 | N | 11.39 | 1.38 | 2.96 | N | 62.81 | 1.73 | 2.96 | N | 949.66 | 2.07 | 2.96 |
| wap06a | 947 × 43571 | N | 11.63 | 1.42 | 3.25 | N | 62.70 | 1.75 | 3.24 | N | 1326.84 | 2.13 | 3.26 |
| wap07a | 1809 × 103368 | N | 31.98 | 3.28 | 15.14 | N | 191.06 | 4.12 | 15.14 | N | 2861.64 | 4.99 | 15.19 |
| wap08a | 1870 × 104176 | N | 32.07 | 3.31 | 16.17 | N | 192.54 | 4.15 | 16.22 | N | 3604.96 | 5.08 | 16.18 |

**Table 1.** Graph $k$-coloring ('–' denotes no answer in at least 30 minutes of CPU-time— '?' means that none of the three solvers gave an answer.)

three systems; the first column of each result indicates whether a solution exists for the problem instance.

A particular class of graph coloring problems listed in [20] originates from encoding a generalized form of the $N$-queens problem. Graphs for the $M$-$N$-queen problems are obtained as follows. The nodes correspond to the cells of a $N \times N$ chessboard. Two nodes $u$ and $v$ are connected by an (undirected) edge if a queen in the cell $u$ attacks the cell $v$. Solving the $M$-$N$-queens problem consists of determining whether or not such graph is $M$-colorable. In the particular case where $M = N$, this is equivalent to finding $N$ independent solutions to the classical $N$-queens problem. Observe that, for $M < N$ the graph cannot be colored. We run a number of tests on this specific class of graphs. Table 2 lists the results obtained for $N = 5, \ldots, 11$ and $M = N - 1, N, N + 1$. For the sake of completeness, we also experimented, on these instances, using the library `ugraphs` of SICStus Prolog (a library independent from the library `clpfd`), where the `coloring/3` predicate is provided as a built-in feature. `ugraphs` is slower than CLP(FD) for small instances, however, it finds solutions in acceptable time for some larger instances, whereas CLP(FD) times out.

| Instance | | Solvability for $M = N - 1$ | | | | Solvability for $M = N$ | | | | Solvability for $M = N + 1$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | $V \times E$ | SModels | CModels | CLP(FD) | ugraph | SModels | CModels | CLP(FD) | ugraph | SModels | CModels | CLP(FD) | ugraph |
| 5 | $25 \times 320$ | N | 0.06 | 0.07 | 0.01 | <0.01 | Y | 0.06 | 0.07 | <0.01 | <0.01 | Y | 0.07 | 0.08 | <0.01 | <0.01 |
| 6 | $36 \times 580$ | N | 1.00 | 0.11 | 0.01 | <0.01 | N | 63.80 | 198.65 | 1.33 | 0.02 | Y | 0.66 | 0.19 | <0.01 | 0.16 |
| 7 | $49 \times 952$ | N | 341.17 | 0.20 | 0.02 | 0.03 | Y | 1.95 | 0.18 | <0.01 | 0.29 | Y | 0.54 | 14.08 | 0.02 | 0.35 |
| 8 | $64 \times 1456$ | N | – | 0.42 | 0.16 | 0.89 | N | – | – | – | 224.11 | Y | 116.50 | 1.28 | 1.04 | 807.22 |
| 9 | $81 \times 2112$ | N | – | 0.85 | 1.37 | 106.64 | ? | – | – | – | – | Y | – | – | 138.85 | 131.27 |
| 10 | $100 \times 2940$ | N | – | 3.63 | 14.53 | – | ? | – | – | – | – | ? | – | – | – | – |
| 11 | $121 \times 3960$ | N | – | 10.62 | 148.74 | – | ? | – | – | – | – | ? | – | – | – | – |

**Table 2.** The $M$-$N$-Queens problem ('–' denotes no answer in 10 min. of CPU-time).

## 4 Hamiltonian Circuit

In this section we deal with the problem of establishing whether a directed graph admits an Hamiltonian circuit. The graph representations adopted are the same as in the previous section.

**CLP(FD):** A possible CLP(FD) encoding is the following:

```
hc(N, Edges) :- length(Path, N), domain(Path, 1, N),
        make_domains(Path, 1, Edges, N),
        circuit(Path), labeling([ff], Path).
make_domains([], _, _, _).
make_domains([X|Y], Node, Edges, N) :-
    findall(Z, member([Node,Z], Edges), Successors),
    reduce_domains(N, Successors, X),
    Node1 is Node+1, make_domains(Y, Node1, Edges, N).
reduce_domains(0, _, _) :- !.
reduce_domains(N, Successors, Var) :- N>0, member(N,Successors),
    !, N1 is N-1, reduce_domains(N1, Successors, Var).
reduce_domains(N, Successors, Var) :-
    Var #\= N, N1 is N-1, reduce_domains(N1, Successors, Var).
```

We use the built-in predicate `circuit`, provided by `clpfd` in SICStus. In the literal `circuit(List)`, the `List` is a list of domain variables or integers. The goal `circuit`$([X_1, \ldots, X_n])$ constrains the variables so that the set of edges $\langle 1, X_1\rangle$, $\langle 2, X_2\rangle, \ldots, \langle n, X_n\rangle$ is an Hamiltonian circuit. The predicate `make_domains` restricts the admissible values for the variable $X_i$ to the successors of node $i$ in the graph.

**ASP:** The following program for Hamiltonian circuit comes from the ASP literature [15]:

```
(1)   1 {hc(X,Y) : edge(X,Y)} 1 :- node(X).
(2)   1 {hc(Z,X) : edge(Z,X)} 1 :- node(X).
(3)   reachable(X) :- node(X), hc(1,X).
(4)   reachable(Y) :- node(X), node(Y), reachable(X), hc(X,Y).
(5)   :- not reachable(X), node(X).
```

The description of the search space is given by rules `(1)` and `(2)`. They state that, for each node `X`, exactly one outgoing edge `(X,Y)` and one incoming edge `(Z,X)` belong to the circuit (represented by the predicate `hc`). Rules `(3)` and `(4)` define the transitive closure of the relation `hc` starting from node number `1`. The "test" phase is expressed by the ASP-constraint `(5)`, which weeds out the answer sets that do not represent solutions to the problem. Also in this case, the ASP approach permits a more compact encoding (even if in CLP(FD) we exploited the built-ins `circuit` and `findall`).

| Instance | node×edges | Hamiltonian? | | | |
|---|---|---|---|---|---|
| | | | SModels | CModels | CLP(FD) |
| hc1 | 200×1250 | Y | 2.99 | 37.59 | 0.34 |
| hc2 | 200×1250 | Y | 2.99 | 1394.15 | 0.34 |
| hc3 | 200×1250 | Y | 3.03 | 20.06 | 0.32 |
| hc4 | 200×1250 | Y | 2.98 | 93.10 | 0.34 |
| hc5 | 200×1250 | N | 1.44 | 0.22 | 0.24 |
| hc6 | 200×1250 | N | 1.44 | 0.21 | 0.10 |
| hc7 | 200×1250 | N | 1.44 | 0.20 | 0.25 |
| hc8 | 200×1250 | N | 1.44 | 0.20 | 0.26 |
| np10c | 10×90 | Y | 0.01 | 0.05 | 0.0 |
| np20c | 20×380 | Y | 0.07 | 0.82 | 0.0 |
| np30c | 30×870 | Y | 0.26 | 0.27 | 0.01 |
| np40c | 40×1560 | Y | 0.91 | 4.38 | 0.02 |
| np50c | 50×2450 | Y | 2.59 | 118.18 | 0.03 |
| np60c | 60×3540 | Y | 7.38 | 24.81 | 0.05 |
| np70c | 70×4830 | Y | 15.68 | 9.47 | 0.07 |
| np80c | 80×6320 | Y | 27.79 | 12.55 | 0.11 |
| np90c | 90×8010 | Y | 45.66 | 128.25 | 0.15 |
| 2xp30 | 60×316 | N | 0.14 | 0.02 | 0.03 |
| 2xp30.1 | 60×318 | Y | 0.18 | 4.61 | 0.02 |
| 2xp30.2 | 60×318 | Y | – | 2.69 | 5.38 |
| 2xp30.3 | 60×318 | Y | – | 2.70 | 5.38 |
| 2xp30.4 | 60×318 | N | – | – | – |
| 4xp20 | 80×392 | N | 0.24 | 0.04 | 0.04 |
| 4xp20.1 | 80×395 | N | – | 1.47 | 0.04 |
| 4xp20.2 | 80×396 | Y | 0.37 | 3.32 | 0.03 |
| 4xp20.3 | 80×396 | N | 0.24 | 2.65 | – |
| nv50a260 | 50×239 | Y | 0.12 | 0.95 | 0.01 |
| nv50a280 | 50×263 | Y | 0.13 | 0.51 | 0.02 |
| nv50a300 | 50×280 | Y | 0.14 | 0.16 | 0.02 |
| nv50a340 | 50×303 | Y | 0.14 | 1.25 | 0.03 |
| nv50a360 | 50×329 | Y | 0.15 | 1.14 | 0.02 |
| nv50a380 | 50×354 | Y | 0.15 | 0.62 | 0.03 |
| nv50a400 | 50×365 | Y | 0.16 | 0.17 | 0.02 |
| nv50a420 | 50×375 | Y | 0.15 | 0.18 | 0.01 |

| Instance | node×edges | Hamiltonian? | | | |
|---|---|---|---|---|---|
| | | | SModels | CModels | CLP(FD) |
| nv50a440 | 50×401 | Y | 0.16 | 3.08 | 0.01 |
| nv50a460 | 50×416 | Y | 0.17 | 0.22 | 0.02 |
| nv50a480 | 50×422 | Y | 0.17 | 3.03 | 0.02 |
| nv50a500 | 50×438 | Y | 0.17 | 0.71 | 0.03 |
| nv50a520 | 50×459 | Y | 0.18 | 0.70 | 0.03 |
| nv50a540 | 50×480 | Y | 0.18 | 1.18 | 0.01 |
| nv50a560 | 50×500 | Y | 0.19 | 0.26 | 0.02 |
| nv50a580 | 50×509 | Y | 0.18 | 0.42 | 0.03 |
| nv60a320 | 60×304 | Y | 0.19 | 0.79 | 0.04 |
| nv60a360 | 60×343 | Y | 0.19 | 8.15 | 0.03 |
| nv60a420 | 60×389 | Y | 0.21 | 0.96 | 0.03 |
| nv60a440 | 60×412 | Y | 0.23 | 8.90 | 0.03 |
| nv60a460 | 60×423 | Y | 0.22 | 0.84 | 0.04 |
| nv60a480 | 60×425 | Y | 0.22 | 1.60 | 0.03 |
| nv60a500 | 60×455 | Y | 0.23 | 2.18 | 0.04 |
| nv60a520 | 60×582 | Y | 0.23 | 0.35 | 0.03 |
| nv60a540 | 60×587 | Y | 0.24 | 0.55 | 0.02 |
| nv60a560 | 60×513 | Y | 0.26 | 0.20 | 0.03 |
| nv60a580 | 60×532 | Y | 0.25 | 0.99 | 0.04 |
| nv70a300 | 70×287 | Y | 0.21 | 0.79 | 0.04 |
| nv70a320 | 70×306 | Y | 0.23 | 4.24 | 0.04 |
| nv70a340 | 70×328 | Y | 0.24 | 1.87 | 0.05 |
| nv70a360 | 70×346 | Y | 0.24 | 1.44 | 0.02 |
| nv70a380 | 70×359 | Y | 0.25 | 0.44 | 0.04 |
| nv70a400 | 70×386 | Y | 0.26 | 4.22 | 0.04 |
| nv70a420 | 70×404 | Y | 0.27 | 4.63 | 0.04 |
| nv70a440 | 70×423 | Y | 0.28 | 1.33 | 0.05 |
| nv70a460 | 70×429 | Y | 0.28 | 3.00 | 0.03 |
| nv70a480 | 70×460 | Y | 0.29 | 1.66 | 0.06 |
| nv70a500 | 70×473 | Y | 0.29 | 1.73 | 0.03 |
| nv70a520 | 70×478 | Y | 0.29 | 0.36 | 0.05 |
| nv70a540 | 70×507 | Y | 0.31 | 4.19 | 0.04 |
| nv70a560 | 70×516 | Y | 0.32 | 0.62 | 0.05 |
| nv70a580 | 70×540 | Y | 0.32 | 1.00 | 0.04 |

**Table 3.** Hamiltonian circuit ('–' denotes no answer within 30 minutes of CPU-time).

**Results:** Most of the problem instances have been taken from the benchmarks used to compare ASP-solvers [13]. Graphs `hc1`–`hc8` are drawn from `www.cs.uky.edu/ai/benchmark-suite/hamiltonian-cycle.html`. All other graphs are chosen from `assat.cs.ust.hk/Assat-2.0/hc-2.0.html`. The graphs `np`$n$`c` are complete directed graphs with $n$ nodes and one edge $\langle u, v\rangle$ for each pair of distinct nodes. The graphs `nv`$v$`a`$a$ graphs are randomly generated graphs having at most $v$ nodes and $a$ edges. The instances `2xp30` (resp., `4xp20`) are obtained by joining 2 (resp., 4) copies of the graph `p30` (resp., `p20`) plus 2 (resp., 3–4) new edges. Graphs `p20` and `p30` are graphs provided in the SModels' distribution [19]. Table 3 lists the results.

## 5 Schur Numbers

A set $S \subseteq \mathbb{N}$ is *sum-free* if the intersection of $S$ and the set $S + S = \{x + y : x \in S, y \in S\}$ is empty. The *Schur number* $S(P)$ is the largest integer $n$ for which the interval $[1..n]$ can be partitioned in $P$ sum-free sets. For instance, $\{1, 2, 3, 4\}$ can be partitioned in $S_1 = \{1, 4\}$ and $S_2 = \{2, 3\}$. Observe that the sets $S_1 + S_1 = \{2, 5, 8\}$ and $S_2 + S_2 = \{4, 5, 6\}$ are sum-free. The set $\{1, 2, 3, 4, 5\}$, instead, originates at least 3 sum-free subsets, thus, $S(2) = 4$. It is easy to verify that the problem of determining whether $S(P) \geq N$, for a given $N$ and $P$, is in NP. It should be noted that so far only 4 Schur numbers have been computed, i.e., $S(1) = 1$, $S(2) = 4$, $S(3) = 13$, and $S(4) = 44$. The best known bound for $S(5)$ is $160 \leq S(5) \leq 315$ [18].

**CLP(FD):** The following CLP(FD) code checks if $N$ can be partitioned into $P$ sum-free parts, i.e., if $S(P) \geq N$.

```
schur(N,P) :- length(List,N), domain(List,1,P),
              constraints(List,N), labeling([ff],List).
```

| Instance | is $Schur(P) \geq N$? | | | Instance | is $Schur(P) \geq N$? | | |
|---|---|---|---|---|---|---|---|
| $\langle P, N\rangle$ | | SModels | CModels | CLP(FD) | $\langle P, N\rangle$ | | SModels | CModels | CLP(FD) |
| $\langle 3, 11\rangle$ | Y | 0.01 | 0.04 | <0.01 | $\langle 5, 100\rangle$ | Y | – | 0.39 | 0.31 |
| $\langle 3, 12\rangle$ | Y | 0.01 | 0.04 | <0.01 | $\langle 5, 101\rangle$ | Y | – | 0.43 | 0.29 |
| $\langle 3, 13\rangle$ | Y | 0.01 | 0.04 | <0.01 | $\langle 5, 102\rangle$ | Y | – | 0.41 | 0.35 |
| $\langle 3, 14\rangle$ | N | 0.02 | 0.04 | 0.01 | $\langle 5, 103\rangle$ | Y | – | 0.74 | 0.36 |
| $\langle 3, 15\rangle$ | N | 0.02 | 0.04 | 0.03 | $\langle 5, 104\rangle$ | Y | – | 5.01 | 0.34 |
| $\langle 3, 16\rangle$ | N | 0.02 | 0.04 | 0.03 | $\langle 5, 105\rangle$ | Y | – | 27.88 | 0.37 |
| $\langle 4, 40\rangle$ | Y | 0.22 | 0.10 | 0.30 | $\langle 5, 106\rangle$ | Y | – | 38.55 | 0.40 |
| $\langle 4, 41\rangle$ | Y | 0.24 | 0.23 | 1.17 | $\langle 5, 107\rangle$ | Y | – | 5.07 | 0.44 |
| $\langle 4, 42\rangle$ | Y | 0.25 | 0.21 | 2.61 | $\langle 5, 108\rangle$ | Y | – | 2.80 | 0.43 |
| $\langle 4, 43\rangle$ | Y | 0.27 | 0.24 | 2.51 | $\langle 5, 109\rangle$ | Y | – | 13.97 | 0.44 |
| $\langle 4, 44\rangle$ | Y | 0.29 | 3.35 | 4.18 | $\langle 5, 110\rangle$ | Y | – | 33.12 | 54.71 |
| $\langle 4, 45\rangle$ | N | 510.01 | 891.66 | – | $\langle 5, 111\rangle$ | Y | – | 0.52 | 56.72 |
| $\langle 4, 46\rangle$ | N | 561.80 | 813.34 | – | $\langle 5, 112\rangle$ | Y | – | 0.54 | 58.44 |
| $\langle 4, 47\rangle$ | N | 767.80 | 791.57 | – | $\langle 5, 113\rangle$ | Y | – | 0.54 | 207.46 |
| $\langle 4, 48\rangle$ | N | 978.84 | 805.33 | – | $\langle 5, 114\rangle$ | Y | – | 11.63 | 1032.43 |
| $\langle 4, 49\rangle$ | N | 1258.57 | 678.19 | – | $\langle 5, 115\rangle$ | Y | – | 82.13 | 1069.54 |
| $\langle 4, 50\rangle$ | N | 1680.34 | 890.05 | – | $\langle 5, 116\rangle$ | Y | – | 60.53 | 1108.02 |
| $\langle 4, 51\rangle$ | N | 1892.36 | 1046.73 | – | $\langle 5, 117\rangle$ | Y | – | 761.11 | 1150.25 |

**Table 4.** Schur numbers ('–' denotes no answer within 30 minutes of CPU-time).

```
constraints(List, N) :- recursion(List,1,1,N).
recursion( _  ,I,_,N):- I>N, !.
recursion(List,I,J,N):- I+J>N, !, I1 is I+1, recursion(List,I1,1,N).
recursion(List,I,J,N):- I>J, !, J1 is J+1, recursion(List,I,J1,N).
recursion(List,I,J,N):- K is I+J, J1 is J+1, nth(I,List,BI),
      nth(J,List,BJ), nth(K,List,BK), (BI #= BJ) #=> (BK #\= BI),
      recursion(List,I,J1,N).
```

The term `List` is a list of `N` variables (associated to the numbers `1`,...,`N`), each of them taking values in `1..P`. The value of the $i$th variable identifies the block of the partition $i$ belongs to. The predicate `recursion` states that for all `I` and `J`, with $\texttt{1} \leq \texttt{I} \leq \texttt{J} \leq \texttt{N}$, the numbers `I`, `J` and `I+J` must not be all in the same block.

**ASP:** The following is the ASP solution we employed.

```
(1) number(1..n).      part(1..p).
(2) 1 { inpart(X,P) : part(P) } 1 :- number(X).
(3) :- number(X;Y), part(P), X<=Y, inpart(X,P),
       inpart(Y,P), inpart(X+Y,P).
(4) :- number(X), part(P;P1), inpart(X,P), P1<P, not occupied(X,P1).
(5) occupied(X,P) :- number(X;Y), part(P), Y<X, inpart(Y,P).
```

The atom `inpart(X,P)` represents the fact that number $X$ is assigned to part $P$. Rule `(2)` generates the potential solutions, by assigning each integer to exactly one part. The ASP-constraints `(3)`, `(4)`, and `(5)` remove unwanted solutions: `(3)` states that for any $X$ and $Y$, the three numbers $X$, $Y$, and $X + Y$ cannot belong to the same part. Rules `(4)` and `(5)` remove symmetries, by selecting the free part with lowest index.

**Results:** Table 4 lists the timings we obtained. Let us observe that, unfortunately, we are still far from the best known lower bound of 160 for $S(5)$.

## 6 Protein Structure Prediction

Given a sequence $S = s_1 \cdots s_n$, with $s_i \in \{h, p\}$, the *2D, HP-protein structure prediction problem* (reduced from [3]) is the problem of finding a mapping (*folding*)

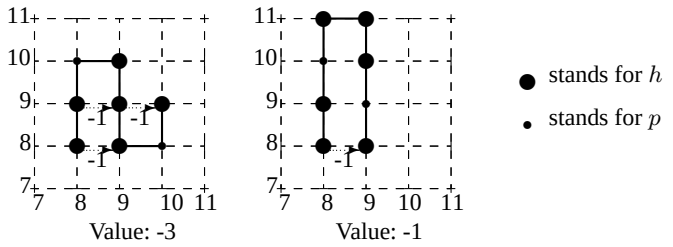



**Fig. 1.** Two foldings for $S = hhphhhph$ ($n = 8$). The leftmost one is minimal.

$\omega : \{1, \ldots, n\} \longrightarrow \mathbb{N}^2$ such that

$(\forall i \in [1, n-1]\})\mathsf{next}(\omega(i), \omega(i+1))$ and $(\forall i, j \in [1, n])(i \neq j \rightarrow \omega(i) \neq \omega(j))$

and minimizing the energy:

$$\sum_{\substack{1 \leq i \leq n-2 \\ i+2 \leq j \leq n}} \mathsf{Pot}(s_i, s_j) \cdot \mathsf{next}(\omega(i), \omega(j))$$

where $\mathsf{Pot}(s_i, s_j) \in \{0, -1\}$ and $\mathsf{Pot} = -1$ if and only if $s_i = s_j = h$. The condition $\mathsf{next}(\langle X_1, Y_1\rangle, \langle X_2, Y_2\rangle)$ holds between two adjacent positions of a given lattice if and only if $|X_1 - X_2| + |Y_1 - Y_2| = 1$. W.l.o.g., we set $\omega(1) = \langle n, n\rangle$ and $\omega(2) = \langle n, n+1\rangle$, to remove some symmetries in the solution space.

Intuitively, we look for a self-avoiding walk that maximizes the number of contacts between occurrences of objects (aminoacids) of kind $h$ (see Figure 1). Contiguous occurrences of $h$ in the input sequence $S$ contribute in the same way to the energy associated to each spatial conformation and thus they are not considered in the objective function. Note that two objects can be in contact only if they are at an odd distance in the sequence (odd property of the lattice). This problem is a version of the protein structure prediction problem, whose decision problem is known to be NP-complete [4].

**CLP(FD):** A complete CLP(FD) encoding of this problem can be found in `www.di.univaq.it/~formisano/CLPASP`. An extension of this code (in 3D, inside a realistic lattice, and with a more complex energy function) has been used to predict the spatial shape of real proteins [5].

**ASP:** As far as we know, there are no ASP formulations of this problem available in the literature. A specific instance of the problem is represented as a set of facts, describing the sequence of aminoacids. For instance, the protein denoted by hpphpphpph (or simply $(\mathsf{hpp})^3\mathsf{h}$ using regular expressions) is described as:

```
prot(1,h). prot(2,p). prot(3,p). prot(4,h). prot(5,p).
prot(6,p). prot(7,h). prot(8,p). prot(9,p). prot(10,h).
```

The ASP code is as follows:

```
(1) size(10).      %%% size(N) where N is input length
(2) range(7..13).  %%% [ N-sqrt{N}, N+sqrt{N} ]
(3) sol(1,N,N)   :- size(N).
(4) sol(2,N,N+1) :- size(N).
(5) 1 { sol(I,X,Y) : range(X;Y) } 1 :- prot(I,Amino).
(6) :- prot(I1,A1), prot(I2,A2),neq(I1,I2),
       sol(I1,X,Y), sol(I2,X,Y), range(X;Y).
(7) :- prot(I1,A1), prot(I2,A2), I2>1, eq(I1,I2-1), not next(I1,I2).
(8) next(I1,I2) :- prot(I1,A1), prot(I2,A2), I1<I2,
```

| Instance | | | Optimization problem | | Decision problem | | |
|---|---|---|---|---|---|---|---|
| Input Sequence | Length | Min | CLP(FD) | SModels | CLP(FD) | SModels | CModels |
| $h^{10}$ | 10 | -4 | 0.14 | 0.91 | 0.01 | 0.65 | 0.69 |
| $h^{15}$ | 15 | -8 | 1.93 | 13.21 | 0.04 | 2.84 | 2.69 |
| $h^{20}$ | 20 | -12 | 201.58 | 1982.44 | 0.29 | 45.63 | 40.70 |
| $h^{25}$ | 25 | -16 | 25576.38 | – | 817.71 | 3181.78 | 1165.26 |
| $(hpp)^3h$ | 10 | -4 | 0.01 | 0.66 | 0.01 | 0.42 | 0.49 |
| $(hpp)^5h$ | 16 | -6 | 0.32 | 26.95 | 0.22 | 22.46 | 16.58 |
| $(hpp)^7h$ | 22 | -6 | 62.69 | 1303.13 | 12.75 | 35.96 | 161.25 |
| $(hpp)^9h$ | 28 | -9 | 6758.45 | – | 1955.28 | 3369.78 | 1217.34 |

**Table 5.** Protein structure prediction ('–' denotes no answer within 10 hours of CPU-time).

```
      sol(I1,X1,Y1), sol(I2,X2,Y2), range(X1;Y1;X2;Y2),
      1==abs(Y1-Y2)+abs(X2-X1).
(9) energy_pair(I1,I2) :- prot(I1,h), prot(I2,h),
      next(I1,I2), I1+2<I2, 1==(I2-I1) mod 2.
(10) seq_proteins(I1,I2) :- prot(I1,A1), prot(I2,A2),
      I1+2<I2, 1==(I2-I1) mod 2.
(11) maximize{ energy_pair(I1,I2) : seq_proteins(I1,I2) }.
```

Rules `(1)` and `(2)`, together with the predicate `prot`, define the domains. The range $N-\sqrt{N}..N+\sqrt{N}$ is a heuristic value used consistently in both the CLP(FD) and ASP encodings. Rule `(5)` implements the "generate" phase: it states that each aminoacid occupies exactly one position. Rules `(3)` and `(4)` fix the positions of the two initial aminoacids (they eliminates symmetric solutions). The ASP-constraints `(6)` and `(7)` state that there are no self-loops and that two contiguous aminoacids must satisfy the `next` property. Rule `(8)` defines the `next` relation, also including the odd property of the lattice. The objective function is defined by Rule `(9)`, which determines the energy contribution of the aminoacids, and rule `(11)`, that searches for a answer sets maximizing the energy.

**Results:** The experimental results for the two programs are reported in Table 5. Since CModels does not support optimization statements, we can only compare the performance of SICStus and SModels. Nevertheless, we performed a series of tests relative to the decision version of this problem, namely, answering the question "can the given protein fold to reach a given energy level?", using the energy results obtained by solving the optimization version of the problem. The results are also reported in Table 5.

## 7 Planning

Planning is one of the most interesting applications of ASP. CLP(FD) has been used less frequently to handle planning problems. A planning problem is based on the notions of `State` (a representation of the world) and `Actions` that change the states. We focus on solving a planning problem in the block world domain. Let us assume to have $N$ blocks (blocks $1,\ldots,N$). In the *initial state*, the blocks are arranged in a single stack, in increasing order, i.e., block 1 is on the table, block 2 is on top of block 1, etc. Block $N$ is on top of the stack. In the *goal state*, there must be two stacks, composed of the blocks with odd and even numbers, respectively. In both stacks the blocks are arranged in increasing order, i.e., blocks 1 and 2 are on the table and blocks $N-1$ and $N$ are on top of the respective stacks. The planning problem consists of finding a sequence of $T$ actions (*plan*) to reach the goal state, starting from the initial state. Some additional

restrictions must be met: first, in each state at most three blocks can lie on the table. Moreover, a block $x$ cannot be placed on top of a block $y$ if $y \geq x$.

**CLP(FD):** We study the encoding of block world planning problem in CLP(FD). The code can be easily generalized as a scheme for encoding general planning problems. The plan can be modeled as a list `States` of $T+1$ states. Each `State` is a $N$-tuple `[B1,...,BN]`, where `Bi=j` means that block `i` is placed on block `j`. The case `j=0` represents the fact that the block `i` lies on the table. The initial state and the final state are represented by the lists $[0,1,2,3,\ldots,\mathtt{N}-1]$ and $[0,0,1,2,\ldots,\mathtt{N}-2]$.

```
planning(NBlocks,NTime) :- init_domains(NBlocks,NTime,States),
  initial_state(States), final_state(States),
  init_actions(NBlocks,NTime,Actions),
  forward(Actions,States), no_rep(Actions),
  action_properties(Actions,States), term_variables(Actions,Vars),
  labeling([leftmost],Vars).
init_domains(NBlocks,NTime,States) :- T1 is NTime+1,
  length(States,T1), init_domains(NBlocks,States).
init_domains(_,[]).
init_domains(N,[S|States]) :- length(S,N),
  init_domains(N,States), domain(S,0,N), count(0,S,'#=<',3).
initial_state([State|_]) :- increasing_list(State).
final_state(Sts) :- append(_,[[0|FS]],Sts), increasing_list(FS).
init_actions(_,0,[]) :- !.
init_actions(N,T,[[Block,To_Block]|Acts]) :- T1 is T-1,
  Block#\=To_Block, Block in 1..N, To_Block in 0..N,
  (Block#<To_Block #=> To_Block#=0), init_actions(N,T1,Acts).
forward([],_).
forward([[Block,To_Block]|B],[CurrState,NextState|Rest]) :-
  element(Block,NextState,To_Block), is_clear(CurrState,Block),
  is_clear(CurrState,To_Block), element(Block,CurrState,Old),
  Old#\=To_Block, forward(B,[NextState|Rest]).
is_clear([],_).
is_clear([A|B],X) :- (X#\=0 #=> A#\=X), is_clear(B,X).
no_rep([_]).
no_rep([[X1,_],[X2,Y2]|Rest]) :- X1#\=X2, no_rep([[X2,Y2]|Rest]).
action_properties([],_).
action_properties([[Block,_To]|Rest],[Current,Next|States]) :-
  inertia(1,Block,Current,Next),
  action_properties(Rest,[Next|States]).
inertia(_,_,[],[]).
inertia(N,X,[A|B],[C|D]) :-
  N1 is N+1, inertia(N1,X,B,D), (X#\=N #=> A#=C).
increasing_list(List) :- sequence(List,0).
sequence([],_).
sequence([N|R],N) :- M is N+1, sequence(R,M).
```

The code follows the usual constrain-and-generate methodology. The `init_domains` predicate generates the list of the `NTime` states and fixes the maximum number of objects admitted on the table in each state (using the built-in constraint `count`). After

| Instance | | Plan | SModels | CModels | SICStus | Instance | | Plan | SModels | CModels | SICStus |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Blocks | Length | exists | | | CLP(FD) | Blocks | Length | exists | | | CLP(FD) |
| 5 | 11 | N | 0.23 | 0.11 | 0.01 | 7 | 51 | N | – | 991.56 | 824.61 |
| 5 | 12 | N | 0.29 | 0.12 | 0.01 | 7 | 52 | N | – | 1091.54 | 1097.13 |
| 5 | 13 | Y | 0.33 | 0.16 | 0.02 | 7 | 53 | N | – | 2044.34 | 1509.35 |
| 6 | 22 | N | 2.61 | 8.16 | 0.11 | 7 | 54 | Y | – | 431.32 | 1104.16 |
| 6 | 23 | N | 3.60 | 9.86 | 0.13 | 8 | 40 | N | 193.31 | 115.40 | 21.73 |
| 6 | 24 | N | 4.73 | 6.46 | 0.18 | 8 | 41 | N | 234.96 | 300.55 | 29.48 |
| 6 | 25 | N | 6.44 | 13.40 | 0.25 | 8 | 42 | N | 279.35 | 126.93 | 41.57 |
| 6 | 26 | N | 8.64 | 8.31 | 0.32 | 8 | 43 | N | 335.08 | 196.62 | 55.66 |
| 6 | 27 | Y | 12.17 | 6.56 | 0.26 | 8 | 44 | N | 404.43 | 874.70 | 78.75 |
| 7 | 33 | N | 38.64 | 175.96 | 2.49 | 8 | 45 | N | 475.71 | 231.80 | 110.08 |
| 7 | 34 | N | 47.34 | 222.07 | 3.42 | 8 | 46 | N | 579.54 | 351.47 | 158.71 |
| 7 | 35 | N | 58.01 | 153.02 | 4.71 | 8 | 47 | N | 682.70 | 193.02 | 205.57 |
| 7 | 36 | N | 71.40 | 106.57 | 6.36 | 8 | 48 | N | 808.11 | 52.04 | 285.94 |
| 7 | 37 | N | 87.84 | 115.96 | 8.70 | 8 | 49 | N | 947.37 | 463.65 | 386.23 |
| 7 | 38 | N | 107.11 | 157.32 | 11.94 | 8 | 50 | N | 1123.94 | 379.32 | 544.91 |
| 7 | 39 | N | 150.11 | 84.98 | 16.33 | 8 | 51 | N | 1328.18 | 192.87 | 748.38 |
| 7 | 40 | N | 177.69 | 115.40 | 22.22 | 8 | 52 | N | 1566.62 | 172.24 | 1049.58 |
| 7 | 41 | N | 253.65 | 217.10 | 31.19 | 8 | 53 | N | 1877.88 | 3440.71 | 1436.14 |
| 7 | 42 | N | 355.66 | 220.00 | 42.83 | 8 | 54 | N | 2257.87 | 212.60 | 2028.70 |
| 7 | 43 | N | 565.60 | 74.19 | 58.91 | 8 | 55 | N | 2717.22 | 178.01 | 2760.98 |
| 7 | 44 | N | 1126.52 | 169.01 | 80.59 | 8 | 56 | N | 3308.28 | 4667.86 | 3875.05 |
| 7 | 45 | N | 2710.53 | 139.66 | 111.98 | 8 | 57 | N | 4290.26 | 866.58 | 5101.24 |
| 7 | 46 | N | 7477.13 | 299.01 | 158.03 | 8 | 58 | N | 5672.42 | 287.16 | 7240.92 |
| 7 | 47 | N | – | 180.63 | 217.26 | 8 | 59 | N | 7791.38 | 1769.51 | 9838.83 |
| 7 | 48 | N | – | 209.73 | 299.31 | 8 | 60 | N | 11079.03 | 903.10 | 13917.36 |
| 7 | 49 | N | – | 463.56 | 417.63 | 8 | 61 | N | 18376.59 | 488.78 | 19470.35 |
| 7 | 50 | N | – | 542.98 | 586.73 | 8 | 62 | N | 35835.76 | 4639.58 | 27030.19 |

**Table 6.** Planning in blocks world ('–' denotes no answer in less than 3 hours ).

that, the initial and final states are initialized. The predicate `init_actions` specifies that a block can be moved either to the table or to another block having a smaller number. `forward` states that if a block is placed on another one, then both of them must be *clear*, i.e., without any block on top of them. The predicate `no_rep` guarantees that two consecutive actions cannot move the same block. Finally, `action_properties` forces the inertia laws (i.e., if a block is not moved, then it remains in its position).

**ASP:** There are several ways to encode a block world in ASP (e.g., [12, 2]). In our experiments we adopted the code reported in `www.di.univaq.it/~formisano/CLPASP`.

**Results:** Table 6 reports the execution times from the three systems, for different number of blocks and plan lengths.

## 8 Knapsack

In this section we discuss a generalization of the knapsack problem. Let us assume to have $n$ types of objects, and each object of type $i$ has size $w_i$ and it costs $c_i$. We wish to fill a knapsack with $X_1$ object of type 1, $X_2$ objects of type 2, and so on, so that:

$$\sum_{i=1}^{n} X_i w_i \leq \texttt{max\_size} \quad \text{and} \quad \sum_{i=1}^{n} X_i c_i \geq \texttt{min\_profit}. \tag{1}$$

where `max_size` is the capacity of the knapsack and `min_profit` is the minimum profit required.

**CLP(FD):** We represent the types of objects using two lists (containing the size and cost of each type of object), e.g., `objects([2,4,8,16,32,64], [2,5,11,23,47,95])`. The CLP(FD) encoding is:

```
knapsack(Max_Size,Min_Profit) :- objects(Weights,Costs),
       length(Sizes,N), length(Vars,N), domain(Vars,0,Max_Size),
       scalar_product(Sizes,Vars,#=<,Max_Size),
       scalar_product(Costs,Vars,#>=,Min_Profit),
       labeling([ff],Vars).
```

Observe that we used the built-in `scalar_product` for implementing (1).[4]

**ASP:** The ASP-formulation of the knapsack problem we experimented with, is easily obtainable from the classical encoding of the standard knapsack problem (i.e., with $X_i \in \{0, 1\}$). The different kinds of objects are so represented as follows:

```
item(1..10).
#weight size(1,X1) = 2*X1.     #weight size(2,X2) = 4*X2.
#weight size(3,X3) = 8*X3.     #weight size(4,X4) = 16*X4.
#weight size(5,X5) = 32*X5.    #weight size(6,X6) = 64*X6.
#weight size(7,X7) = 128*X7.   #weight size(8,X8) = 256*X8.
#weight size(9,X9) = 512*X9.   #weight size(10,X10) = 1024*X10.
#weight cost(1,X1) = 2*X1.    #weight cost(2,X2) = 5*X2.
#weight cost(3,X3) = 11*X3.   #weight cost(4,X4) = 23*X4.
#weight cost(5,X5) = 47*X5.   #weight cost(6,X6) = 95*X6.
#weight cost(7,X7) = 191*X7.  #weight cost(8,X8) = 383*X8.
#weight cost(9,X9) = 767*X9.  #weight cost(10,X10) = 1535*X10.
```

The ASP encoding is the following:

```
(1) occs(0..max_size).
(2) 1 { in_sack(I,XI) : occs(XI) } 1 :- item(I).
(3) size(I,XI) :- item(I), occs(XI), in_sack(I,XI).
(4) cost(I,XI) :- item(I), occs(XI), in_sack(I,XI).
(5) cond_cost :- min_profit [ cost(I,XI) : item(I) : occs(XI) ].
(6) :- not cond_cost.
(7) cond_weight :- [ size(I,XI) : item(I) : occs(XI) ] max_size.
(8) :- not cond_weight.
```

Fact `(1)` fixes the domain for the variable `XI`. The Rule `(2)` states that, for each type of objects `I`, there is only one fact `in_sack(I,XI)` in the answer set, representing the number of objects of type `I` in the knapsack. Rules `(3)` and `(4)` get the total `size` and `cost` for each type of object present in the knapsack. Rules `(5)`–`(8)` establish the constraints of minimum profit and maximum size. The two constants `max_size` and `min_profit` must be provided to lparse during grounding.

**Results:** Table 7 reports some of the results we obtained. The right-hand side columns regard runs with 10-fold increase in objects' costs and `min_profit`. We were not able to obtain any result from CModels. For any of the instances we experimented with (except the smallest ones, involving at most five types of objects) the corresponding process was terminated by the operative system. The reason for this could be found by observing that the run-time images of such processes grow very large in size (up to

[4] The built-in predicate `knapsack`, available in SICStus Prolog, is a special case of `scalar_product` where the third argument is the equality constraint.

| max_size | min_profit | Answer | SICStus | SModels | max_size | min_profit | Answer | SICStus | SModels |
|---|---|---|---|---|---|---|---|---|---|
| 255 | 374 | Y | 0.02 | 0.34 | 255 | 3740 | Y | 0.02 | 0.35 |
| 255 | 375 | N | 0.03 | 6.25 | 255 | 3750 | N | 0.03 | 6.27 |
| 511 | 757 | Y | 0.36 | 0.85 | 511 | 7570 | Y | 0.36 | (**) |
| 511 | 758 | N | 0.36 | 248.26 | 511 | 7580 | N | 0.36 | 0.50 |
| 1023 | 1524 | Y | 8.81 | 2.59 | 1023 | 15240 | Y | 8.75 | (**) |
| 1023 | 1525 | N | 8.75 | – | 1023 | 15250 | N | 8.69 | 1.03 |
| 2047 | 3059 | Y | 368.50 | (*) | 2047 | 30590 | Y | 369.83 | (**) |
| 2047 | 3060 | N | 366.79 | (*) | 2047 | 30600 | N | 368.24 | 1.83 |

**Table 7.** Knapsack instances ('–' denotes no answer within 30 minutes of CPU-time).

4.5GB, in some instances). The mark (*) in Table 7, denotes instances where SModels is not able to process the ground program—in such cases, SModels stops with the message "`sum of weights in weight rule too large...`". The (**) denotes instances where SModels reports the incorrect answer (No). For $(511, 7570)$, setting the value `#weight cost(10,Q) = 1177*Q` we obtain the correct solution (that does not use objects of type 10), while with values greater than $1177$ something goes astray—probably improperly handled large integers. This shows that currently ASP is more sensible to number size w.r.t. CLP(FD). The behavior of SModels on $(511, 7580)$, $(1023, 15250)$, and $(2047, 30600)$ should be wrong, as well, but the absence of solutions does not allow to point out it.

## 9 Discussion and Conclusions

We tested the CLP(FD) and ASP codes for various combinatorial problems. In the Tables 1–7 we reported the running times (in seconds) of the solutions to these problems on different problem instances. Let us try here to analyze these results.

First of all, from the benchmarks, it is clear that ASP provides a more compact, and probably more declarative, encoding; in particular, the reliance on grounding and domain-restricted variables allows ASP to avoid use of recursion in many situations.

As far as running times are concerned, CLP(FD) definitely wins the comparison vs. SModels. In a few cases, the running times are comparable, but in most of the cases CLP(FD) runs significantly faster. Observe also that CModels is, in most of the problems, faster than SModels; part of this can be justified by the fact that the programs we are using are mostly tight [7], and by the high speed of the underlying SAT solver used by CModels.

The comparison between CLP(FD) and CModels is more interesting. In the $k$-coloring and $N$-$M$-queens cases, running times are comparable. In some of the classes of graphs, CModels performs slightly better on all instances. More in general, whenever the instances of a single class are considered, one of the two systems tends to always outperform the other. This indicates that the behavior of the solver is significantly affected by the nature of the specific problem instances considered (recall that each class of graphs comes from encodings of instances of different problems [20]).

As one may expect, the bottom-up search strategy of ASP is less sensitive to the presence of solutions w.r.t. the top down search strategy of CLP(FD). As a matter of fact, CLP(FD) typically runs faster than CModels when a solution exists. Moreover, CLP(FD) behaves better on small graphs. For the Hamiltonian circuit problem, CLP(FD) runs significantly faster—we believe this is due to the use of the built-in global constraint `circuit`, which guarantees excellent constraint propagation. In this case, only in absence of solutions the running times are comparable—i.e., when the two

approaches are forced to traverse the complete search tree. A similar situation arises in computing Schur numbers. When the solution exists and numbers are low, CLP(FD) performs better. For larger instances (even with solutions), the running times are favorable to CModels.

Regarding the protein folding problem, CLP(FD) solves the problems much faster than ASP. Also in this case, however, the ASP code appears to be simpler and more compact than the CLP(FD) one. In general, in designing the CLP code, the programmer cannot easily ignore knowledge about the inference strategy implemented in the CLP engine. The fact that CLP(FD) adopts a top-down depth-first strategy influences programmer's choices in encoding the algorithms.

For the planning problem, we observe that SModels runs faster than CModels for small instances. In general, CLP(FD) performs better for small dimensions of the problem. On the other hand, when the dimension of the problem instance becomes large, the behavior of CLP(FD) and SModels become comparable while CModels provides the best performance. In fact, the performance of CModels does not seem to be significantly affected by the growth in the size of the problem instance, as clearly happens for CLP(FD) and SModels. The same phenomenon can be also observed in other situations, e.g., in the Hamiltonian circuit and Schur numbers problems. In these cases, the time spent by CModels to obtain a solution does not appear to be directly related to the raw dimension of the problem instance. Initial experiments reveal that this phenomenon arises even when different SAT-solvers are employed. Further studies are needed to better understand to which extent the intrinsic structure of an instance biases CModels' behavior, in particular the way in which CModels' engine translates an ASP program into a SAT-instance.

For the Knapsack problem, CModels is not applicable. CLP(FD) runs definitively faster than SModels; furthermore, SModels becomes inapplicable and unreliable for large problem instances.

Table 9 intuitively summarizes our observations drawn from the different benchmarks. Although these experiments are quite preliminary, they actually provide already some concrete indications that can be taken into account when choosing a paradigm to tackle a problem. We can summarize the main points as follows:

- graph-based problems have nice compact encodings in ASP and the performance of the ASP solutions is acceptable and scalable;
- problems requiring more intense use of arithmetic and/or numbers are declaratively and efficiently handled by CLP(FD);
- for problems with no arithmetic, the exponential growth w.r.t. the input size is less of an issue for ASP.

| | Coloring | Hamilton | Schur | PF | Planning | Knapsack |
|---|---|---|---|---|---|---|
| CLP(FD) | + | ++ | + | + | + | + |
| ASP CModels | ++ | + | ++ | - | + | - |

**Table 8.** Schematic results' analysis. + (-) means that the formalism is (not) applicable. ++ that it is the best when the two formalisms are applicable.

In the future we plan to extend our analysis to other problems and to other constraint solvers (e.g., BProlog, ILOG) and ASP-solvers (e.g., ASSAT, aspps, DLV). In particular, we are interested in answering the following questions:

- is it possible to formalize domain and problem characteristics to lead the choice of which paradigm to use?
- is it possible to introduce strategies to split problem components and map them to cooperating solvers (using the best solver for each part of the problem)?

In particular, we are interested in identifying those contexts where the ASP solvers perform significantly better than CLP. It seems reasonable to expect this behavior, for instance, whenever incomplete information comes into play.

## References


[1] C. Anger, T. Schaub, and M. Truszczynski. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004.

[2] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.

[3] P. Clote and R. Backofen. *Computational Molecular Biology*. Wiley & Sons, 2001.

[4] P. Crescenzi et al. On the complexity of protein folding. In *STOC*, pages 597–603, 1998.

[5] A. Dal Palù, A. Dovier, and F. Fogolari. Constraint logic programming approach to protein structure prediction. *BMC Bioinformatics*, 5(186):1–12, 2004.

[6] I. Elkabani, E. Pontelli, and T. C. Son. SModels with CLP and Its Applications: A Simple and Effective Approach to Aggregates in ASP. In *ICLP*, 73–89, 2004.

[7] E. Erdem and V. Lifschitz. Tight Logic Programs. In *TPLP*, 3:499-518, 2003.

[8] A. J. Fernandez and P. M. Hill. A Comparative Study of 8 Constraint Programming Languages Over the Boolean and Finite Domains. *Constraints*, 5(3):275–301, 2000.

[9] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP*, pages 1070–1080, MIT Press, 1988.

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

[11] Y. Lierler and M. Maratea. CModels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *LPNMR*, pages 346–350. Springer Verlag, 2004.

[12] V. Lifschitz. Answer Set Planning. In *Logic Programming and Non-monotonic Reasoning*, pages 373–374. Springer Verlag, 1999.

[13] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In *AAAI*, pages 112–117. AAAI/MIT Press, 2002.

[14] V. W. Marek and M. Truszczynski. Autoepistemic Logic. *JACM*, 38(3):588–619, 1991.

[15] V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm*, 375–398. Springer, 1999.

[16] K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[17] M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On Benchmarking Constraint Logic Programming Platforms. *Constraints*, 9(1):5–34, 2004.

[18] E. W. Weisstein. *Schur Number*. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/SchurNumber.html`.

[19] Web references for some ASP solvers. ASSAT: `assat.cs.ust.hk`. CCalc: `www.cs.utexas.edu/users/tag/cc`. CModels: `www.cs.utexas.edu/users/tag/cmodels`. DeReS and aspps: `www.cs.uky.edu/ai`. DLV: `www.dbai.tuwien.ac.at/proj/dlv`. SModels: `www.tcs.hut.fi/Software/smodels`.

[20] Web site of COLOR02/03/04: Graph Coloring and its Applications: `http://mat.gsia.cmu.edu/COLORING03`.

# Un formalismo basato sulla Frame Logic per il ragionamento spaziale. Un'applicazione al problema della visualizzazione di relazioni spaziali qualitative

**Licenziato Luca*, Mele Francesco****
*Università di Napoli Federico II
Via Cinzia Napoli,80126 Napoli licenzia@studenti.unina.it
** Istituto di Cibernetica Consiglio Nazionale delle Ricerche,
Via dei Campi Flegrei 34 Pozzuoli (NA) f.mele@cib.na.cnr.it

## Abstract

Il principale obiettivo del presente lavoro è stato quello di costruire un formalismo basato su una rappresentazione a frame, che catturasse tutti gli aspetti significativi (anche complessi) fra le parti e la totalità di un oggetto fisico composto, e le relazioni spaziali qualitative delle parti appartenenti alla totalità.
Il lavoro è stato svolto attraverso una prima fase di rigorosa scelta, della rappresentazione di base che ci ha portato al formalismo proposto, il quale è stato definito rispettando una serie di requisiti guida esistenti nel settore, tra questi, quello di adottare un approccio esplicito per la rappresentazione delle totalità. Inoltre l'implementazione della base di conoscenza è stata realizzata utilizzando un paradigma basato su prototipi – paradigma che ha come nozioni di base il prototipo e la delega, diversamente dal paradigma basato su classi, che ha la classe e l'ereditarietà.
Una seconda fase è stata dedicata alla costruzione di una rappresentazione, e di un'assiomatica, per il ragionamento spaziale qualitativo, integrata nei requisiti guida. Infine, una terza fase è stata dedicata alla costruzione di un ragionatore, sviluppato in Frame Logic, che fosse in grado, utilizzando l'apparato inferenziale presente nell'assiomatica, di eseguire il completamento e il rendering di oggetti 3D.

## 1 Introduzione

Il ragionamento spaziale è stato, fino ad ora, un argomento che si è sviluppato all'interno di diverse discipline come la Robotica, la Fisica Qualitativa e l'Analisi delle immagini. Recentemente è diventato un argomento di studio con una precisa missione e connotazione. La missione è quella di definire formalismi, al fine di potere *catturare* le molteplici tipologie di relazioni e inferenze spaziali esistenti, in diversi domini conoscitivi (quasi ogni dominio possiede una specificità propria) al fine di potere *effettuare* ragionamenti spaziali intorno ad oggetti collocati nello spazio. Un caso particolare costituisce lo studio del ragionamento spaziale qualitativo umano. Modello ritenuto interessante perché si crede possa essere adoperato in diverse tipologie di applicazioni che coinvolgono il ragionamento spaziale.
L'ambito di collocazione del ragionamento spaziale, invece, è quello delle ontologie formali [2], settore che ha messo a disposizione molti strumenti per affrontare le varietà di problemi esistenti nel settore in esame.
Un'attiva ricerca specifica, nel settore del ragionamento spaziale, riguarda lo studio delle relazioni fra una totalità e le sue parti [1]. In questo studio le relazioni spaziali (sta_sotto_a, sta_a_destra_di, ect) sono usate per descrivere la struttura di tali oggetti composti.
Riguardo all'approccio metodologico adottato in questo lavoro, faremo riferimento ai requisiti di base presentati in [1]. Nel citato lavoro, si propone un insieme di requisiti (minimi a parere degli autori) per costruire un modello idoneo "a catturare l'ontologica natura delle parti e delle totalità" di oggetti fisici composti:

1- Adottare un approccio esplicito per la definizione delle totalità
2- Eseguire una chiara distinzione fra le parti ed altri attributi di una totalità
3- Costruire in maniera built-in la relazione di transitività fra parti

4- Possibilità di riferirsi alle parti mediante generici nomi
5- Capacità di esprimere relazioni fra parti e totalità
6- Capacità di esprimere relazioni fra le parti di una totalità

Una rappresentazione di oggetto composto è implicita quando nella definizione della totalità non sono presenti (esplicitamente) le parti che compongono la totalità. In questi tipi di rappresentazioni sono le parti che *hanno conoscenza* a quali totalità appartengono, conoscenza espressa localmente nella definizione delle parti stesse.
Contrariamente, una rappresentazione è esplicita se nella definizione della totalità, che rappresenta l'oggetto composto, sono presenti esplicitamente le connessioni con le parti che la compongono, in altre parole, quando la totalità *conosce* le sue parti. Un esempio dei due tipi di rappresentazione è il seguente:

**Approccio implicito**

colonna1 : stile : dorico

capitello1 : descrizione non struttuale: xxxx
: tipoforma : cubo
: altezza : 1.0 m
: parte_di: colonna1

fusto1 : descrizione non struttuale : xxxx
: tipoforma : cilindro
: altezza : 4.0 m
: raggio_base : 5
: parte_di: colonna1

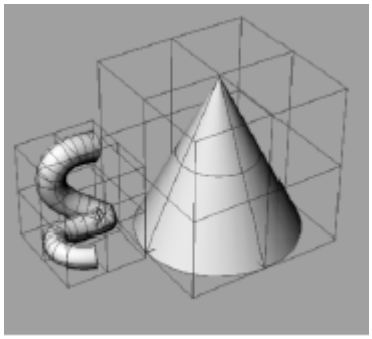
**Approccio esplicito**

colonna1 : stile : dorico
: formata_da : capitello1, fusto1

capitello1 : descrizione non struttuale: xxxx
: tipoforma : cubo
: altezza : 1.0 m

fusto1 : descrizione non struttuale : xxxx
: tipoforma : cilindro
: altezza : 4.0 m
: raggio_base : 5

Una rappresentazione esplicita comporta alcuni vantaggi.
In primo luogo *espressività* e *dichiaratività:* nella definizione di una totalità le parti sono esplicitamente dichiarate, contrariamente a quello che accade in una rappresentazione implicita, dove, per avere il *quadro* della completa struttura, bisogna *andare a ricercare* (in maniera procedurale), nelle definizioni delle parti, quelle che appartengono alla totalità che si sta esaminando.
In secondo luogo il *riuso*: è semplice convincersi che, se si rappresentano le parti in maniera da non contenere l'attributo di appartenenza alla totalità, tali definizioni non dipendono dal contesto di appartenenza (dalle totalità). Per tale ragione risulta che una definizione di parte, così strutturata, può essere utilizzata da più definizioni di totalità.
Se ad un approccio esplicito si aggiunge la possibilità di riferirsi alle parti mediante generici nomi (punto 4 della precedente lista), si conferisce ad un formalismo proprietà di composizionalità: avere la possibilità di costruire nuove totalità, in maniera incrementale, ossia, comporre nuove aggregazioni di oggetti senza dover modificare le definizioni di altre precedentemente definite.



Come faremo vedere nel formalismo di questo lavoro, abbiamo scelto un approccio esplicito per la rappresentazione delle totalità. Nell'approccio proposto, una totalità è definita mediante l'insieme delle sue parti (come nel caso di definizione esplicita dell'esempio riportato). In aggiunta, nella nostra proposta, una totalità è definita a) mediante l'insieme delle relazioni della totalità con le sue parti b) mediante l'insieme delle relazioni fra le parti stesse.
I punti a) e b) ci hanno permesso di raggiungere i requisiti guida 5) e 6) della lista. Per il punto 5) "la capacità di esprimere relazioni fra parti e totalità" abbiamo utilizzato una struttura geometrica di riferimento: il BoundingBox[1]. L'oggetto-totalità è la composizione dei suoi oggetti-parti, ed il BoundingBox rappresenta il risultato di questa

[1] Il BoundingBox è il parallelepipedo, con i lati paralleli agli assi cartesiani, che contiene un oggetto tridimensionale nello spazio, nel nostro caso contiene la totalità.

composizione. Nello specifico delle relazioni spaziali, invece, una parte si *relaziona male* con la sua totalità, per il semplice fatto che, da un punto di vista strutturale-geometrico, la totalità stessa possiede pochi attributi intrinseci descrittivi. Scegliendo come riferimento il BoundingBox abbiamo potuto esprimere, nel formalismo proposto, tutte le relazioni spaziali di base fra gli oggetti, definendo una teoria di composizione tra le parti per rappresentare le totalità.
A riguardo del punto 6), invece, abbiamo definito un formalismo, che utilizza un insieme esteso di relazioni spaziali qualitative, *vincolate a rispettare* un insieme di assiomi e un insieme di proprietà (si veda avanti).Il formalismo è un sottoinsieme di Flora2[6] un'implementazione della Frame Logic[15]. Le regole, le estensioni, i vincoli ed il ragionatore, che proponiamo in questo lavoro, sono interamente definiti in Flora2, anche se, per motivi che riporteremo in avanti, abbiamo implementato un meccanismo prototype-based differente da quello standard, calss-based, di Flora2. Gli elementi di base del linguaggio Flora2 sono i seguenti:

(per la definizione delle classi e le relazioni tassonomiche fra classi)

| | |
|---|---|
| x::z | x è una sottoclasse z; |
| x[s =>y] | s è un attributo di tipo y della classe x; |
| y[s=>> {t1,t2, ..}] | s è uno slot di y a valori multipli di tipo t1,t2, ..; |

(per la definizione delle istanza)

| | |
|---|---|
| a:x | a è istanza di y; |
| idx: x[s->vx] | vx è il valore dell'attributo s della classe x; |
| idx: x[s->>{v1,v2, ..}] | v1, v2, .. sono i valori dell'attributo s della classe x. |

(idx è l'indentificatore logico dell'oggetto - conosciuto nella programmazione ad oggetti come Oid)

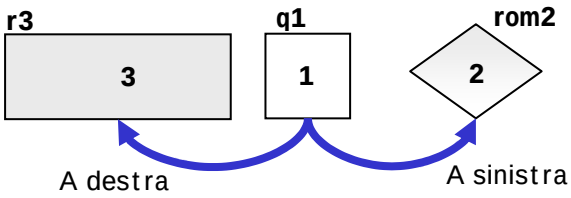
Riportiamo un esempio:

**/* Tassonomia e definizione delle classi */**

```
parallelogrammi::figure_piane.
rettangoli::parallelogrammi.
rombi::parallelogrammi.
quadrati::rettangoli.
triangoli::figure_piane.
figure_piane[numero_figura=>integer,
             colore=>string,
             a_sinistra=>figure_piane,
             a_destra=>figure_piane].
rombi[diagonale1=> integer, diagonale2=> integer].
rettangoli[base=> integer, altezza=> integer]
```

**/* Istanze */**

```
q1:quadrati[base->3, colore->bianco,
            numero_figura->1, a_destra->r3].
r3:rettangoli[altezza->3, base->7,
              colore->grigio, numero_figura->3].
rom2:rombi[colore>biancogrigio,numero_figura->2,
           diagonale1->3,diagognale2->5,
           a_destra->q1].
```




**/* Assiomi */**

```
Y[a_sinistra->X]:-X[a_destra->Y]. (inverse slot)
X[a_destra->Y]:-Y[a_sinistra->X].
X[a_sinistra->Z]:-X[a_sinistra>Y],Y[a_sinistra->Z].
X[a_destra->Z]:-X[a_destra->Y],Y[a_destra->Z]
quadrati[altezza->X, base->X].
```

**/* Esempi di query */**

```
?- I:romb1[colore->Y].
?- L:romb1[a_sinistra->X].
?- I:C[A->V].
```

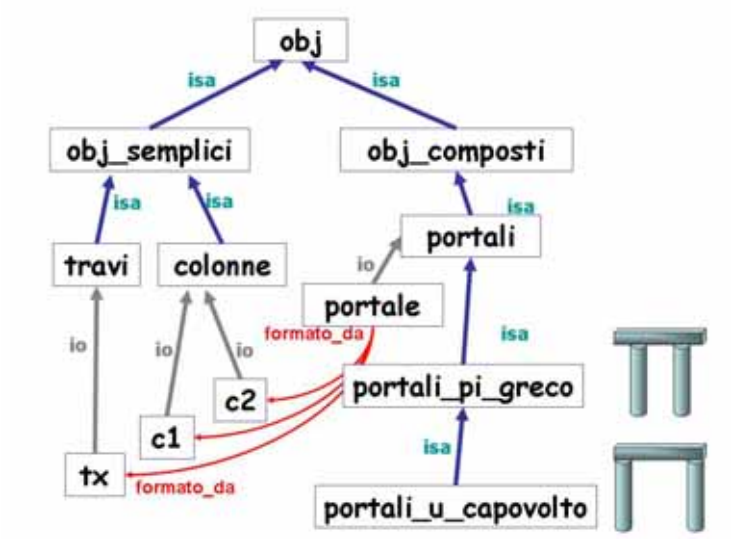
L'approccio da noi utilizzato, a riguardo del paradigma computazionale, è stato quello di adottare una metodologia che si fonda su un paradigma denominato in letteratura, prototype – based[16]. Esistono due modi di rappresentare i concetti generali: tramite insiemi e tramite prototipi. Un primo modo è quello di costruire il concetto di classe, un'entità che astrae le caratteristiche comuni ad un insieme di oggetti i quali sono istanze di questa classe. Caratteristiche aggiuntive possono essere contenute nelle sottoclassi, e la condivisione tra

classe e sottoclasse avviane mediante l'ereditarietà. Un secondo modo è quello basato sui prototipi e sulla delega[16]. La delega rappresenta l'operazione di affidare ad un altro oggetto un compito, qualora un oggetto non è in grado soddisfare una particolare richiesta. E' possibile mediante un tale approccio definire prototipi rappresentanti "famiglia" di oggetti, dove ogni oggetto condivide con il prototipo tutte le informazioni comuni, e nello tesso tempo ha immagazzinato nella propria struttura lo caratterizzano e lo differenziano dal prototipo.

Nell'approccio di rappresentazione che proponiamo, al fine di costruire tassonomie di concetti, utilizziamo gli standard concetti di classe, sottoclasse ed istanza dei linguaggi orientati agli oggetti e della Frame Logic. Ma, diversamente a quanto accada in tali approcci, *arricchiamo* le definizioni delle classi con prototipi – strutture che sono utilizzate per effettuare eventuali completamenti che si rendono necessari per le istanziazioni degli oggetti.

## 2 Formalismo e paradigma

In questo paragrafo presenteremo un formalismo per il ragionamento spaziale qualitativo, mostrando, di volta in volta, come sia conforme alle linee guida, elencati nell'introduzione, presentati in [1]. Il nostro formalismo è orientato al ragionamento al fine di costruire una scena, a partire da relazioni spaziali anche non espressamente date. Ciò è stato fatto costruendo un apparato deduttivo(assiomatica).




Uno dei principali obiettivi che ci siamo posti per la costruzione del formalismo, è stato quello di voler definire un linguaggio che permettesse di rappresentare classi di oggetti fisici quanto più dettagliate e specifiche, e nello stesso tempo, che permettesse di *catturare* eventuali *sfumature,* fra le classi di una tassonomia, adottando un graduale meccanismo di raffinamento. La sintassi e la semantica del formalismo proposto ricalcano, ovviamente, quella della Frame Logic, ed è presentato di seguito:

sottoclasse :: classe. --definizione sottoclasse
istanza : classe. --definizione di istanza/prototipo
istanza_ogg_semplice[ha_forma -> primitive, height->float, width->float, depth->float].
--definizione delle dimensioni del boundingbox e della primitive degli oggetti semplici nella dichiarazione di istanza/prototipo
istanza_ogg_composto[formato_da->>{lista_oggetti},relazioni_spaziali->>{lista_relazioni}].
--definizione di oggetto composto, mediante F-molecola a due slot:
*formato_da* (contenente la lista degli oggetti semplici o composti),
*relazioni_spaziali (*contenente la lista delle relazioni spaziali sugli oggetti componenti)
istanza_relazione_spaziale[obj1->oggetto1, obj2->oggetto2].
--definizione di relazione spaziale, i due slot *obj1* e *obj2* rappresentano gli oggetti (semplici o composti) ai quali si riferisce la relazione.

La BNF dell'intero formalismo è riportata in Appendice A. Per fornire una prima idea della rappresentazione adoperata, riportiamo un esempio nel quale è rappresentata una gerarchia di portali definiti all'interno della base di conoscenza, utilizzando oggetti semplici (obj_semplici) e composti (obj_composti)[2].

```
/* Classi di oggetti semplici e composti */
obj_semplici::obj.
obj_composti::obj.
portali::obj_composti.
travi::obj_semplici.
colonne::obj_semplici.

/*Definizione prototipo per portali */
portale1:portali.
portale1[ formato_da ->> {tx, c1, c2} ].
/* Definizione prototipo per portale Pi-greco */
portali_pigreco::portali.
portale_pigreco1:portali_pigreco.
portale_pigreco1[ relazioni_spaziali->>{s1,s2,i1,i2},
                  delega ->> {portale1} ].
/*Definizione prototipo per portale u capovolta*/
portali_u_cap::portali_pigreco.
portale_u_cap1:portali_u_cap.
portale_u_cap1[relazioni_spaziali ->>{adx1, asx1},
               delega ->> {portale_pigreco1}].
```

```
/* Oggetti semplici */
tx:travi.
c1:colonne.
c2:colonne.
travi [ha_forma -> parallepipedi].
colonne [ha_forma -> cilindri].

/* Relazioni spaziali la classe portali Pi-greco*/
s1:sopra[obj1->tx, obj2->c1].
s2:sopra[obj1->tx, obj2->c2].
i1:in_contatto[obj1->tx, obj2->c1].
i2:in_contatto[obj1->tx, obj2->c2].

/*Relazioni spaziali per prototipo portali_u_cap*/
asx1:allineato_asinistra[obj1->tx, obj2->c1].
adx1:allineato_adestra[obj1->tx, obj2->c2].
```

Come si può osservare, il formalismo permette di rappresentare le totalità (ad esempio portale1) in maniera tale da contenere esplicitamente, nella definizione, le parti di cui esse sono formate (portali[ formato_da ->> {tx, c1, c2}), in altre parole il formalismo è esplicito.
Le parti stesse (tx, c1, c2), inoltre, possono essere sia oggetti semplici che a loro volta composti. Nell'esempio, ovviamente, la classe portali gioca il ruolo di classe astratta. Le classi concrete sono definite, nel formalismo proposto, definendo o ulteriori relazioni spaziali qualitative fra le parti o aggiungendo nuove parti. Per la classe portali_pigreco, definita come sottoclasse della classe portali, le relazioni[3] s1, s2, i1 e i2 sono appunto relazione spaziali qualitative fra le parti c1, c2 e tx della totalità portali_pigreco. Nell'esempio, infine, a partire dalla classe portali_pigreco è stata definita una sottoclasse portale_u_cap, dove le relazioni spaziali qualitative adx1, asx1 conferiscono una particolare caratterizzazione alla classe (mediante il suo prototipo), ossia, di avere entrambe le colonne poste alle estremità della trave.
Vogliamo sottolineare che il formalismo rispetta le linee guida elencate in precedenza. Oltre ad adottare un approccio esplicito (punto 1), permette una chiara distinzione fra le parti e altri attributi di una totalità (punto 2). Il formalismo, inoltre, permette di riferirsi alle parti mediante generici nomi (punto 4), in modo tale da potere utilizzare una definizione di parte in più definizioni di totalità. A riguardo dei punti 5) e 6), come si può osservare anche dall'esempio riportato, il formalismo permette di esprimere relazioni fra parti e totalità (es. formato_da ->> {tx, c1, c2}) e fra le parti di una totalità (es. s1:sopra[obj1->tx, obj2->c1], i1:in_contatto[obj1->tx, obj2->c1], asx1:allineato_asinistra[obj1->tx, obj2->c1]).
Come si può notare, inoltre, le istanze/prototipi di oggetti composti possiedono uno slot denominato *delega*, che contiene la lista di oggetti con i quali l'oggetto dato condivide

[2] La sintassi di tale formalismo è simile a quella proposta in [4][22].
[3] Nell'esempio abbiamo riportato una forma semplificata della relazione *sopra,* definita nel formalismo. In avanti riporteremo la forma completa di tale relazione.

informazioni (slot formato_da e slot relazioni_spaziali) secondo il meccanismo proprio del paradigma prototype – based.
Per supportare il paradigma *prototype – based*, dato che il modello computazionale è diverso da quello *class – based* in quanto non sfrutta l'ereditarietà, è stato necessario implementare un meccanismo di delega, per far si che qualora un oggetto non fosse in grado di rispondere ad una certa richiesta, questi attivasse uno degli oggetti presenti nella sua lista di delega. Per condividere la conoscenza tra prototipi, abbiamo introdotto la regola:

```
Obj[_Slot->>Part] :- Obj[delega->>Deleg], Deleg[_Slot->>Part].
```

Questo meccanismo permette di condividere conoscenze fra prototipi anche non appartenenti alla stessa "famiglia".

## 3 Rappresentazione e apparato deduttivo per il ragionamento spaziale qualitativo

Uno degli obiettivi principali del lavoro è stato quello di voler definire un formalismo in grado di rappresentare i concetti di base e nello stesso tempo, le peculiarità, i dettagli e le *sfumature* degli artefatti complessi. Un tale tipo di formalismo, riteniamo, risulta adeguato per la realizzazione di algoritmi orientati alla visualizzazione di artefatti partendo da descrizioni qualitative. Nella descrizione qualitativa di una scena, le informazioni possono essere parziali e contenere degli errori. Il completamento delle descrizioni, o la correzione degli errori, possono essere effettuati (in modalità automatica o semiautomatica), solo se si è in grado di inserire, in un sistema specializzato a eseguire inferenze su relazioni spaziali (ragionatore), completi prototipi ai quali un ragionatore può fare riferimento. L'approccio computazionale che suggeriamo, quindi, è quello di avere descrizioni dettagliate ed *espressive* di istanze di classe, per le definizioni dei prototipi di riferimento. In maniera tale da permettere, per il problema del completamento, un confronto fra l'istanza che si vuole disegnare e il prototipo esistente nella base di conoscenza. La scoperta dei dati e delle relazioni mancanti, può essere in tal modo effettuata, da parte di un sistema che esegue il confronto sopra citato, in base all'assiomatica di relazioni spaziali definita (per maggiori dettagli di un tale sistema si veda avanti).

Le relazioni spaziali qualitative di base da noi scelte, per la costruzione dell'assiomatica, sono le direzionali e le topologiche.

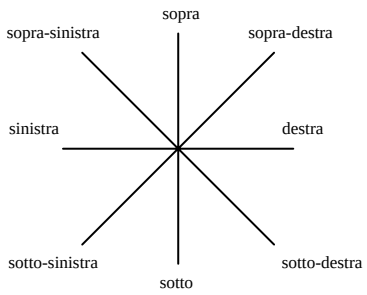
### 3.1 Relazioni direzionali

La scelta dell'insieme di relazioni spaziali direzionali è ricaduta su alcune relazioni per le quali c'è poca ambiguità in termini interpretativi (semantici), ma soprattutto perché, tali relazioni, permettono di descrivere la scena dal punto di vista dell'osservatore. I concetti di base scelti sono:

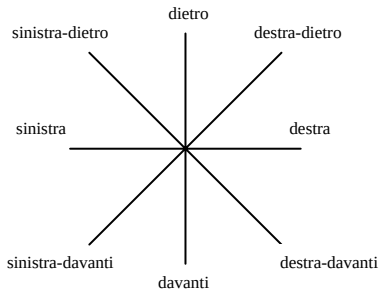
```
RelDir = {sopra, sotto, destra, sinistra, davanti, dietro}
```

Tali relazioni di base possono essere utilizzate nelle istanze congiuntamente (eccetto conflitti) per rappresentare altre relazioni direzionali. Nello schema che segue mostriamo le combinazioni che si possono ottenere combinando le relazioni di base RelDir.

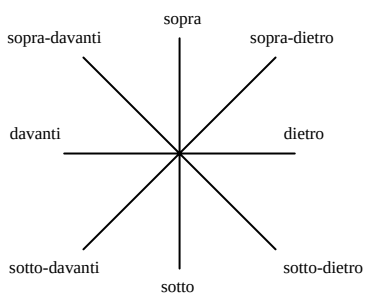
Alle relazioni qualitative RelDir sono state associate alcune proprietà: la transitività e la dualità.




**Vista frontale**




**Vista superiore**




**Vista destra**

```
trans(R1-R2):Rel[obj1->X, obj2->Z] :-
        R1:Rel[obj1->X, obj2->Y], R2:Rel[obj1->Y, obj2->Z].

dual(R): Relinv[obj1->X, obj2->Y] :-
        R:Rel[obj1 -> Y, obj2 -> X], inv(Rel,Relinv).
```

Il primo predicato inferisce che, esiste una relazione Rel tra X e Z, se esiste Rel tra X e Y, e Rel Y e Z (transitività). La seconda inferisce che esiste una relazione Relinv tra X e Y, se esiste una relazione Rel tra Y e X, e una relazione inv(Rel, RelInv) (RelInv è la relazione inversa della relazione Rel). Per ciascuna relazione spaziale qualitativa direzionale, sono state definite le inverse:

```
inv(sopra,sotto), inv(sotto,sopra), inv(sinistra,destra),
inv(destra,sinistra), inv(davanti,dietro), inv(dietro,davanti).
```

Le proprietà delle relazioni qualitative sono state definite nell'assiomatica come predicati del secondo ordine (adoperando quantificazioni sui predicati). Tale scelta è giustificata dal fatto che nell'apparato di deduzione è possibile estendere tutte le relazioni spaziali qualitative di base (non solo quelle direzionali), senza la necessità di aggiungere per ogni nuova relazione altre regole.
In aggiunta, alle relazioni direzionali precedenti, abbiamo definito altre relazioni che permettono di descrivere l'allineamento relativo tra oggetti, sfruttando la conoscenza della loro posizione e dimensione. Le relazioni sono:

```
RelAllign = {allineato_inalto, allineato_inbasso, allineato_adestra, allineato_asinistra,
        allineato_suldavanti, allineato_suldietro}.
```

Anche per queste ultime relazioni qualitative, abbiamo associato le proprietà di simmetria e di transitività:

```
simm(R):Rel_allineamento[obj1->X, obj2->Y] :-
        R:Rel_allineamento[obj1->Y, obj2->X].

trans(R1-R2): Rel_allineamento[obj1 -> X, obj2 -> Z] :-
        R1: rel_allineamento[obj1 -> X, obj2 -> Y],
        R2: rel_allineamento[obj1 -> Y, obj2 -> Z], X\=Z.
```

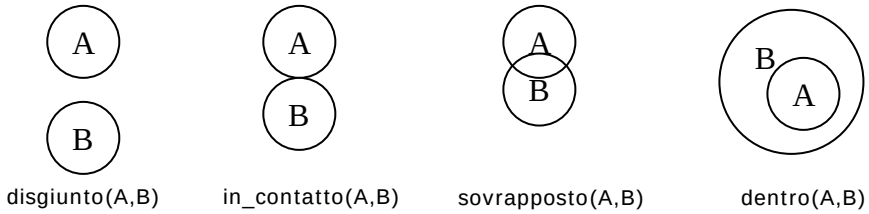
La prima relazione permette di inferire che esiste un allineamento tra gli oggetti X e Y, se esiste un allineamento tra Y e X. La seconda permette di inferire che esiste un allineamento, tra gli oggetti X e Z, se esistono gli allineamenti tra X e Y, e tra Y e Z (con X diverso da Z).

## 3.2 Relazioni topologiche

Per il formalismo proposto, l'insieme delle relazione topologiche di base è il seguente:

```
RelTopol = {disgiunto, in_contatto, sovrapposto, dentro}
```

Nella figura seguente diamo una caratterizzazione di tali relazioni topologiche:




disgiunto(A,B) in_contatto(A,B) sovrapposto(A,B) dentro(A,B)

La proprietà di simmetria delle relazioni disgiunto, in_contatto e sovrapposto, sono state definite nel modo seguente:

```
simm(R):disgiunto[obj1->X, obj2->Y] :-
        R:disgiunto[obj1->Y, obj2->X].
simm(R):sovrapposto[obj1->X, obj2->Y] :-
        R:sovrapposto[obj1->Y,obj2->X].
simm(R): in_contatto [obj1->X, obj2->Y] :-
        R: in_contatto [obj1->Y, obj2->X].
```

La proprietà transitiva della relazione dentro:

```
trans(R1-R2):dentro[obj1->X, obj2->Z] :-
        R1:dentro[obj1->X, obj2->Y],
        R2:dentro[obj1->Y, obj2->Z].
```

Per le relazioni dentro e sovrapposto abbiamo definito una regola di composizione che permette di derivare che due oggetti X e Z sono sovrapposti, se Y è dentro un oggetto X, e Y è sovrapposto a Z.

```
comp(R1-R2):sovrapposto[obj1->X, obj2->Z] :-
        R1:dentro[obj1->Y, obj->X],
        R2:sovrapposto[obj1->Y, obj2->Z].
```

Per le relazioni dentro e disgiunto, invece, abbiamo definito una regola di composizione che permette di derivare che due oggetti X e Z sono disgiunti se X è dentro un oggetto Y e Y è disgiunto da Z.

```
comp(R1-R2):disgiunto[obj1->X, obj2->Z] :-
        R1:dentro[obj1->X, obj2->Y], R2:disgiunto[obj1->Y, obj2->Z].
```

## 3.3 Proprietà algebriche delle relazioni: idempotenza, antiriflessiva e composizione delle inverse

Nel formalismo proposto abbiamo implementato le proprietà algebriche di idempotenza, di antiriflessività e composizione delle inverse mediante le seguenti regole[4]:

```
idempotenza():-
        while((R1:Rel[obj1->X,obj2->Y], R2:Rel[obj1->X,obj2->Y], (R1\=R2)))
        do ( delete{R2:Rel}).

antiriflessiva() :-
        deleteall{ R:Dir | R:Dir[obj1->X, obj2->X] }.

cancella_inverse() :-
        deleteall{R1:rel(Rel1,Dist), R2:rel(Rel2,Dist) |
           R1:rel(Rel1,Dist)[obj1->X, obj2->Y],
           R2:rel(Rel2,Dist)[obj1->X, obj2->Y],
```

[4] I predicati extra-logici presenti in tali regole sono costrutti dell'implementazione utilizzata Flora2:
*while(Condition)…do(Action)* è semanticamente identico al costrutto dei linguaggi imperativi, cerca tutte le derivazioni nelle quali è verificata la *Condition* ed esegue *l'Action*.
*Deleteall*, come *delete*,*insertall* ed *insert* sono predicati Flora2, mediante i quali è possibile, rispettivamente: cancellare e inserire tutti o uno i predicati passati come parametro (similmente al *retract* e *assert* del Prolog)

```
    Rel1::rel_direz, Rel2::rel_direz,
    inv(Rel1,Rel2), Dist::dist_qual},
deleteall{R1:rel(Rel,Dist), R2:rel(Rel,Dist) |
    R1:rel(Rel,Dist)[obj1->X, obj2->Y],
    R2:rel(Rel,Dist)[obj1->Y, obj2->X],
    Rel::rel_direz, Dist::dist_qual}.
```

(l'idempotenza elimina tutte le relazioni uguali lasciandone una sola, l'antiriflessiva elimina tutte le relazioni che sono applicate al medesimo oggetto, la composizione delle inverse elimina tutte le relazioni che sono una l'inversa dell'altra).

## 3.4 Distanza qualitativa e sue proprietà

Nell'assiomatica di ragionamento qualitativo, abbiamo adottato una nozione di distanza qualitativa definita mediante sei valori qualitativi di riferimento:

D = {moltovicino, vicino, media, lontano, moltolontano}

Tali valori sono stati scelti come traduzione dei valori utilizzati nei sistemi *multi-step distance* presentati in [8] nel quale sono utilizzati i simboli *C, c, f, F* come abbreviazioni di *close(vicino)* e *far(lontano)*, ed utilizzando le lettere maiuscole e le minuscole per enfatizzare la più o meno vicinanza o lontananza. In aggiunta abbiamo inserito un valore mediano per una raffinazione ulteriore.
In corrispondenza di tali valori, il ragionatore da noi definito, per il ragionamento spaziale qualitativo, associa distanze (quantitative) in funzione del particolare contesto, questo tipo di intuizione è presente in [17]. Il contesto, nel nostro caso, viene stabilito dai tipi, dalle dimensioni e dalle relazioni (direzionali) degli oggetti presenti nella scena. La distanza media è calcolata dalle dimensioni dei BoundingBox dei due oggetti X (Xweight, Xheight, Xdepth) e Y (Yweight, Yheight, Ydepth) interessati dalla relazione, utilizzando la regola (le altre calcolate in funzione di tale regola):

```
dist[valDist->DistMedia, obj1->X, obj2->Y] :-
        boundingboxX[wbb->Xweight, hbb->Xheight, dbb->Xdepth ],
        boundingboxY[wbb->Yweight, hbb->Yheight, dbb->Ydepth ],
        DimBBX = ((Xweight+Xheight+Xdepth)/3.0),
        DimBBY = ((Yweight+Yheight+Ydepth)/3.0),
        DistMedia = ((DimBBX + DimBBY)/2.0).
```

Per la nozione di distanza qualitativa, abbiamo definito un ordinamento tra distanze (qualitative) e un'operazione di addizione di distanze addDistq/3.
Abbiamo utilizzato il predicato succ(D1,D2) per indicare che "una distanza D1 segue nell'ordinamento una distanza D2", ordinando i valori qualitativi di riferimento nel seguente modo:

succ(moltolontano, lontano), succ(lontano, media), succ(media, vicino), succ(vicino, moltovicino)

A partire da tali relazioni di precedenza, abbiamo definito il predicato max/3 che calcola la distanza più grande tra due distanze qualitative:

```
max(X,X,X):- !.
max(X,Y,X) :- succ(X,Y), !.
max(X,Y,Y) :- succ(Y,X), !.
max(X,Y,X) :- succ(X,Z), max(Z,Y,Z),!.
max(X,Y,Y) :- succ(Y,Z), max(Z,X,Z),!.
```

In altri approcci relativi la gestione delle distanze qualitative[8] sono suggeriti tre diversi metodi di gestire la composizione dei valori di distanza, siamo dell'opinione che non esista una scelta migliore universalmente, ma piuttosto riteniamo che a seconda del dominio sia necessario scegliere l'approccio che meglio lo approssima. La definizione di somma di due distanze qualitative, da noi utilizzata, addDistq(D1, D2, Somma) con D1 *e* D2 $\in$ D, è:

```
addDistq(D1,D2,R) :- R=moltolontano,!.
addDistq (D1, D2, R) :- max(D1, D2, R1), (R1\=moltolontano, succ(R,R1)),!.
```

Alcuni esempi di applicazione di tale definizione sono:

```
?- addDistq (moltovicino, moltovicino, R)    →    R=vicino
?- addDistq (moltovicino, lontano, R)        →    R=moltolontano
```

### 3.5 Inferenze fra direzioni e distanze

Per costruire connessioni fra direzioni e distanze del tipo rel(sopra,vicino) e rel(destra, lontano), abbiamo introdotto il predicato del secondo ordine rel(Rel,Dist)[obj1->X, obj2->Y] a partire dal quale è possibile definire regole estremamente compatte, come la regola dual:

```
dual(R):rel(Rel,Dist)[obj1->X, obj2->Y] :-
            R:rel(Relinv,Dist)[obj1->Y, obj2->X], inv(Rel,Relinv).
```

Una possibile istanziazione della regola appena riportata è:

```
id:rel(sopra,vicino)[obj1->capitello, obj2->fusto] ->
                    dual(id):rel(sotto,vicino)[obj1->fusto, obj2->capitello]
```

Come si può notare, l'adozione di regole del secondo ordine, permette un notevole risparmio di definizioni di regole. Quantificando sulle variabili della regola, infatti, è possibile ragionare su tutti i predicati di relazioni spaziali e su tutti quelli delle distanze qualitative.
Le relazioni sono transitive lungo la stessa direzione e stessa distanza, applicando la regola di somma precedentemente introdotta:

```
trans(R1-R2): rel(Rel,DistNew)[obj1->X, obj2->Z] :-
                    R1:rel(Rel,Dist)[obj1->X, obj2->Y],
                    R2:rel(Rel,Dist)[obj1->Y, obj2->Z],
                    (X\=Z), succ(DistNew,Dist).
```

La seconda regola, invece, si applica a relazioni con la stessa direzione ma a distanze diverse:

```
trans(R1-R2): rel(Rel,Dist2)[obj1->X, obj2->Z] :-
                    R1:rel(Rel,Dist1)[obj1 -> X, obj2 -> Y],
                    R2:rel(Rel,Dist2)[obj1 -> Y, obj2 -> Z],
                    (X\=Z), Rel::rel_direz, Dist1::dist_qual,
                    Dist1::dist_qual, max(Dist2,Dist1).
```

Se esistono due relazioni uguali applicate una a X e Y con distanza Dist1, l'altra a Y e Z con Dist2, con Dist2=>Dist1, si può inferire che X è in relazione con Z con distanza Dist2.

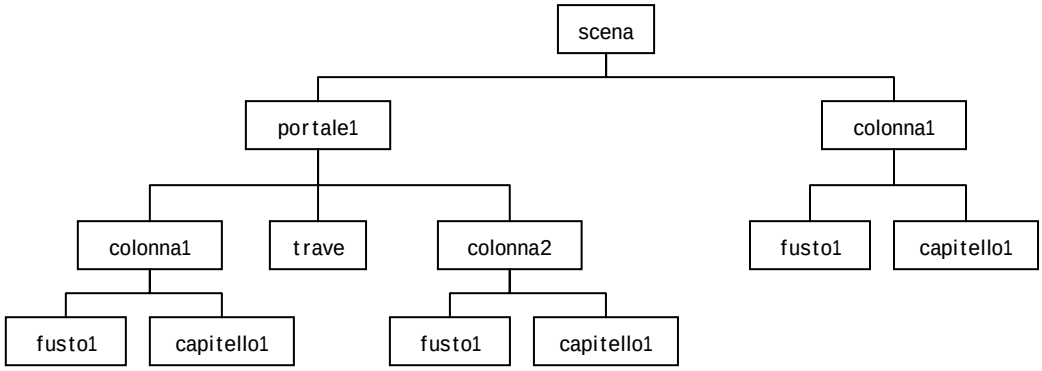
## 4 Applicazione al problema della visualizzazione di relazioni spaziali qualitative

L'applicazione che proponiamo, realizza il completamento di oggetti, delle relative relazioni spaziali qualitative, delle distanze tra gli oggetti, e la generazione delle coordinate necessarie per il disegno della scena: realizzando una visione tridimensionale della scena descritta mediante il formalismo introdotto.

Il primo passo è quello di effettuare il completamento dell'istanza confrontandola con l'ontologia di riferimento. Il formalismo definito permette di introdurre classi che vincolano fortemente le istanze. Ciò permette di completare le istanze degli oggetti che si cerca di visualizzare. Il completamento interessa sia le parti dell'oggetto, sia le relazioni spaziali qualitative tra di esse. Le informazioni dell'ontologia coprono anche informazioni riguardanti le dimensioni degli oggetti primitivi, e la primitiva da utilizzare per il rendering. Combinando queste informazioni, con le caratteristiche derivate durante il calcolo degli oggetti composti, è possibile definire geometricamente la scena.

## 4.1 Procedura di visualizzazione

L'istanza, che descrive la scena, è rappresentata mediante un albero. La scena è analizzata, in maniera ricorsiva, per singolo nodo.




Si analizza ogni nodo nel suo sistema di riferimento (cartesiano) locale, partendo dai nodi che hanno come figli, solo foglie (oggetti primitivi). Successivamente, sono calcolate le coordinate di ogni oggetto (figlio del nodo considerato), applicando le proprietà e gli assiomi precedentemente introdotti e applicando regole di conversione per passare dalle relazioni spaziali qualitative, alle coordinate di posizionamento degli oggetti. Riportiamo un sottoinsieme di tali regole:

Ist:rel(destra,Dist)[obj1->A,obj2->B] :- X(A) = X(B) + Dist.
Ist:rel(sinistra,Dist)[obj1->A,obj2->B] :- X(A) = X(B)- Dist.
Ist:rel(sopra,Dist)[obj1->A,obj2->B] :- Y(A) = Y(B) + Dist.
Ist:rel(sotto,Dist)[obj1->A,obj2->B] :- Y(A) = Y(A) - Dist.
Ist:rel(dietro,Dist)[obj1->A,obj2->B] :- Z(A) = Z(B) + Dist.
Ist:rel(davanti,Dist)[obj1->A,obj2->B] :- Z(A) = Z(B) - Dist.

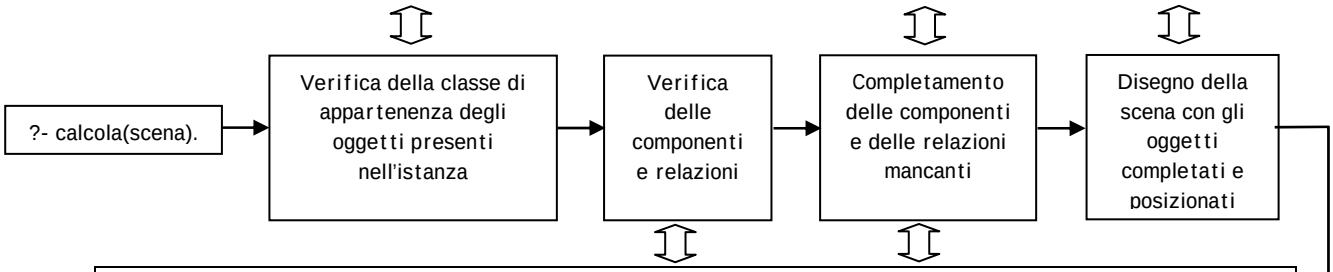
Le coordinate di un oggetto A sono calcolate da quelle dell'oggetto B sommando (sottraendo) la distanza Dist che li separa. Seguendo questa metodologia, partendo da un oggetto posto al centro della scena, si posizionano tutti gli altri oggetti componenti il nodo considerato. Terminato il posizionamento, si calcola il BoundingBox del nodo e si itera il calcolo sino alla radice dell'albero. Posizionati, infine, localmente i singoli oggetti si effettua un calcolo che permette il posizionamento globale nella scena.
In questo lavoro abbiamo realizzato un sistema per la visualizzazione delle relazioni spaziali qualitative, secondo l'approccio sopra riportato. Nello schema seguente riportiamo in maniera schematica i passi del funzionamento di tale sistema.

**Istanza di scena**

```
scena[formato_da->>{portale1,portale2},
      relazioni_spaziali->>{dxpp1,dipp1}].
dxpp1:rel(destra,vicino)[obj1->portale1, obj2->portale2].
dipp1:rel(dietro,vicino)[obj1->portale1, obj2->portale2].
portale1:portali_u_cap.
portale2: portali_u_cap.
portale1[formato_da->>{trave,col1,col2},
         relazioni_spaziali ->>{sot1,sot2,all1}].
portale2[formato_da->>{trave,col1,col2},
         relazioni_spaziali ->>{sot1,sot2,all1}].
trave:tx.
col1:c1.
col2:c2.
sot1:rel(sopra,zero)[obj1->trave, obj2->col1].
sot2:rel(sopra,zero)[obj1->trave, obj2->col2].
all1:allineato_suldietro[obj1->col2, obj2->col1].

col1[formato_da->>{fusto1, capitello1},
     relazioni_spaziali->>{}].
fusto1:fusto.
capitello1:capitello.
sotto1:rel(sotto,zero)[obj1->fusto1, obj2->capitello1].

col2[formato_da->>{fusto1, capitello1},
     relazioni_spaziali ->>{sotto1}].
fusto1:fusto.
capitello1:capitello.
sotto1:rel(sotto,zero)[obj1->fusto1, obj2->capitello1].
```




**Ontologia di riferimento**

```
/* Definizione classe Portale Pi-greco */
portali_pigreco::portali.
portale_pigreco1:portali_pigreco.
portale_pigreco1[relazioni_spaziali ->> {s1, s2, i1, i2},
                 delega ->> {portale1}].
/* Definizione classe Portale ad u capovolta */
portali_u_cap::portali_pigreco.
portale_u_cap1: portali_u_cap.
portale_u_cap1[relazioni_spaziali ->> {adx1, asx1},
               delega ->> {portale_pigreco1}].

/* Relazioni spaziali la classe portali Pi-greco*/
s1:sopra[obj1 -> tx, obj2 -> c1].
s2:sopra[obj1 -> tx, obj2 -> c2].
i1:in_contatto[obj1 -> tx, obj2-> c1].
i2:in_contatto[obj1 -> tx, obj2-> c2].

/* Relazioni spaziali per la classe portali_u_cap*/
asx1:allineato_asinistra[obj1 -> tx, obj2 -> c1].
adx1:allineato_adestra[obj1 -> tx, obj2 -> c2].
```

Rendering 3D

## 4.2 Nota sull'architettura software

Il sistema di visualizzazione di relazioni spaziali qualitative realizzato ha reso necessario l'integrazione di due ambienti di programmazione: Flora2, implementazione di Frame Logic basata su XSB [5] (un sistema Prolog OpenSource) e Java, messi in connessione mediante un bridge denominato Java2Flora, che permette effettuare *interrogazioni* Flora2 da Java.
Per la realizzazione e il disegno della scena sono state utilizzate le API Java3D della SUN. Java3D è una libreria in grado di effettuare disegno tridimensionale implementando le caratteristiche dei moderni motori tridimensionali uniti alle *capacità* del linguaggio Java.

## 5 Lavori di riferimento e discussione

Nel lavoro [3] è stato proposto un formalismo per la rappresentazione di strutture architettoniche classiche. In tale lavoro si è affrontato il problema di rappresentare relazione meronomiche parte-totalità utilizzando un approccio esplicito. Il formalismo permette di esprimere rappresentazioni del tipo:

<Concept identifier (187)> <Synonyms (colonna)> <Superconcept (159)> < Form (132)> < Stuff (158)> <Components (361 (exactly 1)) (362 (exactly 1)) (37 (exactly 1))> <Position (upon 362 361) (upon 37 362)>
<Glossa (Piedritto a sezione circolare composto di base, fusto monolitico o a segmenti cilindrici, capitello, con funzione portante o decorativa)> con 159 “piedritto”, 361 “base”, 362 “fusto”, 37 “capitello”, 53 “trabeazione”, 132 “cilindro”, 158 “pietra”.

Tale formalismo è stato definito nello specifico settore linguistico per l’archeologia, e riteniamo che poca attenzione sia stata data agli aspetti deduttivi-computazionali. La rappresentazione appare avere, comunque, qualche limite in termini di *espressività*. Con tale formalismo, ad esempio, non risulta possibile definire relazioni spaziali differenti per parti che appartengono ad una stessa categoria.
Relativamente ad altri formalismi per il ragionamento spaziale come [18][19][21] riteniamo che le informazioni, sulle quali si effettuano ragionamenti, siano insufficienti nel dominio di interesse del nostro lavoro, tali formalismi (denominati RCC) si occupano prevalentemente di relazioni topologiche che risultano essere inutilizzabili per ricreare la configurazione di una scena in tre dimensioni. Infatti, una relazione topologica ci può *indicare* dove un oggetto non si trova, ma non dove esso si trova. Dire che A è disgiunto da B, ci informa del fatto che A non è a contatto con B, ma che si può trovare in qualsiasi punto della scena. Questo è poco utile nel dominio da noi analizzato.
Riguardo al problema del ragionamento spaziale qualitativo, alcuni lavori [9] si sono occupati degli oggetti estesi (non puntiformi), utilizzando relazioni topologiche (*adiacenti*, *sovrapposti*,….), non affrontando, però, il problema delle distanze. Tale approccio è stato seguito anche da altri [8]. Altri lavori (relativi a sistemi GIS) hanno cercato di *catturare* il ragionamento spaziale qualitativo mediante rappresentazioni algebriche [7][10]. In [7] è proposto un approccio dove si definiscono le relazioni spaziali, e le loro inferenze, in un scenario spaziale composto di oggetti di tipo contenitori e superfici. Il lavoro suggerisce una metodologia di approccio generale al problema del ragionamento spaziale. Purtroppo esso si occupa soltanto di due tipologie di oggetti (contenitori e superfici). L'investigazione delle deduzioni possibili è completa, ma presenta il problema di non considerare né le dimensioni, né la categoria di appartenenza di ciascun oggetto.

## 6 Ringraziamenti

Ringraziamo Giovanni Criscuolo e Nicola Guarino per gli utilissimi suggerimenti che ci hanno fornito in questo lavoro. Uno speciale ringraziamento va a Antonio Sorgente per il suo aiuto su alcune definizione dell’apparato inferenziale realizzato.

## 7 Bibliografia

[1] A. Artale, E. Franconi, N. Guarino, L. Pazzi, *Part-Whole Relations in Object-Centered Systems: An overview,* Data & Knowledge Engineering (DKE) journal 20 347-383 – North-Holland, Elsevier, 1996.
[2] N. Guarino, *Formal Ontology in Information Systems,* Proceedings of FOIS’98, Trento, Italy. Amsterdam, IOS Press, 6-8 June 1998.
[3] Cappelli A., Novella Catarsi A., Nichelassi P., Moretti L, *Un formalismo per rappresentare strutture architettoniche classiche,* Atti del Convegno, Contesti Virtuali e Fruizione dei Beni Culturali, Napoli, 22-23 Maggio 2003.
[4] A. Calabrese, F. Mele, L. Serino, A. Sorgente, O. Talamo, *Rappresentazioni di conoscenze spaziali di siti archeologic*i, AI*IA 2003 - Ottavo Congresso Nazionale dell'AI*IA, Polo didattico "L. Fibonacci", Università di Pisa, 23-26 Settembre 2003.
[5] XSB Project v2.6 - http://xsb.sourceforge.net/
[6] Flora Project v0.92 - http://flora.sourceforge.net/

[7] M. Egenhofer and A. Rodríguez, *Relation Algebras over Containers and Surfaces: An Ontological Study of a Room Space*, Journal of Spatial Cognition and Computation, Vol. 1, No. 2, pp. 155-180, 1999.
[8] A. U. Frank, *Qualitative Spatial Reasoning about Distances and Directions in Geographic Space*, Journal of Visual Languages and Computing 3, 343-371, 1992.
[9] D. Papadias and T. Sellis, *Qualitative Representation of Spatial Knowledge in Two-Dimensional Space* VLDB Journal,3, 479-516, Ralf Hartmut Gfiting Editor, 1994.
[10] C. Eschenbach and L. Kulik, *An axiomatic approach to the spatial relations underlying left–right and in front of–behind*, KI-97 Advances in Artificial Intelligence (pp. 207 – 218). Berlin:Springer, 1997.
[11] D. M. Mark, S Svorou and D. Zubin, *Spatial terms and spatial concepts; geographic, cognitive, and linguistic perspectives*, International Geographic Infonnation Systems (IGIS) Symposinm, 1987.
[12] G. Retz-Schmidt, *Various views on spatial prepositions*, AI Magazine, 9, 95–105, 1988
[13] W.J.M. Levelt, *Some perceptual limitations on talking about space*, A.J. van Doorn, W.A. van der Grind & J.J. Koenderink (eds.): Limits in Perception, 1984.
[14] R. Casati, C. V. Achille, *I trabocchetti della rappresentazione spaziale*, Sistemi Intelligenti 11:1, 1999.
[15] M. Kifer, G. Lausen, J. Wu, *Logical Foundations of Object-Oriented and Frame-Based Languages*, Journal of ACM, 1995.
[16] H. Lieberman, *Using prototypical objects to implement shared behavior in object oriented systems*, Artificial Intelligence Laboratory Massachusetts Institute of Technology Cambridge, 1986.
[17] M. Worboys, *Nearness relations in environmental space*, Journal of GIS,2001.
[18] B. Bennett, *Logical Representations for Automated Reasoning about Spatial Relationships* PhD thesis, School of Computer Studies, University of Leeds, 1997.
[19] A.G. Cohn and S.M. Hazarika, *Qualitative spatial representation and reasoning: an overview*, Fundamenta Informaticae, 45:1--29, 2001.
[20] J. Renz and B. Nebel, *On the complexity of qualitative spatial reasoning: A maximal tractable fragment of the region connection calculus*,Artificial Intelligence, 108(1-2),1999
[21] M. Wessel, *On Spatial Reasoning with Description Logics*, Proceedings of DL, 2002.
[22] A.Sorgente, *Rappresentazione di relazioni spaziali mediante ontologie,*Tesi di laurea in Informatica, Università di Napoli Federico II, a.a. 2002-2003

## 8 Appendice A : BNF del formalismo

Presentiamo la BNF del formalismo proposto. Il formalismo si compone della definizione degli oggetti nella base di conoscenza e la descrizione della scena e delle istanze.

### BNF

```
Start ← Def_sottoclasse | Def_istanza | Def_istanza_ogg_semplice | Def_istanza_ogg_composto | Def_istanza_rel_spaziali.
/****************************** Definizione classe-sottoclasse ************************************************/
Def_sottoclasse ← Def_obj_semplici | Def_obj_composti | Def_rel_spaziali.
Def_obj_semplici ← Id :: IdOS | Id :: obj_semplici.
Def_obj_composti ← Id :: IdOC | Id :: obj_composti.
Def_rel_spaziali ← Id :: IdRel | Id :: Relazioni_spaziali.
/****************************** Definizione istanza ***********************************************************/
Def_istanza ← Def_ist_obj_semplici | Def_ist_obj_composti | Def_ist_rel_spaziali.
Def_ist_obj_semplici ← IdIOS : IdOS.
Def_ist_obj_composti ← IdIOC : IdOC.
Def_ist_rel_spaziali ← IdIRel : IdRel | IdIRel : Relazioni_spaziali.
/****************************** Definizione classe oggetto semplice *******************************************/
Def_classe_ogg_semplice ← IdOS[Def_slot_proprieta_ogg_semplice].
Def_slot_proprieta_ogg ← Forma_ogg, Dimensioni_BB, Dimensioni_ogg, Rotazioni_ogg|
                         Forma_ogg, Dimensioni_BB, Rotazioni_ogg |
                         Forma_ogg, Dimensioni_BB, Dimensioni_ogg|
                         Forma_ogg, Dimensioni_BB.
Forma_ogg ← ha_forma, *=> Forme.
Dimensioni_BB ← hBB *=> float, wBB*=> float, dBB*=> float.
Dimensioni_ogg ← dimensione *=> Dimensioni.
```

```
Rotazioni_ogg ← rotazione *=> Rotazioni.
/****************************** Definizione istanza oggetto semplice ********************************************/
Def_classe_ogg_semplice ← IdOS[Def_slot_proprieta_ist_ogg_semplice].
Def_slot_ist_proprieta_ogg ←  Forma_ist_ogg, Dimensioni_ ist_BB, Dimensioni_ ist_ogg, Rotazioni_ ist_ogg|
                              Forma_ ist_ogg, Dimensioni_ ist_BB, Rotazioni_ ist_ogg |
                              Forma_ ist_ogg, Dimensioni_ ist_BB, Dimensioni_ ist_ogg|
                              Forma_ ist_ogg, Dimensioni_ ist_BB.
Forma_ ist_ogg ←  ha_forma, -> Forme.
Dimensioni_ ist_BB ← hBB -> float, wBB -> float, dBB -> float.
Dimensioni_ ist_ogg ← dimensione -> Dimensioni.
Rotazioni_ ist_ogg ← rotazione -> Rotazioni.
/****************************** Definizione istanza oggetto composto ********************************************/
Def_istanza_ogg_composto ← IdIOC[ Def_slot_proprieta_ist_ogg_composto ].
Def_slot_proprieta_ist_ogg_composto ←
                  Def_componenti, Def_rel_spaziali, Dimensioni_ogg, Rotazioni_ogg, Delega_ogg |
                  Def_componenti, Def_rel_spaziali, Rotazioni_ogg, Delega_ogg |
                  Def_componenti, Def_rel_spaziali, Dimensioni_ogg, Delega_ogg |
                  Def_componenti, Dimensioni_ogg, Rotazioni_ogg, Delega_ogg |
                  Def_componenti, Rotazioni_ogg, Delega_ogg |
                  Def_componenti, Dimensioni_ogg, Delega_ogg |
                  Def_componenti, Delega_ogg.
                  Def_componenti.
Def_componenti ← formato_da ->> { Istanze_oggetti }.
Def_rel_spaziali ← relazioni_spaziali ->> { Istanze_rel_spaziali }.
Istanze_oggetti ← Istanza_oggetto | Istanza_oggetto, Istanze_oggetti.
Istanze_rel_spaziali ← Istanza_rel_spaziale | Istanza_rel_spaziale, Istanze_rel_spaziali.
Dimensioni_ogg ← dimensione -> Dimensioni.
Rotazioni_ogg ← rotazione -> Rotazioni.
Delega_ogg ← delega -> { Istanze_oggetti }.

Istanza_oggetto ← IdIOS | IdIOC.
Istanza_rel_spaziale ← IdIRel.
/****************************** Definizione classe relazione spaziale qualitativa ***********************************/
Def_classe_rel_spaziali ← IdRel [ Def_obj1_rel_spaz, Def_obj2_rel_spaz ].
Def_obj1_rel_spaz ← obj1 *=> Classe_oggetto.
Def_obj2_rel_spaz ← obj2 *=> Classe_oggetto.
/****************************** Definizione generale simboli non-terminali e terminali*******************************/
Forme ← rectangle | box | cylinder | sphere | ...
Dimensioni ← moltopiccolo | piccolo | normale | grande | moltogrande.
Rotazioni ← ruotato_asinistra | ruotato_adestra | ... | inclinato_inavanti | inclinato_asinistra | ... | sottosopra | notdef.
Relazioni_spaziali ← rel(destra, moltovicino) | rel(destra,vicino) | ... | rel(sotto,vicino) | ... | disgiunto | in_contatto | ... |
                  allineato_adestra | ...
Id ← stringa con primo carattere minuscolo.
IdOS ← stringa. (nome di un oggetto semplice).
IdOC ← stringa. (nome di un oggetto composto).
IdRel ← stringa. (nome di una relazione spaziale).
```

# Un Algoritmo per l'Apprendimento di Concetti Basato su Controfattuali

Floriana Esposito, Nicola Fanizzi, Luigi Iannone, Ignazio Palmisano, Domenico Redavid, e Giovanni Semeraro

Dipartimento di Informatica
Università degli Studi di Bari
Via Orabona 4, 70125 Bari, Italy
{*esposito, fanizzi, iannone, palmisano, redavid, semeraro*}@di.uniba.it

**Sommario** La realizzazione della visione del Semantic Web necessita di algoritmi e strumenti per la costruzione automatica di ontologie, poiché la costruzione manuale di ontologie non può reggere il passo, sia per questioni di costo che di tempo, con l'enorme quantità di conoscenza disponibile sul Web attualmente. In questo articolo presentiamo un algoritmo, basato su metodi di Machine Learning, che permette di automatizzare parte del processo di costruzione di ontologie da parte degli ingegneri della conoscenza.

## 1 Introduzione

Dalla creazione del nome Semantic Web (SW), da parte di Tim Berners-Lee, per identificare la sua visione del nuovo web [4], molti ricercatori, soprattutto dell'area Knowledge Representation & Reasoning (KRR), si sono impegnati per trovare il formalismo migliore per rappresentare la conoscenza nel SW. Il nome Semantic Web identifica infatti un insieme di specifiche che hanno l'obiettivo di rendere l'informazione (o meglio la conoscenza) direttamente elaborabile dalle macchine. Questo insieme di specifiche si basa su tecnologie già esistenti e di larghissima diffusione, come XML[1] e URI[2]. Su questi standard, poggia un framework per la rappresentazione di metadati, che hanno lo scopo di esplicitare la conoscenza contenuta in risorse web esistenti: RDF[3]. Per garantire l'interoperabilità, i metadati dovrebbero essere espressi facendo riferimento a *vocabolari* condivisi, in cui sono definite classi e relazioni tra classi. L'evoluzione delle specifiche per l'espressione di questi *vocabolari* è iniziata da RDFSchema [10] ed è giunta ad OWL (Web Ontology Language) [8], rendendo chiaro che il formalismo scelto nel SW per la rappresentazione di concetti e relazioni nei metadati è una

[1] eXstensible mark-up language `http://www.w3.org/XML`
[2] Uniform Resource Identifiers `http://www.w3.org/Addressing/`
[3] Resource Description Framework - `http://www.w3.org/RDF`

particolare logica descrittiva [2]. Il termine *ontologia* è preso in prestito dalla filosofia, ma con un significato differente: un'ontologia per il SW è "la specificazione di una concettualizzazione" [7]. Ciò rappresenta la base per l'evoluzione del Semantic Web: attualmente, sono disponibili specifiche per la scrittura di documenti portabili (XML), specifiche per la scrittura di metadati (RDF) per questi o per altri documenti (per esempio pagine HTML), e per la costruzione di ontologie per i metadati, che sono insiemi di relazioni tassonomiche e non tassonomiche tra le classi definite per i metadati. Poiché queste ontologie sono basate sulle Description Logics, esse hanno una semantica formale, e pertanto offrono la possibilità di implementare algoritmi di inferenza su rappresentazioni basate su ontologie. Illustreremo ora le problematiche emergenti, le quali necessitano di un approccio basato su tecniche di Machine Learning (Sezione 2). Di seguito, presenteremo la nostra soluzione sia dal punto di vista teorico (Sezione 3) che dal punto di vista pratico (Sezione 4). Infine, nella Sezione 5, trarremo alcune conclusioni e presenteremo gli sviluppi futuri di questo lavoro.

## 2 Motivazione

Il Semantic Web assicura l'interoperabilità semantica grazie ad ontologie che specificano il significato dei metadati in termini delle loro relazioni con le entità che compongono il dominio. Il problema è che, attualmente, il Web non ha ancora abbastanza ontologie, e quelle disponibili, oltre ad essere poche, si riferiscono a domini ristretti. Inoltre, costruire un'ontologia da zero è un compito molto pesante e difficile [9], tenendo anche conto del fatto che due esperti del dominio progetteranno ontologie diverse per lo stesso dominio. Anche se le discrepanze tra le ontologie possono apparire banali, esse possono dipendere da diversi fattori (livello di granularità, punti di vista differenti), e non possono essere armonizzate e integrate facilmente dalle macchine. Di conseguenza, è necessario un approccio alla costruzione di ontologie che sia:

- Semi-automatico (minimalmente).
- Prevedibile e controllabile, nel senso che, fissati i parametri di input e il dominio, i risultati non variano tra un'esecuzione e l'altra.

In questo contesto, un approccio basato su Machine Learning è appropriato, poiché fornisce sia la necessaria flessibilità che il supporto formale per l'acquisizione di ontologie. In particolare, il problema che affrontiamo è la costruzione della definizione di un concetto a partire da esempi positivi e negativi per il concetto in questione. Nel seguito (Sezione 5) vedremo come questo approccio può essere utilizzato per la costruzione di intere ontologie. Tuttavia, una situazione pratica in cui questo algoritmo si rivela utile è quella in cui un'ontologia preesistente deve evolversi per includere una nuova definizione. L'ingegnere della conoscenza può seguire due strade:

- Esprimere la definizione intensionale del nuovo concetto nel linguaggio scelto (per esempio OWL).

– Usare un algoritmo di apprendimento automatico supervisionato per indurre la definizione del nuovo concetto a partire da esempi positivi e negativi per tale concetto.

Anche se la prima soluzione può apparire più semplice, essa ha alcuni svantaggi. Per esempio, nessuno può garantire che la nuova definizione sia consistente con gli esempi (istanze/individui) già presenti nella base di conoscenza. Inoltre, nella scrittura della definizione l'ingegnere potrebbe non considerare caratteristiche importanti ma non evidenti se non si guarda agli esempi. Un esempio pratico consiste nell'estensione di un interessante lavoro di I. Astrova [1], in cui l'autrice propone una metodologia per la costruzione di ontologie da database relazionali. Supponiamo che, dopo il processo, ci si accorga che l'ontologia risultante manca di alcune classi (concetti) che possono essere costruiti a partire dall'ontologia di base costruita in precedenza. Invece di scrivere la definizione mancante da zero, l'ingegnere della conoscenza può selezionare adeguati esempi positivi e negativi (in questo caso un esempio corrisponde a una o più tuple nel database) ed eseguire l'algoritmo per l'apprendimento di concetti che noi proponiamo.

## 3 Algoritmo per l'Apprendimento di Concetti

In questa sezione illustriamo il nostro algoritmo dal punto di vista teorico, cioè l'apprendimento di una definizione espressa in uno dei linguaggi della summenzionata famiglia di formalismi per la rappresentazione della conoscenza chiamata Description Logics (DL). In particolare, mostriamo i risultati per una particolare DL chiamata $\mathcal{ALC}$. I formalismi DL differiscono tra loro per i costrutti consentiti. Per facilità di lettura, riportiamo sintassi e semantica per $\mathcal{ALC}$; per una descrizione più approfondita si rimanda a [11]. In ogni DL, i concetti primitivi $N_C = \{C, D, \ldots\}$ sono interpretati come un sottinsieme di un dominio di oggetti (risorse) mentre i ruoli primitivi $N_R = \{R, S, \ldots\}$ sono interpretati come relazioni binarie su questo dominio (proprietà). Descrizioni di concetti più complesse sono costruite usando concetti atomici e ruoli primitivi attraverso l'uso dei costruttori in Tabella 1. Il loro significato è definito attraverso un'*interpretazione* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, dove $\Delta^{\mathcal{I}}$ è il *dominio* dell'interpretazione e il funtore $\cdot^{\mathcal{I}}$ sta per *funzione di interpretazione*. Una base di conoscenza $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ contiene due componenti: una T-Box $\mathcal{T}$ e una A-box $\mathcal{A}$. $\mathcal{T}$ è un insieme di definizioni di concetti $C \equiv D$, che vuol dire $C^{\mathcal{I}} = D^{\mathcal{I}}$, dove $C$ è il nome del concetto e $D$ la descrizione in termini dei costruttori del linguaggio. In realtà, esistono T-Box generiche che permettono anche assiomi come $C \sqsubseteq D$ o $C \sqsupseteq D$ e cicli nelle definizioni, ma in questo articolo ci limitiamo a quelle che in letteratura sono chiamate T-Box *acicliche*, in cui vi sono solo definizioni di concetti. Queste definizioni sono nella forma $ConceptName \equiv D$ (si vede facilmente che esse sono equivalenti a T-Box acicliche con concetti complessi su entrambi i lati del segno di equivalenza). $\mathcal{A}$ contiene asserzioni estensionali su concetti e ruoli, per esempio $C(a)$ e $R(a, b)$, intendendo, rispettivamente, che $a^{\mathcal{I}} \in C^{\mathcal{I}}$ e $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$. La nozione semantica di *sussunzione* tra concetti (o ruoli) può essere data in termini delle interpretazioni:

| NOME | SINTASSI | SEMANTICA |
|---|---|---|
| concetto top | $\top$ | $\Delta^{\mathcal{I}}$ |
| concetto bottom | $\bot$ | $\emptyset$ |
| negazione di un concetto | $\neg C$ | $\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$ |
| congiunzione di concetti | $C_1 \sqcap C_2$ | $C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$ |
| disgiunzione di concetti | $C_1 \sqcup C_2$ | $C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$ |
| restrizione esistenziale | $\exists R.C$ | $\{x \in \Delta^{\mathcal{I}} \mid \exists y\ (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$ |
| restrizione universale | $\forall R.C$ | $\{x \in \Delta^{\mathcal{I}} \mid \forall y\ (x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$ |

**Tabella 1.** I costruttori per le descrizioni $\mathcal{ALC}$ e la loro interpretazione.

**Definizione 3.1 (sussunzione)** *Date due descrizioni di concetti $C$ e $D$ in $\mathcal{T}$, $C$* sussume *$D$, denotato con $C \sqsupseteq D$, se e solo se per ogni interpretazione $\mathcal{I}$ di $\mathcal{T}$ vale $C^{\mathcal{I}} \supseteq D^{\mathcal{I}}$. Pertanto, $C \equiv D$ equivale a $C \sqsupseteq D$ e $D \sqsupseteq C$.*

**Esempio 3.1** *Una possibile definizione di concetto nel linguaggio proposto è:*
$Father \equiv Human \sqcap Male \sqcap \exists hasChild.Human$
*che traduce la frase:* "a father is a male human that has some humans as his children"*. Le asserzioni della A-Box sono del tipo:*
$Father(Tom), Father(Bill), hasChild.Human(Bill, Joe)$ *e così via.*
*Ora, se definiamo due esempi:*
$FatherWithoutSons \equiv Human \sqcap Male \sqcap \exists hasChild.Human \sqcap \forall hasChild.(\neg Male)$
$Parent \equiv Human \sqcap (Male \sqcup Female) \sqcap \exists hasChild.Human$
*è facile vedere che $Father \sqsupseteq FatherWithoutSons$ e $Parent \sqsupseteq Father$, mentre $Father \not\sqsupseteq Parent$ e $FatherWithoutSons \not\sqsupseteq Father$*

Sottolineiamo che la sussunzione impone una relazione d'ordine parziale su qualunque insieme di concetti DL. Nel seguito, infatti, considereremo un insieme di definizioni di concetti ordinati con la sussunzione come spazio di ricerca $(\mathcal{S}, \succeq)$ in cui l'algoritmo deve trovare una definizione consistente per il concetto da definire. Il nostro problema di induzione, nella sua forma più semplice, può essere ora definito formalmente come un problema di apprendimento supervisionato:

**Definizione 3.2 (problema di apprendimento induttivo)** *In uno spazio di ricerca $(\mathcal{S}, \succeq)$, data una base di conoscenza $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ e un insieme di asserzioni positive e negative $\mathcal{A}_C = \mathcal{A}_C^+ \cup \mathcal{A}_C^-$ riguardanti l'appartenenza (o non appartenenza) di alcuni individui ad un generico concetto $C$ tali che: $\mathcal{A} \not\models_{\mathcal{T}}$[4] $\mathcal{A}_C$*
***Trovare** una nuova T-Box $\mathcal{T}' = (\mathcal{T} \setminus \{C \equiv D\}) \cup \{C \equiv D'\}$ tale che: $\mathcal{A} \models_{\mathcal{T}'} \mathcal{A}_C$*

Quindi, se un concetto $C$ non è definito nella terminologia $\mathcal{T}$, abbiamo un caso di *problema di induzione* che richiede di trovare le definizioni $C \equiv D$ di

[4] $\not\models_{\mathcal{T}}$ può essere definita informalmente come "$\not\models$ secondo la T-Box $\mathcal{T}$"; per una definizione più formale, consultare [2]

cui sono conseguenza logica le (nuove) asserzioni $\mathcal{A}_C$. D'altra parte, quando una definizione esistente in $\mathcal{T}$ si rivela scorretta, cioè ci sono asserzioni positive in $\mathcal{A}_C$ non coperte (definizione *incompleta*) oppure ci sono asserzioni negative coperte (definizione *inconsistente*), questo crea un *problema di raffinamento* dove una nuova definizione corretta $C \equiv D'$ deve essere trovata, sulla base degli esempi precedenti e di quelli nuovi. Entrambi i problemi possono essere visti come problemi di ricerca nello spazio di tutte le possibili definizioni di concetti in $\mathcal{ALC}$. Per muoversi in questo spazio, sono necessari operatori per spostarsi tra concetti in due direzioni (poiché lo spazio è parzialmente ordinato con la sussunzione). difatti, data una definizione di concetto in questo spazio, si può:

- Ottenere una definizione più generale (operatore di raffinamento verso l'alto).
- Ottenere una definizione più specifica (operatore di raffinamento verso il basso).

Dipendentemente dalla DL scelta, si possono immaginare molti operatori di raffinamento. Tipicamente, si tratta di operatori che manipolano la struttura sintattica del concetto, che preventivamente è stato posto in una particolare *forma normale*. Questo tipo di riscrittura sintattica, di solito, ha il vantaggio di semplificare gli operatori di raffinamento e di esistere per ogni descrizione di concetto nella DL scelta. Alcuni esempi di operatori di raffinamento si possono trovare in [3] e in [6] per due differenti DL. Dal punto di vista teorico, si possono studiare alcune proprietà degli operatori di raffinamento che garantiscono che l'implementazione sarà efficace nell'attraversamento dello spazio di ricerca. Queste proprietà sono:

- Finitezza locale, che vuol dire che i possibili concetti ottenibili attraverso il raffinamento sono in numero finito.
- Correttezza, che assicura che che ogni passo di raffinamento produce un concetto strettamente più generale (o più specifico) del concetto di partenza.
- Completezza, che garantisce che ogni concetto che sussume (o che è sussunto) il concetto di partenza è raggiungibile con una catena di raffinamenti (cioè $n$ applicazioni di raffinamento a partire dal concetto iniziale).
- Minimalità, cioè ogni possibile raffinamento di un concetto non può essere raggiunto da due catene di raffinamenti diversi (raffinamento non ridondante).

Un operator di raffinamento *ideale* ha la proprietà di essere localmente finito, completo e corretto. Per $\mathcal{ALC}$, nessuno ha ancora trovato un operatore con queste caratteristiche; l'implementazione di un operatore completo e ridondante, peraltro, non sarebbe sensata in termini di efficienza. Risulterebbe, infatti, essere una strategia di tipo *generate and test*, consistente nella generazione di tutti i possibili raffinamenti dei concetti in input e nel test di correttezza e consistenza rispetto agli esempi positivi e negativi nel problema di learning. Questo approccio, oltre a essere scarsamente performante, non sfrutta l'informazione disponibile negli esempi utilizzabile per la costruzione del raffinamento del concetto. In questo lavoro, illustreremo in particolare un operatore di raffinamento verso il basso

(specializzazione) basato su esempi, che specializza definizioni troppo generali (definizioni che coprono esempi negativi, che invece non dovrebbero essere coperti), mentre la generalizzazione viene di proposito lasciata all'implementatore, poiché ci sono scelte particolarmente efficienti (soprattutto in $\mathcal{ALC}$), come vedremo nel seguito. L'idea di base per la specializzazione è che, esaminando una definizione troppo generale, è necessario *rimuovere* la porzione della definizione che è responsabile per l'inclusione delle istanze negative. Un'idea per scegliere la parte di definizione da rimuovere consiste nel trovare il residuo [12] tra la definizione sbagliata e gli esempi negativi come spiegato nel seguito. Quindi, una volta che il concetto responsabile è stato individuato, esso può essere negato (poiché in $\mathcal{ALC}$ la negazione è permessa davanti a concetti complessi) e si può calcolare l'intersezione tra la definizione troppo generale e il residuo negato. Ovviamente, si tratta di una specializzazione, poiché nella teoria degli insiemi si ha che se $A, B$ sono due insiemi, allora $A \cap B \subseteq A$; prendendo $A = C^I$ e $B = (\neg D)^I$, dove $C$ è la definizione originale sbagliata e $D$ il residuo calcolato, allora abbiamo $C \sqsupset C \sqcap \neg D$. Il residuo negato è chiamato *controfattuale* [13] e può essere generalizzato per eliminare il maggior numero di negativi possibile dalla definizione inconsistente di partenza, come specificato nella prossima sezione.

### 3.1 L'Algoritmo

Il processo di apprendimento può iniziare quando ci sono esempi e controesempi nella A-Box di un concetto per cui una nuova definizione è richiesta. Si assume che la classificazione degli esempi sia data dall'ingegnere della conoscenza. Tuttavia, la metodologia si può applicare anche in un contesto analogo, dove esista già una definizione per il concetto da apprendere, ma essa risulta errata (troppo generale) perché da essa derivano concetti che sono stati classificati come negativi per il concetto da apprendere. Per applicare l'algoritmo alla seconda situazione, è sufficiente chiamare la routine *counterfactuals* dell'algoritmo descritto nel seguito. Le asserzioni non sono elaborate direttamente; un rappresentante nel linguaggio per la descrizione dei concetti (*single representation trick*) viene derivato preliminarmente, nella forma di concetto più specifico (*most specific concept*, *msc*). Il *msc* richiesto per l'algoritmo è la descrizione di un concetto DL massimamente specifica da cui deriva l'asserzione in questione. Poiché in alcune DL tale definizione non esiste, consideriamo la sua approssimazione fino a una certa profondità [5]. Di conseguenza, nell'algoritmo gli esempi positivi e negativi saranno descrizioni congiuntive molto specifiche. L'algoritmo si basa su due coroutine (vedere Figura 1) che eseguono, rispettivamente, generalizzazione e controfattuali, e si chiamano l'un l'altra per convergere a una definizione corretta. L'algoritmo di generalizzazione cerca di spiegare gli esempi positivi costruendo una definizione disgiuntiva. Ad ogni iterazione più esterna, una definizione molto specializzata (l'*msc* di un esempio) viene selezionata come punto di partenza per una nuova generalizzazione parziale; quindi, iterativamente, l'ipotesi è generalizzata per mezzo dell'operatore di generalizzazione $\delta$ (non definito qui, ma l'implementatore dovrebbe scegliere un operatore con euristiche per massimizzare il numero di positivi coperti con la generalizzazione) finché

**generalizzazione**(*Positivi*, *Negativi*, *Generalizzazione*)
**input** *Positivi*, *Negativi*: istanze positive e negative (definizioni);
**output** *Generalizzazione*: definizione generalizzata
**begin**
$RisPositivi \leftarrow Positivi$
$Generalizzazione \leftarrow \bot$
**while** $RisPositivi \neq \emptyset$ **do**
    $ParGen \leftarrow$ select_seed($RisPositivi$)
    $PosCoperti \leftarrow \{Pos \in RisPositivi \mid ParGen \sqsupseteq Pos\}$
    $NegCoperti \leftarrow \{Neg \in Negativi \mid ParGen \sqsupseteq Neg\}$
    **while** $PosCoperti \neq RisPositivi$ **and** $NegCoperti = \emptyset$ **do**
        $ParGen \leftarrow$ select($\delta(ParGen)$, $RisPositivi$)
        $PosCoperti \leftarrow \{Pos \in RisPositivi \mid ParGen \sqsupseteq Pos\}$
        $NegCoperti \leftarrow \{Neg \in Negativi \mid ParGen \sqsupseteq Neg\}$
    **if** $NegCoperti \neq \emptyset$ **then**
        $K \leftarrow$ **counterfactuals**($ParGen$, $PosCoperti$, $NegCoperti$)
        $ParGen \leftarrow ParGen \sqcap \neg K$
    $Generalizzazione \leftarrow Generalizzazione \sqcup ParGen$
    $RisPositivi \leftarrow RisPositivi \setminus PosCoperti$
**return** *Generalizzazione*
**end**

**counterfactuals**($ParGen$, $PosCoperti$, $NegCoperti$, $K$)
**input** *ParGen*: definizione inconsistente
    *PosCoperti*, *NegCoperti*: descrizioni positive e negative coperte
**output** $K$: counterfactual
**begin**
$NuoviPositivi \leftarrow \emptyset$
$NuoviNegativi \leftarrow \emptyset$
**for each** $N_i \in NegCoperti$ **do**
    $NewP_i \leftarrow$ residual($N_i$, $ParGen$)
    $NuoviPositivi \leftarrow NuoviPositivi \cup \{NewP_i\}$
**for each** $P_j \in PosCoperti$ **do**
    $NewN_j \leftarrow$ residual($P_j$, $ParGen$)
    $NuoviNegativi \leftarrow NuoviNegativi \cup \{NewN_j\}$
$K \leftarrow$ **Generalizzazione**($NuoviPositivi$, $NuoviNegativi$)
**return** $K$
**end**

**Figura 1.** Le coroutine usate nel metodo.

tutti i rappresentanti dei concetti positivi sono coperti o tutti i rappresentanti dei negativi sono spiegati. In questo caso, la definizione corrente *ParGen* deve essere specializzata attraverso i controfattuali. La coroutine, che riceve gli esempi coperti come input, trova una sottodescrizione $K$ che sia capace di eliminare gli esempi negativi presentati. Nella routine per la costruzione dei controfattuali, data un'ipotesi precomputata *ParGen*, che si suppone completa (che copre le asserzioni positive) ma inconsistente rispetto a nuove asserzioni negative, si mira a trovare i controfattuali da congiungere all'ipotesi iniziale per tornare a una definizione corretta che permetta di escludere le istanze negative. L'algoritmo è basato sulla costruzione di problemi di apprendimento residui basati su sottodescrizioni che causano la sussunzione di esempi negativi, rappresentati dai rispettivi *msc*. In questo caso, per ogni modello il residuo è derivato considerando la parte della definizione scorretta *ParGen* che non ha avuto parte nella sussunzione. Il residuo viene poi usato come un'istanza positiva di quella parte della descrizione che dovrebbe essere esclusa dalla definizione (attraverso la negazione). Analogamente, l'*msc* derivato dalle asserzioni positive assumerà il ruolo opposto di istanza negativa per il problema di apprendimento residuo in costruzione. Infine, il problema viene risolto chiamando la coroutine che generalizza queste descrizioni di esempi e quindi congiungendo la negazione dei risultati ritornati.

## 4 Un Esempio

In questa sezione, presentiamo un esempio per illustrare l'algoritmo attraverso una esecuzione completa.

**Esempio 4.1 (Apprendimento Supervisionato)** *Supponemdo che la A-Box di partenza sia*
$\mathcal{A} = \{M(d), r(d,l), r(j,s), \neg M(m), r(m,l), \neg M(a), w(a,j), r(a,s), F(d), F(j), \neg F(m), \neg F(a)\}$
*(assumendo $F \equiv$ Father, $M \equiv$ Man $r \equiv$ parentOf (role), $w \equiv$ wifeOf per rendere l'esempio comprensibile)*
*$F$ è il concetto da apprendere, pertanto gli esempi e i controesempi sono, rispettivamente: Positivi $= \{d, j\}$ and Negativi $= \{m, a\}$*
*Gli* msc *approssimati sono:*

$msc(j) = \exists r.\top$
$msc(d) = M \sqcap \exists r.\top$
$msc(m) = \neg M \sqcap \exists r.\top$
$msc(a) = \neg M \sqcap \exists r.\top \sqcap \exists w.\top$

*L'esecuzione dell'algoritmo è:*

***generalize**:*
*ResiduoPositivi* $\leftarrow$ {*msc(d), msc(j)*}
*Generalizzazione* $\leftarrow \bot$
*/* while esterno */*

$ParGen \leftarrow msc(d) = M \sqcap \exists r.\top$
$PosCoperti \leftarrow \{msc(d)\}$
$NegCoperti \leftarrow \{\}$
$ParGen \leftarrow \exists r.\top$ /* *M rimosso nel loop interno* */
$PosCoperti \leftarrow \{msc(d), msc(j)\}$
$NegCoperti \leftarrow \{msc(m), msc(a)\}$
*Call* ***counterfactuals***$(\exists r.\top, \{msc(d), msc(j)\}, \{msc(m), msc(a)\})$

***counterfactuals****:*

$NewP_1 \leftarrow \neg M \sqcap \exists r.\top \sqcup \neg\exists r.\top = \neg M$
$NuoviPositivi \leftarrow \{\neg M\}$
$NewP_2 \leftarrow \neg M \sqcap \exists r.\top \sqcap \exists w.\top \sqcup \neg(\exists r.\top) = \neg M \sqcap \exists w.\top$
$NuoviPositivi \leftarrow \{\neg M, \neg M \sqcap \exists w.\top\}$
$NewN_1 \leftarrow M \sqcap \exists r.\top \sqcup \neg\exists r.\top = M$
$NuoviNegativi \leftarrow \{M\}$
$NewN_2 \leftarrow \top$
$NuoviNegativi \leftarrow \{M, \top\}$
*Call* ***generalize***$(\{\neg M, \neg M \sqcap \exists w.\top\}, \{M, \top\})$
...

*Il risultato è:* $F = M \sqcap \exists r.\top$

## 5 Conclusioni e Sviluppi futuri

In questo lavoro abbiamo affrontato il problema della costruzione di ontologie in maniera semiautomatica. In particolare, abbiamo presentato un algoritmo capace di inferire descrizioni di concetti nella Description Logic $\mathcal{ALC}$ da istanze di concetti disponibili in una A-Box. L'algoritmo rappresenta la base per un potente strumento per l'ingegnere della conoscenza. Esso è stato implementato in un sistema chiamato *YINYANG* (Yet another INduction Yields to ANother Generalization), e, al momento attuale, è sottoposto a sperimentazione empirica per valutare estensivamente l'applicabilità pratica dell'approccio. Inoltre, nei problemi reali è possibile che le A-Box siano inconsistenti, cosa che risulterebbe nel fallimento dell'algoritmo. Pertanto, un'altra linea di ricerca futura è l'indagine sul modo migliore di affrontare questo problema.

### 5.1 Riconoscimenti

Lo sviluppo di questa ricerca è stato finanziato in parte dalla Comunità Europea sotto il Progetto Integrato IST VIKEF - Virtual Information and Knowledge Environment Framework (Contratto n. 507173 - altre informazioni a `http://www.vikef.net`) e dal Ministero dell'Istruzione, dell'Università e della Ricerca, progetto COFIN 2003 "Tecniche di intelligenza artificiale per il reperimento di informazione di qualità sul Web".

## Riferimenti bibliografici


[1] Irina Astrova. Reverse engineering of relational databases to ontologies. In *ESWS*, pages 327–341, 2004.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[3] L. Badea and S.-H. Nienhuys-Cheng. A refinement operator for description logics. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *LNAI*, pages 40–59. Springer, 2000.

[4] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.

[5] S. Brandt, R. Küsters, and A.-Y. Turhan. Approximation and difference in description logics. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the International Conference on Knowledge Representation*, pages 203–214. Morgan Kaufmann, 2002.

[6] Floriana Esposito, Nicola Fanizzi, Luigi Iannone, Ignazio Palmisano, and Giovanni Semeraro. Knowledge-intensive induction of terminologies from metadata. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proceedings of the 3rd International Semantic Web Conference, ISWC2004*, volume 3298 of *LNCS*, pages 411–426. Springer, 2004.

[7] Thomas R. Gruber. A translation approach to portable ontology specifications, 1993.

[8] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[9] Alexander Maedche and Steffen Staab. Ontology learning for the semantic web. *IEEE Intelligent Systems*, 16(2), March/April 2001.

[10] RDF-Schema. RDF Vocabulary Description Language 1.0: RDF Schema, 2003. Available at: http://www.w3c.org/TR/rdf-schema.

[11] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[12] G. Teege. A subtraction operation for description logics. In P. Torasso, J. Doyle, and E. Sandewall, editors, *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 540–550. Morgan Kaufmann, 1994.

[13] S.A. Vere. Multilevel counterfactuals for generalizations of relational concepts and productions. *Artificial Intelligence*, 14:139–164, 1980.

# Declarative Programming in Java using JSetL

Elio Panegai and Gianfranco Rossi

Dipartimento di Matematica
Università di Parma, Parma (Italy)
panegai@cs.unipr.it, gianfranco.rossi@unipr.it

**Abstract.** JSetL is a Java library that endows Java with a number of facilities to support declarative programming like those usually found in constraint logic programming (CLP) languages. In this paper we show, through a number of simple examples, how these facilities can be exploited to support a declarative programming style in Java.

## 1 Introduction

While it is undeniable that declarative programming (DP) languages (such as Prolog and CLP languages) provide valuable support for programming in many application areas, it is also a reality that most real-world software development is still done using traditional, possibly object-oriented (OO), programming languages, such as C++ and Java.

Development of new programming languages that integrate features of both declarative programming languages and conventional and object-oriented languages is feasible (e.g., Mozart/Oz). This solution however requires a lot of effort and does not provide any guarantee that the new language is accepted: it is known that real-world programmers tend to be quite conservative.

An alternative approach is making DP features available as part of a *library* for some existing language. In this way one can exploit (at least, at some extent) the DP paradigm while retaining all the advantages of constructs for programming and software structuring that are typical of conventional programming languages. One of the first proposal that integrates DP features—specifically, constraints and logical variables—in a conventional OO framework following the *library-based approach* is the ILOG Solver [12, 9]. Other similar, more recent, proposals are, JSolver [2], Choco [1], and the Java Constraints Library (JCL) [10], all focusing on constraint programming. Frappè [3] and Gisela [8], instead, face the problem of declarative programming in a wider sense, though focusing on specific applications. Another proposal that can be cited in this context is `tu`Prolog [5], a Java package that implements Prolog.

In [13] we have presented a Java library—called JSetL—that endows Java with a number of facilities to support declarative programming like those usually found in constraint logic programming (CLP) languages. A detailed presentation of JSetL can be found in [11].[1] Differently from other related proposals which are concerned with only some specific concepts and mechanisms used to support declarative programming (in particular, constraint handling), JSetL aims at providing a comprehensive collection of facilities to support declarative programming. Moreover, our proposal aims to be general-purpose, not devoted to any specific application.

In this paper we describe, through a number of simple examples, how the new facilities provided by JSetL can be exploited to support a declarative programming style in Java. The notion of logical variable, along with unification and constraint solving, are fundamental tools to allow execution order to be immaterial, hence to support a declarative reading of programs. Constraint solving, in particular, allows complex (set) expressions to be build and checked for satisfiability disregarding their order and the instantiation of logical variables possibly occurring in them. Moreover, the use of partially specified data structures,

[1] JSetL is fully implemented in Java and is available at URL `www.math.unipr.it/~gianfr/JSetL`.

along with the nondeterminism naturally supported by operations over sets, are fundamental features to allow the language to be used as a highly declarative modeling tool.

In is important to note that the focus of our work is on declarative programming; we are deliberately assuming that execution efficiency is not a primary requirement. Indeed, JSetL is mainly conceived as a tool for rapid software prototyping, where easiness of program development and program understanding prevail over efficiency.

The paper is organized as follows. In Section 2 we introduce the use of logical variables and unification. In Section 3 we introduce the fundamental data structures of JSetL, namely sets and lists, and we show how to deal with partially specified lists and sets. In Section 4 we show how to exploit nondeterminism embedded in set-theoretical operations (in particular we present a solution for the well-known *traveling salesman problem*). In Section 5 we introduce constraint solving in JSetL (showing, in particular, a solution for the well-known *n-queens problem*). In Section 6 we show how to implement intensional set definitions in JSetL. In Section 7 we show how the user can implement nondeterministic methods by exploiting the facilities for new constraint definition provided by JSetL. Finally, in Section 8 we draw some conclusion and we briefly discuss future work.

## 2 Logical variables

The use of logical variables is at the heart of the declarative computation model (see, e.g., [14]). Logical variables are single-assignment variables. Initially, a logical variable is uninitialized. Once it has been bound to a value, the variable remains bound throughout the computation.

In JSetL logical variables are instances of the `Lvar` class. A value can be assigned to a (uninitialized) logical variable as the result of processing some constraint involving the variable, in particular, equality constraints. Values can be of any type.

In order to illustrate the programming style supported by the use of logical variables we show a simple example of a Java program using JSetL.

**Example**. Translate the name of the days of the week from Italian to English and vice versa.

```
class Translator {
   static SolverClass Solver = new SolverClass();

   public static void weekTranslate(Lvar g, Lvar d) {
       if (Solver.boolSolve(g.eq("lunedi").and(d.eq("monday"))))       return;
       if (Solver.boolSolve(g.eq("martedi").and(d.eq("tuesday"))))     return;
       if (Solver.boolSolve(g.eq("mercoledi").and(d.eq("wednesday")))) return;
       if (Solver.boolSolve(g.eq("giovedi").and(d.eq("thursday"))))    return;
       if (Solver.boolSolve(g.eq("venerdi").and(d.eq("friday"))))      return;
       if (Solver.boolSolve(g.eq("sabato").and(d.eq("saturday"))))     return;
       if (Solver.boolSolve(g.eq("domenica").and(d.eq("sunday"))))     return;
       throw new Failure;
   }
...
}
```

The equality operator `eq` implements unification (actually, *set unification* [7]). To solve equalities, as well as other operations on logical variables, one needs to invoke the `boolSolve` method over an instance of `SolverClass`. `Solver` is the instance of the `SolverClass` class that will be used in examples throughout the paper.

The use of logical variables and unification, in place of assignment and standard equality, allows the same methods to be used for different purposes. For example, if `answer` is an uninitialized logical variable, `lunedi` and `monday` are initialized logical variables, declared as follows:

```
Lvar answer = new Lvar();
Lvar lunedi = new Lvar("", "lunedi");
Lvar monday = new Lvar("", "monday");
```

the following three are all legal invocations of the method `weekTranslate`:

```
weekTranslate(lunedi,answer);
weekTranslate(answer,monday);
weekTranslate(lunedi,monday);
```

That is, the same method can be used both to test a given translation and to translate a given week name into another one. No assumption is made about which are input and which are output parameters. In the first invocation, the first parameter is an input and the second is an output, and vice versa in the second invocation, while in the third invocation both are input parameters. Actually, the method `weekTranslate` defines a binary *relation*, not a function that maps one or more inputs to exactly one output. [2]

If no matches are found for the passed values, the `weekTranslate` method raises a `Failure` exception. The default action for this exception is the immediate termination of the current thread. The exception, however, can be caught by the program and dealt with as preferred. In general, we say that the invocation of a method terminates with *failure* if its execution causes the `Failure` exception to be raised; otherwise we say that it terminates with *success*.

The real power of logical variables should become apparent after considering their usage in defining and handling dynamic data structures, namely lists and sets.

## 3 Data structures and partially specified values

JSetL provides two distinct kinds of data structures: lists and sets. The main difference between them is that, while in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter.

Data structures that contain uninitialized logical variables in place of single elements or as a part of the data structure itself represent *partially specified values*. This is another feature that is often advocated as fundamental for the declarative computational model [14].

List and sets in JSetL are declared as instances of classes `Lst` and `Set`, respectively. Moreover, methods are provided to build lists and sets out of their elements. Elements can be of any type, including `Lvar`, `Lst`, and `Set` objects. Both lists and sets can be partially specified, as shown in the following two examples. Hereafter we will use the Prolog notation as an abstract syntax for lists.

**Example**. Check whether a list `L` contains at least two elements.

```
public static void at_least_two(Lst L)
throws Failure {
    Lvar x = new Lvar();
    Lvar y = new Lvar();
    Lst R  = new Lst();  // an uninitialized list
    Solver.solve(L.eq(R.ins1(y).ins1(x)));  // L = [x,y|R]
    return;
}
```

The problem is solved—in a truly declarative way—by requiring that the given list `L` satisfies the equality `L` = [`x`, `y`|`R`]. The `ins1` method is used to insert the value of the specified

[2] Note that the use of uninitialized logical variables as output parameters provides a simple solution to the problem caused by the fact that Java always passes primitive types and object references to methods by value.

argument as the first element of the list on which it is invoked. The `solve` method differs from the `boolSolve` method in that it does not return an explicit result, but it can cause a `Failure` exception to be generated if its argument expression turns out to be not satisfiable.

Observe that JSetL methods are declarative. For instance, the list element insertion and removal methods do not modify the list on which they are invoked: rather they build and return a new list obtained by adding/removing the elements to/from the input list.

As two further examples of declarative programs using partially specified lists consider the following two simple problems.

**Example**. Deterministic concatenation. Given two lists `L1` and `L2` computes the new list `L3` as the concatenation of `L1` and `L2`.

```
public static void concat(Lst L1, Lst L2, Lst L3)
throws Failure {
    if (Solver.boolSolve(L1.eq(Lst.empty))) {
        Solver.solve(L2.eq(L3));    // L3 = L2
        return;
    }
    else {
        Lvar x    = new Lvar();
        Lst R     = new Lst();
        Lst L3new = new Lst();
        Solver.solve(L1.eq(R.ins1(x)).and(L3.eq(L3new.ins1(x))));
                                    // L1 = [x|R] ∧ L3 = [x|L3new]
        concat(R,L2,L3new);
        return;
    }
}
```

Observe that in this version of `concat` `L1` and `L2` are required to be input parameters. `L3` instead can be either an output or an input parameter (i.e., `concat` can be used also to check whether a list is the concatenation of two other lists). A fully invertible definition of `concat` requires non-determinism and will be shown in Sect. 7.

**Example**. Check whether all elements of a set `s` are pairs, i.e., they have the form $[x1, x2]$, for any $x1$ and $x2$.

```
public static void all_pairs(Set s)
throws Failure {
    Lvar x1  = new Lvar();
    Lvar x2  = new Lvar();
    Lvar[] localVars = {x1, x2};
    Lvar x   = new Lvar();
    Solver.solve(forall(x,s,localVars,x.eq(Lst.empty.ins1(x1).insn(x2))));
    return;
}
```

The logical meaning of the `forall` expression in the above example is:

$$\forall \mathtt{x}((\mathtt{x} \in \mathtt{s}) \rightarrow \exists \mathtt{x1}, \mathtt{x2}(\mathtt{x} = [\mathtt{x1}, \mathtt{x2}])).$$

`localVars` specifies the variables which are to be intended as "local" to the `forall` expression. Concretely, a new copy of these variables is created for each element `x` of the set `s`.

## 4 Nondeterministic operations

Nondeterminism provides a very high-level control abstraction. The ability to state solutions in a nondeterministic way, along with the availability of logical variables, represents another valuable support to obtain declarative programs.

The notion of nondeterminism fits into that of set very naturally. Set unification and many other set operations are inherently and naturally nondeterministic. For example, the evaluation of $x \in \{1,2,3\}$, with $x$ an uninitialized variable, nondeterministically returns one among $x = 1$, $x = 2$, $x = 3$. The main source of nondeterminism in JSetL, therefore, are set operations. In this section we show three examples where set operations—in particular, membership and set unification—are exploited to nondeterministically bind values to logical variables.

**Example**. Compute the maximum of a set of integers `s`.

```
public static Lvar max(Set s)
throws Failure {
    Lvar x = new Lvar();
    Lvar y = new Lvar();
    Solver.solve(x.in(s));
    Solver.solve(forall(y, s, y.leq(x)));
    return x;
}
```

The declarative reading of this method is: an element `x` of `s` is the maximum of `s`, if for each element `y` of `s` it holds that `y` $\leq$ `x`.

Set unification is nondeterministic. For example, solving the set unification problem {x,y,z}={1,2,3} nondeterministically returns, one after the other, six different solutions:

```
x = 1, y = 2, z = 3,
x = 1, y = 3, z = 2,
x = 2, y = 3, z = 1, ...
```

and so on. This feature of set unification can be immediately exploited to provide a simple (declarative) solution to the common problem of computing permutations.

**Example**. Compute all permutations (one at a time) of the set {1,2,3,4}.[3]

```
public static void main (String[] args)
throws IOException, Failure {
    int[] I_elems = {1,2,3,4};
    Set I = new Set(I_elems);    // I = {1,2,3,4}
    Lvar[] S_elems = {new Lvar(), new Lvar(), new Lvar(), new Lvar()};
    Set S = new Set(S_elems);    // S = {_S1,_S2,_S3,_S4}
    Solver.solve(S.eq(I));       // {_S1,_S2,_S3,_S4} = {1,2,3,4}
    S.output();

    System.out.print("Do you want another solution(y/n)?");
    InputStreamReader reader = new InputStreamReader(System.in);
    BufferedReader input     = new BufferedReader(reader);

    while (input.readLine().equals("y")) {
        if (Solver.nextSolution()) {
            S.output();
            System.out.print("Do you want another solution(y/n)?");
        }
        else {
            System.out.println("no");
            break;
        }
```

[3] For the sake of simplicity we assume the problem deals with a specific set of integers. However, the solution could be easily generalized to any set: the cardinality `n` of the set can be computed using the `size` method of class `Set`, while the set of `n` uninitialized variables to be used in the set unification problem can be constructed calling the `mkSet(n)` method of class `Set`.

```
        }
    }
```

In this example we show also how to implement interaction with the user that allows the user to get one solution at a time, when requested (a'la Prolog). The `nextSolution` method allows to obtain a new solution, if any, for the last issued `solve` (thus, in the example, a new solution for the equality `S.eq(I)`). If no further solution exists, `nextSolution` returs a false result.

As a more complete example, showing the use of nondeterminism in connection with sets, we show a possible JSetL solution to the well-known problem of the traveling salesman problem.

**Example**. Given a directed labeled graph $G$, the *traveling salesman problem (TSP)* consists in determining whether there is a path in $G$ starting from a source node, passing exactly once for every other node, and returning in the initial node (here we are making the simplification of not considering costs attached to edges; the problem, and the proposed solution, however, can be straightforwardly extended to weighted graphs as well, where the goal is that of finding a path of global cost less than a constant $k$).

A directed labeled graph $G$ can be represented as a pair $\langle N, E\rangle$ where $N$ is the set of nodes and $E$ is the set of edges, and each edge has the form $\langle n_1, n_2\rangle$, with $n_1, n_2 \in N$. This representation of graphs has an immediate implementation using JSetL's data structures: $N$ can be implemented as a `Set` object containing simple elements (e.g., strings), and $E$ as a `Set` object whose elements are lists (i.e., `Lst` objects) of two nodes. If $E$ contains the list $[a, b]$ it means that the graph contains an arc from node $a$ to node $b$ (note that arcs can be conveniently implemented using sets—instead of lists—if the graph is undirected).

The following is the core part of the Java program which is able to solve the (simplified) TSP problem using JSetL.

```
import JSetL.*;
import java.io.*;

class Tsp {
   static Set visited = Set.empty;
   static Lst path    = Lst.empty;
   static int n;

   public static void main (String[] args)
   throws IOException, Failure {
       InputStreamReader reader = new InputStreamReader(System.in);
       BufferedReader in        = new BufferedReader(reader);
       Set nodes   = readNodes(in);
       Set edges   = readArcs(in, nodes);
       Lvar source = readSource(in);

       Solver.solve(source.in(nodes));
       visited = visited.ins(source);
       path    = path.ins1(source);
       tsp(edges, source, source);

       showPath();
       return;
   }

   public static void tsp(Set edges, Lvar source, Lvar startNode)
   throws Failure {
       Lvar nextNode = new Lvar();
       Lst newArc    = new Lst(Lst.empty.ins1(startNode).insn(nextNode));
```

```
        Solver.solve(newArc.in(edges).and(nextNode.nin(visited)));
        visited = visited.ins(nextNode);
        path    = path.insn(nextNode);
        if (path.size() < n) tsp(edges, source, nextNode);
        else tspLast(edges, source, nextNode);
        return;
    }

    public static void tspLast(Set edges, Lvar source, Lvar lastNode)
    throws Failure {
        Lst backArc = new Lst(Lst.empty.ins1(source).ins1(lastNode));
        Solver.solve(backArc.in(edges));
        return;
    }
}
```

The set `visited` is the set of all the already examined nodes, while the list `path` is used to store the computed path. Initially they are both empty. The `main` method first asks the user to supply the nodes and the edges of the graph, and then the initial node which is stored in the logical variable `source`. We assume the interaction with the user is implemented by methods `readNodes`, `readArcs`, `readSource` in some reasonable way.

Methods `tsp` and `tspLast` provide our solution to the TSP. Statements in the `main` method preceding the invocation of the `tsp` method state that the source node must belong to the set of nodes of the given graph $G$ and that it must be added to the set of visited nodes and to the list that represents the computed path (as the first element). Then the `tsp` method is called with `source` as the start node. The definition of the `tsp` method simply states that an edge leaving from node `startNode` and ending in a node `nextNode` not yet visited must belong to the set of edges of $G$. If these conditions are true, then `nextNode` is added to the set of visited nodes as well as to the computed path, and the `tsp` method is called recursively with `nextNode` as the new initial node, unless all nodes have been already visited. In the last case, the `tspLast` method is called instead; this method simply states that there must be an edge (`backArc`) from the last visited node to the initial one. If the problem admits at least one solution the invocation of the `tsp` method within the `main` method terminates successfully. Thus the `main` method can invoke `showPath()` which prints the solution stored in `path`.

The following is an example of a possible run of the program.

```
Insert the set of all nodes (separated by blanks): a b c d e

Insert the nodes reachable from a : b c
Insert the nodes reachable from b : a e
Insert the nodes reachable from c : b
Insert the nodes reachable from d : a
Insert the nodes reachable from e : d

Insert the initial node: a

Path: a -> c -> b -> e -> d -> a
```

## 5 Constraints

Declarative programming is often associated with *constraint programming* (CP). As a matter of fact, the ability to deal with constraints allows to compute with partially specified values. For example, the disequality

$$[x] \neq [1],$$

with $x$ an uninstantiated logical variable, can be solved if one can state (and remember) that it must hold that $x \neq 1$.[4] As another example, the expression

$$\texttt{less}(\{x\}, 1, r),$$

where $x$ is an uninstantiated logical variable, $r$ is an uninitialized set, and the intended meaning of $\texttt{less}(s, e, r)$ is $r = s \backslash \{e\}$, admits two possible solutions, corresponding to two possible constraints: $x = e \wedge r = \{\}$ and $x \neq e \wedge r = \{x\}$. Furthermore, an expression such as

$$\texttt{un}(x, y, z),$$

with $x$, $y$, $z$ uninstantiated logical variables, can not be further simplified and must be kept unaltered as a constraint over the possible values that can be subsequently assigned to the variables.

In JSetL basic set-theoretical operations, as well as equalities, inequalities and integer comparison expressions, are dealt with as *constraints*. Constraints are handled by a *constraint solver*, which is an instance of the class `SolverClass`. The current collection $\mathcal{C}$ of active constraints for a constraint solver $S$ is stored in the *constraint store* (CS) of $S$. Constraints are added to the CS of a solver using the `add` method. After constraints have been stored in the CS, one can invoke their resolution by calling the `solve` method on the related solver. The solver nondeterministically searches for a solution that satisfies all constraints introduced in the constraint store, by rewriting them to a simpler form. Constraint solving in JSetL is basically the same developed for CLP($\mathcal{SET}$) [6].

The ability to solve constraints disregarding the instantiation of variables possibly occurring in them allows the programmer to ignore (in some cases) the order of execution of statements in a program (hence, to have a more declarative reading of programs). For example, if `x` is an uninstantiated logical variable, executing

```
x.read();
Solver.add(x.neq(0));
Solver.solve();
```

is the same as executing

```
Solver.add(x.neq(0));
x.read();
Solver.solve();
```

Programming with constraints allows the programmer to express solutions to a possibly complex problem as a collection of equations and disequations, that will be solved in a subsequent moment, with no concern to the order in which they occur in the program. As such, CP languages constitute powerful modeling tool, particularly suitable to coincisely express solutions for artificial intelligence and constraint-satisfaction problems (e.g, combinatorial problems). The following is an example of one such well-known problem.

**Example**. The $n$-queens problem.

The $n$-queens problem consists in trying to place $n$ queens on a $n \times n$ chess board such that no two queens attack each other, i.e., no two queens are placed on the same row, the same column or the same diagonal.

To model the problem we observe that any solution will have a queen on each row and one on each column. We can then represent each row with a logical variable, $r_0, ..., r_{n-1}$. For every row there are $n$ columns on which we can place the queen. If we identify the columns with the integer $0, ..., n-1$, the resolution of the problem consists in assigning a value among 0 and $n-1$ to each of the $n$ logical variables $r_0, ..., r_{n-1}$ such that

$(i)$ $r_i \neq r_j$
$(ii)$ $r_j + j - r_i \neq i$
$(iii)$ $r_i + j - r_j \neq i$

[4] The alternative would be to prohibit evaluation of not completely specified expressions, raising an exception indicating an instantiation error.

for each $i \neq j$, $0 \leq i,j \leq n-1$. The first inequality states that two queens can not be placed on the same column, while the other two inequalities state that two queens can not be placed on the same diagonal.

The following is a Java program for the $n$-queens problem using JSetL (with $n = 4$).

```
import JSetL.*;
import java.io.*;

class Queens {
  public static final int n = 4;

  public static void main (String[] args)
  throws IOException, Failure {
    Set colmns = new Set(0, n-1); // a set of n integers representing columns
    Lst rows   = Lst.mkLst(n);    // a list of n Lvar's representing rows

    for (int i = 0; i < n; i++) // assign a distinct int. in 0..n-1 to each var. in rows
      Solver.add(((Lvar)rows.get(i)).in(colmns));
    Solver.allDifferent(rows);

    for (int i = 0; i < n-1; i++)  // add constraints on diagonals
      for (int j = i+1; j < n; j++) {
        Solver.add(((Lvar)rows.get(j)).sum(j).sub((Lvar)rows.get(i)).neq(i));
        Solver.add(((Lvar)rows.get(i)).sum(j).sub((Lvar)rows.get(j)).neq(i));
    }
    Solver.solve(); // check constraints

    for (int i = 0; i < n; i++) {  // print a solution
      System.out.print("\n Queen "+i+" = ");
      ((Lvar)rows.get(i)).print();
    }
    return;
  }
}
```

get is an utility methods for lists: `l.get(i)` returns the `i`-th element of the list `l`. `allDifferent` is used to add constraints to the CS: `Solver.allDifferent(a)`, where `a` is either a (bounded) list or set containing $n$ elements $e_1,\ldots,e_n$, adds the atomic constraints $e_i \neq e_j$ to the CS of `Solver`, for all $i$, $j$ in $1,\ldots,n$. `sum` and `sub` are methods of class `Lvar` which implement addition and subtraction over integers, respectively: they allow to write usual arithmetic expressions but involving logical variables.

Integers in set `colmns` indicate the columns on which the queens can be placed. Logical variables in list `rows` indicate the rows of the chess board. A value $c$ in the $i$-th `Lvar` of `rows` indicates a queen placed in row $i$ at column $c$. The `allDifferent` method assures that only one queen is placed on each different column. The other two constraints added to the CS through the `add` method implement inequalities (*ii*) and (*iii*) (note that `rows.get(i)` and `rows.get(j)` denote the i-th and the j-th `Lvar` of the list `rows`, respectively).[5]

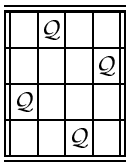
A possible solution printed by this program is:

```
Queen0 = 1
Queen1 = 3
```

[5] The current version of the JSetL constraint solver is not able to deal with integer arithmetic constraints in a complete way: specifically, it is not able to decide satisfiability of integer constraints containing uninitialized variables when the `solve` method is invoked. These limitations will be possibly overcome in future releases, following, for example, the approach described in [4], where the $\mathcal{SET}$ solver is integrated with an efficient solver over *finite domains* which allows the language to deal with arithmetic constraints with almost no restriction on the instantiation of expressions that occur in them.

```
Queen2 = 0
Queen3 = 2
```

that represents the situation depicted below



JSetL allows the user to define new constraints. User-defined constraints are dealt with as the built-in constraints: they can be added to the constraint store using the `add` method and solved using the `SolverClass` methods for constraint solving. Definition of user-defined constraints, however, requires some programming conventions to be respected. In particular, all definitions must be provided as part of a class named `NewConstraints`. This class extends the `SolverClass` and must be defined as part of the JSetL package (see [11] for a detailed description).

The user can exploit the ability to define new constraints, in particular, whenever he/she wants a part of his/her program to be "sensible" to backtracking. As an example, let us consider the following trivial program fragment, where we assume that `x` is an uninitialized logical variable and `s` is the set `{0,1}`:

```
Solver.solve(x.in(s));
x.output();
Solver.solve(x.neq(0));
```

Executing this program will print only one of the value nondeterministically assigned to `x` by the `x.in(s)` constraint, namely, the first one. The problem is caused by the fact that nondeterminism in JSetL is confined to constraint solving: backtracking allows the computation to go back to the nearest open choice point *within* the constraint solver, but it does not allow to "re-execute" user program code. In this example, the `output` statement is not re-executed when the second alternative for the `x.in(s)` constraint is taken into account. The problem can be solved using the `NewConstraints` class. The piece of code that requires to be re-executed is defined as the body of a new constraint in the `NewConstraints` class, with proper parameter passing:

```
public static void printVar(Lvar x) {
    x.output();
    return;
}
```

Then the sample program fragment is modified as follows:

```
Solver.solve(x.in(s));
Solver.solve(NewConstraints.printVar(x));
Solver.solve(x.neq(0));
```

With these changes, execution of this program causes all values assigned to `x` to be printed, that is:

```
x = 0
x = 1
```

Observe that the definition of the `printVar` method shown above is not the Java code which is really executed. The real code is obtained through a straightforward preprocessing

of the `NewConstraints` class code, that inserts all auxiliary definitions which allow to deal with methods defined by the user in this class as new constraints. Refer to [11] for more details.[6]

Another, more sophisticated, usage of the new constraint facility will be presented in Section 7.

## 6 Intensional sets

A very common way to represent a set is by its *intensional definition*, that is by providing a condition $\varphi$ that is necessary and sufficient for an element $x$ to belong to the set itself: $\{ x : \varphi \}$. Intensional set definitions represent a powerful tool for supporting a declarative programming style, as shown for instance in [6].

JSetL provides a limited form of intensional set definition through the use of the `setof` method. The invocation `Solver.setof(x,`$C$`)`, allows the program to collect into a set all values of the logical variable `x`, satisfying the given constraint $C$.

As an example, the following code defines, in a truly declarative way, a method to compute and print the set of all pairs `[x,y]` such that `x` and `y` belong to a given `s` and `x` $\neq$ `y`:

```
public static Lvar allPairs(Set s)
throws Failure {
    Lvar x    = new Lvar();
    Lvar y    = new Lvar();
    Lst pair  = new Lst(Lst.empty.ins1(x).insn(y));  // a pair [x,y]
    Set Pairs = new Set();
    Solver.setof(pair, x.in(s).and(y.in(s)).and(x.neq(y)));
                     // collect all solutions for x
                     // satisfying the constraint x ∈ s ∧ y ∈ s ∧ x ≠ y
    Pairs.output();
    return;
}
```

If `s` is the set {`1,2,3,4`} (and the CS is initially empty), the set printed by executing this method is

{`[1,2],[1,3],[1,4],[2,1],[2,3],[2,4],[3,1],[3,2],[3,4],[4,1],[4,2],[4,3]`}.

Observe that, the logical meaning of the above constraint is:

$$\forall\, \texttt{pair}((\texttt{pair} \in \texttt{s}) \leftrightarrow \exists \texttt{x}, \texttt{y}(\texttt{x} \in \texttt{s} \wedge \texttt{y} \in \texttt{s} \wedge \texttt{x} \neq \texttt{y})).$$

That is, the uninitialized logical variables possibly occurring in the constraint are considered as existentially quantified within the scope of the intensional definition of the set itself.

The same effect of `Solver.setof(x,`$C$`)` can be obtained by invoking first the `add` method on $C$, `Solver.add(`$C$`)`, then the `setof` method with only one parameter, `Solver.setof(x)`, which implicitly considers the constraint stored in the current CS. This form of the `setof` method can be particularly convenient when the constraint to be dealt with is quite complex.

**Example**. Prime numbers.

Compute the set of all prime numbers smaller or equal to a given integer `n`.

```
public static Set primes(Lvar n)
throws Failure {
    Lvar x = new Lvar();
```

[6] This preprocessing facility is not yet provided by the JSetL library; hence, the required program transformation of methods of the `NewConstraints` class must be performed at hand.

```
        Solver.add(x.in(Set.interv(2,n)));   // x in 2..n
        // check whether x is prime
        Lvar y = new Lvar();
        Lvar m = new Lvar();
        Solver.add(m.eq(x.sub(1)));          // m = x - 1
        Solver.forall(y, Set.interv(2,m),(x.mod(y)).neq(0));
        // compute the set of all primes numbers in 2..n
        return Solver.setof(x);
    }
```

In more abstract terms, the `setof` method collects the set $\{$`x : x ∈ 2..n ∧ ∀y (y ∈ 2..x-1) x mod y ≠ 0`$\}$.

## 7 Nondeterminism between statements

Besides exploiting the nondeterminism embedded in the predefined constraints, the user can implement her/his own nondeterministic procedures by exploiting the facilities for the new constraint definition provided by the JSetL library. In fact, JSetL allows the user to specify, within methods of the `NewConstraints` class, the nondeterministic choice (with backtracking) among two or more different blocks of statements.

As an example, the following code fragment shows a fully nondeterministic version of the list concatenation operation shown in Section 3. To make the code more readable, we assume a special construct, ***either*** $S_1$ ***orelse*** $S_2 \ldots$ ***orelse*** $S_n$, where $S_1,\ldots,S_n$ are Java statements, is available. The logical meaning of this construct is the disjunction $S_1 \vee,\ldots,\vee S_n$, while its computational interpretation is that of exploring, through backtracking, all possible alternatives $S_1,\ldots,S_n$, starting from $S_1$. The *either-orelse* construct can be used only within the `NewConstraints` class. This construct must be replaced, through a straightforward preprocessing phase, by usual Java code endowed with JSetL facilities for handling nondeterminism. Refer to [11] for more details.

```
class NewConstraints {
    public static void concat(Lst l1, Lst l2, Lst l3) {
        either {
            Solver.add(l1.eq(Lst.empty));
            Solver.add(l2.eq(l3));
        }
        orelse {
            Lvar x    = new Lvar();
            Lst l1new = new Lst();
            Lst l3new = new Lst();
            Solver.add(l1.eq(l1new.ins1(x)));
                                // l1 = [x | l1new]
            Solver.add(l3.eq(l3new.ins1(x)));
                                // l3 = [x | l3new]
            Solver.add(concat(l1new, l2, l3new));
        }
        return;
    }
}
```

This version of the `concat` method can be used not only to compute `l3` out of `l1` and `l2` or to test whether `l3` is the concatenation of l1 and l2, but also to compute (nondeterministically) all possible lists l1 and l2 from l3. Such flexibility is obtained by using unification, instead of standard assignment, and nondeterminism, instead of a deterministic **if** statement.

A simple example where we use `concat` to split a list into two sub-lists is the following definition of a method `prefix` that checks if a list `l1` is a prefix of a list `l2`:

```
public static void prefix(Lst l1, Lst l2)
throws Failure {
    Lst l = new Lst();
    Solver.add(NewConstraints.concat(l1, l, l2));
    Solver.solve();
    return;
}
```

Generally speaking, the ability to express nondeterminism allows the user to define methods which are completely "invertible", that is that do not impose any distinction between input and output parameters. In particular we can define methods that map a value to a set of different values (differently from functions where every element of the domain can be mapped to at most one element of the range). For instance, as shown above, we can use `concat` as a mapping from a list of $n$ elements (`l3`) to the set of $n+1$ pairs of lists (`l1` and `l2`) in which the given list can be splitted.

Moreover, the ability to choice nondeterministically one action among many can be conveniently exploited whenever one has to implement an algorithm which requires to explore a search space. Again, the presence of a high-level control abstraction such as that represented by the ***either-orelse*** construct can be of great help to devise solutions in a highly declarative style. As an example, the following is a completely nondeterministic method to find a path between two nodes in a directed labeled graph, represented as a set of pairs constituting the edges of the graph, as done in Section 4. To allow nondeterminism to be fully exploited, `path` is defined as a new constraint in the `NewConstraints` class.

```
public static void path(Set G, Lvar source, Lvar dest) {
    either {
        Lst finalEdge = Lst.empty.ins1(source).insn(dest));
        Solver.add(finalEdge.in(G));
    }
    orelse {
        Lvar x = new Lvar();
        Lst intermediateEdge = Lst.empty.ins1(source).insn(x));
        Solver.add(intermediateEdge.in(G));
        Solver.add(path(G, x, dest);
    }
    return;
}
```

As an example, if `g1` is the graph

`{['a','b'],['a','c'],['c','b'],['b','d'],['c','d']}`,

`s1` is an `Lvar` with value `'a'`, and `d` is an uninitialized variable, the invocation

`Solver.solve(NewConstraints.path(g1,s1,d))`

nondeterministically assigns to `d` the values `'b'`, `'c'`, `'d'`, which represent all nodes reachable from `'a'`.

## 8 Conclusions and future work

We have shown how facilities provided by the JSetL library can be used to write programs that exhibit a quite good declarative reading, while maintaining all the features of conventional Java programs.

As noted in the Introduction, computation efficiency has not been a primary requirement in the development of JSetL. As a matter of fact, various improvements to the current implementation can be devised, such as the use of more sophisticated techniques to handle choice points and backtracking, the use of a more efficient data representation and processing of sets and lists in the ground case, and so on. A more critical source of possible inefficiency, however, lies in the kind of constraint solving technique supported at present by our constraint solver, which is basically a "generate & test" technique (see, for instance, solution of the $n$-queens problem in Sect. 5). To overcome limitations of this approach—which are the same present in CLP($\mathcal{SET}$)—we plan to move along the lines suggested in [4]: integrating our (set) constraint solver with a more efficient constraint solver over *finite domains* (FD). Whenever the JSetL solver detects constraints that can be handled as FD constraints (e.g., membership constraints over finite sets of integer numbers) it delegates their resolution to the FD solver. This allows us to maintain the expressive power and flexibility of the JSetL constraint domain in the general case, while permitting efficient FD-constraint processing in many particular cases.

As another future work, simple preprocessing tools should be developed to make JSetL facilities simpler and more natural to use.

Finally, making the solver an instantiable class (not a static one, as in the first release) opens a number of interesting new possibilities:

- more solvers can coexist and cooperate
- solvers can be different threads
- constructors of the solver class can be parametric w.r.t. constraint solving policies.

## Acknowledgments

This work is partially supported by MIUR project *AIDA—Abstract Interpretation Design and Applications*.

## References

1. Choco. Available at `http://www.choco-constraints.net`
2. A.Chun. Constraint programming in Java with JSolver. In *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.
3. A.Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages, PADL 2001*, LNCS, Vol. 1990, Springer-Verlag, 29–44, 2001.
4. A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. In *PPDP'03 — Proc. of the Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press, 219–229, 2003.
5. E.Denti, A.Omicini, and A.Ricci. tuProlog: a light-weight Prolog for internet applications and infrastructures. In *Practical Aspects of Declarative Languages, PADL 2001*, LNCS, Vol. 1990, Springer-Verlag, 184–198, 2001.
6. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.
7. A. Dovier, E. Pontelli, and G. Rossi. Set unification. TR-CS-001/2001, Dept. of Computer Science, New Mexico State University, USA, January 2001 (available at `www.cs.nmsu.edu/TechReports`).
8. G.Falkman, O.Torgersson. Enhancing Usefulness of Declarative Programming Frameworks through Complete Integration. In *Proc. of the 12th Int. Workshop on Logic Programming Environments*, July 2002 (available at `http://xxx.lanl.gov/abs/cs.SE/0207054`).
9. ILOG Optimisation Suite - White Paper. Available at `www.ilog.com/products/optimisation/tech/optimisation/whitepaper.pdf`.
10. JCL - the Java Constraints Library. Available at `http://liawww.epfl.ch/JCL/`

11. E.Panegai, E.Poleo, and G.Rossi. JSetL User's Manual - Version 1.0. Research Report "Quaderno del Dipartimento di Matematica", n. 384, Università di Parma, November 2004.
12. J.-F.Puget and M.Leconte. Beyond the Glass Box: Constraints as Objects. In *Proc. of the 1995 Int'l Symposium on Logic Programming*, MIT press, pp. 513-527.
13. G.Rossi and E.Poleo. JSetL: Declarative Programming in Java with Sets. *In CF '04, 2004 ACM International Conference on Computing Frontiers*, ACM Press, 2004.
14. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

# Verifying Parameterized Protocols by Transforming Stratified Logic Programs

(Preliminary Version)

Alberto Pettorossi[1], Maurizio Proietti[2], Valerio Senni[1]

(1) DISP, University of Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy
pettorossi@info.uniroma2.it, senni@disp.uniroma2.it
(2) IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
proietti@iasi.rm.cnr.it

**Abstract.** We propose a method for the specification and the automated verification of temporal properties of parameterized protocols. Our method is based on logic programming and program transformation. We specify the properties of parameterized protocols by using an extension of stratified logic programs. This extension allows premises of clauses to contain first order formulas over arrays of parameterized length. A property of a given protocol is proved by applying suitable unfold/fold transformations to the specification of that protocol. We demonstrate our method by proving that the parameterized Peterson's protocol among $N$ processes, for any $N \geq 2$, ensures the mutual exclusion property.

## 1 Introduction

Protocols are rules that govern the interactions among concurrent processes. In order to guarantee that these interactions enjoy some desirable properties, many sophisticated protocols have been designed and proposed in the literature. These protocols are, in general, difficult to verify because of their complexity and ingenuity. This difficulty has motivated the development of methods for the formal specification and the automated verification of properties of protocols. One of the most successful methods is *model checking* [4]. It can be applied to every protocol that can be formalized as a *finite state system*, that is, a set of transitions over a finite set of states.

Usually, the number of the interacting concurrent processes is not known in advance. Thus, people have designed protocols which can work properly for any number of interacting processes. These protocols are said to be *parameterized* w.r.t. the number of processes.

In this paper we will address the problem of proving properties of parameterized protocols by using the program transformation methodology. The formulas which we manipulate in our transformations, describe the parameterized protocols and the processes themselves. These formulas include the so called *array formulas* which will be presented below.

We will demonstrate our proof method by considering the parameterized Peterson's protocol. It is used for ensuring mutually exclusive access to a given resource which is shared among $N$ processes. Each of these $N$ processes wants to access and possibly modify the shared resource. The number $N$ is the parameter of the parameterized protocol.

We assume that for any $i$, with $1 \leq i \leq N$, the $i$-th process consists of an infinite loop whose body is made out of two portions of code: (i) a portion called *critical section*, denoted *cs*, in which the process accesses and modifies the resource, and (ii) a portion called *non-critical section*, denoted *ncs*, in which the process does not access the resource. We also assume that every process is initially in its non-critical section.

We want the following property of the computation of the given system of $N$ processes to hold.

*Mutual Exclusion*: the statements of the critical section are executed by any one of the $N$ processes while no other process is executing a statement of the critical section.

The parameterized Peterson's protocol consists in adding two portions of code to every process: a first portion to be executed before entering into the critical section, and a second portion to be executed after exiting from the critical section.

We have two arrays $Q[1,\ldots,N]$ and $S[1,\ldots,N-1]$ of integers which are shared among the $N$ processes. The $N$ elements of the array $Q$ are initially set to 0 and may get values from 0 to $N-1$. The $N-1$ elements of the array $S$ are initially set to 1 and may get values from 1 to $N$.

We also have the array $J[1,\ldots,N]$ whose elements are initially set to 1 and may get values from 1 to $N$. The array $J$ is *not* shared, because for $i = 1,\ldots,N$, every process $i$ reads and writes $J[i]$ only, but it does nothing on the other elements of the array $J$.

Process $i$ is of the form given below (see also Figure 1 where it is represented in the form of a finite state diagram).

---

**while** *true* **do** // Process $i$
  non-critical section of process $i$;
  **for** $J[i] = 1,\ldots,N-1$ **do**
    **begin** $Q[i] := J[i]$; $S[J[i]] := i$;
      $\lambda$: **if** $(\forall k\,(k \neq i \rightarrow Q[k] < J[i]) \vee (S[J[i]] \neq i)$ **then** *skip* **else** *goto* $\lambda$;
    **end**;
  critical section of process $i$;
  $Q[i] := 0$;
    **od**

---

The $N$ processes execute their portions of code in a concurrent way. We assume that some operations are atomic and, in particular:
- the tests '$J[i] < N$' and '$\forall k\,(k \neq i \rightarrow Q[k] < J[i]) \vee (S[J[i]] \neq i)$' are atomic, and

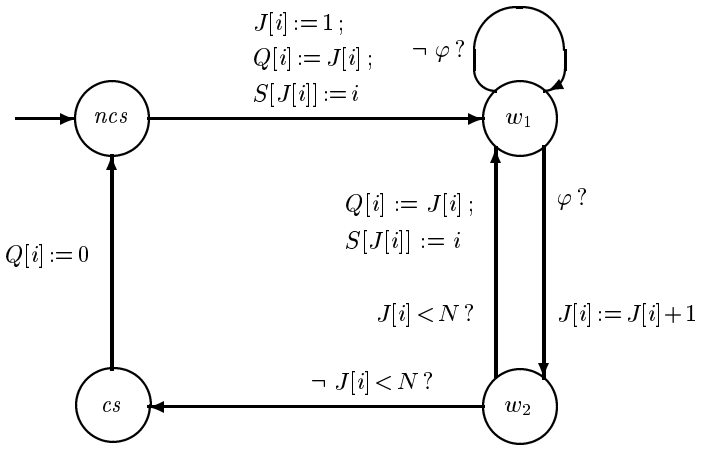



**Fig. 1.** Finite state diagram corresponding to process $i$ of a system of $N$ processes using Peterson's protocol. The formula $\varphi$ is $\forall k(k \neq i \rightarrow Q[k] < J[i]) \vee S[J[i]] \neq i$.

- the assignments '$Q[i] := 0;$' and '$J[i] := J[i] + 1;$', and the two sequences of assignments '$J[i] := 1;\ \ Q[i] := J[i];\ \ S[J[i]] := i;$' and '$Q[i] := J[i];\ \ S[J[i]] := i;$' are all atomic.

Actually, less stringent atomicity conditions are sufficient for ensuring that Peterson's protocol guarantees mutual exclusion.

We assume that the value of $N$ does *not* change over time, in the sense that while the computation progresses, neither a new process is constructed nor an existing process is destroyed.

In the original paper [15] G. L. Peterson does *not* prove the mutual exclusion property of its parameterized protocol in a formal way. He leaves that proof to the reader by saying that the correctness of the protocol in the case of $N$ processes, for $N \geq 2$, can be derived from the informal proof provided for the case of two processes because, for each value of $J[i] = 1, \ldots, N-1$, at least one process is discarded from the set of those which may enter into the critical section. In other words, when going from 'level' $J[i]$ to 'level' $J[i]+1$, at least one process is eliminated from the set of those competing for entering into the critical section.

In Peterson's protocol, the value of the variable $J[i]$ of process $i$ indicates the level that process $i$ has reached since it first requested to enter into its critical section. When process $i$ has completed its non-critical section and requests to enter into the critical section, it goes to the state $w_1$, while its level $J[i]$ has value 1. When process $i$ goes from state $w_1$ back to state $w_1$ through state $w_2$, it increases its level from $J[i]$ to $J[i]+1$.

For each level $J[i] = 1, \ldots, N-1$, process $i$ tests whether or not the following property $\varphi$ holds, where:

$$\varphi \equiv \quad \forall k(k \neq i \rightarrow Q[k] < J[i]) \vee S[J[i]] \neq i$$

If $\varphi$ holds then process $i$ enters the next level $J[i]+1$. When process $i$ has successfully tested the property $\varphi$ at the final level $N-1$, it can enter into its critical section.

In order to formally show that Peterson's protocol ensures mutual exclusion we cannot use directly the model checking technique. Indeed, since the parameter $N$ is unbounded, the parameterized Peterson's protocol can be viewed as a system with an infinite number of states. Now, in order to reduce a system with an infinite number of states to a system with a finite number of states, and thus, be able to apply model checking, one needs to apply an abstraction which, however, is not easily mechanizable.

In this paper we propose an alternative method for the specification and the automated verification of properties of parameterized protocols which does *not* need an abstraction. Our method is based on logic programming and program transformation.

We consider properties of parameterized protocols that can be expressed by using the CTL branching time temporal logic [4]. We formally specify these temporal properties by using an extension of stratified logic programs where premises of clauses may contain first order formulas over arrays of parameterized length. Our specification method is demonstrated in Section 2 by considering the mutual exclusion property of the parameterized Peterson's protocol. Then, in Section 3, we show that this mutual exclusion property can be proved by transforming the given specification using the unfold/fold transformation rules. Finally, in Section 4 we briefly illustrate the related work in the area of the verification of parameterized protocols.

## 2 Specifying Parameterized Protocols

In this section we present our method for the specification of temporal properties of parameterized protocols. Our method is an extension of other methods based on logic programming and constraint logic programming [6,7,10,12,19]. The main new feature of our method is that in the specifications of protocols we use a first order theory of arrays.

Similarly to the model checking approach, we represent a protocol as a set of transitions between states which are assumed to belong to a possibly infinite set. The transition relation is specified as a binary predicate $t$, defined by a set of statements of the form:

$$t(a, a') \leftarrow \tau$$

where $a$ and $a'$ are terms representing states and $\tau$ is a first order formula. We assume that the transition $t(a, a') \leftarrow \tau$ is not recursive, that is, $t$ does not occur in $\tau$.

For the representation of states and transitions, it is often useful to consider arrays of length $N$, where $N$ is the number of processes that participate in the protocol. Thus, in order to specify the transition relation, we assume that the

formula $\tau$ is an *array formula*, that is, a formula of the first order theory of arrays.

Array formulas are arbitrarily quantified formulas constructed as usual in first order logic starting from the following predicates and function symbols: (i) the equality $=$ between constants (such as the labels *ncs*, $w_1$, $w_2$, and *cs* of the states of Figure 1), between natural numbers, and between arrays, (ii) the inequalities $<$ and $\leq$, and the disequality $\neq$ between natural numbers, (iii) the constant 0, (iv) the successor $+1$ and predecessor $-1$, (v) the unary function *length* denoting the length of an array, and (vi) the binary function $-[-]$ such that $A[i]$ denotes the $i$-th element of the array $A$.

Statements with array formulas in their premises extend the usual syntax of clauses in logic programs. However, we can translate a non-recursive statement of the form $H \leftarrow \tau$ into a *stratified* set of clauses. Indeed, the predicates used in array formulas can be easily defined by logic programs. For example, the predicate $A[i] = e$ can be defined by a ternary predicate *member* as follows:

$member([E|A], 1, E) \leftarrow$
$member([E_1|A], I, E) \leftarrow I = J+1 \wedge member(A, J, E)$

Moreover, any non-recursive statement of the form $H \leftarrow \tau$ can be translated into a stratified set of clauses by applying the Lloyd-Topor transformation [13]. For instance, given the statement:

$p \leftarrow \forall n\, \exists A\, \forall i\, (i \geq 1 \ \wedge \ n \geq i \ \rightarrow \ \exists e\, A[i] = e)$

by applying the Lloyd-Topor transformation we get:

$p \leftarrow \neg newp1$
$newp1 \leftarrow \neg newp2(N)$
$newp2(N) \leftarrow \neg newp3(N, A)$
$newp3(N, A) \leftarrow I \geq 1 \wedge N \geq I \wedge \neg newp4(A, I)$
$newp4(A, I) \leftarrow member(A, I, E)$

When specifying protocols we will use statements with array formulas, instead of the corresponding clausal translations, because statements are more concise and intuitive. By abuse of language, we will use the term 'clause' also to indicate any statement in which array formulas may occur.

Let us now show how the parameterized Peterson's protocol is specified by using clauses with array formulas.

A *state* is represented by a term of the form $s(P, J, Q, S)$, where:

1. $P$ is the array $P[1, \ldots, N]$ such that, for $i = 1, \ldots, N$, $P[i]$ belongs to the set $\{cs, ncs, w_1, w_2\}$ and represents the state of process $i$. The constants *cs* and *ncs* denote the critical and the non-critical section, respectively, while the constants $w_1$ and $w_2$ denote two distinct waiting states.
2. $Q$ and $S$ are shared arrays such that, for $i = 1, \ldots, N$, $Q[i]$ belongs to $\{0, \ldots, N-1\}$ and, for $i = 1, \ldots, N-1$, $S[i]$ belongs to $\{1, \ldots, N\}$. These two arrays can be read and modified by the individual processes.
3. $J[1, \ldots, N]$ is an array such that, for $i = 1, \ldots, N$, $J[i]$ belongs to $\{1, \ldots, N\}$ and represents a local variable that can be read and modified by process $i$ only.

The transition relation is defined by six clauses $T_1, \ldots, T_6$, where for $r = 1, \ldots, 6$, $T_r$ is of the form:

$T_r: \quad t(s(P,J,Q,S), s(P',J',Q',S')) \leftarrow \tau_r(s(P,J,Q,S), s(P',J',Q',S'))$

and $\tau_r(s(P,J,Q,S), s(P',J',Q',S'))$ is an array formula defined as follows:

transition $ncs \rightarrow w_1$ :

$$\begin{aligned}
&\tau_1(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\
&\quad \exists i(P[i]=ncs \wedge \\
&\qquad P'[i]=w_1 \ \wedge \ J'[i]=1 \ \wedge \ Q'[i]=J'[i] \ \wedge \ S'[J'[i]]=i \wedge \\
&\qquad \forall k(k \neq i \ \rightarrow \ (P'[k]=P[k] \ \wedge \ Q'[k]=Q[k] \ \wedge \ J'[k]=J[k])) \wedge \\
&\qquad \forall k(k \neq J[i] \ \rightarrow \ S'[k]=S[k]))
\end{aligned}$$

transition $w_1 \rightarrow w_1$ :

$$\begin{aligned}
&\tau_2(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\
&\quad \exists i(P[i]=w_1 \ \wedge \ \exists k(k \neq i \ \wedge \ Q[k] \geq J[i]) \ \wedge \ S[J[i]]=i) \wedge \\
&\quad P'=P \ \wedge \ J'=J \ \wedge \ Q'=Q \ \wedge \ S'=S
\end{aligned}$$

transition $w_1 \rightarrow w_2$ :

$$\begin{aligned}
&\tau_3(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\
&\quad \exists i(P[i]=w_1 \ \wedge \ (\forall k(k \neq i \ \rightarrow \ Q[k] < J[i]) \ \vee \ S[J[i]] \neq i) \wedge \\
&\qquad P'[i]=w_2 \ \wedge \ J'[i]=J[i]+1 \wedge \\
&\qquad \forall k(k \neq i \ \rightarrow \ P'[k]=P[k] \ \wedge \ J'[k]=J[k])) \wedge \\
&\quad Q'=Q \ \wedge \ S'=S
\end{aligned}$$

transition $w_2 \rightarrow w_1$ :

$$\begin{aligned}
&\tau_4(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\
&\quad \exists i(P[i]=w_2 \ \wedge \ J[i] < length(P) \wedge \\
&\qquad P'[i]=w_1 \ \wedge \ Q'[i]=J[i] \ \wedge \ S'[J[i]]=i \wedge \\
&\qquad \forall k(k \neq i \ \rightarrow \ (P'[k]=P[k] \ \wedge \ Q'[k]=Q[k])) \wedge \\
&\qquad \forall k(k \neq J'[i] \ \rightarrow \ S'[k]=S[k])) \wedge \\
&\quad J'=J
\end{aligned}$$

transition $w_2 \rightarrow cs$ :

$$\begin{aligned}
&\tau_5(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\
&\quad \exists i(P[i]=w_2 \ \wedge \ J[i]=length(P) \ \wedge \ P'[i]=cs \wedge \\
&\qquad \forall k(k \neq i \ \rightarrow \ P'[k]=P[k])) \wedge \\
&\quad J'=J \ \wedge \ Q'=Q \ \wedge \ S'=S
\end{aligned}$$

transition $cs \rightarrow ncs$ :

$$\begin{aligned}
&\tau_6(s(P,J,Q,S), s(P',J',Q',S')) \equiv \\
&\quad \exists i(P[i]=cs \ \wedge \ P'[i]=ncs \ \wedge \ Q'[i]=0 \wedge \\
&\qquad \forall k(k \neq i \ \rightarrow \ (P'[k]=P[k] \ \wedge \ Q'[k]=Q[k]))) \wedge \\
&\quad S'=S \ \wedge \ J'=J
\end{aligned}$$

We express the properties of interest of the parameterized protocols by using the CTL branching time temporal logic [4]. In particular, the mutual exclusion property of Peterson's protocol is expressed by the temporal formula:

$$initial \rightarrow \neg EF \ unsafe$$

where *initial* and *unsafe* are atomic properties of states which we will specify below. This temporal formula holds at a state $a$ iff if $a$ is an *initial* state then there exists no *unsafe* state in the future of $a$.

The truth of a CTL temporal formula is defined by the following locally stratified logic program, where the predicate $holds(a, f)$ means that the temporal formula $f$ holds at state $a$:

$holds(X, F) \leftarrow atomic(X, F)$
$holds(X, \neg F) \leftarrow \neg holds(X, F)$
$holds(X, F \rightarrow G) \leftarrow \neg holds(X, F)$
$holds(X, F \rightarrow G) \leftarrow holds(X, F) \wedge holds(X, G)$
$holds(X, ef(F)) \leftarrow holds(X, F)$
$holds(X, ef(F)) \leftarrow t(X, X') \wedge holds(X', ef(F))$

The unary constructor $ef$ encodes the temporal logic operator $EF$. Other temporal operators can be defined by using locally stratified logic programs [7,12]. For reasons of simplicity, here we have restricted ourselves to the operator $EF$ which is the only operator needed for specifying the mutual exclusion property of Peterson's protocol.

The atomic properties of the states are specified by a set of non-recursive clauses of the form:

$atomic(a, p) \leftarrow \psi$

where $\psi$ is an array formula stating that the atomic formula $p$ holds at state $a$. In particular, *initial* and *unsafe* are atomic properties specified as follows.

$atomic(s(P, J, Q, S), initial) \leftarrow \forall i\ (P[i]{=}ncs \wedge J[i]{=}1 \wedge Q[i]{=}0 \wedge S[i]{=}1)$

The premise of the above clause will also be written as $init\_state(s(P, J, Q, S))$, and it means that in an initial state every process is in its non-critical section, $J$ is an array whose elements are all 1's, $Q$ is an array whose elements are all 0's, and $S$ is an array whose elements are all 0's.

$atomic(s(P, J, Q, S), unsafe) \leftarrow \exists i, j\ (P[i] = cs \wedge P[j] = cs \wedge i \neq j)$

The premise of the above clause will also be written as $unsafe\_state(s(P, J, Q, S))$, and it means that in an unsafe state two distinct processes are in critical section.

Let *Peterson* be the program consisting of the clauses that define the binary predicates *holds* and *atomic*, and the binary transition relation $t$. *Peterson* is a locally stratified program which, therefore, has a perfect model [1], denoted by $M(Peterson)$. We will prove that the temporal formula $initial \rightarrow \neg EF\ unsafe$ holds for every state, by showing that:

$M(Peterson) \models \forall X\ holds(X, initial \rightarrow \neg ef(unsafe))$

Notice that $X$ ranges over terms of the form $s(P, J, Q, S)$ where the length of the array $P$ of the states of the processes is the parameter $N$. Thus, by showing the above formula we show that Peterson's protocol ensures mutual exclusion for any number $N$ of processes.

## 3 Transformational Verification of the Parameterized Peterson's Protocol

In this section we prove that the mutual exclusion property holds for the parameterized Peterson's protocol by using program transformation. As a first step we introduce the statement:

$mutex \leftarrow \forall X\ holds(X, initial \rightarrow \neg ef(unsafe))$

which, by applying the Lloyd-Topor transformation [13], is transformed into the following set of clauses:

1. $mutex \leftarrow \neg new1$
2. $new1 \leftarrow new2(X)$
3. $new2(X) \leftarrow holds(X, initial) \wedge holds(X, ef(unsafe))$

We have that:

$M(Peterson) \models \forall X\ holds(X, initial \rightarrow \neg ef(unsafe))$ iff

$M(Peterson \cup \{1,2,3\}) \models mutex$

We will show that $M(Peterson \cup \{1,2,3\}) \models mutex$ by applying unfold/fold transformation rules that preserve the perfect model [9,20] and deriving from the program $Peterson \cup \{1,2,3\}$ a new program $T$ which contains the clause $mutex \leftarrow$.

The unfold/fold transformation rules are guided by a transformation strategy similar to the ones presented in [7,16]. We do not indicate this strategy here and we only show it in action in our verification of Peterson's protocol.

First we transform clause 3 with the objective of deriving the specialized definitions corresponding to the instances of the atom $holds(X, F)$ for various values of the state $X$ and the formula $F$. Indeed, by doing so we will discover that $new2(X)$, which corresponds to the instance of $holds(X, F)$ where $X$ is an initial state and $F$ is $ef(unsafe)$, is false. Then, by unfolding, we can easily derive the clause $mutex \leftarrow$.

Starting from clause 3, we apply the following transformation steps: (i) we unfold clause 3, thereby deriving a new set, say $G$, of clauses, (ii) we manipulate the array formulas occurring in the clauses of that set $G$, by replacing these formulas by equivalent ones, and we remove each clause whose body contains an unsatisfiable formula, (iii) we introduce new predicate definitions and we fold every instance of $holds(X, F)$. Starting from every new predicate definition which has been introduced, we repeat the above transformation steps (i), (ii), and (iii) until we are able to fold every instance of $holds(X, F)$ by using a predicate definition introduced at a previous step.

As already mentioned in [7,16], this unfolding/definition/folding procedure is not ensured to terminate in general, because properties of programs are in general undecidable. However, for many classes of programs and properties, this procedure terminates and, for those classes, it behaves as a decision procedure.

Let us now show some of the transformation steps for verifying that Peterson's protocol enjoys the mutual exclusion property. By unfolding clause 3 we get:

4. $new2(s(P,J,Q,S)) \leftarrow init\_state(s(P,J,Q,S)) \ \wedge \ unsafe\_state(s(P,J,Q,S))$
5. $new2(s(P,J,Q,S)) \leftarrow init\_state(s(P,J,Q,S)) \wedge$
   $t(s(P,J,Q,S), s(P',J',Q',S')) \wedge$
   $holds(s(P',J',Q',S'), ef(unsafe))$

By unfolding clause 5 w.r.t. the atom $t$ we get six new clauses, one for each clause $T_1, \ldots, T_6$ defining the transition relation (see previous Section 2). The clauses

derived from $T_2, \ldots, T_6$ are removed because their bodies contain unsatisfiable array formulas. Also clause 4 is removed because the formula

$$\mathit{init\_state}(s(P,J,Q,S)) \ \wedge \ \mathit{unsafe\_state}(s(P,J,Q,S))$$

occurring in its body is unsatisfiable. Thus, the only clause derived from clause 3 after the unfolding and removal steps is:

$$\begin{array}{lll} 6.\ \mathit{new2}(s(P,J,Q,S)) \leftarrow & \mathit{init\_state}(s(P,J,Q,S)) \wedge \\ & \tau_1(s(P,J,Q,S), s(P',J',Q',S')) \wedge \\ & \mathit{holds}(s(P',J',Q',S'), \mathit{ef}(\mathit{unsafe})) \end{array}$$

where $\tau_1(s(P,J,Q,S), s(P',J',Q',S'))$ is the array formula defined in the previous section. Let us consider the formula $c_1(s(P',J',Q',S'))$ defined as follows:

$$\begin{array}{l} c_1(s(P',J',Q',S')) \equiv \\ \quad \exists P,J,Q,S \ (\mathit{init\_state}(s(P,J,Q,S)) \wedge \tau_1(s(P,J,Q,S), s(P',J',Q',S'))) \end{array}$$

This formula $c_1(s(P',J',Q',S'))$ characterizes the set of successor states of the state $s(P,J,Q,S)$. We have that the following equivalence holds:

$$\begin{array}{l} c_1(s(P',J',Q',S')) \equiv \exists i(P'[i] = w_1 \ \wedge \ Q'[i] = J'[i] \ \wedge \ S'[J'[i]] = i \ \wedge J'[i] = 1 \wedge \\ \qquad\qquad \forall k(k \neq i \rightarrow (P'[k] = ncs \ \wedge \ Q'[k] = 0))) \end{array}$$

In order to fold the atom $\mathit{holds}(s(P',J,Q',S'), \mathit{ef}(\mathit{unsafe}))$ in the body of clause 6 by applying the folding rule of [9], we need to introduce a new predicate definition of the form:

$$\begin{array}{lll} 7.\ \mathit{new3}(s(P,J,Q,S)) \leftarrow & \mathit{genc}_1(s(P,J,Q,S)) \wedge \\ & \mathit{holds}(s(P,J,Q,S), \mathit{ef}(\mathit{unsafe})) \end{array}$$

where $\mathit{genc}_1(s(P,J,Q,S))$ is a *generalization* of $c_1(s(P,J,Q,S))$, in the sense that $\forall P,J,Q,S \ (c_1(s(P,J,Q,S)) \rightarrow \mathit{genc}_1(s(P,J,Q,S)))$ holds. As usual in the program transformation approach, this generalization step requires ingenuity. Here we will not address the problem of how to mechanize the generalization steps. This crucial issue is left for future research. In our example we continue the derivation by introducing the following array formula $\mathit{genc}_1(s(P,J,Q,S))$ which expresses the fact that, for any set of processes which are in state $w_1$, there exists a process $i$ which has been the last one to enter $w_1$ and to set $S[J[i]] = i$:

$$\begin{array}{l} \mathit{genc}_1(s(P,J,Q,S)) \equiv \exists i(P[i] = w_1 \ \wedge \ Q[i] = J[i] \ \wedge \ J[i] = 1 \ \wedge \ S[J[i]] = i) \wedge \\ \qquad\qquad \forall k((P[k] = ncs \ \wedge \ Q[k] = 0) \vee \\ \qquad\qquad\quad (P[k] = w_1 \ \wedge \ Q[k] = J[k] \wedge J[k] = 1)) \end{array}$$

By folding clause 6 using the newly introduced clause 7 we get:

$$\begin{array}{lll} 6.\mathrm{f}\ \ \mathit{new2}(s(P,Q,S,J)) \leftarrow & \mathit{init\_state}(s(P,J,Q,S)) \wedge \\ & \tau_1(s(P,J,Q,S), s(P',J',Q',S')) \wedge \\ & \mathit{new3}(s(P',J',Q',S')) \end{array}$$

Now, starting from clause 7, we repeat the transformation steps (i), (ii), and (iii) described above, until we are able to fold every instance of $holds(X, F)$ by using a predicate definition introduced at a previous step. By doing so we derive the following program $S$ where:
- $genc_1$ is defined as indicated above,
- $genc_2, \ldots, genc_8$ are defined as indicated in the Appendix,
- $\tau_1, \ldots, \tau_6$ are the array formulas that define the transition relation as indicated in Section 2, and
- the arguments $a$ and $a'$ stand for the states $s(P, J, Q, S)$ and $s(P', J', Q', S')$, respectively.

---

Program $S$

1. $mutex \leftarrow \neg new1$
2. $new1 \leftarrow new2(X)$

6.f $new2(a) \leftarrow initial(a) \wedge \tau_1(a, a') \wedge new3(a')$

8. $new3(a) \leftarrow genc_1(a) \wedge \tau_1(a, a') \wedge new3(a')$
9. $new3(a) \leftarrow genc_1(a) \wedge \tau_2(a, a') \wedge new3(a')$
10. $new3(a) \leftarrow genc_1(a) \wedge \tau_3(a, a') \wedge new4(a')$
11. $new3(a) \leftarrow genc_1(a) \wedge \tau_3(a, a') \wedge new5(a')$

12. $new4(a) \leftarrow genc_2(a) \wedge \tau_1(a, a') \wedge new8(a')$
13. $new4(a) \leftarrow genc_2(a) \wedge \tau_2(a, a') \wedge new4(a')$
14. $new4(a) \leftarrow genc_2(a) \wedge \tau_3(a, a') \wedge new4(a')$
15. $new4(a) \leftarrow genc_2(a) \wedge \tau_4(a, a') \wedge new4(a')$
16. $new4(a) \leftarrow genc_2(a) \wedge \tau_5(a, a') \wedge new9(a')$

17. $new5(a) \leftarrow genc_3(a) \wedge \tau_1(a, a') \wedge new5(a')$
18. $new5(a) \leftarrow genc_3(a) \wedge \tau_2(a, a') \wedge new5(a')$
19. $new5(a) \leftarrow genc_3(a) \wedge \tau_3(a, a') \wedge new4(a')$
20. $new5(a) \leftarrow genc_3(a) \wedge \tau_3(a, a') \wedge new5(a')$
21. $new5(a) \leftarrow genc_3(a) \wedge \tau_4(a, a') \wedge new6(a')$
22. $new5(a) \leftarrow genc_3(a) \wedge \tau_5(a, a') \wedge new7(a')$

23. $new6(a) \leftarrow genc_4(a) \wedge \tau_1(a, a') \wedge new6(a')$
24. $new6(a) \leftarrow genc_4(a) \wedge \tau_2(a, a') \wedge new6(a')$
25. $new6(a) \leftarrow genc_4(a) \wedge \tau_3(a, a') \wedge new6(a')$
26. $new6(a) \leftarrow genc_4(a) \wedge \tau_4(a, a') \wedge new6(a')$
27. $new6(a) \leftarrow genc_4(a) \wedge \tau_5(a, a') \wedge new7(a')$

28. $new7(a) \leftarrow genc_5(a) \wedge \tau_2(a, a') \wedge new7(a')$
29. $new7(a) \leftarrow genc_5(a) \wedge \tau_6(a, a') \wedge new6(a')$

30. $new8(a) \leftarrow genc_6(a) \wedge \tau_1(a, a') \wedge new8(a')$
31. $new8(a) \leftarrow genc_6(a) \wedge \tau_2(a, a') \wedge new8(a')$
32. $new8(a) \leftarrow genc_6(a) \wedge \tau_3(a, a') \wedge new8(a')$
33. $new8(a) \leftarrow genc_6(a) \wedge \tau_4(a, a') \wedge new8(a')$
34. $new8(a) \leftarrow genc_6(a) \wedge \tau_5(a, a') \wedge new10(a')$

35. $new9(a) \leftarrow genc_7(a) \wedge \tau_1(a, a') \wedge new10(a')$
36. $new9(a) \leftarrow genc_7(a) \wedge \tau_6(a, a') \wedge new2(a')$

37. $new10(a) \leftarrow genc_8(a) \wedge \tau_1(a, a') \wedge new10(a')$
38. $new10(a) \leftarrow genc_8(a) \wedge \tau_2(a, a') \wedge new10(a')$
39. $new10(a) \leftarrow genc_8(a) \wedge \tau_3(a, a') \wedge new10(a')$
40. $new10(a) \leftarrow genc_8(a) \wedge \tau_4(a, a') \wedge new10(a')$
41. $new10(a) \leftarrow genc_8(a) \wedge \tau_6(a, a') \wedge new8(a')$

---

An inspection of program $S$ reveals that predicates $new1$ through $new10$ are *useless*, that is, for every predicate $p$ in the set $\{new1, \ldots, new10\}$ and for every clause $C$ that defines $p$ in $S$, there exists a predicate $q$ in $\{new1, \ldots, new10\}$ which occurs positively in the body of $C$. The removal of the clauses that define useless predicates preserves the perfect model of the program [9]. Thus, we remove clauses 2, 6.f, and 8 through 41, and we derive a program consisting of clause 1 only. By unfolding clause 1 we get the final program $T$, which consists of the clause $mutex \leftarrow$ only. Thus, $M(T) \models mutex$ and we have proved that:

$$M(Peterson) \models \forall X \; holds(X, initial \rightarrow \neg ef(unsafe))$$

that is, for any initial state and for any number $N$ of processes, the mutual exclusion property holds for Peterson's protocol for $N$ processes.

## 4 Related Work and Conclusions

The protocol verification method presented in this paper is based on the program transformation approach proposed in [16] for the verification of properties of locally stratified logic programs. We use locally stratified logic programs which are extended with array formulas and the properties we consider are temporal properties of parameterized systems, that is, systems consisting of an *arbitrary* number of *finite state* processes.

As yet, our method is *not* fully mechanical and human intervention is needed for the following two tasks: (i) the verification of array formulas and (ii) the introduction of new definitions by generalization.

The problem of verifying array formulas has been addressed in several papers (see [21] for a short survey). In general this problem is undecidable. However, some decidable fragments such as the *quantifier-free extensional theory of arrays*, have been identified [21]. Unfortunately, the array formulas considered in this paper cannot be reduced to formulas of the quantifier-free extensional theory of arrays. As an alternative approach we are working on the design of (necessarily incomplete) transformational strategies which would allow us to check validity of most array formulas that occur in practice in the verification of parameterized protocols.

The introduction of suitable new definitions by generalization is a typical issue of the program transformation methodology [3] and it corresponds to the discovery of suitable invariants of the protocol to be verified. There is no universal method for automating these generalization steps. However, we believe that suitable techniques can be devised by focusing on specific classes of protocols.

Other verification methods based on the transformational approach presented in [16] are those described in [7] and [8]. In [7] it is presented a method for verifying CTL properties of systems consisting of a *fixed number* of infinite state processes. The method of [7] makes use of locally stratified constraint logic programs, where the constraints are linear equations and disequations on real numbers. In this paper we have followed a paradigm similar to constraint logic programming where, however, the constraints are array formulas. The method presented here can be easily extended to deal with parameterized infinite state systems by considering, for instance, arrays of infinite state processes.

The paper [7] describes the verification of the mutual exclusion property for the parameterized Bakery protocol [11]. That paper uses locally stratified logic programs extended with formulas of the Weak Monadic Second Order Theory of $k$-Successors (WSkS), which describes monadic properties of strings. The array formulas considered in this paper are more expressive than WSkS formulas, because array formulas may express polyadic properties. However, as already mentioned, the general theory of array formulas is undecidable, while the theory WSkS is decidable.

Other transformational approaches to the verification of concurrent systems have been proposed in [12,18,19].

The method described in [12] uses partial deduction and abstract interpretation of logic programs for verifying safety properties of infinite state systems. Partial deduction is strictly less powerful than unfold/fold program transformation, which, on the other hand, is more difficult to mechanize when unrestricted transformations are considered. One of the main objectives of our future research is the design of suitably restricted unfold/fold transformations which are powerful enough for verification purposes and yet amenable to mechanization.

The work presented in [18,19] is the most similar to ours. Indeed, the authors of [18,19] use unfold/fold rules for transforming programs and proving properties of parameterized concurrent systems. Our paper differs from [18,19] in that, instead of using definite logic programs, we use logic programs with locally stratified negation and array formulas for the specification of concurrent systems and their properties. As a consequence, also the transformation rules we consider are different and more general than those used in [18,19].

Besides the above mentioned transformational methods, some more verification methods based on (constraint) logic programming have been proposed in the literature [6,10,14,17].

The methods proposed in [14,17] deal with finite state systems only. In particular, the method presented in [14] uses CLP with finite domains, extended with constructive negation and tabled resolution, for finite state local model checking, and the method described in [17] uses tabled logic programming to efficiently verify $\mu$-calculus properties of finite state systems expressed in the CCS calculus.

The methods presented in [6,10] deal with infinite state systems. In particular, the method presented in [6] uses constraint logic programs to represent infinite state systems. This method can be applied to verify CTL properties of

infinite state systems by computing approximations of least and greatest fixed points via abstract interpretation. An extension of this method has also been used for the verification of parameterized cache coherence protocols [5]. The method described in [10] uses logic programs with linear arithmetic constraints and Presburger arithmetic to verify safety properties of Petri nets. However, parameterized systems that use arrays, like Peterson's protocol, cannot be directly specified and verified using the methods in [6,10], because in general array formulas cannot be encoded as constraints over the real numbers or Presburger formulas.

Several verification techniques for parameterized systems have been presented also outside the area of logic programming (see [22] for a survey of some of these techniques). These techniques extend finite state model checking with various forms of *induction* (for proving properties for every value of the parameter) or *abstraction* (for reducing the verification of a parameterized system to the verification of a finite state system).

We do not have the space here for discussing the relationships of our work with all these techniques. We only want to mention the technique presented in [2], which is also applied for the verification of the parameterized Peterson's protocol. The technique proposed in [2] can be applied for verifying in an automatic way safety properties of all systems that satisfy a so-called *stratification* condition. Indeed, when this restriction holds for a given parameterized system, then the verification task can be reduced to the verification of a number of finite-state systems that are instances of the given parameterized system for suitable values of the parameter. However, Peterson's protocol does *not* satisfy the stratification condition and its treatment with the technique proposed in [2] requires a substantial amount of ingenuity.

Finally, we want to notice that techniques based on deduction and transformation which have been developed in the area of logic programming, seem particularly promising when moving from the problem of verifying finite state systems to the problem of verifying infinite state systems and parameterized systems, because in the latter verification logical reasoning plays a crucial role.

## Appendix

Below we give the definitions of the array formulas $genc_2$ through $genc_8$ occurring in the program $S$.

$$
\begin{array}{l}
genc_2(s(P,J,Q,S)) \equiv \\
\quad \exists i,k(1 \leq k \leq length(P) \wedge \\
\qquad ((P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=k-1 \ \wedge \ S[J[i]]=i) \vee \\
\qquad \ (P[i]=w_2 \ \wedge \ Q[i]=J[i]-1 \ \wedge \ J[i]=k \ \wedge \ S[J[i]-1]=i)) \wedge \\
\qquad \forall j(j \neq i \ \rightarrow \ P[j]=ncs \ \wedge \ Q[j]=0))
\end{array}
$$

$$
\begin{array}{l}
genc_3(s(P,J,Q,S)) \equiv \\
\quad \exists i(P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=1 \ \wedge \ S[J[i]]=i) \wedge
\end{array}
$$

$$\forall k((P[k]=ncs \ \wedge \ Q[k]=0) \vee$$
$$(P[k]=w_1 \ \wedge \ Q[k]=J[k] \ \wedge \ J[k]=1) \wedge$$
$$(P[k]=w_2 \ \wedge \ Q[k]=J[k]-1 \ \wedge \ J[k]=2))$$

$$genc_4(s(P,J,Q,S)) \equiv$$
$$\exists m(m \leq length(P) \wedge$$
$$\forall k(1 \leq k \leq m \ \rightarrow \ \exists i(P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=k-1 \ \wedge \ S[J[i]]=i)) \wedge$$
$$\forall j((P[j]=ncs \ \wedge \ Q[j]=0) \vee$$
$$\exists k(k \leq m \ \wedge \ ((P[j]=w_1 \ \wedge \ Q[j]=J[j] \ \wedge \ J[j]=k-1) \vee$$
$$(P[j]=w_2 \ \wedge \ Q[j]=J[j]-1 \ \wedge \ J[j]=k)))))$$

$$genc_5(s(P,J,Q,S)) \equiv$$
$$\exists i(P[i]=cs \ \wedge \ Q[i]=J[i]-1 \ \wedge \ J[i]=length(P) \ \wedge \ S[J[i]-1]=i) \wedge$$
$$\forall k(1 \leq k \leq length(P)-1 \ \rightarrow$$
$$\exists i(P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=k \ \wedge \ S[J[i]]=i))$$

$$genc_6(s(P,J,Q,S)) \equiv$$
$$\exists l,n(n \leq length(P) \wedge$$
$$((P[l]=w_1 \ \wedge \ Q[l]=J[l] \ \wedge \ J[l]=n-1 \ \wedge \ S[J[l]]=l) \vee$$
$$(P[l]=w_2 \ \wedge \ Q[l]=J[l]-1 \ \wedge \ J[l]=n \ \wedge \ S[J[l]-1]=l)) \wedge$$
$$\exists m(m \leq n \wedge$$
$$\forall k(1 \leq k \leq m \ \rightarrow \ \exists i(P[i]=w_1 \ \wedge \ Q[i]=J[i] \ \wedge \ J[i]=k-1 \ \wedge \ S[J[i]]=i)) \wedge$$
$$\forall j((P[j]=ncs \ \wedge \ Q[j]=0) \vee$$
$$\exists k(k \leq m \ \wedge \ ((P[j]=w_1 \ \wedge \ Q[j]=J[j] \ \wedge \ J[j]=k) \vee$$
$$(P[j]=w_2 \ \wedge \ Q[j]=J[j]-1 \ \wedge \ J[j]=k))))))$$

$$genc_7(s(P,J,Q,S)) \equiv$$
$$\exists i(P[i]=cs \ \wedge \ Q[i]=J[i]-1 \ \wedge \ J[i]=length(P) \ \wedge \ S[J[i]-1]=i \wedge$$
$$\forall k(k \neq i \ \rightarrow \ (P[k]=ncs \ \wedge \ Q[k]=0)))$$

$$genc_8(s(P,J,Q,S)) \equiv$$
$$\exists i(P[i]=cs \ \wedge \ Q[i]=J[i]-1 \ \wedge \ J[i]=length(P) \ \wedge \ S[J[i]-1]=i \wedge$$
$$\exists m(m \leq length(P) \wedge$$
$$\forall l(1 \leq l \leq m \ \rightarrow \ \exists j(P[j]=w_1 \ \wedge \ Q[j]=J[j] \ \wedge \ J[j]=l-1 \ \wedge \ S[J[j]]=j)) \wedge$$
$$\forall j(j \neq i \ \rightarrow \ ((P[j]=ncs \ \wedge \ Q[j]=0) \vee$$
$$\exists k(k \leq m \wedge$$
$$((P[j]=w_1 \ \wedge \ Q[j]=J[j] \ \wedge \ J[j]=k-1) \vee$$
$$(P[j]=w_2 \ \wedge \ Q[j]=J[j]-1 \ \wedge \ J[j]=k)))))))$$

## References


1. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
2. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 221–234. ACM, July 2001.

3. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
4. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
5. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
6. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'01, Florence (Italy)*, Technical Report DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of sets of infinite state systems using program transformation. In A. Pettorossi, editor, *Proceedings of LOPSTR 2001, Eleventh International Workshop on Logic-based Program Synthesis and Transformation*, Lecture Notes in Computer Science 2372, pages 111–128. Springer-Verlag, 2002.
9. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 292–340. Springer, 2004.
10. L. Fribourg and H. Olsén. A decompositional approach for computing least fixed-points of Datalog programs with z-counters. *Constraints*, 2(3/4):305–335, 1997.
11. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
12. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proceedings of LOPSTR '99, Venice, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 1999.
13. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
14. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In J. W. Lloyd, editor, *First International Conference on Computational Logic, CL 2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 384–398. Springer-Verlag, 2000.
15. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
16. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, editor, *First International Conference on Computational Logic, CL 2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 613–628. Springer, 2000.
17. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV '97*, Lecture Notes in Computer Science 1254, pages 143–154. Springer-Verlag, 1997.
18. A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *CAV 2001*, pages 25–37, 2001.
19. A. Roychoudhury and C. R. Ramakrishnan. Unfold/fold transformations for automated verification. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 261–290. Springer, 2004.

20. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
21. A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *16th IEEE Symposium on Logic in Computer Science*, pages 29–37. IEEE Press, 2001.
22. L. D. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3-4):139–169, 2004.

# Computational Logic in GeoPKDD*

Fosca Giannotti[1], Alessandra Raffaetà[2], Chiara Renso[1]

[1] KDD LAB - ISTI CNR - Pisa - {giannotti, renso}@isti.cnr.it
[2] Dipartimento di Informatica - Università Ca' Foscari Venezia - raffaeta@dsi.unive.it

**Abstract.** The rapidly growing collections of privacy-sensitive telecommunication data from mobile phones and other location-aware devices, are enabling new classes of applications. The space-time trajectories of the personal mobile devices and their human companions offer interesting practical opportunities to find behavioural patterns, to be exploited for several challenging applications. In this setting the MIUR project GeoPKDD (Geographic Privacy-aware Knowledge Discovery and Delivery) has been proposed with the aim of devising a novel geographic privacy-preserving knowledge discovery process. This paper describes the context and the objectives of the GeoPKDD project, focusing on the role played by computational logic inside such a new discovery process.

## 1 Introduction and Motivation

Spatio-temporal datasets are, and will be, growing rapidly, in particular, due to the collection of telecommunication data from mobile phones and other location-aware devices, as well as the daily collection of transaction data through database systems, network traffic controllers, web servers, sensors. The large availability of these forms of geo-referenced information is expected to enable novel classes of applications, where the discovery of consumable, concise, and applicable knowledge is the key step. As a distinguishing example, the presence of a large number of location-aware wirelessly connected mobile devices presents a growing possibility to access space-time trajectories of these personal devices and their human companions: trajectories are indeed the traces of moving objects and individuals. These trajectories contain detailed information about personal and vehicular mobile behaviour, and therefore offer interesting practical opportunities to find behavioural patterns, to be used for instance in traffic and sustainable mobility management, e.g., to study the accessibility to services. Clearly, in these applications privacy is a concern. As a prototypical application scenario, assume that source data are log transactions from mobile cellular phones, reporting user's movements among the cells in the network; these are streams of raw data (log entries) about users entering a cell – *(userID, time, cellID)* – or, in the near future, even user's position within a cell – *(userID, time, cellID, X, Y)* and, in the case of GPS/Galileo equipped devices, user's absolute position.

In this context, the MIUR project GeoPKDD (Geographic Privacy-aware Knowledge Discovery and Delivery) has been proposed with the aim of devising a novel geographic privacy-preserving knowledge discovery process. Such a process, illustrated in Fig. 1, consists of three main steps: trajectories reconstruction, knowledge extraction, and interpretation and delivery of the obtained information.

* This work has been partially supported by the MIUR Italian Project GeoPKDD.

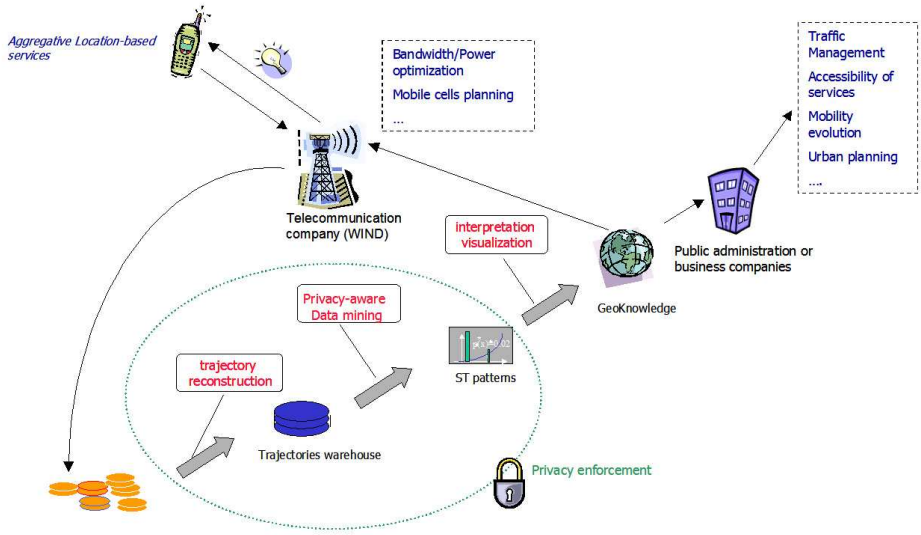


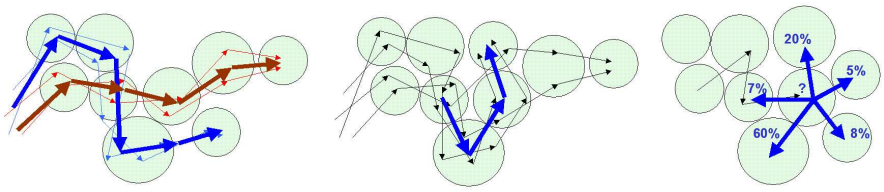

**Fig. 1.** The GeoPKDD process

**Trajectory reconstruction.** In this basic phase, a stream of raw data about moving people has to be processed to obtain ready-to-use trajectories, building a privacy-aware trajectories warehouse. Reconstruction of trajectories is *per se* a challenging problem. The reconstruction accuracy of trajectories, as well as their level of spatio-temporal granularity, depend on the quality of the log entries, since the precision of the position may range from the granularity of a cell of varying size to the relative (approximated) position within a cell. Indeed, each moving object trajectory is typically represented as a set of localisation points of the tracked device, called *sampling*. This representation has intrinsic imperfection due to mainly two aspects. The first source of imperfection is the error measurement of the tracking device. The second one is related to the sampling rate and involves the trajectory reconstruction process that approximates the movement of the object between two localisation points. Although for some application, linear interpolation can be an acceptable approximation of the real trajectory, we believe that this could be a too coarse approximation and more sophisticated techniques are to be investigated to take into account the spatial, and possibly temporal, imperfection in the reconstruction process.

**Knowledge extraction.** Spatio-temporal data mining methods must be developed to extract useful patterns out of trajectories. Spatio-temporal data mining is still in its infancy ([1, 3, 7, 13, 14]), and even the most basic questions in this field are still largely unanswered: what kinds of patterns can be extracted from trajectories? Which methods and algorithms should be applied to extract them? How can such patterns be effectively used to improve the comprehension of the application domain and to deliver better

services? How can privacy be guaranteed? The following basic examples give a glimpse of the wide variety of patterns and possible applications it is expected to manage:

- *Clustering*, the discovery of groups of "similar" trajectories, together with a summary of each group (see Fig. 2.(a)). Knowing which are the main routes (represented by clusters) followed by people during the day can represent a precious information for improving several different services to citizens. E.g., trajectory clusters may highlight the presence of important routes not adequately covered by the public transportation service.
- *Frequent patterns*, the discovery of frequently followed (sub)-paths (see Fig. 2.(b)). Such information can be useful in urban planning, e.g., by spotlighting frequently followed inefficient vehicle paths, which can be the result of a mistake in the road planning.
- *Classification*, the discovery of behaviour rules, aimed at explaining the behaviour of current users and predicting that of future ones (see Fig. 2.(c)). Urban traffic simulations are a straightforward example application for this kind of knowledge, since a classification model can represent a sophisticated alternative to the simple ad hoc behaviour rules, provided by domain experts, on which actual simulators are based.




(a) Clusters (b) Frequent patterns (c) Classification

**Fig. 2.** Examples of extracted patterns.

**Knowledge delivery.** Extracted patterns are very seldom geographic knowledge ready-to-use: it is necessary to reason on patterns and on pertinent background knowledge, evaluate patterns interestingness, refer them to geographic information, find out appropriate presentations and visualisations.

We believe that a computational logic component could be profitably used as knowledge representation formalism and inference engine supporting reasoning.

In the next section we show some preliminary ideas on the role computational logic techniques play in the realisation of the knowledge interpretation and reasoning functionalities. Privacy issues will not show up in our discussion, since, according to the project plan, we think that they should be mainly addressed in the construction of the trajectory warehouse and in the definition of the spatio-temporal data mining algorithms.

## 2 Computational Logic for Knowledge Management

The availability of a suitable query language is clearly fundamental for a profitable use of data. Current (geographical) information systems, e.g., Oracle or ArcGIS, allow the user to exploit some consolidated mechanisms such as SQL-like queries, statistical functions and some spatial operations, thus offering an efficient and mature technology to query standard relational or spatial data. However they do not provide high level operations to handle the time and space related dimensions: nowadays the development of a spatio-temporal application with commercial systems often requires the programming of some ad-hoc components by means of procedural languages provided by the system itself for customisation purposes. The absence of specific spatio-temporal operations makes the customisation of applications extremely complex.

In order to provide a more efficient and flexible treatment of spatio-temporal data, several attempts to exploit the deductive capabilities of logics to reason on geographic data have been done in the literature [15, 11, 4, 2, 5]. Our research activity, briefly outlined below, follows this line.

**The language.** We defined the language STACLP [9], a constraint logic programming language with spatial and temporal annotations. The pieces of spatio-temporal information are given by pairs of annotations which specify the spatial extent of an object or of a property at a certain time period. The use of annotations makes time and space explicit while avoiding the proliferation of spatial and temporal variables and quantifiers. Annotated formulae come with inference rules that allow to combine annotations in order to derive new information. Moreover, STACLP supports both definite and indefinite spatial and temporal information. Definite information is modelled by $\texttt{atp}(X,Y)$, $\texttt{at}\,T$, $\texttt{thr}\,[(X_1,X_2),(Y_1,Y_2)]$ and $\texttt{th}\,[T_1,T_2]$ annotations specifying, respectively, the spatial and time point, the spatial region and the time interval in which a property holds. On the other hand, indefinite information is expressed by $\texttt{inr}\,[(X_1,X_2),(Y_1,Y_2)]$ and $\texttt{in}\,[T_1,T_2]$ annotations, expressing the fact that there are some point(s) in a given spatial region or temporal interval - which ones may not be known - in which a property holds. It is worth noticing that time and space can be discrete or dense, it depends on the application domain. Finally, STACLP allows to establish a dependency between space and time, thus permitting to model continuously moving points and regions.

*Example 1.* A moving point can be modelled easily by using a clause of the form:

$$\mathit{moving\_point}\,\texttt{atp}(X,Y)\,\texttt{at}\,T \leftarrow \mathit{constraint}(X,Y,T)$$

For instance, consider a car running on a straight road with speed $v$ and assume that its initial position at time $t_0$ is $(x_0,y_0)$. The moving car can be represented by the clause

$$\mathit{car\_position}\,\texttt{atp}(X,Y)\,\texttt{at}\,T \leftarrow X = x_0 + v(T-t_0), Y = y_0 + v(T-t_0)$$

whose body "computes" in the obvious way, the position $(X,Y)$ of the car at time $T$.

Such a language offers facilities for handling spatio-temporal information and has been used successfully as a high level interface on the top of GIS [10, 5]. Rules can be used to represent general knowledge about the collected data (background knowledge),

and deductive capabilities can provide answers to queries that require some inference besides the crude manipulation of the data. In addition, as shown e.g. in [8] the language can reconciliate both deductive and inductive inference mechanisms. Induction can help extracting implicit knowledge from data and, according to the impressive success in the knowledge discovery in the database field, it can provide a powerful support to decision making.

**Trajectories.** In STACLP, starting from a set of spatio-temporal locations and by using linear interpolation, an approximation of trajectories can be easily modelled: the location points are represented by means of `atp`/`at` annotations and then the straight line between the two end points is expressed as a constraint.

Specifically, a set of localisations of each object $o$, say $(x_i, y_i, t_i)$ for $i = 1, \ldots, N$, can be represented by the following $N$ STACLP facts:

```
fix(o) atp (x1, y1) at t1
fix(o) atp (x2, y2) at t2
⋮
fix(o) atp (xN, yN) at tN
```

Such localisations define the *core* of the trajectory of object $o$, which is then *completed* by defining all the intermediate points through linear interpolation using the following STACLP rules:

```
traj(O) atp (X, X) at T :- fix(O) atp (X, Y) at T.
traj(O) atp (X, Y) at T :- fix(O) atp (X1, Y1) at T1,
                           fix(O) atp (X2, Y2) at T2,
                           succ(T1,T2), T1 < T < T2,
                           X=(X1(T2-T)+X2(T-T1))/(T2-T1),
                           Y=(Y1(T2-T)+Y2(T-T1))/(T2-T1).
```

In the body of the second rule, approximate points $(x,y)$ are computed by using the equation for the line passing through two given points. The presence of the (standard) *successor* predicate `succ`, defined as true for all and only the couples of (strictly) consecutive location points, ensures that no other observation exists between times $t_1$ and $t_2$, i.e., the interpolation is performed only between consecutive location points.

**Uncertainty**. In applications of geographical nature, data rarely come truly free of errors: imperfection in some form or another is an endemic feature of geographic information. In particular, trajectories of moving objects - our data of interest - may not be completely specified or identified with sufficient accuracy (e.g. we may only know that a given object crossed an area, without knowing the exact path followed while crossing such an area). A representative case is given by the *log* transactions: each entry just represents the presence of the device in the cell, thus identifying each position of the moving object with a degree of imprecision. This feature can be immediately represented in STACLP exploiting the `in` annotation for indefinite spatial localisations.

A further challenging issue to investigate is the possibility of extending our framework to cope with different aspects of uncertainty in spatio-temporal data (i.e. degrees of certainty, fuzzy and rough sets). Following a typical idea in the related literature (for

a survey, see [12]), a possible way to deal with uncertain data in our formalism consists of considering a lattice of annotations that model different degrees of certainty (e.g., the interval [0,1] with the usual ordering is the lattice commonly used for fuzzy logic).

**Reasoning Architecture for GeoPKDD**. The knowledge interpretation phase can be intended as a reasoning step that takes place only after the extraction of patterns from raw data (see Fig. 1). The rules apply to patterns, possibly combining them with background knowledge, to perform some kind of high level reasoning. However, it can be convenient to see, more generally, knowledge interpretation as an activity that applies at various steps of the discovery process, from raw data and trajectories representation, to extracted patterns and background knowledge, possibly to final end-user application.

In this sense, an architecture which looks promising combines the efficiency of data management systems (i.e., DBMS, trajectories warehouse or Geographical Information System), with the flexibility and the expressive power of computational logic (e.g., STACLP) in a two-layered structure, as illustrated in Fig. 3. The data of interest (log transactions, trajectories, patterns) are exploited at two levels: they can be kept in the data management systems or exported at the logical level. The reasoning, logic-based layer can ask the data management systems to access data and to perform some demanding operations on data, e.g. visualisation or data mining operations, which are offered as primitive (and thus implemented efficiently) by such systems. The data stored in the lower layer can be exported to the logic-based layer in order to perform complex, deductive and/or inductive reasoning functionalities, which are not supported by standard data management systems. The strategy will be to exploit the efficiency of the data management systems as much as possible, exporting data into the logical representation only if strictly necessary to perform analysis not provided by the underlying systems.

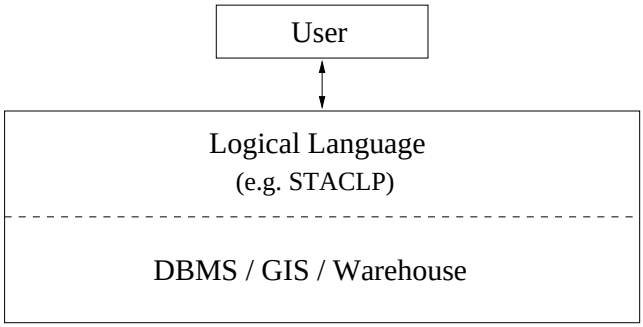



**Fig. 3.** Architecture of the system.

As an example, in the case of our project dealing with trajectories, the data management system could include the *trajectory warehouse* and offer some (spatio-temporal) data mining primitives. The logical layer could make use of STACLP, enriched with a collection of primitive predicates, e.g., *trajectory(Tr_id, prop)* which calls the trajectory warehouse in order to obtain suitably identified trajectories, or *cluster(dataset,patterns)*

which calls an external data mining algorithm, getting back the extracted patterns. Trajectories could be exported and represented at the logic layer, using the language STA-CLP as already discussed.

This two-layered architecture has been already experimented in [5] where the logical language is on top of a GIS. A similar approach is followed also in [6], where the deductive database language *LDL++* is extended to deal with data mining primitives implemented as external calls. We believe that these approaches can represent a starting point for the definition of the reasoning architecture for GeoPKDD.

## References


1. T. Abraham. *Knowledge Discovery in Spatio-Temporal Databases*. PhD thesis, School of Computer and Information Science, Faculty of Information Technology, University of South Australia, 1999.
2. J. Chomicki and P.Z. Revesz. Constraint-Based Interoperability of Spatiotemporal Databases. *GeoInformatica*, 3(3):211–243, 1999.
3. S. Gaffney and P. Smyth. Trajectory clustering with mixture of regression models. In *KDD Conf.*, pages 63–72. ACM, 1999.
4. S. Grumbach, P. Rigaux, and L. Segoufin. Spatio-Temporal Data Handling with Constraints. *GeoInformatica*, 5(1):95–115, 2001.
5. P. Mancarella, A. Raffaetà, C. Renso, and F. Turini. Integrating Knowledge Representation and Reasoning in Geographical Information Systems. *International Journal of GIS*, 18(4):417–446, June 2004.
6. G. Manco. *Foundations of a Logic-Based Framework for Intelligent Data Analysis*. PhD thesis, Dipartimento di Informatica, University of Pisa, 2001.
7. M. Nanni. *Clustering Methods for Spatio-Temporal Data*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 2002.
8. M. Nanni and F. Turini A. Raffaetà, C. Renso. A Declarative Framework for Reasoning on Spatio-Temporal Data. In *Spatio-Temporal Databases. Flexible Querying and Reasonig*, pages 75–104. Springer, 2004.
9. A. Raffaetà and T. Frühwirth. Spatio-Temporal Annotated Constraint Logic Programming. In *PADL'01*, volume 1990 of *LNCS*, pages 259–273. Springer, 2001.
10. A. Raffaetà, C. Renso, and F. Turini. Enhancing GISs for Spatio-Temporal Reasoning. In *ACM GIS'02*, pages 35–41. ACM Press, 2002.
11. S. Spaccapietra, editor. *Spatio-Temporal Data Models & Languages (DEXA Workshop)*. IEEE Computer Society Press, 1999.
12. V. S. Subrahmanian. Uncertainty in Databases and Knowledge Bases. In *Advanced database systems*. Morgan Kaufmann Publishers Inc., 1997.
13. N. Sumpter and A. Bulpitt. Learning spatio-temporal patterns for predicting object behaviour. *Image and Vision Computing*, 18(9):697–704, 2000.
14. I. Tsoukatos and D. Gunopulos. Efficient mining of spatiotemporal patterns. In *Advances in Spatial and Temporal Databases, 7th International Symposium, SSTD 2001*, volume 2121 of *LNCS*, pages 425–442, 2001.
15. M. F. Worboys. *GIS - A Computing Perspective*. Taylor & Francis, 1995.

# An ILP Approach to Spatial Clustering

Antonio Varlaro, Annalisa Appice, Antonietta Lanza, and Donato Malerba

Dipartimento di Informatica, Università degli Studi di Bari
via Orabona, 4 - 70126 Bari - Italy
{varlaro,appice,lanza,malerba}@di.uniba.it

**Abstract.** Spatial clustering is a fundamental task in Spatial Data Mining where the goal is to group nearby sites and form clusters of homogeneous regions. Spatial clustering must be driven by the discrete spatial structure of data that expresses the (spatial) relational constraints between separate sites. Only similar sites (transitively) connected in the discrete spatial structure may be clustered together. CORSO is a novel clustering method that resorts to an ILP approach to mine spatial data and exploits the concept of neighborhood to capture relational constraints. Spatial data are expressed in a first-order formalism and similarity among structured objects is computed as degree of matching with respect to a common generalization. Two real-world applications are described.

## 1 Introduction

In recent times the growing interest in the extension of data mining methods to spatial domain as well as the application of Inductive Logic Programming (ILP) to knowledge discovery in spatial database is driven by the pressure from the public and private sectors to provide solutions to a wide range of data intensive spatial applications (e.g., environmental analysis or urban planning). We consider a specific task, that is, spatial clustering and investigate spatial clustering tools as advanced means for creating models of spatially distributed phenomena.

Many approaches to discover spatial clusters have been proposed in literature. For instance, Ng and Han [11] have extended the $k$-medoid partitioning algorithm [6] to group point data in a set of $k$ clusters, while Ester et al. [5] have proposed DBSCAN to detect dense clusters of arbitrary shape from point spatial data with noise. Similarly, Wang and Hamilton [16] have proposed a density-based clustering method that estimates density within a cluster on the basis of only non spatial properties of point spatial data. Conversely, Sander et al. have proposed GDBSCAN [14] that is a generalization of DBSCAN in order to cluster not only point data but also spatially extended objects (lines or areas) taking into account both spatial and non spatial attributes. At this aim, GDBSCAN uses the notion of neighborhood, that is, any binary relation that is symmetric and reflexive (e.g. distance, meet). A neighborhood relation can be modeled as graph, namely neighborhood or proximity graph [15]. This graph imposes a discrete spatial structure on data that guides the cluster detection such that only the neighbor objects may be clustered together However, all

these methods suffer from severe limitations due to the single-table assumption [2], that is, data to be mined are stored in a single table of a relational database, such that each row (or tuple) represents an independent unit of the sample population and columns correspond to properties of units. This is a strong restriction in representing spatial objects that have different properties and are modeled by as many data tables (relational data model) as the number of object types. To face this additional level of complexity we resort to the field of (multi-)relational data mining [2] and adopt subsets of first-order logic to express the relational nature of both data to be mined and relational patterns to be discovered. In particular, ILP approach to relational data mining supplies representation and reasoning means appropriate for the spatial domain where relations among objects play a key role and are often inferred by qualitative reasoning. However, ILP methods of clustering [7, 1] generally work in the learning from interpretation setting [13] that allows to mine examples and background knowledge stored as Prolog programs exploiting expressiveness of first-order representation during cluster detection. The interpretation corresponding to each example $e$ given the background knowledge $BK$ is here intended as the minimal Herbrand model of $e \wedge BK$ and the implicit assumption is that separate interpretations are independent. This leads to ignore relational constraints eventually relating separate interpretations (e.g. geographic contiguity of areal units). To overcome these deficiencies, we propose to combine a graph-based partitioning algorithm with a relational clustering method to mine both relational constraints imposing the discrete spatial structure and relational data representing structured objects (spatial unit) to be clustered.

The paper is organized as follows. In the next section we discuss a new relational clustering method working in presence of the discrete spatial structure. Two applications of spatial clustering for topographic map interpretation and geo-referenced census data analysis are reported in Section 3. Finally Section 4 presents some conclusions and future works.

## 2 The Method

In a quite general formulation, the problem of clustering structured objects (e.g., complex areal units), which are related by links representing persistent relations between objects (e.g., spatial correlation), can be defined as follows: *Given*: (i) a set of structured objects $O$, (ii) a background knowledge $BK$ and (iii) a binary relation $R$ expressing links among objects in $O$; *Find* a set of homogeneous clusters $\mathbf{C} \subseteq \wp(O)$ that is feasible with $R$.

Each structured object $o_i \in O$ can be described by means of a conjunctive ground formula (conjunction of ground selectors) in a first-order formalism, while background knowledge $BK$ is expressed with first-order clauses that support some qualitative reasoning on $O$. In both cases, each basic component (i.e., *selector*) is a relational statement in the form $f(t_1, \ldots, t_n) = v$, where $f$ is a function symbol or *descriptor*, $t_i$ are constants or variables (but not functions) and $v$ is a value taken from the categorical or numerical range of $f$. Structured

objects are then related by $R$ that is a binary relation $R \subseteq O \times O$ imposing a discrete structure on $O$. This relation may be either purely spatial (topological, distance and directional relations) or hybrid, which mixes both spatial and non spatial properties (e.g. two regions are connected by a road). The relation $R$ can be described by the graph $G = (N_O, A_R)$ where $N_O$ is the set of nodes $n_i$ representing each structured object $o_i$ and $A_R$ is the set of arcs $a_{i,j}$ describing links between each pair of nodes $\langle n_i, n_j \rangle$ according to the discrete structure imposed by $R$. This means that there is an arc from $n_i$ to $n_j$ only if $o_i R o_j$. Let $N_R(n_i)$ be the *R-neighborhood* of a node $n_i$ such that $N_R(n_i) = \{n_j |$ there is an arc linking $n_i$ to $n_j$ in $G\}$, a node $n_j$ is *R-reachable* from $n_i$ if $n_j \in N_R(n_i)$, or $\exists n_h \in N_R(n_i)$ such that $n_j$ is $R$-reachable from $n_h$.

**Algorithm 1** Top-level description of CORSO algorithm.

```
1: function CORSO(O, BK, R, h − threshold) → CList;
2: CList ← ∅; O_BK ←saturate(O,BK); C ← newCluster( );
3: for each seed ∈ O_BK do
4:   if seed is UNCLASSIFIED then
5:     N_seed ← neighborhood(seed,O_BK,R);
6:     for each o ∈ N_seed do
7:       if o is assigned to a cluster different from C then
8:         N_seed = N_seed/o;
9:       end if
10:    end for
11:    T_seed ← neighborhoodModel(N_seed);
12:    if homogeneity(N_seed, T_seed) ≥ h − threshold then
13:      C.add(seed); seedList ← ∅;
14:      for each o ∈ N_seed do
15:        if o in UNCLASSIFIED or o is NOISE then
16:          C.add(o); seedList.add(o);
17:        end if
18:      end for
19:      ⟨C, T_C⟩ ←expandCluster(C,seedList,O_BK,R,T_seed,h − threshold);
20:      CLabel=clusterLabel(T_C); CList.add(⟨C, CLabel⟩); C ← newCluster( );
21:    else
22:      seed ← NOISE;
23:    end if
24:  end if
25: end for
26: return CList;
```

According to this graph-based formalization, a clustering $\mathbf{C} \subseteq \wp(O)$ is feasible with the discrete structure imposed by $R$ when each cluster $C \in \mathbf{C}$ is a subgraph $G_C$ of the graph $G(N_O, A_R)$ such that for each pair of nodes $\langle n_i, n_j \rangle$ of $G_C$, $n_i$ is $R$-reachable from $n_j$, or vice-versa.

CORSO is a clustering method that both integrates a neighborhood-based graph partitioning to obtain clusters which are feasible with $R$ discrete structure

**Algorithm 2** Expand current cluster by merging homogeneous neighborhood.

```
   function expandCluster(C, seedList, O_BK, R, T, h − threshold) → ⟨C, T⟩;
 2: while (seedList is not empty) do
      seed ← seedList.first(); N_seed ← neighborhood(seed,O_BK,R);
 4:   for each o ∈ N_seed do
        if o is assigned to a cluster different from C then
 6:       N_seed = N_seed/o;
        end if
 8:   end for
      T_seed ← neighborhoodModel(N_seed);
10:   if homogeneity(N_seed, {T, T_seed}) ≥ h − threshold then
        for each o ∈ N_seed do
12:       C.add(o); seedList.add(o);
        end for
14:     seedList.remove(seed); T ← T ∪ T_seed;
      end if
16: end while
   return ⟨C, T⟩;
```

and resorts to a multi-relational approach to evaluate similarity among structured objects and form homogeneous clusters (i.e., clusters grouping structured objects sharing a similar relational description according to some similarity criterion). Top-level description of the method is presented in Algorithm 1. The key idea is to exploit the $R$-neighborhood construction and build clusters feasible with $R$-discrete structure by merging partially overlapping homogeneous neighborhood units. Cluster construction starts from an arbitrary node $n$ (seed node) of $G$ such that the $R$-neighborhood $N_R(n)$ is an homogeneous cluster $C$ whose model $T_C$ is a generalization of $C$ and then iteratively expands the cluster in question by including the objects of each $R$-neighborhood $N_R(n_i)$ of a node $n_i \in G_C$ (neighborhood expansion) when $N_R(n_i)$ is an homogeneous set (see Algorithm 2). The $R$-neighborhoods involved in the construction of any cluster contain only the objects not yet classified into a different cluster. Moreover, homogeneity within a neighborhood is estimated by comparing each object in the neighborhood with the model of the cluster currently built. Obviously the choice of the seed node affects clustering since neither reassignments nor multiple-cluster assignments are performed.

Although CORSO appears quite similar to GDBSCAN in detecting spatial clusters by exploiting a neighborhood-based partitioning of the graph representing the discrete spatial structure, CORSO differs from GDBSCAN in evaluating homogeneity within a neighborhood to be added to the current cluster. Indeed, GDBSCAN retrieves all spatial objects density-reachable from an arbitrary core object by building successive neighborhoods, but it checks the density within a neighborhood by ignoring the entire cluster. This yields a density-connected set, where density is efficiently estimated independently from the neighborhoods already merged in forming the current cluster. However, this approach may lead

to merge connected neighborhoods sharing some objects but modeling different phenomena. Moreover, GDBSCAN computes density within each neighborhood according to a weighted cardinality function (e.g. aggregation of non spatial values) that assumes single table data representation. CORSO overcomes these limitations by computing density within a neighborhood in terms of degree of similarity among all relationally structured objects falling in the neighborhood with respect to the model of the entire cluster currently built. In particular, following the suggestion given in [10], we evaluate homogeneity within a neighborhood $N_R(n_i)$ to be added to the cluster $C$ as the average degree of matching between objects of $N_R(n_i)$ and the cluster model $T_{C'}$ with $C' = C \cup N_R(n_i)$. Details on cluster model determination, neighborhood homogeneity estimation and cluster labeling are reported below.

### 2.1 Cluster model generation

Let $C$ be the cluster currently built by merging $w$ neighborhood sets $N_1, \ldots, N_w$, we assume that the cluster model $T_C$ is a set of first-order theories $\{T_1, \ldots, T_w\}$ for the concept $C$ where $T_i$ is a model for the neighborhood set $N_i$. More precisely, $T_i$ is a set of first-order clauses:

$$T_i : \{cluster(X) = c \leftarrow H_{i1}, \ldots, cluster(X) = c \leftarrow H_{iz}\},$$

where each $H_{ij}$ is a conjunctive formula describing a sub-structure shared by one or more objects in $N_i$ and $\forall o_i \in N_i$, $BK \cup T_i \models o_i$, that is, each object in $N_i$ can be explained by the model $T_i$ and the background knowledge $BK$.

Such model can be learned by resorting to the ILP system ATRE [8] that adopts a separate-and-conquer search strategy to learn a model of structured objects from a set of training examples and eventually counter-examples. In this context, ATRE learns a model for each neighborhood set without considering any counter-examples. To this aim, since only consistent clauses can be induced without negative examples and ATRE searches the hypotheses space until a fixed number of consistent clauses have been generated, the learning parameters are opportunely fixed to make the learner able to generate an adequate number of clauses and to choose the clause covering the highest number of examples and with the highest number of literals in the body. The search of a model starts with the most general clause, that is, $cluster(X) = c \leftarrow$, and proceeds top-down by adding selectors (literals) to the body according to some preference criteria according to which a clause can be considered better than another one (e.g. number of objects covered or number of literals). Selectors involving both numerical and categorical descriptors are handled in the same way, that is, they have to comply with the property of linkedness and are sorted according to preference criteria. The only difference is that selectors involving numerical descriptors are generalized by computing the closed interval that best covers positive examples and eventually discriminates from counter-examples, while selectors involving categorical descriptors with same function value are generalized by simply turning all ground arguments into the corresponding variables and assigning as function value the set containing only the value taken from the input

categorical descriptors. In both cases, the generalization of two selectors is in the form $f(X_1, \ldots, X_n) \in v$, where $v$ is an interval or a set.

### 2.2 Neighborhood homogeneity estimation

The homogeneity $h$ of a neighborhood set $N$ to be added to the cluster $C$ is computed as follows:

$$h(N, T_{C \cup N}) = \frac{1}{\#N} \sum_i h(o_i, T_{C \cup N}) = \frac{1}{\#N} \sum_i \frac{1}{w+1} \sum_j h(o_i, T_j), \quad (1)$$

where $\#N$ is the cardinality of the neighborhood set $N$ and $T_{C \cup N}$ is the cluster model of $C \cup N$ formed by both $\{T_1, \ldots, T_w\}$, i.e., the model of $C$, and $T_{w+1}$, i.e., the model of $N$ built as explained above. It is clear that, as defined above, the homogeneity of a neighborhood is calculated according to the entire cluster rather than just the neighborhood itself. Since $T_j = \{H_{1j}, \ldots, H_{zj}\}$ $(z \geq 1)$ with each $H_{rj}$ a conjunctive formula in first-order formalism, we assume that:

$$h(o_i, T_j) = \frac{1}{z} \sum_r fm(o_i, H_{rj}), \quad (2)$$

where $fm$ is a function returning the degree of matching of an object $o_i \in N$ against the conjunctive formula $H_{rj}$. In this way, the definition of homogeneity of a neighborhood set $N = \{o_1, \ldots, o_n\}$ with respect to some logical theory $T_{C \cup N}$ is closely related to the problem of comparing (matching) the conjunctive formula $f_i$ representing an object $o_i \in N$[1] with a conjunctive formula $H_{rj}$ forming the model $T_j$ in order to discover likenesses or differences [12]. This is a directional similarity judgment involving a *referent* $R$, that is the description or prototype of a class (cluster model) and a *subject* $S$ that is the description of an instance of a class (object to be clustered).

In the classical matching paradigm, the matching of $S$ against $R$ corresponds to compare them just for equality. In particular, when both $S$ and $R$ are conjunctive formulas in first-order formalism, matching $S$ against $R$ corresponds to check the existence of a substitution $\theta$ for the variables in $R$ such that $S = \theta(R)$. This last condition is generally weakened by requiring that $S \Rightarrow \theta(R)$, where $\Rightarrow$ is the logical implication. However, the requirement of equality, even in terms of logical implication, is restrictive in presence of noise or variability of the phenomenon described by the referent of matching. Therefore a flexible definition of matching is needed that aims at comparing two descriptions and identifying similarities rather than equalities. The result of such a flexible matching is a number in the interval $[0, 1]$ that is the probability of precisely matching $S$ against $R$, provided that some change described by $\theta$ is possibly made in the description $R$.

The problem of computing flexible matching to compare structures is not novel. Esposito et al. [4] have formalized a computation schema for flexible

[1] The conjunctive formula $f_i$ is here intended as the description of $o_i \in N$ saturated according to the $BK$.

matching on formulas in first-order formalism whose basic components (selectors) are the relational statements, that is, $f_i(t_1, \ldots, t_n) = v$, which are combined by applying different operators such as conjunction ($\wedge$) or disjunction ($\vee$) operator. In this work, we focus on the computation of flexible matching $fm(S, R)$ when both $S$ and $R$ are described by conjunctive formulas and $fm(S, R)$ looks for the substitution $\theta$ returning the best matching of $S$ against $R$, as:

$$fm(S, R) = \max_{\theta} \prod_{i=1,\ldots,k} fm_{\theta}(S, r_i). \tag{3}$$

The optimal $\theta$ that maximizes the above conditional probability is here searched by adopting the branch and bound algorithm that expands the least cost partial path by performing quickly on average [4]. According to this formulation, $fm_{\theta}$ denotes the flexible matching with the tie of the substitution fixed by $\theta$ computed on each single selector $r_i \equiv f_{r_i}(t_{r_1}, \ldots, t_{r_n}) = v_{r_i}$ of the referent $R$ where $f_{r_i}$ is a function descriptor with either numerical (e.g. area or distance) or categorical (e.g. intersect) range. In the former case the function value $v_{r_i}$ is an interval value ($v_{r_i} \equiv [a, b]$), while in the latter case $v_{r_i}$ is a subset of values ($v_{r_i} \equiv \{v_1, \ldots, v_M\}$) from the range of $f_{r_i}$. This faces with a referent $R$ that is obtained by generalizing a neighborhood of objects in $O$. Conversely for the subject $S$, that is, the description of a single object $o \in O$, the function value $w_{s_j}$ assigned to each selector $s_j \equiv f_{s_j}(t_{s_1}, \ldots, t_{s_n}) = w_{s_j}$ is an exactly known single value from the range of $f_{s_j}$. In this context, the flexible matching $fm_{\theta}(S, r_i)$ evaluates the degree of similarity $fm(s_j, \theta(r_i))$ between $\theta(r_i)$ and the corresponding selector $s_j$ in the subject $S$ such that both $r_i$ and $s_j$ have the same function descriptor $f_r = f_s$ and for each pair of terms $\langle t_{r_i}, t_{s_i} \rangle$, $\theta(t_{r_i}) = t_{s_i}$. More precisely,

$$fm(s_j, \theta(r_i)) = fm(w_{s_j}, v_{r_i}) = \max_{v \in v_{r_i}} P(equal(w_{s_j}, v)). \tag{4}$$

The probability of the event $equal(w_{s_j}, v)$ is then defined as the probability that an observed $w_{s_j}$ is a distortion of $v$ (i.e. $w_{s_j}$ is the observed value different from the real value $v$ because of the presence of noise), that is:

$$P(equal(w_{s_j}, v)) = P(\delta(X, v) \geq \delta(w_{s_j}, v)) \tag{5}$$

where $X$ is a random variable assuming value in the domain $D$ representing the range of $f_r$ while *delta* is a distance measure. The computation of $P(equal(w_{s_j}, v))$ clearly depends on the probability density function of $X$. For categorical descriptors, that is, $D$ is a discrete set with cardinality $\#D$, it has be proved [4] that:

$$P(equal(w_{s_j}, v)) = \begin{cases} 1 \text{ if } w_{s_j} = v \\ (\#D - 1)/\#D \text{ otherwise} \end{cases} \tag{6}$$

when $X$ is assumed to have a uniform probability distribution on $D$ and $\delta(x, y) = 0$ if $x = y$, 1 otherwise. Although similar results have been reported for both linear non numerical and tree-structured domains, no result appears for numerical

domains. Therefore, we have extended definitions reported in [4] to make flexible matching able to deal with numerical descriptors and we have proved that:

$$fm(c,[a,b]) = \begin{cases} 1 & \text{if } a \leq c \leq b \\ 1-2(a-c)/(\beta-\alpha) & \text{if } c < a \wedge 2a-c \leq \beta \\ (c-\alpha)/(\beta-\alpha) & \text{if } c < a \wedge 2a-c > \beta \\ (\beta-c)/(\beta-\alpha) & \text{if } c > b \wedge 2b-c < \alpha \\ 1-2(c-b)/(\beta-\alpha) & \text{if } c > b \wedge 2b-c \geq \alpha \end{cases} \quad (7)$$

by assuming that $X$ has uniform distribution on $D \equiv [\alpha, \beta]$ and $\delta(x,y) = |x-y|$. A proof of formula 7 is reported in the Appendix A of this paper.

### 2.3 Cluster labeling

A cluster $C$ can be naturally labeled with $T_C$ that is the set of first-order clauses obtained from the generalization of neighborhoods merged in $C$. Each first-order clause is in the form $C \leftarrow s_1, \ldots, s_n$, where $C$ represents the cluster label and each $s_i$ denotes a selector in the form $f_i(X_{i_1}, \ldots, X_{i_l}) \in v_i$ with $v_i$ an interval value or a set value. In this formalization, two selectors $s_1 : f_1(X_{1_1}, \ldots, X_{1_l}) \in v_1$ and $s_2 : f_2(X_{2_1}, \ldots, X_{2_l}) \in v_2$ are comparable according to some substitution $\theta$ when they involve the same descriptor ($f_1 = f_2 = f$) and each pair of terms $\langle X_{1_i}, X_{2_i} \rangle$ is unifiable according to $\theta$, i.e., $X_{1_i}\theta = X_{2_i}\theta = X_i$ ($\forall i = 1 \ldots l$). In this case, the selector $s : f(X_1, \ldots, X_l) \in v_1 \cup v_2$ is intended as a generalization for both $s_1$ and $s_2$. In particular, the selectors $s_1$ and $s_2$ are equal when they are comparable and $v_1 \equiv v_2 \equiv v$. In this case the generalization of $s_1$ and $s_2$ is built as $s : f(X_1, \ldots, X_l) \in v$. Similarly, the selector $s_1$ ($s_2$) is contained in the selector $s_2$ ($s_1$) when they are comparable and $v_1 \subseteq v_2$ ($v_2 \subseteq v_1$), while the generalization $s$ is $f(X_1, \ldots, X_l) \in v_2$ ($f(X_1, \ldots, X_l) \in v_1$). Note that equality of selectors implies containment, but not vice-versa. Similarly, the first-order clauses $H_1 : C \leftarrow s_{1_1}, \ldots, s_{1_n}$ and $H_2 : C \leftarrow s_{2_1}, \ldots, s_{2_n}$ are comparable according to some substitution $\theta$ when each pair of selectors $\langle s_{1_i}, s_{2_i} \rangle$ is comparable according to $\theta$. Hence, $H_1$ is equal (contained) to $H_2$ when $s_{1_i}$ is equal (contained) to $s_{2_i}$ for each $i = 1, \ldots, n$. In both these cases (equality and containment condition), the pair of clauses $H_1, H_2$ can be replaced without lost of information with the clause $H$ that is the generalization of $H_1, H_2$ built by substituting each pair of comparable selectors $\langle s_{1_i}, s_{2_i} \rangle \in \langle H_1, H_2 \rangle$ with the generalization obtained as stated above. This suggests the idea of merging a pair of comparable clauses $H_1, H_2$ in a single clause $H$ by preserving the equivalence of coverage, that is:

- if $H_1, H_2, BK \models o$ then $H, BK \models o$ and vice-versa,
- if $H_1, H_2, BK \not\models o$ then $H, BK \not\models o$ and vice-versa,

where $o$ is a structured object and $BK$ is a set of first-order clauses. The equivalence of coverage between $\{H_1, H_2\}$ and $H$ is obviously guaranteed when $H_1$ is either equal or contained in $H_2$ or vice-versa, but this equivalence cannot be guaranteed when $H_1$ and $H_2$ are comparable first-order clauses but neither equality condition nor containment condition are satisfied.

**Example 1**: Let us consider the pair of comparable first-order clauses:

$H_1 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [5..10], type(X_2) \in \{street\}$

$H_2 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [3..7], type(X_2) \in \{river\}$

where neither $H_1$ is equal to $H_2$ nor $H_1(H_2)$ is contained in $H_2(H_1)$. The first-order clause obtained by generalizing pairs of comparable selectors in both $H_1$ and $H_2$, is:

$H : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [3..10], type(X_2) \in \{street, river\}$

where $H \models o$ with $o : distance(X_1, X_2) = 3 \wedge type(X_2) = street$, but neither $H_1 \models o$ nor $H_2 \models o$.

The requirement of equality between $H_1$ and $H_2$ can be relaxed while preserving equivalence of coverage with respect to the generalization $H$. Indeed, when

$H_1 : C \leftarrow s_1(\_) \in v_1, \ldots s_k(\_) \in v_k, \ldots, s_n(\_) \in v_n$

$H_2 : C \leftarrow s_1(\_) \in v_1, \ldots s_k(\_) \in w_k, \ldots, s_n(\_) \in v_n$

are comparable first-order clauses differing only in the function value of a single selector (i.e. $s_k$), the first-order clause:

$H : C \leftarrow s_1(\_) \in v_1, \ldots s_k(\_) \in v_k \cup w_k, \ldots, s_n(\_) \in v_n$

continues to preserve the equivalence of coverage with $\{H_1, H_2\}$.

**Example 2**: Let us consider the pair of comparable first-order clauses:

$H_1 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [3..7], type(X_2) \in \{street\},$
$\quad length(X_2) \in [3, 5]$

$H_2 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [3..7], type(X_2) \in \{street\},$
$\quad length(X_2) \in [7, 10]$

which differ only in the value of a single selector (length), the first-order clause obtained by generalizing the pairs of comparable selectors in both $H_1$ and $H_2$ is $H : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [3..7], type(X_2) \in \{street\}$, $length(X_2) \in [3, 5] \cup [7, 10]$ that is equivalent in coverage to the pair $\{H_1, H_2\}$.

Following this idea, it is possible to compactly describe the cluster theory $T_C$ finally associated to a cluster $C$ by iteratively replacing pairs of comparable first-order clauses $H_1, H_2$ with the generalization $H$, when $H$ results equivalent in coverage to $\{H_1, H_2\}$ (see Algorithm 3).

**Example 3**: Let us consider $T_C$ that is the set of first-order clauses including:

$H_1 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [5..10], color(X_2) \in \{red\}$

$H_2 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [5..6], color(X_2) \in \{blue\}$

$H_3 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [5..10], color(X_2) \in \{blue\}$

$H_4 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [6..10], area(X_2) \in [30..40]$

$T_C$ can be transformed in the set of first-order clauses:

$H'_1 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [5..10], color(X_2) \in \{red, blue\}$

$H'_2 : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [6..10], area(X_2) \in [30..40]$

where $H'_1$ results by firstly merging $H_1$ and $H_3$, which are comparable and differ only in the function value of a selector ($color(X_2) \in \{red\}$ vs $color(X_2) \in \{blue\}$), and obtaining $H_{13} : cluster(X_1) = c \leftarrow distance(X_1, X_2) \in [5..10], color(X_2) \in \{red, blue\}$ and then merging $H_{13}$ and $H_2$ since $H_2$ is contained in $H_{13}$.

**Algorithm 3** Build a compact theory to describe a cluster $C$.

```
1: function clusterLabel(T_C) → T'_C;
2: T'_C ← ∅; merge ← false;
3: while T_C is not empty do
4:    H is a first-order clause in T_C; T_C = T_C/H;
5:    for each H' ∈ T_C do
6:       if H and H' are generalizable without lost of information then
7:          H = generalize(H,H'); T_C = T_C/H'; merge = true;
8:       end if
9:    end for
10:   T'_C = T'_C ∪ H;
11: end while
12: if merge is true then
13:   T'_C ←clusterLabel(T'_C);
14: end if
15: return T'_C;
```

## 3 The Application: two case studies

In this section, we describe the application of CORSO to two distinct real-world problems, namely topographic map interpretation and geo-referenced census data analysis. In the former problem, a topographic map is treated as a grid of square cells of same size, according to a hybrid tessellation-topological model such that adjacency among cells allows map-reading from a cell to one of its neighbors in the map. For each cell, geographical data is represented by means of a geometric (or physical) representation, describing the physical entities (point, line or region) corresponding to the geographical objects, and a thematic (or logical) representation, expressing the corresponding semantics (e.g., hydrography, vegetation and so on). Spatial clustering in this case aims at identifying a mosaic of nearly homogeneous clusters (areas) including adjacent cells such that geographical data inside each cluster properly models the spatial continuity of some morphological environment, while separate clusters model spatial variation over the entire space. In the second problem, the goal is to perform a joint analysis of both socio-economic factors represented in census data and geographical factors represented in topographic maps. The discovery of homogeneous areal clusters on spatially distributed socio-economic phenomena (e.g. social and economical deprivation) can be a valuable support to good public policy. In this case, spatial objects are territorial units for which census data are collected as well as entities of geographical layers such as urban and wood areas. In both applications, running time of CORSO refers to execution performed on a 2 Ghz IBM notebook with Windows XP and 256 Mb of RAM.

### 3.1 Topographic map interpretation

In this study we discuss an application of spatial clustering to characterize spatial continuity of some morphological elements over the topographic map of Canosa

(Apulia, Italy). The examined area covers 45 $km^2$ and is segmented into square areal units of 1 $Km^2$ each. Thus, the problem of recognizing spatial continuity of morphological elements in the map is formulated as the problem of grouping adjacent cells resulting in a morphologically homogeneous area. The discrete spatial structure is then imposed by the relation of "adjacency" among cells.

Since several geographical objects (e.g., almond tree, olive tree, fountain, etc) are collected within each cell, we apply algorithms derived from geometrical and topological reasoning [9] to obtain cell descriptions in first-order formalism. We consider descriptions including spatial descriptors encompassing geometrical properties (e.g., *area* and *extension*) and topological relations (e.g., *regionToRegion*, *lineToLine*, *pointToRegion*) as well as non spatial descriptors (e.g., *typeOf* and *subtypeOf*). The descriptor *partOf* is used to define the physical structure of a logical object. An example is: $typeOf(f_1) = fountain \wedge partOf(f_1, x_1) = true$, where $f_1$ denotes a fountain which is physically represented by a point referred with the constant $x_1$. Each cell is here described by a conjunction of 946.866 ground selectors in average. To support some qualitative reasoning, a BK is expressed in form of clauses. An example of BK is:
$fountainToCulture(Font, Culture) = Relation \leftarrow typeOf(Font) = fountain$,
$partOf(Font, Point) = true, typeOf(Culture) = culture$,
$partOf(Culture, Region) = true, pointToRegion(Point, Region) = Relation$
that allows to move from a physical to a logical level in describing the topological relation between the point that physically represents the fountain and the region that physically represents the culture and that are, respectively, referred to as the variables $Font$ and $Culture$. The goal is to model the spatial continuity of some morphological environment (e.g. cultivation setting) within adjacent cells over the map. It is noteworthy that granularity of partitioning changes by varying homogeneity threshold (see Figure 1). In particular, when $h-threshold = 0.95$, CORSO clusters adjacent cells in five regions in 1821 secs. Each cluster is compactly labeled as follows:

$C_1 : cluster(X_1) = c_1 \leftarrow containAlmondTree(X_1, X_2) \in \{true\}$,
$cultivationToCulture(X_2, X_3) \in \{outside\}$,
$areaCulture(X_3) \in [328..420112], fountainToCulture(X_4, X_3) \in \{outside\}$.

$C_2 : cluster(X_1) = c_2 \leftarrow containAlmondTree(X_1, X_2) \in \{true\}$,
$cultivationToCulture(X_2, X_3) \in \{inside\}, areaCulture(X_3) \in [13550..$
$187525], cultivationToCulture(X_2, X_4) \in \{outside\}$.

$C_3 : cluster(X_1) = c_3 \leftarrow containGrapevine(X_1, X_2) \in \{true\}$,
$cultivationToCulture(X_2, X_3) \in \{inside\}, areaCulture(X_3) \in [13550..$
$212675], cultivationToCulture(X_2, X_4) \in \{outside\}$.
$cluster(X_1) = c_3 \leftarrow containGrapevine(X_1, X_2) \in \{true\}$,
$cultivationToCulture(X_2, X_3) \in \{outside\}, areaCulture(X_3) \in [150..$
$212675], cultivationToCulture(X_2, X_4) \in \{outside, inside\}$.

$C_4 : cluster(X_1) = c_4 \leftarrow containStreet(X_1, X_2) \in \{true\}$
$streetToCulture(X_2, X_3) \in \{adjacent\}, areaCulture(X_3) \in [620..$
$230326], cultureToCulture(X_3, X_4) \in \{outside, inside\}$.

$C_5 : cluster(X_1) = c_5 \leftarrow containOliveTree(X_1, X_2) \in true$,

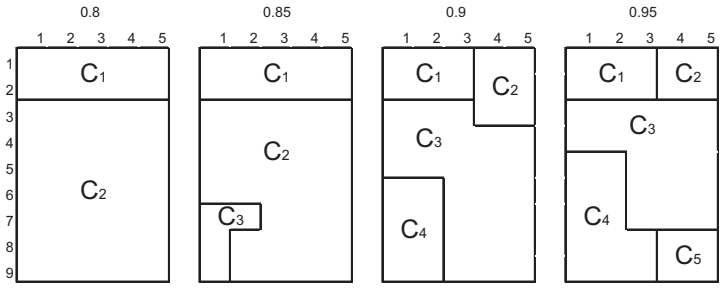



**Fig. 1.** Spatial clusters detected on map data from the zone of Canosa by varying $h-threshold$ value in {0.8,0.85,0.9,0.95}.

$coltivationToCulture(X_2, X_3) \in \{outside\}, areaCulture(X_3) \in [620..144787], cultivationToCulture(X_2, X_4) \in \{outside\}$.

Notice that each detected cluster effectively includes adjacent cells sharing a similar morphological environment, while separate clusters describe quite different environments.

### 3.2 Geo-referenced census data analysis

In this application, we consider both census and digital map data concerning North West England (NWE) area that is decomposed into censual sections or wards for a total of 1011 wards. Census data is available at ward level and provides some measures of deprivation level in the ward according to index scores that combine information provided by 1998 Census. These scores are used in the evaluation of the need for primary care, in health-related analysis and in targeting urban regeneration funds. The higher the index value the more deprived a ward is. We focus attention on investigating continuity of socio-economic deprivation joined to geographical factors represented in linked topographic maps.

Both ward-referenced census data and map data are stored in Oracle Spatial 9i database as a set of spatial tables, one for each layer. Each spatial table includes a geometry attribute that allows storing the geometrical representation and the positioning of a spatial object with respect to some reference system. We adopt a topological algorithm based on the 9-intersection model [3] to detect both adjacency relation between NWE wards (i.e. wards which share some boundary) and overlapping relation between wards and urban areas (or woods). The former imposes a discrete spatial structure over NWE wards such that only adjacent wards may be grouped in the same cluster while the latter contributes to define the spatial structure embedded in each ward not only in terms of observed values of deprivation scores but also extension of urban areas and/or woods overlapping each ward. No $BK$ is defined for this problem.

Granularity of partitioning changes when varying the value of $h-threshold$, that is, CORSO detects 79 clusters with $h-threshold = 0.80$, 89 clusters with $h-threshold = 0.85$, 122 clusters with $h-threshold = 0.90$ and 163 clusters

with $h - threshold = 0.95$. In particular, when $h - threshold = 0.95$, CORSO clusters NWE area in 2160 secs and identifies adjacent regions modeling differently relational patterns involving deprivation and geographical environment. For instance, by analyzing these spatial clusters, we discover three adjacent areas, namely $C_1$, $C_2$ and $C_3$ compactly labeled as follows:

$C_1 : cluster(X_1) = c_1 \leftarrow townsend(X_1) \in [-4.7..-0.6],$
$doe(X_1) \in [-12.4..2.7], carstairs(X_1) \in [-4.5..-0.9],$
$jarman(X_1) \in [-32.7..7.5], overlapped\_by\_wood(X1, X2) \in \{true\}.$
$cluster(X_1) = c_1 \leftarrow townsend(X_1) \in [-5.4..-2.3],$
$doe(X_1) \in [-10.9..-0.5], carstairs(X_1) \in [-4.2..-1.6],$
$jarman(X_1) \in [-22.8..0.6], overlapped\_by\_wood(X1, X2) \in \{true\}.$
$cluster(X_1) = c_1 \leftarrow townsend(X_1) \in [-5.4..-3.2],$
$doe(X_1) \in [-8.8..-2.1], carstairs(X_1) \in [-4.4..-2.5],$
$jarman(X_1) \in [-22.8..-2.4], overlapped\_by\_wood(X1, X2) \in \{true\}.$

$C_2 : cluster(X_1) = c_1 \leftarrow townsend(X_1) \in [-2.0..0.6],$
$doe(X_1) \in [-4.2..1.6], carstairs(X_1) \in [-2.6..2.1],$
$jarman(X_1) \in [-9.7..8.8], overlapped\_by\_largeUrbArea(X1, X2) \in \{true\}.$
$cluster(X_1) = c_1 \leftarrow townsend(X_1) \in [-2.7..2.8],$
$doe(X_1) \in [-4.2..4.0], carstairs(X_1) \in [-2.2..2.7],$
$jarman(X_1) \in [-8.8..21.3], overlapped\_by\_largeUrbArea(X1, X2) \in \{true\}$

$C_3 : cluster(X_1) = c_1 \leftarrow townsend(X_1) \in [-3.4..0.4],$
$doe(X_1) \in [-8.2..-0.2], carstairs(X_1) \in [-3.7..0.6],$
$jarman(X_1) \in [-27.7..-1.5],$
$overlapped\_by\_smallUrbArea(X1, X2) \in \{true\}.$

$C_1$, $C_2$ and $C_3$ cover adjacent areas with quite similar range value for deprivation indexes but $C_1$ models the presence of woods while $C_2$ and $C_3$ model the presence of small urban areas and large urban areas, respectively. Discontinuity of geographical environments modeled by these clusters is confirmed by visualizing map data about the area (see Figure 2).

## 4 Conclusions

This paper presents a novel approach to discover clusters from structured spatial data taking into account relational constraints (e.g. spatial correlation) forming the discrete spatial structure representable as a graph. In this way, the concept of graph neighborhood is exploited to capture relational constraints embedded in the graph edges. Moreover, we resort to a relational approach to mine data scattered in multiple relations describing the structure that is naturally embedded in spatial data. As a consequence, only spatial units associated with (transitively) graph connected nodes can be clustered together according to judgment of similarity on relational descriptions representing their internal (spatial) structure. As future work, we intend to investigate the possibility of assigning the same unit to multiple clusters. In addition, we plan to employ CORSO in analyzing geo-referenced data of air pollution in order to support a good policy of environmental planning.

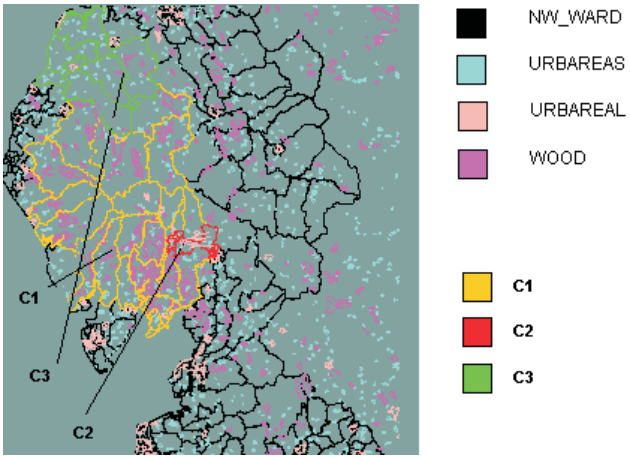



**Fig. 2.** Spatial clusters detected on NWE with $h - threshold = 0.95$.

## 5 Acknowledgment

The work presented in this paper is partial fulfillment of the research objective set by the ATENEO-2004 project on "Metodi di Data Mining Multi-relazionale per la scoperta di conoscenza in basi di dati".

## References

1. H. Blockeel. *Top-down induction of first order logical decision trees*. PhD thesis, Department of Computer Science, Katholieke Universiteit, Leuven, Belgium, 1998.
2. S. Džeroski and N. Lavrač. *Relational Data Mining*. Springer-Verlag, 2001.
3. M. Egenhofer. Reasoning about binary topological relations. In *Symposium on Large Spatial Databases*, pages 143–160, 1991.
4. F. Esposito, D. Malerba, and G. Semeraro. Flexible matching for noisy structural descriptions. In *International Joint Conference on Artificial Intelligence*, pages 658–664, 1991.
5. M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Knowledge Discovery in Databases*, pages 226–231, 1996.
6. L. Kaufmann and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.
7. M. Kirsten and S. Wrobel. Relational distance-based clustering. In *Inductive Logic Programming, 8th International Conference*, volume 1446, pages 261–270. Springer-Verlag, 1998.
8. D. Malerba. Learning recursive theories in the normal ilp setting. *Fundamenta Informaticae*, 57(1):39–77, 2003.
9. D. Malerba, F. Esposito, A. Lanza, F. A. Lisi, and A. Appice. Empowering a gis with inductive learning capabilities: The case of ingens. *Journal of Computers, Environment and Urban Systems, Elsevier Science*, 27:265–281, 2003.

10. D. Mavroeidis and P. Flach. Improved distances for structured data. In T. Horváth and A. Yamamoto, editors, *Inductive Logic Programming, 13th International Conference*, volume 2835, pages 251–268. Springer-Verlag, 2003.
11. R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Very Large Data Bases, 20th International Conference*, pages 144–155. Morgan Kaufmann Publishers, 1994.
12. D. Patterson. *Introduction to Artificial Intelligence and expert systems.* Prentice-Hall, 1991.
13. L. D. Raedt and S. Dzeroski. First-order jk-clausal theories are pac-learnable. *Artificical Intelligence*, 70(1-2):375–392, 1994.
14. J. Sander, E. Martin, H.-P. Kriegel, and X. Xu. Density-based clustering in spatial databases: The algorithm gdbscan and its applications. *Data Mining and Knowledge Discovery*, 2(2):169–194, 1998.
15. G. Toussaint. Some unsolved problems on proximity graphs. In D. Dearholt and F. Harary, editors, *First Workshop on Proximity Graphs*, 1991.
16. X. Wang and H. J. Hamilton. Dbrs: A density-based spatial clustering method with random sampling. In *PAKDD*, pages 563–575, 2003.

# A Appendix

Let us recall definitions (4) and (5) and apply them to numerical case. We have:

$$fm(c,[a,b]) = \max_{v\in[a,b]} P(equal(c,v)) = \max_{v\in[a,b]} P(\delta(X,v) \geq \delta(c,v))$$

By assuming that $X$ has a uniform distribution on domain $D = [\alpha,\beta]$ with density function $f_D(x) = 1/(\beta-\alpha), \forall x \in D$ and fixing $\delta(x,y) = |x-y|$, $P(\delta(X,v) \geq \delta(c,v))$ can be rewritten as $P(|X-v| \geq |c-v|)$ that is maximized when minimizing $|c-v|$.
If $a \leq c \leq b$ then $\max_{v\in[a,b]} P(|X-v| \geq |c-v|) = P(|X-v| \geq |c-c|) = 1$.
If $c < a$ then $\max_{v\in[a,b]} P(|X-v| \geq |c-v|)$ is written as $\max_{v\in[a,b]} P(|X-v| \geq v-c)$. Since the maximum of $P(|X-v| \geq v-c)$ is obtained for $v = a$, we have that $\max_{v\in[a,b]} P(|X-v| \geq v-c) = P(|X-a| \geq a-c) = P(X-a \geq a-c) + P(X-a \leq c-a) = P(X \geq 2a-c) + P(X \leq c)$ where:

1. $P(X \geq 2a-c) = \int_{\beta}^{2a-c} 1/(\beta-\alpha)dx = (\beta-2a+c)/(\beta-\alpha)$ if $2a-c \leq \beta$, 0 otherwise;
2. $P(X \leq c) = (c-\alpha)/(\beta-\alpha)$.

Hence, we obtain that:

$$\max_{v\in[a,b]} P(|X-v| \geq v-c) = \begin{cases} 1-2(a-c)/(\beta-\alpha) & \text{if } c < a \wedge 2a-c \leq \beta \\ (c-\alpha)/(\beta-\alpha) & \text{if } c < a \wedge 2a-c > \beta \end{cases}$$

If $c > b$ then $\max_{v\in[a,b]} P(|X-v| \geq |c-v|)$ can be equivalently written as $\max_{v\in[a,b]} P(|X-v| \geq c-v)$ that is obtained for $v = b$. Therefore, $\max_{v\in[a,b]} P(|X-v| \geq c-v) = P(|X-b| \geq c-b) = P(X-b \geq c-b) + P(X-b \leq b-c) = P(X \geq c) + P(X \leq 2b-c)$. We have that:

$$\max_{v\in[a,b]} P(|X-v| \geq c-v) = \begin{cases} (\beta-c)/(\beta-\alpha) & \text{if } c > b \wedge 2b-c < \alpha \\ 1-2(c-b)/(\beta-\alpha) & \text{if } c > b \wedge 2b-c \geq \alpha \end{cases}$$

# On Learning in $\mathcal{AL}$-log

Francesca A. Lisi and Floriana Esposito

Dipartimento di Informatica, Università degli Studi di Bari, Italy
lisi@di.uniba.it

**Abstract.** In this paper we provide a general framework for learning in $\mathcal{AL}$-log, a hybrid language that integrates the description logic $\mathcal{ALC}$ and the function-free Horn clausal language DATALOG. In this framework inductive hypotheses are represented as constrained DATALOG clauses, organized according to the $\mathcal{B}$-subsumption relation, and evaluated against observations by applying coverage relations that depend on the representation chosen for the observations. The framework is valid whatever the scope of induction (description vs. prediction) is. Yet we concentrate on an instantiation of the framework which corresponds to the logical setting of *characteristic induction from intepretations* and is particularly suitable for descriptive data mining tasks such as frequent pattern discovery (and its variants).

## 1 Introduction

*Hybrid systems* are a special class of knowledge representation and reasoning (KR&R) systems which are constituted by two or more subsystems dealing with distinct portions of a knowledge base and specific reasoning procedures [11]. The characterizing feature of hybrid systems is that the whole system is in charge of a single knowledge base, thus combining knowledge and reasoning services of the different subsystems in order to answer user questions. Indeed the motivation for building hybrid systems is to improve on two basic features of knowledge representation formalisms, namely *representational adequacy* and *deductive power*. Hybrid systems such as CARIN [13] and $\mathcal{AL}$-log [8] are particularly interesting because they bridge the gap between two fragments of first-order logic, description logics (DLs) [1] and Horn clausal logic, that are incomparable as for the expressive power [4]. In particular, $\mathcal{AL}$-log integrates $\mathcal{ALC}$ [23] and DATALOG [6] by using $\mathcal{ALC}$ concept assertions essentially as type constraints on variables.

In this paper we provide a *general framework for learning in $\mathcal{AL}$-log*. Inductive hypotheses are represented as constrained DATALOG clauses, organized according to the $\mathcal{B}$-subsumption relation, and evaluated against observations by applying coverage relations that depend on the representation chosen for the observations. In defining this learning framework we resort to the methodological apparatus of Inductive Logic Programming (ILP). ILP has been historically concerned with concept learning from examples and background knowledge within the representation framework of Horn clausal logic and with the aim of prediction [19]. More recently ILP has moved towards either different first-order logic fragments (e.g., DLs) or new learning goals (e.g., description).

The framework proposed is valid whatever the scope of induction (description vs. prediction) is. Yet we concentrate on an instantiation of the framework which corresponds to the logical setting of *characteristic induction from intepretations* and is particularly suitable for descriptive data mining tasks such as frequent pattern discovery (and its variants) [7].

The paper is organized as follows. Section 2 introduces the basic notions of $\mathcal{AL}$-log. Section 3 defines the framework for learning in $\mathcal{AL}$-log. Section 4 illustrates the instantiation of the framework in the case of characteristic induction from intepretations. Section 5 concludes the paper with final remarks.

## 2 Basics of $\mathcal{AL}$-log

The system $\mathcal{AL}$-log [8] integrates two KR&R systems: Structural and relational.

**Table 1.** Syntax and semantics of $\mathcal{ALC}$.

| | | |
|---|---|---|
| bottom (resp. top) concept | $\bot$ (resp. $\top$) | $\emptyset$ (resp. $\Delta^{\mathcal{I}}$) |
| atomic concept | $A$ | $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ |
| role | $R$ | $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ |
| individual | $a$ | $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ |
| concept negation | $\neg C$ | $\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$ |
| concept conjunction | $C \sqcap D$ | $C^{\mathcal{I}} \cap D^{\mathcal{I}}$ |
| concept disjunction | $C \sqcup D$ | $C^{\mathcal{I}} \cup D^{\mathcal{I}}$ |
| value restriction | $\forall R.C$ | $\{x \in \Delta^{\mathcal{I}} \mid \forall y\ (x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$ |
| existential restriction | $\exists R.C$ | $\{x \in \Delta^{\mathcal{I}} \mid \exists y\ (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$ |
| equivalence axiom | $C \equiv D$ | $C^{\mathcal{I}} = D^{\mathcal{I}}$ |
| subsumption axiom | $C \sqsubseteq D$ | $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ |
| concept assertion | $a : C$ | $a^{\mathcal{I}} \in C^{\mathcal{I}}$ |
| role assertion | $\langle a, b\rangle : R$ | $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ |

### 2.1 The structural subsystem

The structural part $\Sigma$ is based on $\mathcal{ALC}$ [23] and allows for the specification of knowledge in terms of classes (*concepts*), binary relations between classes (*roles*), and instances (*individuals*). Complex concepts can be defined from atomic concepts and roles by means of constructors (see Table 1). Also $\Sigma$ can state both is-a relations between concepts (*axioms*) and instance-of relations between individuals (resp. couples of individuals) and concepts (resp. roles) (*assertions*). An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for $\Sigma$ consists of a domain $\Delta^{\mathcal{I}}$ and a mapping function $\cdot^{\mathcal{I}}$. In particular, individuals are mapped to elements of $\Delta^{\mathcal{I}}$ such that $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ if $a \neq b$ (*Unique Names Assumption* (UNA) [20]). If $\mathcal{O} \subseteq \Delta^{\mathcal{I}}$ and

$\forall a \in \mathcal{O} : a^{\mathcal{I}} = a$, $\mathcal{I}$ is called *$\mathcal{O}$-interpretation*. Also $\Sigma$ represents many different interpretations, i.e. all its models (*Open World Assumption* (OWA) [1]).

The main reasoning task for $\Sigma$ is the *consistency check*. This test is performed with a *tableau calculus* that starts with the tableau branch $S = \Sigma$ and adds assertions to $S$ by means of *propagation rules* such as

- $S \rightarrow_{\sqcup} S \cup \{s : D\}$ if
  1. $s : C_1 \sqcup C_2$ is in $S$,
  2. $D = C_1$ and $D = C_2$,
  3. neither $s : C_1$ nor $s : C_2$ is in $S$
- $S \rightarrow_{\forall} S \cup \{t : C\}$ if
  1. $s : \forall R.C$ is in $S$,
  2. $sRt$ is in $S$,
  3. $t : C$ is not in $S$
- $S \rightarrow_{\sqsubseteq} S \cup \{s : C' \sqcup D\}$ if
  1. $C \sqsubseteq D$ is in $S$,
  2. $s$ appears in $S$,
  3. $C'$ is the NNF concept equivalent to $\neg C$
  4. $s : \neg C \sqcup D$ is not in $S$
- $S \rightarrow_{\bot} \{s : \bot\}$ if
  1. $s : A$ and $s : \neg A$ are in $S$, or
  2. $s : \neg\top$ is in $S$,
  3. $s : \bot$ is not in $S$

until either a contradiction is generated or an interpretation satisfying $S$ can be easily obtained from it.

### 2.2 The relational subsystem

The relational part of $\mathcal{AL}$-log allows one to define DATALOG[1] programs enriched with *constraints* of the form $s : C$ where $s$ is either a constant or a variable, and $C$ is an $\mathcal{ALC}$-concept. Note that the usage of concepts as typing constraints applies only to variables and constants that already appear in the clause. The symbol & separates constraints from DATALOG atoms in a clause.

**Definition 1.** *A* constrained DATALOG clause *is an implication of the form* $\alpha_0 \leftarrow \alpha_1, \ldots, \alpha_m \& \gamma_1, \ldots, \gamma_n$ *where* $m \geq 0$, $n \geq 0$, $\alpha_i$ *are* DATALOG *atoms and* $\gamma_j$ *are constraints. A* constrained DATALOG program $\Pi$ *is a set of constrained* DATALOG *clauses.*

An *$\mathcal{AL}$-log knowledge base* $\mathcal{B}$ is the pair $\langle \Sigma, \Pi \rangle$ where $\Sigma$ is an $\mathcal{ALC}$ knowledge base and $\Pi$ is a constrained DATALOG program. For a knowledge base to be acceptable, it must satisfy the following conditions:

- The set of DATALOG predicate symbols appearing in $\Pi$ is disjoint from the set of concept and role symbols appearing in $\Sigma$.

[1] For the sake of brevity we assume the reader to be familiar with DATALOG.

– The alphabet of constants in $\Pi$ coincides with the alphabet $\mathcal{O}$ of the individuals in $\Sigma$. Furthermore, every constant in $\Pi$ appears also in $\Sigma$.
– For each clause in $\Pi$, each variable occurring in the constraint part occurs also in the DATALOG part.

These properties state a *safe* interaction between the structural and the relational part of an $\mathcal{AL}$-log knowledge base, thus solving the semantic mismatch between the OWA of $\mathcal{ALC}$ and the CWA of DATALOG [21]. This interaction is also at the basis of a model-theoretic semantics for $\mathcal{AL}$-log. We call $\Pi_D$ the set of DATALOG clauses obtained from the clauses of $\Pi$ by deleting their constraints. We define an *interpretation* $\mathcal{J}$ for $\mathcal{B}$ as the union of an $\mathcal{O}$-interpretation $\mathcal{I}_\mathcal{O}$ for $\Sigma$ (i.e. an interpretation compliant with the unique names assumption) and an Herbrand interpretation $\mathcal{I}_\mathcal{H}$ for $\Pi_D$. An interpretation $\mathcal{J}$ is a *model* of $\mathcal{B}$ if $\mathcal{I}_\mathcal{O}$ is a model of $\Sigma$, and for each ground instance $\alpha'_0 \leftarrow \alpha'_1, \ldots, \alpha'_m \& \gamma'_1, \ldots, \gamma'_n$ of each clause $\alpha_0 \leftarrow \alpha_1, \ldots, \alpha_m \& \gamma'_1, \ldots, \gamma'_n$ in $\Pi$, either there exists one $\gamma'_i$, $i \in \{1, \ldots, n\}$, that is not satisfied by $\mathcal{J}$, or $\alpha'_0 \leftarrow \alpha'_1, \ldots, \alpha'_m$ is satisfied by $\mathcal{J}$. The notion of *logical consequence* paves the way to the definition of answer set for queries. *Queries* to $\mathcal{AL}$-log knowledge bases are special cases of Definition 1. An *answer* to the query $Q$ is a ground substitution $\sigma$ for the variables in $Q$. The answer $\sigma$ is *correct* w.r.t. a $\mathcal{AL}$-log knowledge base $\mathcal{B}$ if $Q\sigma$ is a logical consequence of $\mathcal{B}$ ($\mathcal{B} \models Q\sigma$). The *answer set* of $Q$ in $\mathcal{B}$ contains all the correct answers to $Q$ w.r.t. $\mathcal{B}$.

Reasoning for $\mathcal{AL}$-log knowledge bases is based on *constrained SLD-resolution* [8], i.e. an extension of SLD-resolution to deal with constraints. In particular, the constraints of the resolvent of a query $Q$ and a constrained DATALOG clause $E$ are recursively simplified by replacing couples of constraints $t : C$, $t : D$ with the equivalent constraint $t : C \sqcap D$. The one-to-one mapping between constrained SLD-derivations and the SLD-derivations obtained by ignoring the constraints is exploited to extend known results for DATALOG to $\mathcal{AL}$-log. Note that in $\mathcal{AL}$-log a derivation of the empty clause with associated constraints does not represent a refutation. It actually infers that the query is true in those models of $\mathcal{B}$ that satisfy its constraints. Therefore in order to answer a query it is necessary to collect enough derivations ending with a constrained empty clause such that every model of $\mathcal{B}$ satisfies the constraints associated with the final query of at least one derivation.

**Definition 2.** *Let $Q^{(0)}$ be a query $\leftarrow \beta_1, \ldots, \beta_m \& \gamma_1, \ldots, \gamma_n$ to a $\mathcal{AL}$-log knowledge base $\mathcal{B}$ . A* constrained SLD-refutation *for $Q^{(0)}$ in $\mathcal{B}$ is a finite set $\{d_1, \ldots, d_s\}$ of constrained SLD-derivations for $Q^{(0)}$ in $\mathcal{B}$ such that:*

1. *for each derivation $d_i$, $1 \leq i \leq s$, the last query $Q^{(n_i)}$ of $d_i$ is a constrained empty clause;*
2. *for every model $\mathcal{J}$ of $\mathcal{B}$, there exists at least one derivation $d_i$, $1 \leq i \leq s$, such that $\mathcal{J} \models Q^{(n_i)}$*

Constrained SLD-refutation is a complete and sound method for answering *ground* queries.

**Lemma 1.** *[8] Let $Q$ be a ground query to an $\mathcal{AL}$-log knowledge base $\mathcal{B}$. It holds that $\mathcal{B} \vdash Q$ if and only if $\mathcal{B} \models Q$.*

An answer $\sigma$ to a query $Q$ is a *computed answer* if there exists a constrained SLD-refutation for $Q\sigma$ in $\mathcal{B}$ ($\mathcal{B} \vdash Q\sigma$). The set of computed answers is called the *success set* of $Q$ in $\mathcal{B}$. Furthermore, given *any* query $Q$, the success set of $Q$ in $\mathcal{B}$ coincides with the answer set of $Q$ in $\mathcal{B}$. This provides an operational means for computing correct answers to queries. Indeed, it is straightforward to see that the usual reasoning methods for DATALOG allow us to collect in a finite number of steps enough constrained SLD-derivations for $Q$ in $\mathcal{B}$ to construct a refutation - if any. Derivations must satisfy both conditions of Definition 2. In particular, the latter requires some reasoning on the structural component of $\mathcal{B}$. This is done by applying the tableau calculus as shown in the following example.

Constrained SLD-resolution is *decidable*. Furthermore, because of the safe interaction between $\mathcal{ALC}$ and DATALOG, it supports a form of *closed world reasoning*, i.e. it allows one to pose queries under the assumption that part of the knowledge base is complete.

## 3 The general framework for learning in $\mathcal{AL}$-log

In our framework for learning in $\mathcal{AL}$-log we represent inductive hypotheses as constrained DATALOG clauses and data as an $\mathcal{AL}$-log knowledge base $\mathcal{B}$. In particular $\mathcal{B}$ is composed of a *background knowledge* $\mathcal{K}$ and a set $O$ of *observations*. We assume $\mathcal{K} \cap O = \emptyset$.

To define the framework we resort to the methodological apparatus of ILP which requires the following ingredients to be chosen:

- the *language $\mathcal{L}$ of hypotheses*
- a *generality order* $\succeq$ for $\mathcal{L}$ to structure the space of hypotheses
- a *relation* to test the *coverage* of hypotheses in $\mathcal{L}$ against observations in $O$ w.r.t. $\mathcal{K}$

The framework is **general**, meaning that it is valid whatever the scope of induction (description/prediction) is. Therefore the DATALOG literal $q(\boldsymbol{X})$[2] in the head of hypotheses represents a concept to be either discriminated from others (*discriminant induction*) or characterized (*characteristic induction*).

### 3.1 The language of hypotheses

To be suitable as language of hypotheses, constrained DATALOG clauses must satisfy the following restrictions.

First, we impose constrained DATALOG clauses to be linked and connected (or range-restricted) as usual in ILP.

[2] $\boldsymbol{X}$ is a tuple of variables

**Definition 3.** *Let $H$ be a constrained* DATALOG *clause. A term $t$ in some literal $l_i \in H$ is* linked *with linking-chain of length 0, if $t$ occurs in $head(H)$, and is linked with linking-chain of length $d+1$, if some other term in $l_i$ is linked with linking-chain of length $d$. The link-depth of a term $t$ in some $l_i \in H$ is the length of the shortest linking-chain of $t$. A literal $l_i \in H$ is linked if at least one of its terms is linked. The clause $H$ itself is linked if each $l_i \in H$ is linked. The clause $H$ is* connected *if each variable occurring in $head(H)$ also occur in $body(H)$.*

Second, we impose constrained DATALOG clauses to be compliant with the bias of Object Identity (OI) [24]. This bias can be considered as an extension of the unique names assumption from the semantic level to the syntactic one of $\mathcal{AL}$-log. We would like to remind the reader that this assumption holds in $\mathcal{ALC}$. Also it holds naturally for ground constrained DATALOG clauses because the semantics of $\mathcal{AL}$-log adopts Herbrand models for the DATALOG part and $\mathcal{O}$-models for the constraint part. Conversely it is not guaranteed in the case of non-ground constrained DATALOG clauses, e.g. different variables can be unified. The OI bias can be the starting point for the definition of either an equational theory or a quasi-order for constrained DATALOG clauses. The latter option relies on a restricted form of substitution whose bindings avoid the identification of terms.

**Definition 4.** *A substitution $\sigma$ is an* OI-substitution *w.r.t. a set of terms $T$ iff $\forall t_1, t_2 \in T\colon\ t_1 \neq t_2$ yields that $t_1\sigma \neq t_2\sigma$.*

From now on, we assume that substitutions are OI-compliant.

### 3.2 The generality relation

The definition of a generality relation for constrained DATALOG clauses can disregard neither the peculiarities of $\mathcal{AL}$-log nor the methodological apparatus of ILP. Therefore we rely on the reasoning mechanisms made available by $\mathcal{AL}$-log knowledge bases and propose to adapt Buntine's generalized subsumption [5] to our framework as follows.

**Definition 5.** *Let $H$ be a constrained* DATALOG *clause, $\alpha$ a ground* DATALOG *atom, and $\mathcal{J}$ an interpretation. We say that $H$* covers *$\alpha$ under $\mathcal{J}$ if there is a ground substitution $\theta$ for $H$ ($H\theta$ is ground) such that $body(H)\theta$ is true under $\mathcal{J}$ and $head(H)\theta = \alpha$.*

**Definition 6.** *Let $H_1$, $H_2$ be two constrained* DATALOG *clauses and $\mathcal{B}$ an $\mathcal{AL}$-log knowledge base. We say that $H_1$* $\mathcal{B}$-subsumes *$H_2$ if for every model $\mathcal{J}$ of $\mathcal{B}$ and every ground atom $\alpha$ such that $H_2$ covers $\alpha$ under $\mathcal{J}$, we have that $H_1$ covers $\alpha$ under $\mathcal{J}$.*

We can define a generality relation $\succeq_{\mathcal{B}}$ for constrained DATALOG clauses on the basis of $\mathcal{B}$-subsumption. It can be easily proven that $\succeq_{\mathcal{B}}$ is a quasi-order (i.e. it is a reflexive and transitive relation) for constrained DATALOG clauses.

**Definition 7.** *Let $H_1$, $H_2$ be two constrained* DATALOG *clauses and $\mathcal{B}$ an $\mathcal{AL}$-log knowledge base. We say that $H_1$ is* at least as general as *$H_2$ under $\mathcal{B}$-subsumption, $H_1 \succeq_{\mathcal{B}} H_2$, iff $H_1$ $\mathcal{B}$-subsumes $H_2$. Furthermore, $H_1$ is* more general than *$H_2$ under $\mathcal{B}$-subsumption, $H_1 \succ_{\mathcal{B}} H_2$, iff $H_1 \succeq_{\mathcal{B}} H_2$ and $H_2 \not\succeq_{\mathcal{B}} H_1$. Finally, $H_1$ is* equivalent *to $H_2$ under $\mathcal{B}$-subsumption, $H_1 \sim_{\mathcal{B}} H_2$, iff $H_1 \succeq_{\mathcal{B}} H_2$ and $H_2 \succeq_{\mathcal{B}} H_1$.*

The next lemma shows the definition of $\mathcal{B}$-subsumption to be equivalent to another formulation, which will be more convenient in later proofs than the definition based on covering.

**Definition 8.** *Let $\mathcal{B}$ be an $\mathcal{AL}$-log knowledge base and $H$ be a constrained* DATALOG *clause. Let $X_1, \ldots, X_n$ be all the variables appearing in $H$, and $a_1, \ldots, a_n$ be distinct constants (individuals) not appearing in $\mathcal{B}$ or $H$. Then the substitution $\{X_1/a_1, \ldots, X_n/a_n\}$ is called a* Skolem substitution *for $H$ w.r.t. $\mathcal{B}$.*

**Lemma 2.** *[15] Let $H_1$, $H_2$ be two constrained* DATALOG *clauses, $\mathcal{B}$ an $\mathcal{AL}$-log knowledge base, and $\sigma$ a Skolem substitution for $H_2$ with respect to $\{H_1\} \cup \mathcal{B}$. We say that $H_1 \succeq_{\mathcal{B}} H_2$ iff there exists a ground substitution $\theta$ for $H_1$ such that (i) $head(H_1)\theta = head(H_2)\sigma$ and (ii) $\mathcal{B} \cup body(H_2)\sigma \models body(H_1)\theta$.*

The relation between $\mathcal{B}$-subsumption and constrained SLD-resolution is given below. It provides an operational means for checking $\mathcal{B}$-subsumption.

**Theorem 1** *Let $H_1$, $H_2$ be two constrained* DATALOG *clauses, $\mathcal{B}$ an $\mathcal{AL}$-log knowledge base, and $\sigma$ a Skolem substitution for $H_2$ with respect to $\{H_1\} \cup \mathcal{B}$. We say that $H_1 \succeq_{\mathcal{B}} H_2$ iff there exists a substitution $\theta$ for $H_1$ such that (i) $head(H_1)\theta = head(H_2)$ and (ii) $\mathcal{B} \cup body(H_2)\sigma \vdash body(H_1)\theta\sigma$ where $body(H_1)\theta\sigma$ is ground.*

*Proof. By Lemma 2, we have $H_1 \succeq_{\mathcal{B}} H_2$ iff there exists a ground substitution $\theta'$ for $H_1$, such that $head(H_1)\theta' = head(H_2)\sigma$ and $\mathcal{B} \cup body(H_2)\sigma \models body(H_1)\theta'$. Since $\sigma$ is a Skolem substitution, we can define a substitution $\theta$ such that $H_1\theta\sigma = H_1\theta'$ and none of the Skolem constants of $\sigma$ occurs in $\theta$. Then $head(H_1)\theta = head(H_2)$ and $\mathcal{B} \cup body(H_2)\sigma \models body(H_1)\theta\sigma$. Since $body(H_1)\theta\sigma$ is ground, by Lemma 1 we have $\mathcal{B} \cup body(H_2)\sigma \vdash body(H_1)\theta\sigma$, so the thesis follows.*

The decidability of $\mathcal{B}$-subsumption follows from the decidability of both generalized subsumption in DATALOG [5] and query answering in $\mathcal{AL}$-log [8].

### 3.3 Coverage relations

When defining coverage relations we make assumptions as regards the representation of observations because it impacts the definition of coverage.

In the logical setting of *learning from entailment* extended to $\mathcal{AL}$-log, an observation $o_i \in O$ is represented as a ground constrained DATALOG clause having a ground atom $q(\boldsymbol{a}_i)$[3] in the head.

[3] $\boldsymbol{a}_i$ is a tuple of constants

**Definition 9.** *Let $H \in \mathcal{L}$ be a hypothesis, $\mathcal{K}$ a background knowledge and $o_i \in O$ an observation. We say that $H$* covers *$o_i$* under entailment *w.r.t $\mathcal{K}$ iff $\mathcal{K} \cup H \models o_i$.*

**Theorem 2** *[14] Let $H \in \mathcal{L}$ be a hypothesis, $\mathcal{K}$ a background knowledge, and $o_i \in O$ an observation. We say that $H$* covers *$o_i$* under entailment w.r.t. *$\mathcal{K}$ iff $\mathcal{K} \cup body(o_i) \cup H \vdash q(\boldsymbol{a}_i)$.*

In the logical setting of *learning from interpretations* extended to $\mathcal{AL}$-log, an observation $o_i \in O$ is represented as a couple $(q(\boldsymbol{a}_i), \mathcal{A}_i)$ where $\mathcal{A}_i$ is a set containing ground DATALOG facts concerning the individual $i$.

**Definition 10.** *Let $H \in \mathcal{L}$ be a hypothesis, $\mathcal{K}$ a background knowledge and $o_i \in O$ an observation. We say that $H$* covers *$o_i$* under interpretations w.r.t. *$\mathcal{K}$ iff $\mathcal{K} \cup \mathcal{A}_i \cup H \models q(\boldsymbol{a}_i)$.*

**Theorem 3** *[14] Let $H \in \mathcal{L}$ be a hypothesis, $\mathcal{K}$ a background knowledge, and $o_i \in O$ an observation. We say that $H$* covers *$o_i$* under interpretations w.r.t. *$\mathcal{K}$ iff $\mathcal{K} \cup \mathcal{A}_i \cup H \vdash q(\boldsymbol{a}_i)$.*

Note that the both coverage tests can be reduced to query answering.

## 4 An instantiation of the framework

As an instantiation of our general framework for learning in $\mathcal{AL}$-log we choose the case of *characteristic induction from interpretations* which is defined as follows.

**Definition 11.** *Let $\mathcal{L}$ be a hypothesis language, $\mathcal{K}$ a background knowledge, $O$ a set of observations, and $M(\mathcal{B})$ a model constructed from $\mathcal{B} = \mathcal{K} \cup O$. The goal of* characteristic induction from interpretations *is to find a set $\mathcal{H} \subseteq \mathcal{L}$ of hypotheses such that (i) $\mathcal{H}$ is true in $M(\mathcal{B})$, and (ii) for each $H \in \mathcal{L}$, if $H$ is true in $M(\mathcal{B})$ then $\mathcal{H} \models H$.*

The logical setting of characteristic induction has been considered very close to that form of data mining, called *descriptive data mining*, which focuses on finding human-interpretable patterns describing a data set $\mathbf{r}$ [7]. *Scalability* is a crucial issue in descriptive data mining. Recently, the setting of learning from interpretations has been shown to be a promising way of scaling up ILP algorithms in real-world applications [3].

### 4.1 A task of characteristic induction

Among descriptive data mining tasks, *frequent pattern discovery* aims at the extraction of all patterns whose cardinality exceeds a user-defined threshold. Indeed each pattern is considered as an intensional description (expressed in a given language $\mathcal{L}$) of a subset of $\mathbf{r}$.

The blueprint of most algorithms for frequent pattern discovery is the *level-wise search* [18]. It is based on the following assumption: If a generality order

$\succeq$ for the language $\mathcal{L}$ of patterns can be found such that $\succeq$ is monotonic w.r.t. the evaluation function *supp*, then the resulting space $(\mathcal{L}, \succeq)$ can be searched breadth-first starting from the most general pattern in $\mathcal{L}$ and by alternating *candidate generation* and *candidate evaluation* phases. In particular, candidate generation consists of a refinement step followed by a pruning step. The former derives candidates for the current search level from patterns found frequent in the previous search level. The latter allows some infrequent patterns to be detected and discarded prior to evaluation thanks to the monotonicity of $\succeq$.

We consider a variant of this task which takes concept hierarchies into account during the discovery process, thus yielding descriptions of $\mathbf{r}$ at multiple granularity levels [17]. More formally, given

- a data set $\mathbf{r}$ including a taxonomy $\mathcal{T}$ where a reference concept $C_{ref}$ and task-relevant concepts are designated,
- a multi-grained language $\mathcal{L} = \{\mathcal{L}^l\}_{1 \leq l \leq maxG}$ of patterns
- a set $\{minsup^l\}_{1 \leq l \leq maxG}$ of support thresholds

the problem of *frequent pattern discovery at $l$ levels of description granularity*, $1 \leq l \leq maxG$, is to find the set $\mathcal{F}$ of all the patterns $P \in \mathcal{L}^l$ frequent in $\mathbf{r}$, namely $P$'s with support $s$ such that (i) $s \geq minsup^l$ and (ii) all ancestors of $P$ w.r.t. $\mathcal{T}$ are frequent in $\mathbf{r}$.

### 4.2 Casting the framework to the task

When casting our general framework for learning in $\mathcal{AL}$-log to the task of frequent pattern discovery at multiple levels of description granularity, the data set $\mathbf{r}$ is represented as an $\mathcal{AL}$-log knowledge base.

*Example 1.* As a running example, we consider an $\mathcal{AL}$-log knowledge base $\mathcal{B}_{\texttt{CIA}}$ that enriches DATALOG facts[4] extracted from the on-line 1996 CIA World Fact Book[5] with $\mathcal{ALC}$ ontologies. The structural subsystem $\Sigma$ of $\mathcal{B}_{\texttt{CIA}}$ focuses on the concepts `Country`, `EthnicGroup`, `Language`, and `Religion`. Axioms like

$\texttt{AsianCountry} \sqsubset \texttt{Country}$.
$\texttt{MiddleEasternEthnicGroup} \sqsubset \texttt{EthnicGroup}$.
$\texttt{MiddleEastCountry} \equiv \texttt{AsianCountry} \sqcap \exists \texttt{Hosts.MiddleEasternEthnicGroup}$.
$\texttt{IndoEuropeanLanguage} \sqsubset \texttt{Language}$.
$\texttt{IndoIranianLanguage} \sqsubset \texttt{IndoEuropeanLanguage}$.
$\texttt{MonotheisticReligion} \sqsubset \texttt{Religion}$.
$\texttt{ChristianReligion} \sqsubset \texttt{MonotheisticReligion}$.
$\texttt{MuslimReligion} \sqsubset \texttt{MonotheisticReligion}$.

define four taxonomies, one for each concept above. Note that Middle East countries (concept `MiddleEastCountry`) have been defined as Asian countries that host at least one Middle Eastern ethnic group. Assertions like

[4] `http://www.dbis.informatik.uni-goettingen.de/Mondial/mondial-rel-facts.flp`
[5] `http://www.odci.gov/cia/publications/factbook/`

```
'ARM':AsianCountry.
'IR':AsianCountry.
'Arab':MiddleEasternEthnicGroup.
'Armenian':MiddleEasternEthnicGroup.
<'ARM','Armenian'>:Hosts.
<'IR','Arab'>:Hosts.
'Armenian':IndoEuropeanLanguage.
'Persian':IndoIranianLanguage.
'Armenian Orthodox':ChristianReligion.
'Shia':MuslimReligion.
'Sunni':MuslimReligion.
```

belong to the extensional part of $\Sigma$. In particular, Armenia (`'ARM'`) and Iran (`'IR'`) are two of the 14 countries that are classified as Middle Eastern.

The relational subsystem $\Pi$ of $\mathcal{B}_{\texttt{CIA}}$ expresses the CIA facts as a constrained DATALOG program. The extensional part of $\Pi$ consists of DATALOG facts like

```
language('ARM','Armenian',96).
language('IR','Persian',58).
religion('ARM','Armenian Orthodox',94).
religion('IR','Shia',89).
religion('IR','Sunni',10).
```

whereas the intensional part defines two views on `language` and `religion`:

```
speaks(CountryID, LanguageN)←language(CountryID,LanguageN,Perc)
                      & CountryID:Country, LanguageN:Language
believes(CountryID, ReligionN)←religion(CountryID,ReligionN,Perc)
                      & CountryID:Country, ReligionN:Religion
```

that can deduce new DATALOG facts when triggered on $\mathcal{B}_{\texttt{CIA}}$.

The language $\mathcal{L}$ for a given problem instance is implicitly defined by a declarative bias specification that allows for the generation of expressions, called $\mathcal{O}$-queries, relating individuals of $C_{ref}$ to individuals of the task-relevant concepts.

**Definition 12.** *Given a $\mathcal{ALC}$ concept $C_{ref}$, an $\mathcal{O}$-*query *$Q$ to an $\mathcal{AL}$-log knowledge base $\mathcal{B}$ is a (linked, connected, and OI-compliant) constrained* DATALOG *clause of the form*

$$Q = q(X) \leftarrow \alpha_1, \ldots, \alpha_m \& X : C_{ref}, \gamma_1, \ldots, \gamma_n$$

*where $X$ is the* distinguished variable *and the remaining variables occurring in the body of $Q$ are the* existential variables.

The $\mathcal{O}$-query $Q_t = q(X) \leftarrow \& X : C_{ref}$ is called *trivial* for $\mathcal{L}$.

*Example 2.* We want to describe Middle East countries (individuals of the reference concept) with respect to the religions believed and the languages spoken (individuals of the task-relevant concepts) at three levels of granularity ($maxG = 3$). To this aim we define $\mathcal{L}_{\texttt{CIA}}$ as the set of $\mathcal{O}$-queries with $C_{ref} =$ `MiddleEastCountry` that can be generated from the alphabet $\mathcal{A}$= {`believes/2`, `speaks/2`} of DATALOG binary predicate names, and the alphabets

$\Gamma^1$= {Language, Religion}
$\Gamma^2$= {IndoEuropeanLanguage, ..., MonotheisticReligion, ...}
$\Gamma^3$= {IndoIranianLanguage, ..., MuslimReligion, ...}

of $\mathcal{ALC}$ concept names for $1 \leq l \leq 3$. Examples of $\mathcal{O}$-queries in $\mathcal{L}_{\texttt{CIA}}$ are:

$Q_t$= q(X) ← & X:MiddleEastCountry
$Q_1$= q(X) ← believes(X,Y) & X:MiddleEastCountry, Y:Religion
$Q_2$= q(X) ← believes(X,Y), speaks(X,Z) & X:MiddleEastCountry,
Y:MonotheisticReligion, Z:IndoEuropeanLanguage
$Q_3$= q(X) ← believes(X,Y), speaks(X,Z) & X:MiddleEastCountry,
Y:MuslimReligion, Z:IndoIranianLanguage

where $Q_t$ is the trivial $\mathcal{O}$-query for $\mathcal{L}_{\texttt{CIA}}$, $Q_1 \in \mathcal{L}^1_{\texttt{CIA}}$, $Q_2 \in \mathcal{L}^2_{\texttt{CIA}}$, and $Q_3 \in \mathcal{L}^3_{\texttt{CIA}}$.

Being a special case of constrained DATALOG clauses, $\mathcal{O}$-queries can be $\succeq_{\mathcal{B}}$-ordered. Also note that the underlying reasoning mechanism of $\mathcal{AL}$-log makes $\mathcal{B}$-subsumption more powerful than generalized subsumption as illustrated in the following example.

*Example 3.* We want to check whether $Q_1$ $\mathcal{B}$-subsumes the $\mathcal{O}$-query

$Q_4$= q(A) ← believes(A,B) & A:MiddleEastCountry, B:MonotheisticReligion

belonging to $\mathcal{L}^2_{\texttt{CIA}}$. Let $\sigma$={A/a, B/b} a Skolem substitution for $Q_4$ w.r.t. $\mathcal{B}_{\texttt{CIA}} \cup \{Q_1\}$ and $\theta$={X/A, Y/B} a substitution for $Q_1$. The condition (i) of Theorem 1 is immediately verified. It remains to verify that (ii) $\mathcal{B}' =$

$$\mathcal{B}_{\texttt{CIA}} \cup \{\texttt{believes(a,b)}, \texttt{a:MiddleEastCountry}, \texttt{b:MonotheisticReligion}\} \models \texttt{believes(a,b)} \;\&\; \texttt{a:MiddleEastCountry}, \texttt{b:Religion}.$$

We try to build a constrained SLD-refutation for

$Q^{(0)}$ = ← believes(a,b) & a:MiddleEastCountry, b:Religion

in $\mathcal{B}'$. Let $E^{(1)}$ be believes(a,b). A resolvent for $Q^{(0)}$ and $E^{(1)}$ with the empty substitution $\sigma^{(1)}$ is the constrained empty clause

$Q^{(1)}$ = ← & a:MiddleEastCountry, b:Religion

The consistency of $\Sigma'' = \Sigma' \cup$ {a:MiddleEastCountry, b:Religion} needs now to be checked. The first unsatisfiability check operates on the initial tableau $S_1^{(0)} = \Sigma' \cup$ {a:¬MiddleEastCountry}. The application of the propagation rule $\rightarrow_\perp$ to $S_1^{(0)}$ produces the final tableau $S_1^{(1)}$ = {a:⊥}. Therefore $S_1^{(0)}$ is unsatisfiable. The second check starts with $S_2^{(0)} = \Sigma' \cup$ {b:¬Religion}. The rule $\rightarrow_\sqsubseteq$ w.r.t. MonotheisticReligion⊑Religion, the only one applicable to $S_2^{(0)}$, produces $S_2^{(1)} = \Sigma \cup$ {b:¬Religion, b:¬MonotheisticReligion⊔Religion}. By applying $\rightarrow_\sqcup$ to $S_2^{(1)}$ w.r.t. Religion we obtain $S_2^{(2)} = \Sigma \cup$ {b:¬Religion, b:Religion} which brings to the final tableau $S_2^{(3)}$ = {b:⊥} via $\rightarrow_\perp$.

Having proved the consistency of $\Sigma''$, we have proved the existence of a constrained SLD-refutation for $Q^{(0)}$ in $\mathcal{B}'$. Therefore we can say that $Q_1 \succeq_{\mathcal{B}} Q_4$. Conversely, $Q_4 \not\succeq_{\mathcal{B}} Q_1$. Similarly it can be proved that $Q_2 \succeq_{\mathcal{B}} Q_3$ and $Q_3 \not\succeq_{\mathcal{B}} Q_2$.

*Example 4.* It can be easily verified that $Q_1$ $\mathcal{B}$-subsumes the following query

$Q_5$= `q(A)` ← `believes(A,B)`, `believes(A,C)` & `A:MiddleEastCountry`, `B:Religion`

by choosing $\sigma$={`A/a`, `B/b`, `C/c`} as a Skolem substitution for $Q_5$ w.r.t. $\mathcal{B}_{\texttt{CIA}} \cup \{Q_1\}$ and $\theta$={`X/A`, `Y/B`} as a substitution for $Q_1$. Note that $Q_5 \not\succeq_{\mathcal{B}} Q_1$ under the OI bias. Indeed this bias does not admit the substitution {`A/X`, `B/Y`, `C/Y`} for $Q_5$ which would make possible to verify conditions (i) and (ii) of Theorem 1.

The coverage test reduces to query answering. An *answer* to an $\mathcal{O}$-query $Q$ is a ground substitution $\theta$ for the distinguished variable of $Q$. The conditions of well-formedness reported in Definition 3 guarantee that the evaluation of $\mathcal{O}$-queries is sound according to the following notions of answer/success set.

**Definition 13.** *An answer $\theta$ to an $\mathcal{O}$-query $Q$ is a* correct (resp. computed) answer *w.r.t. an $\mathcal{AL}$-log knowledge base $\mathcal{B}$ if there exists at least one correct (resp. computed) answer to body($Q$)$\theta$ w.r.t. $\mathcal{B}$.*

Therefore proving that an $\mathcal{O}$-query $Q$ covers an observation $(q(a_i), \mathcal{A}_i)$ w.r.t. $\mathcal{K}$ equals to proving that $\theta_i = \{X/a_i\}$ is a correct answer to $Q$ w.r.t. $\mathcal{B}_i = \mathcal{K} \cup \mathcal{A}_i$.

*Example 5.* With reference to Example 1, the background knowledge $\mathcal{K}_{\texttt{CIA}}$ encompasses the structural part and the intensional relational part of $\mathcal{B}_{\texttt{CIA}}$. We want to check whether the $\mathcal{O}$-query $Q_1$ reported in Example 2 covers the observation $(\texttt{q('IR')}, \mathcal{A}_{\texttt{IR}})$ w.r.t. $\mathcal{K}_{\texttt{CIA}}$. This is equivalent to answering the query

← `q('IR')`

w.r.t. $\mathcal{K}_{\texttt{CIA}} \cup \mathcal{A}_{\texttt{IR}} \cup Q_1$. Note that $\mathcal{A}_{\texttt{IR}}$ contains all the DATALOG facts concerning the individual `IR`.

The *support* of an $\mathcal{O}$-query $Q \in \mathcal{L}$ w.r.t. $\mathcal{B}$ supplies the percentage of individuals of $C_{ref}$ that satisfy $Q$ and is defined as

$$supp(Q, \mathcal{B}) = \mid answerset(Q, \mathcal{B}) \mid / \mid answerset(Q_t, \mathcal{B}) \mid$$

where $answerset(Q, \mathcal{B})$ is the set of correct answers to $Q$ w.r.t. $\mathcal{B}$. We remind the reader that $Q_t$ is the trivial $\mathcal{O}$-query for $\mathcal{L}$.

*Example 6.* Since $\mid answerset(Q_1, \mathcal{B}_{\texttt{CIA}}) \mid = 14$ and $\mid answerset(Q_t, \mathcal{B}_{\texttt{CIA}}) \mid = \mid$ `MiddleEastCountry` $\mid = 14$, then $supp(Q_1, \mathcal{B}_{\texttt{CIA}}) = 100\%$. Analogously the support of the other $\mathcal{O}$-queries listed in Example 2 can be computed.

It has been proved that $\succeq_{\mathcal{B}}$ is monotone w.r.t. *supp* [17]. This has allowed us to implement the levelwise search. The resulting ILP system has been called $\mathcal{AL}$-QUIN ($\mathcal{AL}$-log QUERY INDUCTION) [16,14]. In particular [16] supplies details of candidate generation whereas [14] provides insight into candidate evaluation.

## 5 Conclusions

Hybrid languages like $\mathcal{AL}$-log supply expressive and deductive power which have no counterpart in *pure* Horn clausal logic. This makes them appealing for challenging applications in application domains that require a uniform treatment of both relational and structural features of data.

Learning in DL-based hybrid languages has very recently attracted attention in the ILP community. In [22] the chosen language is CARIN-$\mathcal{ALN}$, therefore example coverage and subsumption between two hypotheses are based on the existential entailment algorithm of CARIN. Following [22], Kietz studies the learnability of CARIN-$\mathcal{ALN}$, thus providing a pre-processing method which enables ILP systems to learn CARIN-$\mathcal{ALN}$ rules [12]. In [17], Lisi and Malerba propose $\mathcal{AL}$-log as a KR&R framework for the induction of association rules. Closely related to DL-based hybrid systems are the proposals arising from the study of many-sorted logics, where a first-order language is combined with a sort language which can be regarded as an elementary DL [9]. In this respect the study of a sorted downward refinement [10] can be also considered a contribution to learning in hybrid languages.

The main contribution of this paper is the definition of a general framework for learning in $\mathcal{AL}$-log. We would like to emphasize that $\mathcal{AL}$-log has been preferred to CARIN for two desirable properties of constrained SLD-resolution which are particularly appreciated in ILP: *decidability* and *closed world reasoning*. We intend to extend the framework towards more expressive hybrid languages along the direction shown in [21] in order to make it closer to the representation standards for the logical layer of the Semantic Web [2].

**Acknowledgements.** We are grateful to Francesco M. Donini and Riccardo Rosati for their precious advice on $\mathcal{AL}$-log. Also we acknowledge the financial support of the 2004 project "Astrazione e Logica Descrittiva in Apprendimento Automatico" funded by the *Università degli Studi di Bari*.

## References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
2. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May, 2001.
3. H. Blockeel, L. De Raedt, N. Jacobs, and B. Demoen. Scaling Up Inductive Logic Programming by Learning from Interpretations. *Data Mining and Knowledge Discovery*, 3:59–93, 1999.
4. A. Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1–2):353–367, 1996.
5. W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.
6. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990.

7. L. De Raedt and L. Dehaspe. Clausal Discovery. *Machine Learning*, 26(2–3):99–146, 1997.
8. F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. $\mathcal{AL}$-log: Integrating Datalog and Description Logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998.
9. A.M. Frisch. The substitutional framework for sorted deduction: Fundamental results on hybrid reasoning. *Artificial Intelligence*, 49:161–198, 1991.
10. A.M. Frisch. Sorted downward refinement: Building background knowledge into a refinement operator for inductive logic programming. In S. Džeroski and P. Flach, editors, *Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 104–115. Springer, 1999.
11. A.M. Frisch and A.G. Cohn. Thoughts and afterthoughts on the 1988 workshop on principles of hybrid reasoning. *AI Magazine*, 11(5):84–87, 1991.
12. J.-U. Kietz. Learnability of description logic programs. In S. Matwin and C. Sammut, editors, *Inductive Logic Programming*, volume 2583 of *Lecture Notes in Artificial Intelligence*, pages 117–132. Springer, 2003.
13. A.Y. Levy and M.-C. Rousset. Combining Horn rules and description logics in CARIN. *Artificial Intelligence*, 104:165–209, 1998.
14. F.A. Lisi and F. Esposito. Efficient Evaluation of Candidate Hypotheses in $\mathcal{AL}$-log. In R. Camacho, R. King, and A. Srinivasan, editors, *Inductive Logic Programming*, volume 3194 of *Lecture Notes in Artificial Intelligence*, pages 216–233. Springer, 2004.
15. F.A. Lisi and D. Malerba. Bridging the Gap between Horn Clausal Logic and Description Logics in Inductive Learning. In A. Cappelli and F. Turini, editors, *AI*IA 2003: Advances in Artificial Intelligence*, volume 2829 of *Lecture Notes in Artificial Intelligence*, pages 49–60. Springer, 2003.
16. F.A. Lisi and D. Malerba. Ideal Refinement of Descriptions in $\mathcal{AL}$-log. In T. Horvath and A. Yamamoto, editors, *Inductive Logic Programming*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 215–232. Springer, 2003.
17. F.A. Lisi and D. Malerba. Inducing Multi-Level Association Rules from Multiple Relations. *Machine Learning*, 55:175–210, 2004.
18. H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.
19. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*. Springer, 1997.
20. R. Reiter. Equality and domain closure in first order databases. *Journal of ACM*, 27:235–249, 1980.
21. R. Rosati. On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics*, 2005. to appear.
22. C. Rouveirol and V. Ventos. Towards Learning in CARIN-$\mathcal{ALN}$. In J. Cussens and A. Frisch, editors, *Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 191–208. Springer, 2000.
23. M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.
24. G. Semeraro, F. Esposito, D. Malerba, N. Fanizzi, and S. Ferilli. A logic framework for the incremental inductive synthesis of Datalog theories. In N.E. Fuchs, editor, *Proceedings of 7th International Workshop on Logic Program Synthesis and Transformation*, volume 1463 of *Lecture Notes in Computer Science*, pages 300–321. Springer, 1998.

# A Semantic Similarity Measure for Expressive Description Logics

Claudia d'Amato, Nicola Fanizzi, Floriana Esposito

Dipartimento di Informatica, Università degli Studi di Bari
Campus Universitario, Via Orabona 4, 70125 Bari, Italy
{claudia.damato, fanizzi, esposito}@di.uniba.it

**Abstract.** A totally semantic measure is presented which is able to calculate a similarity value between concept descriptions and also between concept description and individual or between individuals expressed in an expressive description logic. It is applicable on symbolic descriptions although it uses a numeric approach for the calculus. Considering that Description Logics stand as the theoretic framework for the ontological knowledge representation and reasoning, the proposed measure can be effectively used for agglomerative and divisional clustering task applied to the semantic web domain.

## 1 Introduction

Ontological knowledge plays a key role for interoperability in the Semantic Web perspective. Nowadays, standard ontology markup languages are supported by well-founded semantics of Description Logics (DLs) together with a series of available automated reasoning services [1]. However, several tasks in an ontology life-cycle [2], such as their construction and/or integration, are still almost entirely delegated to knowledge engineers.

In the Semantic Web perspective, the construction of the knowledge bases should be supported by automated inductive inference services. The induction of structural knowledge like the T-box taxonomies is not new in machine learning, especially in the context of *concept formation* [3] where clusters of similar objects are aggregated in hierarchies according to heuristic criteria or similarity measures. Almost all of these methods apply to zero-order representations while, as mentioned above, ontologies are expressed through fragments of first-order logic. Yet, the problem of the induction of structural knowledge turns out to be hard in first-order logic or equivalent representations [4].

In *Inductive Logic Programming* (ILP) attempts have been made to extend relational learning techniques towards hybrid representations based on both clausal and description logics [5, 6, 7]. In order to cope with the problem complexity, these methods are based on a heuristic search and generally implement bottom-up algorithms that tend to induce overly specific concept definitions which may suffer for poor predictive capabilities.

So far, the automated induction of knowledge bases expressed in DLs representations has not been investigated in depth. Classic approaches to learning

DL concept definitions generally adopt heuristic search strategies to cope with the inherent complexity of the problem and generally implement bottom-up algorithms (e.g. [8]). Other approaches propose a top-down search for correct concept definitions [9]. These methods are not completely operational: since refinement operators compute short moves in a vast space of candidate definitions, they become useless when disjoined from proper heuristics based on the available assertions. A more knowledge-intensive method is to be preferred.

In this perspective, we introduce a novel similarity measure between concept descriptions based on semantics, which is suitable for expressive DLs like $\mathcal{ALC}$ [10, 1]. Since a merely syntactic approach has proven too weak to enforce standard inferences (namely subsumption), when expressive DLs are taken into account a different approach (based on semantics) is necessary. Also a similarity measure, then, should be founded on the underlying semantics, rather than on the syntactic structure of concept descriptions. Besides measuring the similarity of two concept descriptions, we propose a manner based on notion of *most specific concept* of an individual [1] for employing the same measure for the individual-concept and individual-individual similarity cases.

Such a measure can be the basis for adapting an existing clustering method to this representation (or devising a new one) operating in a top-down (*partitional*) or bottom-up (*agglomerative*) fashion. Moreover, the similarity measure can be also employed for *Information Retrieval* or *Information Integration* purposes applied to DL knowledge bases and also for *Case-based Reasoning* systems (see the next section).

As discussed in the following, the method can effectively compute the similarity measure with a complexity which depends on the complexity of standard inferences as a baseline. The applicability of this has been tested on examples which have been artificially generated from OWL[1] ontologies. Some of these test examples are reported in this paper.

The remainder of the paper is organized as follows. The next section reviews related work on similarity measures. In Sect. 3 the representation language is presented. The similarity measure is illustrated in Sect. 4 and is discussed in Sect. 5. Possible developments of the method are examined in Sect. 6.

## 2 Related Work

Similarity measure play an important role in information retrieval and information integration. Recent investigations in these fields have emphasized the use of ontologies and semantic similarity functions as a mechanism for comparing concepts and/or concept instances that can be retrieved or integrated across heterogeneous repositories [11, 12, 13, 14].

Semantic similarity is typically determined as a function of the *path distance* between terms in the hierarchical structure underlying the ontology [15, 16, 17]. Other methods to assess semantic similarity within a single ontology are *feature*

[1] http://www.w3.org/TR/owl-ref

*matching* [18] and *information content* [13, 19]. The former approach uses both common and discriminant features among concepts and/or concept instances in order to compute the semantic similarity. The latter methods are founded on *Information Theory*. They define a similarity measure between two concepts within a concept hierarchy in terms of the amount of information conveyed by the immediate super-concept that subsumes two concepts being compared. This is a measure of the variation of information crossing from a description level to a more general one.

A recent work [20] presents a number of measures for comparing concepts located in different and possibly heterogeneous ontologies. The following requirements are made for this measure:

- the formal representation supports inferences such as *subsumption*;
- local concepts in different ontologies inherit their definitional structure from concepts in a shared ontology.

This study assumes that the intersection of sets of concept instances is an indication of the correspondence between these concepts. Three main types of measures for comparing concept descriptions are discussed in this work:

1. *filter* measures based on a path-distance
2. *matching* measures based on graph matching establish one-to-one correspondence between elements of the concept descriptions, and
3. *probabilistic* measures that give the correspondence in terms of the joint distribution of concepts.

Other similarity measures have been developed to compute similarity values among classes belonging to different ontologies. These measures are able to take into account the difference in the levels of explicitness and formalization of the different ontology specifications. Particularly, in [21] a similarity function determines similar entity classes by using a matching process making use of synonym sets, semantic neighborhood, and discriminating features that are classified into parts, functions, and attributes.

Another approach [22], aimed at finding commonalities among concepts or among assertions, employs the *Least Specific Concept* operator (LCS [1]) that computes the most specific generalization of the input concepts (with respect to subsumption, see the next section for a formal definition). This approach is generally intended for information retrieval purposes. Considered a knowledge base and a query concept, a filter mechanism selects another concept from the knowledge base that is relevant for the query concept. Then the LCS of the two concepts is computed and finally all concepts subsumed by the LCS are returned.

Most of the cited works adopt a semantic approach in conjunction with the structure of the considered concept descriptions. Thus, they are liable to the phenomenon of the rapid growth of the description granularity. Besides the syntactic structure of concept descriptions becomes much less important when richer DL

**Table 1.** $\mathcal{ALC}$ constructors and their meaning.

| Name | Syntax | Semantics |
|---|---|---|
| top concept | $\top$ | $\Delta^{\mathcal{I}}$ |
| bottom concept | $\bot$ | $\emptyset$ |
| concept | $C$ | $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ |
| concept negation | $\neg C$ | $\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$ |
| concept conjunction | $C_1 \sqcap C_2$ | $C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$ |
| concept disjunction | $C_1 \sqcup C_2$ | $C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$ |
| existential restriction | $\exists R.C$ | $\{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}}((x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}})\}$ |
| universal restriction | $\forall R.C$ | $\{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}}((x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}})\}$ |

representations are adopted due to the expressive operators that can be employed. For these reasons, we have decided to focus our attention to a measure which is totally based on semantics.

## 3 The Reference Representation Language

In relational learning, several solutions have been proposed for the adoption of an expressive fragment of first-order logic endowed with efficient inference procedures. Alternatively, the data model of a knowledge base can be expressed by means of DL concept languages which are empowered with precise semantics and effective inference services [1]. Besides, most of the ontology markup languages for the Semantic Web (e.g., OWL) are founded in Description Logics: representation languages borrow and implement the typical constructors of the DL languages.

Although it can be assumed that annotations and conceptual models are maintained and transported using the XML-based languages mentioned above, the syntax of the representation adopted here is taken from the standard constructors proposed in the DL literature [1]. These DL representations turn out to be both sufficiently expressive and efficient from an inferential viewpoint.

In this section we recall syntax and semantics for the reference representation $\mathcal{ALC}$ [10] which is adopted in the rest paper for it turns out to be sufficiently expressive to support most of the principal constructors of an ontology markup language for the Semantic Web.

In a DL language, primitive *concepts*, denoted with names taken from $N_C = \{C, D, \ldots\}$, are interpreted as subsets of a certain domain of objects (resources) or equivantly as unary relation on such domain and primitive *roles*, denoted with names taken from $N_R = \{R, S, \ldots\}$, are interpreted as binary relations on such a domain (properties). Complex concept descriptions can be built using primitive concepts and roles by means of the constructors in Table 1. Their semantics is defined by an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is the *domain* of the interpretation and the functor $\cdot^{\mathcal{I}}$ stands for the *interpretation function*, mapping the intension of concepts and roles to their extension.

A *knowledge base* $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ contains two components: A *T-box* $\mathcal{T}$ and an *A-box* $\mathcal{A}$. $\mathcal{T}$ is a set of concept definitions $C \equiv D$, meaning $C^{\mathcal{I}} = D^{\mathcal{I}}$, where $C$ is the concept name and $D$ is a description given in terms of the language constructors. $\mathcal{A}$ contains extensional assertions on concepts and roles, e.g. $C(a)$ and $R(a, b)$, meaning, respectively, that $a^{\mathcal{I}} \in C^{\mathcal{I}}$ and $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$; $C(a)$ and $R(a, b)$ are said respectively instance of the concept $C$ and instance of the role $R$, more generally it is said (without loss of generality) that the individual $a$ is instance of the concept $C$ and the same for the role. A notion of *subsumption* between concepts is given in terms of the interpretations:

**Definition 3.1 (subsumption).** *Given two concept descriptions $C$ and $D$, $C$* subsumes *$D$, denoted by $C \sqsupseteq D$, iff for every interpretation $\mathcal{I}$ it holds that $C^{\mathcal{I}} \supseteq D^{\mathcal{I}}$.*

Axioms based on subsumption ($C \sqsupseteq D$) are generally also allowed in the T-boxes as partial definitions. Indeed, $C \equiv D$ amounts to $C \sqsupseteq D$ and $D \sqsupseteq C$.

*Example 3.1.* An instance of concept definition in the proposed language is:

$$\mathsf{Father} \equiv \mathsf{Male} \sqcap \exists \mathsf{hasChild}.\mathsf{Person}$$

which corresponds to the sentence: *"a father is a male (person) that has some persons as his children"*.

The following are instances of simple assertions:

$$\mathsf{Male}(\mathsf{Leonardo}),\ \mathsf{Male}(\mathsf{Vito}),\ \mathsf{hasChild}(\mathsf{Leonardo}, \mathsf{Vito}).$$

Supposing that $\mathsf{Male} \sqsubseteq \mathsf{Person}$ is known (in the T-Box), one can deduce that: $\mathsf{Person}(\mathsf{Leonardo})$, $\mathsf{Person}(\mathsf{Vito})$ and then $\mathsf{Father}(\mathsf{Leonardo})$.

Given these primitive concepts and roles, it is possible to define many other related concepts:

$$\mathsf{Parent} \equiv \mathsf{Person} \sqcap \exists \mathsf{hasChild}.\mathsf{Person}$$

and

$$\mathsf{FatherWithoutSons} \equiv \mathsf{Male} \sqcap \exists \mathsf{hasChild}.\mathsf{Person} \sqcap \forall \mathsf{hasChild}.(\neg \mathsf{Male})$$

It is easy to see that the following relationships hold: $\mathsf{Parent} \sqsupseteq \mathsf{Father}$ and $\mathsf{Father} \sqsupseteq \mathsf{FatherWithoutSons}$. □

Especially for rich DL languages such as $\mathcal{ALC}$, many semantically equivalent (yet syntactically different) descriptions can be given for the same concept, which is the reason for preferring employing semantic approaches to reasoning over structural ones. Nevertheless equivalent concepts can be reduced to a normal form by means of rewriting rules that preserve their equivalence, such as: $\forall R.C_1 \sqcap \forall R.C_2 \equiv \forall R.(C_1 \sqcap C_2)$ (see [1] for issues related to normalization and simplification).

One of the most important inference services from the inductive learning viewpoint is *instance checking*, that is deciding whether an individual is an instance of a concept (w.r.t. an A-Box). Related to this problem, it is often necessary to solve the *realization problem* that requires to compute, given an A-Box and an individual the concepts which the individual belongs to:

**Definition 3.2 (Most Specific Concept).** *Given an A-Box $\mathcal{A}$ and an individual $a$, the most specific concept of $a$ w.r.t. $\mathcal{A}$ is the concept $C$, denoted $\mathsf{MSC}_{\mathcal{A}}(a)$, such that $\mathcal{A} \models C(a)$ and $\forall D$ such that $\mathcal{A} \models D(a)$, it holds: $C \sqsubseteq D$.*

where $\models$ stands for the standard semantic deduction [23].

In the general case of a cyclic A-Box expressed in a an expressive DL endowed with existential or numeric restriction the $\mathsf{MSC}$ cannot be expressed as a finite concept description [1], thus it can only be approximated.

Since the existence of the $\mathsf{MSC}$ for an individual w.r.t. an A-Box is not guaranteed or it is difficult to compute, generally an approximation of the $\mathsf{MSC}$ is considered up to a certain depth $k$. The maximum depth $k$ has been shown to correspond to the depth of the considered A-Box, as defined in [22].

Henceforth we will indicate generically an approximation to the maximum depth with $\mathsf{MSC}^*$.

## 4 The Similarity Measure

In this section we present a similarity measure which is able to assess the similarity between instances or between instance and concept or even between concepts expressed in Description Logic and in particular in the $\mathcal{ALC}$ logic. We call such elements generically objects. Then we will formalize the measure for the various object types. The presented measure employs the basic set theory. It is mainly founded on the commonality among objects. Particularly, the base criterion for this measure is: the similarity value between objects is not only the result of the common features, but also the result of the different characteristics too. This criterion is in agreement with an information-theoretic definition of similarity [24].

### 4.1 A Similarity Measure between Concepts

Let $C$ and $D$ two concepts description in a T-Box, expressed in the $\mathcal{ALC}$ logic. Now recall that through the instance checking service it is possible to determine the set of all individuals of a given A-Box that are instances of a certain concept. Let $C^{\mathcal{I}}$ and $D^{\mathcal{I}}$ be, respectively, the extensions of the concepts $C$ and $D$ respectively. By Def. 3.1, $D$ subsumes $C$ (written $C \sqsubseteq D$) if the set of individuals that are instances of $C$ is contained in the set of individuals that are instances of $D$, in other words if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Here we refer to the canonical interpretation of the A-Box and the *unique names assumption* (UNA) is made: constants in the A-Box are interpreted as themselves and different names for individuals stand for different domain objects (*canonical interpretation*).

Subsumption is a semantic relationship which induces an order over the space of concept descriptions: if $C \sqsubseteq D$ then $D$ is more general that $C$ or equivalently then $C$ is more specific then $D$. Based on subsumption and set theory, we define a semantic similarity measure:

**Definition 4.1 (Semantic Similarity Measure).** *Let $\mathcal{L}$ be the set of all concepts in $\mathcal{ALC}$ and let $\mathcal{A}$ be an A-Box with canonical interpretation $\mathcal{I}$. The* Semantic Similarity Measure *$s$ is a function*

$$s : \mathcal{L} \times \mathcal{L} \mapsto [0, 1]$$

*defined as follows:*

$$s(C, D) = \frac{|I^{\mathcal{I}}|}{|C^{\mathcal{I}}| + |D^{\mathcal{I}}| - |I^{\mathcal{I}}|} \cdot \max(|I^{\mathcal{I}}|/|C^{\mathcal{I}}|, |I^{\mathcal{I}}|/|D^{\mathcal{I}}|)$$

*where $I = C \sqcap D$ and $(\cdot)^{\mathcal{I}}$ computes the concept extension wrt the interpretation $\mathcal{I}$.*

The measure can be briefly justified as follows.
In case of semantic equivalence of the input concepts ($C \sqsubseteq D$ and $D \sqsubseteq C$), the maximum value of the similarity is assigned.

In case of disjunction, the minimum value of similarity is assigned because the two concepts are totally different: their extensions do not overlap. Indeed, they are semantically unrelated with respect to the generalization order: their intersection amounts to the bottom concept.

In case of overlapping concepts, a value in the range $]0, 1[$ is computed. It expresses the similarity between the two concepts (represented by the factor $|I^{\mathcal{I}}|/(|C^{\mathcal{I}}| + |D^{\mathcal{I}}| - |I^{\mathcal{I}}|)$) reduced by a quantity ($\max(|I^{\mathcal{I}}|/|C^{\mathcal{I}}|, |I^{\mathcal{I}}|/|D^{\mathcal{I}}|)$) which represents the major incidence of the intersection with respect to either concept. This means considering similarity not as an absolute value but as weighted with respect to a degree of non-similarity. Indeed, the higher such factor is the more one of the concepts is likely to be subsumed by the other. This is in accordance to the strong semantic relation between the concepts ensured by subsumption.

*Example 4.1.* Let be consider the knowledge base with the *T-Box* and *A-Box* reported below.

Primitive Concepts: $N_C = \{\mathsf{Female},\ \mathsf{Male},\ \mathsf{Human}\}$.

Primitive Roles: $N_R = \{\mathsf{HasChild},\ \mathsf{HasParent},\ \mathsf{HasGrandParent},\ \mathsf{HasUncle}\}$.

T-Box: $\mathcal{T} = \{$
$\mathsf{Woman} \equiv \mathsf{Human} \sqcap \mathsf{Female}$
$\mathsf{Man} \equiv \mathsf{Human} \sqcap \mathsf{Male}$
$\mathsf{Parent} \equiv \mathsf{Human} \sqcap \exists \mathsf{HasChild}.\mathsf{Human}$

Mother $\equiv$ Woman $\sqcap$ Parent $\exists$HasChild.Human
Father $\equiv$ Man $\sqcap$ Parent
Child $\equiv$ Human $\sqcap$ $\exists$HasParent.Parent
Grandparent $\equiv$ Parent $\sqcap$ $\exists$HasChild.( $\exists$ HasChild.Human)
Sibling $\equiv$ Child $\sqcap$ $\exists$HasParent.( $\exists$ HasChild $\geq$ 2)
Niece $\equiv$ Human $\sqcap$ $\exists$HasGrandParent.Parent $\sqcup$ $\exists$HasUncle.Uncle
Cousin $\equiv$ Niece $\sqcap$ $\exists$HasUncle.($\exists$ HasChild.Human)
}.

A-Box: $\mathcal{A}$ = {Woman(Claudia), Woman(Tiziana),
Father(Leonardo), Father(Antonio), Father(AntonioB),
Mother(Maria), Mother(Giovanna),
Child(Valentina),
Sibling(Martina), Sibling(Vito),
HasParent(Claudia,Giovanna), HasParent(Leonardo,AntonioB),
HasParent(Martina,Maria), HasParent(Giovanna,Antonio),
HasParent(Vito,AntonioB), HasParent(Tiziana,Giovanna),
HasParent(Tiziana,Leonardo), HasParent(Valentina,Maria),
HasParent(Maria,Antonio),
HasSibling(Leonardo,Vito), HasSibling(Martina,Valentina),
HasSibling(Giovanna,Maria), HasSibling(Vito,Leonardo),
HasSibling(Tiziana,Claudia), HasSibling(Valentina,Martina),
HasChild(Leonardo,Tiziana), HasChild(Antonio,Giovanna),
HasChild(Antonio,Maria), HasChild(Giovanna,Tiziana),
HasChild(Giovanna,Claudia), HasChild(AntonioB,Vito),
HasChild(AntonioB,Leonardo), HasChild(Maria,Valentina),
HasUncle(Martina,Giovanna), HasUncle(Valentina,Giovanna) }

Considered this knowledge base, it is possible to compute the similarity value between concepts as shown:

$$s(\mathsf{Grandparent}, \mathsf{Father}) = \frac{|\mathsf{Grandparent} \sqcap \mathsf{Father}|}{|\mathsf{Granparent}| + |\mathsf{Father}| - |\mathsf{Grandparent} \sqcap \mathsf{Father}|} \cdot$$
$$\cdot \ max(\frac{|\mathsf{Grandparent} \sqcap \mathsf{Father}|}{|\mathsf{Grandparent}|}, \frac{|\mathsf{Grandparent} \sqcap \mathsf{Father}|}{|\mathsf{Father}|}) = \frac{2}{2+3-2} \cdot max(\frac{2}{2}, \frac{2}{3}) = 0.67$$

In the same way it is possible to compute the similarity value among all concepts the defined above. □

### 4.2 Derived Similarity Measures Involving Individuals

Let us recall that, for every individual in the A-Box, it is possible to calculate the Most Specific Concept MSC (see Def. 3.2) or at least its approximation $\mathsf{MSC}^*$. In some cases they are equivalent concepts.

Let $a$ and $b$ two individuals in a given A-Box. We can calculate $A^* = \mathsf{MSC}^*(a)$ and $B^* = \mathsf{MSC}^*(b)$. Now the semantic similarity measure $s$ can be applied to

these concept descriptions, thus yielding the similarity value of two instances:

$$\forall a, b: \; s(a, b) = s(A^*, B^*) = s(\mathsf{MSC}^*(a), \mathsf{MSC}^*(b))$$

Analogously, the similarity value between a concept description $C$ and an individual $a$ can be computed by determining the MSC approximation of the individual and then applying the similarity measure:

$$\forall a: \; s(a, C) = s(\mathsf{MSC}^*(a), C)$$

*Example 4.2.* Considering the knowledge base of the previous example, it is possible to show how to determine the similarity value between individuals. Let Claudia and Tiziana be such individuals. First of all, using the $\mathsf{MSC}^*$ operator, we have:

MSC*(Claudia) = Woman ⊓ Sibling ⊓ ∃ HasParent(Mother ⊓ Sibling ⊓ ∃HasSibling(C1) ⊓ ∃HasParent(C2) ⊓ ∃HasChild(C3))

C1 ≡ Mother ⊓ Sibling ⊓ ∃HasParent(Father ⊓ Parent) ⊓ ∃HasChild(Cousin ⊓ ∃HasSibling(Cousin ⊓ Sibling ⊓ ∃HasSibling.⊤))

C2 ≡ Father ⊓ ∃HasChild(Mother ⊓ Sibling)

C3 ≡ Woman ⊓ Sibling ⊓ ∃HasSibling.⊤ ⊓ ∃HasParent(C4)

C4 ≡ Father ⊓ Sibling ⊓ ∃HasSibling(Uncle ⊓ Sibling ⊓ ∃HasParent(Father ⊓ Grandparent)) ⊓ ∃HasParent(Father ⊓ Grandparent ⊓ ∃HasChild(Uncle ⊓ Sibling))

And for the individual Tiziana:

MSC*(Tiziana) = Woman ⊓ Sibling ⊓ ∃HasSibling(Woman ⊓ Sibling ⊓ ∃HasParent(C5)) ⊓ ∃HasParent(Father ⊓ Sibling ⊓ ∃HasSibling(C6) ⊓ ∃HasParent(C7)) ⊓ ∃HasParent(Uncle ⊓ Mother ⊓ Sibling ⊓ ∃HasChild(Woman ⊓ Sibling))

C5 ≡ Mother ⊓ Sibling ⊓ ∃HasSibling(C8) ⊓ ∃HasParent(Father ⊓ ∃HasChild(Mother ⊓ Sibling))

C8 ≡ Mother ⊓ Sibling ⊓ ∃HasParent(Father ⊓ Grandparent) ⊓ ∃HasChild(Cousin ⊓ ∃HasSibling(Cousin ⊓ Sibling ⊓ ∃HasSibling.⊤))

C6 ≡ Uncle ⊓ Sibling ⊓ ∃HasParent(Father ⊓ Grandparent)

C7 ≡ Father ⊓ Granparent ⊓ ∃HasChild(Uncle ⊓ Sibling)

Note that it holds that $\mathsf{MSC}^*(\text{Tiziana}) \not\sqsubseteq \mathsf{MSC}^*(\text{Claudia})$ and $\mathsf{MSC}^*(\text{Claudia}) \not\sqsubseteq \mathsf{MSC}^*(\text{Tiziana})$. Now, since $\mathsf{MSC}^*(\text{Tiziana}) = \{\text{Tiziana}\}$ and $\mathsf{MSC}^*(\text{Claudia}) =$

{Claudia, Tiziana}, the similarity value between these individuals is:

$$s(\mathsf{Claudia}, \mathsf{Tiziana}) = \frac{1}{1+2-1} \cdot max\left(\frac{1}{2}, \frac{1}{1}\right) = 0.5$$

In the same way it could be calculate the similarity value between concept and individual. □

## 5 Discussion

First of all it is important to note that, differently from previously proposed similarity measures (see Sect. 2), this measure is totally semantic. Indeed, it uses only semantic inferences like instance checking (to solve the *retrieval problem* [1], that amounts to computing the extension of a concept given an A-Box); it does not make use of the syntactic structure of the concept description, thus it is independent from the granularity level of descriptions. This fact reflects the intrinsic complexity of expressive DL languages like $\mathcal{ALC}$ for which a structural approach to reasoning is simply ineffective (subsumption is computed using a tableaux rather than a structural algorithm). Therefore, the definition of $s$ employs set theory and semantic services, so it make use of numeric approach despite its application on a symbolic DL representation.

Our similarity measure has been applied on knowledge base written using the $\mathcal{ALC}$ logic. However, for the reasons mentioned above, it is important to note that $s$ is applicable to any DL endowed with the basic reasoning services required by its definitions, namely: instance checking and MSC (approximation).

Similarity measures turn out to be useful in several applications and for many tasks such as commonality-based information retrieval in the context of a terminological knowledge representation system (which is a relatively new application context [22]), realization of semantic search engine, classification, case-based reasoning, clustering.

This last task is our goal. In particular, having defined a measure that is applicable both between concepts and between individuals and between concept and individuals, it is suitable for agglomerative clustering and for divisional clustering too. However we have noticed that $s$ measure is suitable for measuring similarity between concepts but it presents some problem in case of individuals. This is due to the use of the individual's MSC (approximation) which often turn out to be so specific that its extension likely includes only the considered individual; this phenomenon consequently provokes a totally dissimilarity value even if the individuals semantically express similar underlying concepts. So now we are investigating ways to overcome this limitation.

Below we prove that the function $s$ presented is really a similarity measure and discuss the complexity issues related to its computation.

### 5.1 Properties of the Similarity Measure

In this section we prove that $s$ function actually is a similarity measure (or *similarity function* [25]), according to the formal definition:

**Definition 5.1 (Similarity Measure).** *Let* E *be a set of elements among which a similarity measure has to be defined. A* similarity measure $E$ *is a real-valued function* $d$ *defined on the set* $E \times E$ *that fulfills the following properties:*

1. $f(a,b) \geq 0 \quad \forall a,b \in E$ (positive definiteness)
2. $f(a,b) = f(b,a) \quad \forall a,b \in E$ (symmetry)
3. $\forall a,b \in E : f(a,b) \leq f(a,a)$

From the definition given in the previous section it is straightforward to prove that $s$ satisfies the first property because $s$ has value in the real interval $[0,1]$. Then, as previously said, $s$ assigns the maximum value when the concepts subsume each other; this last is the condition of equality of concept, so the third property is satisfied too. The property of symmetry is also trivially verified. Indeed set intersection, sum, product and maximum are commutative. It is straightforward to note that given two concepts $C$ and $D$, it holds that:

$$s(C,D) = \frac{|I^{\mathcal{I}}|}{|C^{\mathcal{I}}|+|D^{\mathcal{I}}|-|I^{\mathcal{I}}|} \cdot \max\left(\frac{|I^{\mathcal{I}}|}{|C^{\mathcal{I}}|}, \frac{|I^{\mathcal{I}}|}{|D^{\mathcal{I}}|}\right) = \frac{|I^{\mathcal{I}}|}{|D^{\mathcal{I}}|+|C^{\mathcal{I}}|-|I^{\mathcal{I}}|} \cdot \max\left(\frac{|I^{\mathcal{I}}|}{|D^{\mathcal{I}}|}, \frac{|I^{\mathcal{I}}|}{|C^{\mathcal{I}}|}\right) = s(D,C)$$

note that $I$ remains the same because of the commutativity of intersection.

### 5.2 Complexity Issues

In order to assess the complexity of $s$, the three different cases of applicability of the measure are discussed separately. They all depend on the complexity of the instance checking inference for the adopted DL language, hereafter indicated with $C(\mathsf{IC})$.

**Similarity between concepts:** $s$ is a numerical measure, all calculus in $s$ need of constant complexity; it holds that:

$$C(s) = 3 \cdot C(\mathsf{IC})$$

because the instance check is repeated three times: for the concept descriptions $C$, $D$ and $I$.

**Similarity between an individual and a concept:** in this case, besides of the instance checking operations required by the previous case, the $\mathsf{MSC}$ approximation of the considered individual is to be computed. Thus, denoted with $C(\mathsf{MSC}^*)$ the complexity of the $\mathsf{MSC}$ approximation, it holds that:

$$C(s) = C(\mathsf{MSC}^*) + 3 \cdot C(\mathsf{IC})$$

**Similarity between individuals:** this case is analogous to the previous one, the only difference is that now two $\mathsf{MSC}^*$ approximations are to be computed for the arguments. So the complexity in this case is:

$$C(s) = 2 \cdot C(\mathsf{MSC}^*) + 3 \cdot C(\mathsf{IC})$$

As clearly shown by these formulæ, the measure complexity is sensible to the choice of the reference DL. For instance, for the $\mathcal{ALC}$ logic, $C(\mathsf{IC})$ is PSPACE (see [1], Ch. 3). For the cases involving individuals it suffices to recall that also the computation of the $\mathsf{MSC}$ approximations depends on instance checking besides of the specific algorithm [22].

## 6 Conclusions and Future Work

We have introduced a new similarity measure $s$ which is primarily meant for computing a similarity value between concept descriptions but, as previously shown, it could be also employed for assessing the similarity between individuals and between a concept description and an individual.

As previously suggested, such a measure could be applied to many tasks, namely clustering and retrieval on DL knowledge bases.

This measure could be improved in the case of non overlapping concepts. In particular, we will try to assess similarity in different cases employing a notion of *distance* between concepts.

This is strictly related to the weakness of the presented semantic similarity measure in cases involving individuals. In particular, we are addressing our research on the tuning of a semantic operator for the generalization of the approximated MSC obtained. In this way, we may overcome the actual problem, because the concept would be less specific then MSC and so it would instantiate more individuals that the selected one for calculating the similarity value. With this fitting we would keep a totally semantic similarity measure, but really applicable in every context concerning DLs representations.

## References


[1] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook. Cambridge University Press (2003)

[2] Staab, S., Studer, R., eds.: Handbook on Ontologies. International Handbooks on Information Systems. Springer (2004)

[3] Thompson, K., Langley, P.: Concept formation in structured domains. In Fisher, D., Pazzani, M., Langley, P., eds.: Concept Formation: Knowledge and Experience in Unsupervised Learning. Morgan Kaufmann (1991)

[4] Haussler, D.: Learning conjuntive concepts in structural domains. Machine Learning **4** (1989) 7–40

[5] Donini, F., Lenzerini, M., Nardi, D., Schaerf, M.: $\mathcal{AL}$-log: Integrating Datalog and description logics. Journal of Intelligent Information Systems **10** (1998) 227–252

[6] Rouveirol, C., Ventos, V.: Towards learning in CARIN-$\mathcal{ALN}$. In Cussens, J., Frisch, A., eds.: Proceedings of the 10th International Conference on Inductive Logic Programming. Volume 1866 of LNAI., Springer (2000) 191–208

[7] Kietz, J.U.: Learnability of description logic programs. In Matwin, S., Sammut, C., eds.: Proceedings of the 12th International Conference on Inductive Logic Programming. Volume 2583 of LNAI., Sydney, Springer (2002) 117–132

[8] Kietz, J.U., Morik, K.: A polynomial approach to the constructive induction of structural knowledge. Machine Learning **14** (1994) 193–218

[9] Badea, L., Nienhuys-Cheng, S.H.: A refinement operator for description logics. In Cussens, J., Frisch, A., eds.: Proceedings of the 10th International Conference on Inductive Logic Programming. Volume 1866 of LNAI., Springer (2000) 40–59

[10] Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artificial Intelligence **48** (1991) 1–26

[11] Lee, J., Kim, M., Lee, Y.: Information retrieval based on conceptual distance in is-a hierarchies. Journal of Documentation **2** (1993) 188–207

[12] Smeaton, A.F., Quigley, I.: Experiments on using semantic distances between words in image caption retrieval. In Frei, H.P., Harman, D., Schäuble, P., Wilkinson, R., eds.: Proceedings of the 19th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR96, ACM (1996)
[13] Jang, J., Conrath, D.: Semantic symilarity based on corpus statistic and lexical taxonomy. In: Proceedings of the International Conference on Computational Linguistics. (1997)
[14] Guarino, N., Masolo, C., Verete, G.: Ontoseek: Content-based access to the web. IEEE Intelligent Systems **3** (1999) 70–80
[15] Collet, C., Huhns, M.N., Shen, W.M.: Resource integration using a large knowledge base in carnot. IEEE Computer **24** (1991) 55–62
[16] Fankhauser, P., Neuhold, E.J.: Knowledge based integration of heterogeneous databases. In Hsiao, D.K., Neuhold, E.J., Sacks-Davis, R., eds.: Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5). IFIP Transactions, North-Holland (1992)
[17] Bright, M.W., Hurson, A.R., Pakzad, S.H.: Automated resolution of semantic heterogeneity in multidatabases. ACM Transaction on Database Systems **19** (1994) 212–253
[18] Tversky, A.: Features od similarity. Psycological Review **84** (1997) 327–352
[19] Resnik, P.: Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language. Journal of Artificial Intelligence Research **11** (1999) 95–130
[20] Weinstein, P., Bimingham, P.: Comparing concepts in differentiated ontologies. In: Proceedings of 12th Workshop on Knowledge Acquisition, Modelling, and Management. (1999)
[21] Rodríguez, M.A., Egenhofer, M.J.: Determining semantic similarity among entity classes from different ontologies. IEEE Transaction on Knowledge and Data Engineering **15** (2003) 442–456
[22] Mantay, T.: Commonality-based ABox retrieval. Technical Report FBI-HH-M-291/2000, Department of Computer Science, University of Hamburg, Germany (2000)
[23] Chang, C., Lee, R.: Symbolic Logic and Mechanical Theorem Proving. Academic Press, San Diego (1973)
[24] Lin, D.: An information-theroretic defintion of similarity. In: Proceedings of the International Conference on Machine Learning, Morgan Kaufmann (1998) 296–304
[25] Bock., H.: Analysis of Symbolic Data: Exploratory Methods for Extracting Statistical Information from Complex Data. Springer-Verlag (1999)

# Interleaving belief revision and reasoning: preliminary report

F. Sadri and F. Toni

Department of Computing, Imperial College London, UK
Email: {fs,ft}@doc.ic.ac.uk

**Abstract.** Most existing work on belief revision and knowledge representation and reasoning tacitly assumes that belief revision is performed off-line, and that reasoning from the beliefs is performed either before or after the beliefs are changed. This imposes that, if a revision occurs while reasoning is performed, reasoning has to be stopped and re-started anew so that the revision is taken into account, with an obvious wastage of reasoning effort. In this paper, we tackle the problem of performing belief revision on-line, while reasoning is taking place by means of an abductive proof procedure.

## 1 Introduction

Traditionally, the field of belief revision (e.g. [1]) has concentrated upon identifying principled ways to incorporate new information or delete existing information from sets of beliefs (or belief bases). On the other hand, the field of knowledge representation and reasoning has concentrated upon identifying techniques for drawing correct conclusions efficiently from beliefs. Beliefs could be held by intelligent agents, and reasoning could aim at identifying plans for the agents' goals or answering queries on the agents' beliefs.

If the revision of beliefs takes place while reasoning is performed, the problem arises as to which bits of the already performed reasoning can be "saved", because it has used up beliefs which have not been affected by the revision. Conventional knowledge representation and reasoning systems overlook this problem, and assume that no bits of reasoning can be saved, by restarting the reasoning process anew.

In this paper, we show how belief revision and reasoning can be interleaved, so that reasoning with beliefs that are not affected by the revision can be saved and, if necessary, used after the revision. We assume that reasoning is performed by means of the abductive proof procedure IFF [4] and that beliefs are represented by means of abductive logic programs [7]. In this paper, we also assume that belief revision amounts to the addition or deletion of rules or facts to the logic programming component of the abductive logic program, or to the addition of integrity constraints in the corresponding component of the abductive logic program. For simplicitly, we do not consider deletion of integrity constraints.

These rules and facts could be generated by learning, or by any other means. In this paper, we do not focus on the belief revision part of the process, but rather on how such revision affects the reasoning process. In order to save any reasoning effort not affected by revisions, we adopt a labeling technique that maintains dependencies. We define a new variant of the IFF procedure, called *LIFF*, with labels occurring in derivations. We formalise the soundness of LIFF, wrt the semantics underlying the IFF procedure.

In this paper, we concentrate on the propositional case only for the definition of LIFF and the results.

Our work is closely related to the work of [5, 6], which share our aims. However, we extend an abductive proof procedure (IFF) for beliefs in the form of abductive logic programs (consisting of logic programs and integrity constraints) whereas [5] extends conventional SLDNF, for ordinary logic programs. Our knowledge representation and reasoning choice paves the way to the use of our techniques for agent applications, following the abductive logic agent approach of [10, 14, 8]. Moreover, we use a labeling technique rather than the (arguably more space consuming) technique based upon full data structures associated to atoms as [5].

Amongst agent applications for which our approach is particularly useful are those where the agents are situated in dynamic environments and need to adjust quickly to the changes in that environment. For example, a situated pro-active agent can plan for its goals while it interleaves the planning with action execution and observations from the environment; such an agent has, for example been described in [12]. The observations that it makes and records from the environment and the success or failure of the actions that it attempts to execute can lead to the agent revising its beliefs. LIFF allows the agent to keep as much of its partial plan as is possible and to replan only those parts that are affected by the changes in its beliefs.

Another agent application that would benefit for our work is information integration. In this application a mediator agent receives the user query specified in some user-oriented high-level language. It then constructs a query plan translating the user query into a form understandable by the sources of information it has at its disposal. It contacts the sources for the necessary information, integrates their answers, and translates the answer into the language of the user. Such an information integration approach based on abductive logic programming and, in particular the IFF proof procedure, has been described in [21, 16]. Now, while the agent is constructing the query plan for the user query, it may receive information about semantic content or access availability changes to the information sources. Here again LIFF would allow the agent to accommodate these changes within its reasoning without having to discard all the work it has done in constructing the (possibly partial) query plan.

The paper is organised as follows. In section 2 we summarise necessary background on abductive logic programming and the abductive proof procedure IFF, in the propositional case. In section 3 we define the effect of (propositional) updates upon given abductive logic programs. In section 4 we give a simple example

of the working of our procedure LIFF, interleaving belief revision and reasoning. In section 5 we define LIFF, in the propositional case. In section 6 we give our results. In section 7 we conclude.

## 2 Preliminaries

*Abductive logic programming* is a general-purpose knowledge representation and reasoning framework that can be used for a number of applications and tasks [7, 10, 3, 13, 15, 14, 19, 20]. It relies upon a symbolic, logic-based representation of beliefs, for any given domain of application, via *abductive logic programs*, and the execution of logic-based reasoning engines, called *abductive proof procedures*, for reasoning with such representations. In the propositional case, an abductive logic program consists of

- A **logic program**, $P$, namely a set of if-rules of the form $Head \leftarrow Body$, where $Head$ is an atom and $Body$ is either $true$ (in which case we write the if-rule simply as $Head$) or a conjunction of (negations of) atoms. $P$ is understood as a (possibly incomplete) set of beliefs.
- A set of **abducible atoms**, $\mathcal{A}$. are assumed not to occur in the $Head$ of any if-rule in $P$. Abducible atoms can be used to "complete" the (beliefs held within the) logic program, subject to the satisfaction of the integrity constraints (see below).
- A set of **integrity constraints**, $I$, namely if-then-rules of the form $Condition \Rightarrow Conclusion$, where $Condition$ is either $true$ (in which case we write the if-then-rule simply as $Conclusion$) or a conjunction of (negations of) atoms, and $Conclusion$ is $false$ or an atom. The integrity constraints are understood as properties that must be "satisfied" by any "acceptable" extension of the logic program by means of atoms of abducibles, in the same way integrity constraints in databases are understood as properties that must be "satisfied" in every database state. We will assume that all integrity constraints in $I$ are satisfiable, namely there exists some extension of $P$ via sets of abducibles that satisfies them. [1]

In the sequel, $\overline{\mathcal{A}}$ will refer to the complement of the set $\mathcal{A}$ wrt the set of all predicates in the vocabulary of $P$, i.e. $\overline{\mathcal{A}}$ is the set of non-abducible predicates.

Both IFF and or new proof procedure LIFF rely upon the **completion** [2] of logic programs. The completion of a predicate $p$ defined, within a propositional logic program, by the if-rules

$p \leftarrow D_1, \ldots, p \leftarrow D_k$

is the **iff-definition**

$p \leftrightarrow D_1 \vee \ldots \vee D_k$

The left and right-hand side are called the *head* and the *body*, respectively. The completion of a predicate $p$ not defined in a given logic program is

[1] Otherwise, no explanation may possibly exist for any query, see below.

$p \leftrightarrow false$.

The **selective completion** $comp_S(P)$ of a logic program $P$ wrt a set of predicates $S$ in the vocabulary of $P$ is the union of the completions of all the predicates in $S$. Both IFF and LIFF use $comp_{\overline{\mathcal{A}}}(P)$.

Abductive proof procedures aim at computing "explanations" (or "abductive answers") for "queries", given an abductive logic program representing knowledge about some underlying domain. A **query** is a (possibly empty) conjunction of literals. Given an abductive logic program $\langle P, \mathcal{A}, I \rangle$, in the propositional case, an **explanation** (or **abductive answer**) for a query $Q$ is a (possibly empty) set $E$ of ground abducibles such that $P \cup E$ *entails* $Q$ and $P \cup E$ *satisfies* $I$. Various notions of entailment and satisfaction can be adopted, for example entailment and satisfaction could be entailment and consistency, respectively, in first-order, classical logic. In this paper, we propose an adaptation of IFF as the chosen abductive proof procedure, that we refer to as LIFF. We thus adopt truth wrt the 3-valued completion semantics of [11] as the underlying notion of entailment and satisfaction, inherited from [4].

IFF generates a **derivation** consisting of a **sequence of goals**, starting from an **initial goal**, which is the given query conjoined with the integrity constraints in $I$. Moreover, IFF computes explanations (referred to as **extracted answers**) for the given query extracting them from disjuncts in a goal in a derivation from the initial goal, such that no further inference rule can be applied to these disjuncts.

Goals in a derivation are obtained by applying **inference rules**. In the simplest case, goals derived by the IFF proof procedure are disjunctions of **simple goals**, which are conjunctions of the form

$A_1 \wedge \ldots \wedge A_n \wedge I_1 \wedge \ldots \wedge I_m \wedge D_1 \wedge \ldots \wedge D_k$

where $n, m, k \geq 0$, $n+m+k > 0$, the $A_i$ are atoms, the $I_i$ are **implications**, with the same syntax of integrity constraints, except that in addition to universally quantified variables, they can also contain existentially quantified or free variables, occurring elsewhere in the goal, and the $D_i$ are disjunctions of conjunctions of literals and implications. Implications are obtained by repeatedly applying the inference rules of the proof procedure to either integrity constraints in the given $\langle P, \mathcal{A}, I \rangle$ or to the result of rewriting negative literals $not\, A$ as $A \Rightarrow false$.

IFF assumes that the inference rules are applied in such a way that every goal in a sequence derived by the proof procedure is a *disjunction of simple goals*. (Note that every initial goal is a simple goal with the integrity constraints and the rewriting of negative literals as the only implications.) LIFF will make the same assumption.

We omit here the definition of the inference rules of IFF, as they can be reconstructed from the inference rules of LIFF, given in section 5, by dropping the labels. In the next section, we illustrate the behaviour of the IFF procedure and of LIFF with a simple example. The example illustrates the main difference between IFF and LIFF, namely the fact that the latter associates labels to literals and implications in goals, to be exploited when predicate definitions are updated.

## 3 Assimilating updates in the abductive logic program

In this paper we do not address the issue of belief revision. As mentioned in the section 1, the belief revision can be done in a number of different ways, for example through learning, or through a process of assimilation similar to the one described in [9]. However the belief revision is handled, at the end of the process, items (facts, rules, integrity constraints) will need to be added or deleted from the abductive logic program.

In this section we define how a given abductive logic program $\langle P, \mathcal{A}, I \rangle$ is revised after (any number of) the following kinds of updates:

- the addition to $P$ of facts (atoms) or if-rules,
- the deletion from $P$ of facts or if-rules,
- the addition to $I$ of if-then-rules.

Note that we ignore the deletion of if-then-rules from $I$, in order to keep the definition of LIFF as simple as possible. This deletion could be easily accommodated with the aid of appropriate additional labels in LIFF. We leave this as a future extension of our work. Note also that we do not directly allow for *modifying* if-rules and integrity constraints, as this can be achieved by suitable combinations of deletions and additions.

We will implicitly assume that the integrity constraints $I'$ in the revised abductive logic program $\langle P', \mathcal{A}', I' \rangle$, after the updates have taken place, are satisfiable by the revised logic program $P'$, as earlier in section 2.

There are a number of different ways that we can accommodate addition and deletion of facts and rules in an ALP. For example, we can decide whether or not a predicate that is originally abducible will, in effect, remain abducible after any updates. Similarly we can decide whether or not a predicate that is originally non-abducible will always be non-abducible, even if all its definitions are deleted from the logic program. LIFF is independent of these choices, and it can deal uniformly with any combination of these choices. Below, to set the scene, we, arbitrarily, make the concrete choice where abducible predicates always remain abducible and non-abducible predicates always remain non-abducible, no matter how many updates they are subjected to.

### 3.1 Addition of facts or if-rules

Let the update be $p \leftarrow B$, with $B \neq false$, [2] and with $B$ possibly $true$ (in which case $p$ is a ground fact). Below, we refer to $p \leftarrow B$ simply as $U$.

Let $\langle P, \mathcal{A}, I \rangle$ be the abductive logic program before the update. We will refer to $\mathcal{A}_B$ as the set of all predicates occurring in $B$ in the update but not already in the vocabulary of $\langle P, \mathcal{A}, I \rangle$.

We distinguish three cases:

[2] The addition of $p \leftarrow false$ is not allowed directly, but it can be achieved by deleting all if-rules with head $p$.

1. $p$ is an abducible in $\mathcal{A}$; then revising $\langle P, \mathcal{A}, I\rangle$ by $U$ gives $U(\langle P, \mathcal{A}, I\rangle)$=$\langle P', \mathcal{A}', I\rangle$, obtained as follows:
   - $P' = P \cup \{U\} \cup \{p \leftarrow p^*\}$,
   - $\mathcal{A}' = ((\mathcal{A} \cup \{p^*\}) - \{p\}) \cup \mathcal{A}_B$,

   where $p^*$ is a new predicate not occurring in the vocabulary of $\langle P, \mathcal{A}, I\rangle$;
2. $p$ is in the vocabulary of $\langle P, \mathcal{A}, I\rangle$ but is not abducible; then revising $\langle P, \mathcal{A}, I\rangle$ by $U$ gives $U(\langle P, \mathcal{A}, I\rangle)$=$\langle P', \mathcal{A}', I\rangle$, obtained as follows:
   - $P' = P \cup \{U\}$,
   - $\mathcal{A}' = \mathcal{A} \cup \mathcal{A}_B$;
3. $p$ is not in the vocabulary of $\langle P, \mathcal{A}, I\rangle$; then revising $\langle P, \mathcal{A}, I\rangle$ by $U$ gives $U(\langle P, \mathcal{A}, I\rangle)$=$\langle P', \mathcal{A}', I\rangle$, obtained as follows:
   - $P' = P \cup \{U\}$,
   - $\mathcal{A}' = \mathcal{A} \cup \mathcal{A}_B$.

Note that we assume that any predicate occurring in the body of any added rule but not already present in the vocabulary of the abductive logic program prior to the update is treated as a new abducible (in $\mathcal{A}_B$). This choice sees new undefined predicates as "open", and thus abducible.

Note that we do not allow the addition of new abducibles explicitly, from the outside. However, new abducibles are added as a by-product of adding definitions for abducibles (case 1 above) or definitions of predicates containing the new abducibles in the body, as predicates that did not occur before in the abductive logic program (cases 1-3 above).

### 3.2 Deletion of facts or if-rules

Let $\langle P, \mathcal{A}, I\rangle$ be the abductive logic program before the update. Let the update be the deletion of $p \leftarrow B \in P$ ($B$ possibly *true*, in which case $p$ is a ground fact), referred to simply as $U$. We will assume that $B$ is different from a $p^*$ atom, namely an atom introduced in case 1 in section 3.1. This means that predicates that are abducible in the initial abductive logic program always remain "abducible", in the sense that they can be assumed to hold by abduction (of atoms in $p^*$).

Revising $\langle P, \mathcal{A}, I\rangle$ by $U$ gives $U(\langle P, \mathcal{A}, I\rangle)$=$\langle P', \mathcal{A}', I\rangle$, obtained as follows:

- $P' = P - \{U\}$;
- $\mathcal{A}' = \mathcal{A} - \{a | a$ is a predicate occurring in $B$ and nowhere else in $P$ or $I\}$.

Note that it is arguable whether or not it is sensible whether to delete all integrity constraints that only mention the deleted abducibles. We assume here that such integrity constraints are not deleted.

### 3.3 Addition of integrity constraints

Let $\langle P, \mathcal{A}, I\rangle$ be the abductive logic program before the update. Let the update be the addition of $C \Rightarrow D$, referred to simply as $U$. Let $\mathcal{A}_U$ be the set of all predicates occurring in $C, D$ in the update but not already in the vocabulary of $\langle P, \mathcal{A}, I\rangle$.

Revising $\langle P, \mathcal{A}, I\rangle$ by $U$ gives $U(\langle P, \mathcal{A}, I\rangle)$=$\langle P, \mathcal{A}', I'\rangle$, obtained as follows:

- $I' = I \cup \{U\}$;
- $\mathcal{A}' = \mathcal{A} \cup \mathcal{A}_U$.

### 3.4 Multiple updates

Here we define the effect of multiple updates upon an initially given abductive logic program $\langle P, \mathcal{A}, I\rangle$. Let the sequence of updates be $U_1, \ldots, U_n$, with $n > 1$. Then, $U_1, \ldots, U_n$ applied to $\langle P, \mathcal{A}, I\rangle$ gives

$$U_n \circ U_{n-1} \circ \ldots \circ U_1(\langle P, \mathcal{A}, I\rangle) = U_n(U_{n-1}(\ldots(U_1(\langle P, \mathcal{A}, I\rangle)))).$$

Later on, in section 5.2, we will allow empty updates to occur in sequences of updates. In this setting, an empty update stands for no update at all. However, empty updates are useful in the formal definition of LIFF derivations, to accommodate updates at the correct place during derivations.

## 4 An illustrative example

In this section we illustrate the application of LIFF and its relationship to IFF via a concrete example.

Given the abductive logic program $\langle P, \mathcal{A}, I\rangle$ with

$P$: $p \leftarrow q$
$\quad p \leftarrow a$
$\quad q \leftarrow d \wedge c$
$\quad n \leftarrow e$
$\mathcal{A}$: $a, b, c, d, e$
$I$: $\quad a \Rightarrow b$
$\quad c \wedge n \Rightarrow false$

and a query $p$, IFF would derive the following sequence of goals:

$p \wedge I$
$\quad$ by unfolding $p$ with $Comp_{\overline{\mathcal{A}}}(P)$:
$(q \vee a) \wedge I$
$\quad$ by splitting (distributing $\vee$ over $\wedge$):
$[q \wedge I] \vee [a \wedge I]$
$\quad$ by unfolding $q$ with $Comp_{\overline{\mathcal{A}}}(P)$:

$[d \wedge c \wedge I] \vee [a \wedge I]$
by propagation (with $a \Rightarrow b \in I$ in the second disjunct):
$[d \wedge c \wedge I] \vee [a \wedge b \wedge I]$
by propagation (with $c \wedge n \Rightarrow false \in I$ in the first disjunct):
$[d \wedge c \wedge [n \Rightarrow false] \wedge I] \vee [a \wedge b \wedge I]$
by unfolding $n$ with $Comp_{\overline{\mathcal{A}}}(P)$:
$[d \wedge c \wedge [e \Rightarrow false] \wedge I] \vee [a \wedge b \wedge I]$

From the final goal in this sequence of goals, IFF would extract the explanations $\{d, c\}$ (from the first disjunct/simple goal) and $\{a, b\}$ (from the second disjunct/simple goal) for the given query $p$. From the goal before last in this sequence, IFF would extract the explanation $\{a, b\}$ (from the second simple goal). No explanation could be extracted from the first simple goal in that goal, as inference rules can still be applied to it.

Given the same $\langle P, \mathcal{A}, I\rangle$ and query, LIFF would derive the following sequence of goals with labels, with updates as indicated. Note that a label $\{\langle \mathtt{X_1}; \mathtt{Y_1}\rangle, \ldots \langle \mathtt{X_n}; \mathtt{Y_n}\rangle\}$ associated with an atom or an implication $Z$ in a goal, with $\mathtt{X_i}$ a predicate, intuitively indicates that "if the definition of $\mathtt{X_i}$ is updated (by adding or deleting if-rules for it), then $Z$ should be replaced by $\mathtt{Y_i}$". Also, $\emptyset$ stands for the empty set of sentences.

$p \wedge I$
by unfolding and splitting:
$[q : \{\langle \mathtt{p}; \mathtt{p}\rangle\} \wedge I] \vee [a : \{\langle \mathtt{p}; \mathtt{p}\rangle\} \wedge I]$
by unfolding:
$[d : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge c : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge I] \vee [a : \{\langle \mathtt{p}; \mathtt{p}\rangle\} \wedge I]$
by propagation:
$[d : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge c : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge I] \vee [a : \{\langle \mathtt{p}; \mathtt{p}\rangle\} \wedge b : \{\langle \mathtt{p}; \emptyset\rangle, \langle \mathtt{a}; \emptyset\rangle\} \wedge I]$
by propagation:
$[d : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge c : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge [n \Rightarrow false] : \{\langle \mathtt{p}; \emptyset\rangle, \langle \mathtt{c}; \emptyset\rangle, \langle \mathtt{q}; \emptyset\rangle\} \wedge I] \vee$
$[a : \{\langle \mathtt{p}; \mathtt{p}\rangle\} \wedge b : \{\langle \mathtt{p}; \emptyset\rangle, \langle \mathtt{a}; \emptyset\rangle\} \wedge I]$
by unfolding:
$[d : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge c : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge$
$[e \Rightarrow false] : \{\langle \mathtt{p}; \emptyset\rangle, \langle \mathtt{q}; \emptyset\rangle, \langle \mathtt{c}; \emptyset\rangle, \langle \mathtt{n}; \mathtt{n} \Rightarrow \mathtt{false} : \{\langle \mathtt{p}; \emptyset\rangle, \langle \mathtt{q}; \emptyset\rangle, \langle \mathtt{c}; \emptyset\rangle\}\rangle\} \wedge I] \vee$
$[a : \{\langle \mathtt{p}; \mathtt{p}\rangle\} \wedge b : \{\langle \mathtt{p}; \emptyset\rangle, \langle \mathtt{a}; \emptyset\rangle\} \wedge I]$

Note that the same answers could be extracted from the two simple goals here as in the case of IFF. However, if, before answer extraction takes place, the definition of $p$ is updated, then LIFF would generate, as the next goal in the sequence,

$[p \wedge p \wedge I] \vee [p \wedge I]$

which collapses to the initial goal $p \wedge I$. So, in this case, LIFF would start the reasoning process from scratch, as would happen if applying IFF (or any other conventional abductive proof procedure).

Suppose $n$ is updated instead. Then, our procedure would obtain

$[d : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge c : \{\langle \mathtt{p}; \mathtt{p}\rangle, \langle \mathtt{q}; \mathtt{q}\rangle\} \wedge [n \Rightarrow false] : \{\langle \mathtt{p}; \emptyset\rangle, \langle \mathtt{q}; \emptyset\rangle, \langle \mathtt{c}; \emptyset\rangle\} \wedge I] \vee$
$[a : \{\langle \mathtt{p}; \mathtt{p}\rangle\} \wedge b : \{\langle \mathtt{p}; \emptyset\rangle, \langle \mathtt{a}; \emptyset\rangle\} \wedge I]$

thus avoiding starting the reasoning process from scratch.

## 5 The LIFF proof procedure

We first define the notion of label.

**Definition 1.** *A* **label** *(for an atom or implication) is a set*

$$\{\langle \mathtt{X_1}; \mathtt{Y_1}\rangle, \ldots, \langle \mathtt{X_n}; \mathtt{Y_n}\rangle\}$$

*$n \geq 0$, with each $\mathtt{X_i}$ a predicate and each $\mathtt{Y_i}$ either*

- *the empty sentence $\emptyset$, or*
- *an atom (possibly with a label), or*
- *an implication (possibly with a label).*

Note that labels can be empty (if $n = 0$). Below (and in the earlier example in section 4), absence of a label corresponds to an empty label. Intuitively, each $\langle \mathtt{X_i}; \mathtt{Y_i}\rangle$ in a label for $Z$ indicates what $Z$ should become if any update is made upon $\mathtt{X_i}$, namely if a new definition is added to $P$ for $\mathtt{X_i}$ or if an existing definition for $\mathtt{X_i}$ is deleted from $P$.

We will refer to labelled atoms/implications simply as atoms/implications. Moreover, the terminology of goals and simple goals will carry through in the presence of labels.

### 5.1 LIFF inference rules

Given an abductive logic program $\langle P, \mathcal{A}, I\rangle$ and an initial query $Q$, we will define *LIFF derivations* for $Q$ in terms of sequences of goals, $G_1, \ldots, G_n$, such that $G_1 = G \wedge I$ [3] and each $G_{i+1}$ is obtained from the previous goal $G_i$ either by enforcing an update (see section 5.2) or by application of one of the inference rules below. The intuition behind each of the labels is to identify the components that have contributed to the derivation of the new element(s).

**Unfolding:** given an atom $p : \mathtt{l}$ and $p \leftrightarrow D_1 \vee \ldots \vee D_n$ in $comp_{\overline{\mathcal{A}}}(P)$

- if the atom is a conjunct of a simple goal in $G_i$, then $G_{i+1}$ is $G_i$ with the atom replaced by $(D_1 \vee \ldots \vee D_n) : \mathtt{l} \cup \{\langle \mathtt{p}; \mathtt{p} : \mathtt{l}\rangle\}$;

[3] In the sequel, $(A_1 \wedge \ldots \wedge A_k) : \mathtt{l}$ $(k > 1)$ stands for $A_1 : \mathtt{l} \wedge \ldots \wedge A_k : \mathtt{l}$ and $(D_1 \vee \ldots \vee D_n) : \mathtt{l}$ stands for $D_1 : \mathtt{l} \vee \ldots \vee D_n : \mathtt{l}$, where $\mathtt{l}$ is a label.

– if $p$ is a conjunct $L_i$ in the body of an implication
$[L_1 \wedge \ldots \wedge L_m \Rightarrow A] : \mathtt{l}'$, $m \geq 1$
which is a conjunct of a simple goal of $G_i$, then $G_{i+1}$ is $G_i$ with the implication replaced by the conjunction
$[L_1 \wedge \ldots \wedge D_1 \wedge \ldots \wedge L_m \Rightarrow A] : \mathtt{l}'' \wedge \ldots \wedge$
$[L_1 \wedge \ldots \wedge D_n \wedge \ldots \wedge L_m \Rightarrow A] : \mathtt{l}''$,
where $\mathtt{l}'' = \mathtt{l}' \cup \{\langle \mathtt{p}; \mathtt{L_1} \wedge \ldots \wedge \mathtt{L_m} \Rightarrow \mathtt{A} : \mathtt{l}' \rangle\}$

**Propagation:** given an atom $p : \mathtt{l}$ and an implication
$[L_1 \wedge \ldots \wedge L_m \Rightarrow A] : \mathtt{l}'$, $m \geq 1$
with $L_i = p$, for some $1 \leq i \leq m$, both conjuncts of the same simple goal in $G_i$, then, if the implication
$[L_1 \wedge \ldots L_{i-1} \wedge L_{i+1} \wedge \ldots \wedge L_m \Rightarrow A] : \{\langle \mathtt{p}; \emptyset \rangle\} \cup \{\langle \mathtt{q}; \emptyset \rangle | \langle \mathtt{q}; \mathtt{r} \rangle \in \mathtt{l} \cup \mathtt{l}'\}$
is not already a conjunct of the simple goal, then $G_{i+1}$ is $G_i$ with the new implication conjoined to the simple goal.

**Negation elimination:** given an implication $not\, A_1 \wedge \ldots \wedge not\, A_m \Rightarrow A : \mathtt{l}$ which is a conjunct of a simple goal in $G_i$, then $G_{i+1}$ is $G_i$ with the implication replaced by the disjunction $[A \vee A_1 \vee \ldots \vee A_m] : \mathtt{l}$. [4]

**Logical simplification** replaces:
– $[B : \mathtt{l} \vee C : \mathtt{l}'] \wedge E : \mathtt{l}''$ by
$[B : \mathtt{l} \wedge E : \mathtt{l}''] \vee [C : \mathtt{l}' \wedge E : \mathtt{l}'']$ (**splitting**);
– $B : \mathtt{l} \wedge B : \mathtt{l}$ by
$B : \mathtt{l}$ where B is an atom or an implication;
– $B : \mathtt{l} \vee B : \mathtt{l}$ by
$B : \mathtt{l}$ where B is an atom or an implication.

Note that we are not including some simplification steps present in IFF, e.g. $A \wedge false \equiv false$. Such simplification steps only affect the "efficiency" of IFF, rather than its correctness, in that they prevent "working" on a disjunct in a goal from which no answer can ever be extracted (e.g. for $A \wedge false \equiv false$, due to the inconsistency of that disjunct). In the case of LIFF, though, these inference steps could undermine correctness, as they would prevent us from keeping track of intermediate steps that might be affected by updates (e.g. $A \wedge false$, if what lead to $false$ is updated).

Note also that we are not allowing "merging" of identical conjuncts but with different labels, as this would prevent us from incorporating all updates correctly. This is illustrated by the following example. Assume $\langle P, \mathcal{A}, I \rangle$ with

$P$: $p \leftarrow a$
$\mathcal{A}$: $\{a\}$
$I$: $a \Rightarrow a$

[4] In [4], negation elimination is defined as follows: $not\, A_1 \wedge Rest \Rightarrow A$ is replaced by $Rest \Rightarrow A \vee A_1$. Operationally, our definition (ignoring the labels) is equivalent to the one in [4].

then, given the query $p$, the following sequence of goals is generated by IFF:

$p \wedge I$
  by unfolding $p$ with $Comp_{\overline{\mathcal{A}}}(P)$:
$a \wedge I$
  by propagation:
$a \wedge I \wedge a$
  by simplification
$a \wedge I$

from which the answer $\{a\}$ can be extracted. The following is the LIFF counterpart of this derivation:

$p \wedge I$
  by unfolding $p$ with $Comp_{\overline{\mathcal{A}}}(P)$:
$a : \{\langle \mathtt{p}; \mathtt{p} \rangle\} \wedge I$
  by propagation:
$a : \{\langle \mathtt{p}; \mathtt{p} \rangle\} \wedge I \wedge a : \{\langle \mathtt{p}; \emptyset \rangle, \langle \mathtt{a}; \emptyset \rangle\}$

from which the answer $\{a\}$ can be extracted, as we will see in section 5.3. If we allowed the merging of the two occurrences of $a$ and their labels to obtain:

$a : \{\langle \mathtt{p}; \mathtt{p} \rangle, \langle \mathtt{a}; \emptyset \rangle\} \wedge I$

the resulting label would give an incorrect indication as to how the goal should be revised if the definition of $a$ weer revised.

### 5.2 LIFF derivations

In LIFF derivations, the application of inference rules (as given in section 5.1) is interleaved with the assimilation of updates within the abductive logic program (as given in section 3) and the application to these updates to goals, defined as follows:

**Definition 2.** *Given a goal $G$ and an update $U$, the* **updated goal wrt $G$ and $U$** *is a goal $G'$ obtained as follows:*

- *if $U$ is the addition of a fact or rule $p \leftarrow B$, then*
  - *if $p$ is not abducible, then $G'$ is $G$ where every conjunct with label containing $\langle p; Q \rangle$ is replaced by $Q$;*
  - *if $p$ is abducible, then $G'$ is identical to $G$.*
- *if $U$ is the deletion of a fact or rule $p \leftarrow B$, then $G'$ is $G$ where every conjunct with label containing $\langle p; Q \rangle$ is replaced by $Q$;*
- *if $U$ is the addition of an integrity constraint $X$ then $G'$ is $G$ where $X$ is conjoined to each simple goal.*

**Definition 3.** *Given a query $Q$, an abductive logic program $\langle P, \mathcal{A}, I\rangle$, and a sequence of updates $U_1, \ldots, U_n$, $n > 0$, a* **LIFF derivation** *is a sequence of tuples:*

$$(G_1, U_1, R_1, ALP_1), \ldots, (G_n, U_n, R_n, ALP_n)$$

*where, for each $1 \leq i \leq n$,*

- *$G_i$ is a goal,*
- *$U_i$ is a (possibly empty) update,*
- *$R_i$ is (the application of) an inference rule or is empty,*
- *exactly one of $U_i$ and $R_i$ is non-empty,*
- *$ALP_i$ is an abductive logic program,*

*and $G_1 = Q \wedge I$, $ALP_1 = \langle P, \mathcal{A}, I\rangle$, and, for each $1 < i \leq n$,*

- *if $U_{i-1}$ is empty then*
  - *$G_i$ is obtained from $G_{i-1}$ by applying $R_{i-1}$*
  - *$ALP_i = ALP_{i-1}$*
- *if $U_{i-1}$ is non-empty then*
  - *$G_i$ is the updated goal wrt $G_{i-1}$ and $U_{i-1}$*
  - *$ALP_i = U_{i-1}(ALP_{i-1})$.*

*The* **end goal** *and* **end abductive logic program** *in a LIFF derivation are $G$ and $ALP$, respectively, such that:*

- *if $U_n$ is empty then*
  - *$G$ is obtained from $G_n$ by applying $R_n$*
  - *$ALP = ALP_n$*
- *if $U_n$ is non-empty then*
  - *$G$ is the updated goal wrt $G_n$ and $U_n$*
  - *$ALP = U_n(ALP_n)$.*

Note that for goals in LIFF derivations to be guaranteed to be disjunctions of simple goals, it is sufficient that every step of unfolding and negation elimination is followed by a step of splitting. We will assume this is the case in all derivations.

We will refer to a LIFF derivation whose updates are all empty as a **static LIFF derivation**.

### 5.3 Successful LIFF derivations and extracted answers

In this section we formalise the notion of answer extraction for LIFF, by adapting the notion of answer extraction for IFF to take into account labels.

Given a LIFF derivation for a query $Q$, abductive logic program $\langle P, \mathcal{A}, I\rangle$ and a given sequence of updates, let $G$ be the end goal of the derivation. Let $D$ be a disjunct of $G$:

- if no inference rule can be applied to $D$, then this is called **conclusive**;
- if $false : \mathtt{l}$, for any label $\mathtt{l}$, is a conjunct in $D$, then this is called **failed**;

– if $D$ is conclusive and not failed then it is called **successful**.

Then, a derivation is a **successful derivation** iff there exists a successful disjunct $D$ in $G$.

Given such a successful derivation, an **answer extracted from** $D$ is the set of all abducible atoms, without labels, in $D$. Basically, our notion of extracted answer is the same as that of IFF, but ignoring the labels.

## 6 Soundness of the LIFF proof procedure

**Theorem 1.** *(*Soundness*) Let us assume that $E$ is an answer extracted from a successful LIFF-derivation wrt $Q$ and $\langle P, \mathcal{A}, I\rangle$, after any number of updates. Let $\langle P', \mathcal{A}', I'\rangle$ be the abductive logic program resulting after the updates. Then $E$ is an explanation for $Q$, wrt $\langle P', \mathcal{A}', I'\rangle$.*

The proof of this theorem relies upon the following lemmas and the correctness of the IFF proof procedure.

**Lemma 1.** *Every non-static LIFF derivation wrt $Q$, $\langle P, \mathcal{A}, I\rangle$, $U_1, \ldots, U_m$, with end goal $G$ and end abductive logic program $\langle P', \mathcal{A}', I'\rangle$, can be mapped onto a static LIFF derivation wrt $Q$, $\langle P', \mathcal{A}', I'\rangle$, ending at (a simplified version of) $G$.*

**Lemma 2.** *Every static LIFF derivation wrt $Q$, $\langle P, \mathcal{A}, I\rangle$, can be mapped onto an IFF derivation wrt $Q$.*

## 7 Conclusions and future work

In this paper we have proposed a dynamic abductive logic programming proof procedure, called LIFF. LIFF modifies the existing proof procedure IFF by adding labels to goals. The labels keep track of dependencies amongst the items (atoms and implications) in goals and between the items and the logic program. By keeping track of these dependencies, after updating the abductive logic program, LIFF can keep parts of the reasoning that have not been affected by the updates and determine how best to replace those parts that have been affected. It thus allows reasoning in dynamic environments and contexts without having to discard earlier reasoning when changes occur. We have considered updates consisting of addition and deletion of facts and rules and addition of integrity constraints.

We have not considered updates in the form of deletion of integrity constraints. These could be easily accommodated by adopting additional labels, associated to items (implications and atoms) in goals, to keep dependencies with any integrity constraints that have originated those items. It would be interesting also to consider the consequences of making different choices, for example enlarging the set of abducibles without adding new rules or integrity constraints.

LIFF manages to do most of its saving of earlier work when the definitions of relatively lower level predicates are modified (by additions and deletions). This kind of update is particularly prominent in the case when we use abductive logic programming and agent-based techniques for information integration [21, 16]. In such an application the higher level predicates are user-level and auxiliary predicates, and the lower level predicates are related to information sources. Changes in various aspects of the information sources, for example their content or availability, amount to updating these lower level predicates.

Our work on LIFF shares the same objectives as that of [5]. However whereas we adapt IFF for abductive logic programming, they adapt conventional SLDNF for ordinary logic programs. We also share some of the objectives of [17, 18]. In their work on speculative computation they allow reasoning to proceed with default values for specific facts and then accommodate updates which explicitly replace the default values, attempting to keep some of the earlier computation. They, however, use SLD derivation for Horn theories (no negation in rules and no integrity constraints). They keep a form of dependency, but it is at a much coarser level compared to ours. They keep dependency information for the whole goal, rather than for the individual items in goals, thus allowing fewer savings in computations.

We have described LIFF and a soundness result for propositional cases. Work is currently in progress for the predicate case, including the additional rules of factoring, case analysis and equality rewriting. Completeness results with respect to completeness results of IFF are subject of future work. Another interesting line of thought is to explore how we can parameterise the extent of savings in reasoning according to what is updated, to identify optimal and non-optimal cases. It would also be worth exploring how this work scales up to substantial applications that would require large knowledge bases and frequent updates.

**Acknowledgements**

This work has been supported by the IST programme of the EC, FET under the IST-2001-32530 SOCS project, within the GC proactive initiative.

## References

1. C. E. Alchourron, P. Gardenfors, and D. Makinson. On the logic of theory change: Partial meet functions for contraction and revision. *Journal of Symbolic Logic*, 50:510–530, 1985.
2. K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
3. P. Dell'Acqua, F. Sadri, and F. Toni. Combining introspection and communication with rationality and reactivity in agents. In J. Dix, L. Fariñas del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence, European Workshop, JELIA'98, Dagstuhl, Germany, October 12–15, 1998, Proceedings*, volume 1489 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 1998.

4. T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.
5. H. Hayashi. Replanning in robotics by dynamic sldnf. In *Proc. Workshop on Scheduling and Planning meet Real-time Monitoring in a Dynamic and Uncertain World*, 1999.
6. H. Hayashi, K. Cho, and A. Ohsuga. Integrating planning, action execution, knowledge updates and plan modifications via logic programming. In *Proc. ICLP2002*, 2002.
7. A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.
8. A.C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In *Proc. ECAI-2004*, 2004.
9. R. A. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.
10. R. A. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3/4):391–419, 1999.
11. K. Kunen. Negation in logic programming. In *Journal of Logic Programming*, volume 4, pages 289–308, 1987.
12. P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Planning partially for situated agents. In João Leite and Paolo Torroni, editors, *5th Workshop on Computational Logic in Multi-Agent Systems (CLIMA V)*, September 29–30 2004.
13. F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In E. Lamma and P. Mello, editors, *AI*IA'99: Advances in Artificial Intelligence, Proceedings of the 6th Congress of the Italian Association for Artificial Intelligence, Bologna*, number 1792 in Lecture Notes in Artificial Intelligence, pages 49–60. Springer-Verlag, 2000.
14. F. Sadri, F. Toni, and P. Torroni. An abductive logic programming architecture for negotiating agents. In S. Greco and N. Leone, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA)*, volume 2424 of *Lecture Notes in Computer Science*, pages 419–431. Springer-Verlag, 2002.
15. F. Sadri, F. Toni, and P. Torroni. Dialogues for negotiation: agent varieties and dialogue sequences. In *Intelligent Agents VIII: 8th International Workshop, ATAL 2001, Seattle, WA, USA, Revised Papers*, volume 2333 of *Lecture Notes in Artificial Intelligence*, pages 405–421. Springer-Verlag, 2002.
16. F. Sadri, F. Toni, and I. Xanthakos. An abductive framework for semantic integration of information. Technical report, Department of Computing, Imperial College London, 2004.
17. K. Satoh, K. Inoue, K. Iwanuma, and C Sakama. Speculative computation by abduction under incomplete communication environments. In *Proc. ICMAS2000*, pages 263–270, 2000.
18. K. Satoh and K. Yamamoto. Speculative computation with multi-agent belief revision. In *Proc. AAMAS2002*, pages 897–904, 2002.
19. F. Toni. Automated information management via abductive logic agents. *Journal of Telematics and Informatics*, 18(1):89–104, 2001.
20. F. Toni and K. Stathis. Access-as-you-need: a computational logic framework for flexible resource access in artificial societies. In *Proceedings of the Third International Workshop on Engineering Societies in the Agents World (ESAW'02)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2002.
21. I. Xanthakos. *Semantic Integration of Information by Abduction*. PhD thesis, Imperial College London, 2003.

# A connection between Similarity Logic Programming and Gödel Modal Logic

Luciano Blandi[1], Lluis Godo[2], and Ricardo O. Rodríguez[3]

[1] DMI-Universita' di Salerno, Via Ponte Don Melillo, 84084 Fisciano (SA), Italy
lblandi@unisa.it
[2] IIIA - CSIC, Campus UAB, 08193 Bellaterra, Spain
godo@iiia.csic.es
[3] Dpto. de Computación, Fac. Ciencias Exactas y Naturales, Universidad de Buenos Aires, Ciudad de Buenos Aires, Argentina
ricardo@dc.uba.ar

**Abstract.** In this paper we relate two logical similarity-based approaches to approximate reasoning. One approach extends the framework of (propositional) classical logic programming by introducing a similarity relation in the alphabet of the language that allows for an extended unification procedure. The second approach is a many-valued modal logic approach where $\Diamond p$ is understood as *approximately p*. Here, the similarity relations are introduced at the level of the Kripke models where possible worlds can be similar to some extent. We show that the former approach can be expressed inside the latter.
**Keywords:** Similarity, Logic Programming, Gödel Logic.

## 1 Introduction and previous works

One of the goals of a variety of approximate reasoning models is to cope with inference patterns more flexible than those of classical reasoning. Among them, the similarity-based reasoning based on the notion of similarity relation [16] which provides a way to manage alternative instances of an entity that can be considered "equal" with a given degree expressed by a value in [0,1].

Two main similarity-based approaches to approximate reasoning are put into relation in this paper. The first one is an approach based on introducing a similarity relation $\mathcal{R}$ (in the sense of a reflexive, symmetric and min-transitive fuzzy relation) in the set of object names in a language of classical propositional Logic Programming. Following [15], [2] and [14], we consider inferences that may be approximated by allowing the antecedent clauses of a rule to match its premises only approximately. In particular, the classical SLD Resolution is modified in order to overcome failure situations in the unification process if the entities involved in the matching have a non-zero similarity degree. Such a procedure allows us to compute numeric values belonging to the interval [0,1], named *approximation degrees*, which provide an approximation measure of the obtained solutions. This framework, which we shall call *Similarity Propositional Logic Programming* (SPLP), is the propositional version of that one proposed by Sessa in [14] which is based upon a first order language. In [7] we find the first proposal to introduce similarity in the frame of the declarative paradigm of Logic Programming. Logic programs on function-free languages are considered and approximate and imprecise information are represented by introducing a similarity relation between constant and predicate symbols. Two transformation techniques of logic programs are defined. In the underlying logic, the inference rule (Resolution rule) as well as the usual crisp representation of the considered universe are not modified. It allows to avoid both the introduction of weights on the clauses, and the use of fuzzy sets as elements of the language. The semantic equivalence between the two inference processes associated to the two kind of transformed programs has been proved by using an abstract interpretation technique. Moreover, the notion of fuzzy least Herbrand model has been introduced. In [13] the generalization of this approach to the case of programs

with function symbols is provided by introducing the general notion of *structural translation* of languages. In [14] the operational counterpart of this extension is faced by introducing a modified SLD Resolution procedure which allows us to perform these kinds of extended computations exploiting the original logic program, without any preprocessing steps in order to transform the given program. [10] presents an extended PROLOG interpreter, named *SiLog*, which implements this inference procedure. Finally, for completeness sake, we also cite [6] where a first and different (it takes into account substitutions of variable with sets of symbols) generalized unification algorithm based on similarity has been proposed.

The second approach is based in introducing a similarity relation $S$ in the set of interpretations or possible worlds. This kind of approach was started by Ruspini [12] by proposing a similarity-based semantics for fuzzy logic, trying to capture inference patterns like the so-called *generalized modus ponens*. The basic idea of this "similarity approach" is that the degree of truthlikeness of a sentence $\varphi$ depends on the similarities between the states of affairs allowed by $\varphi$ and the true state of the world. Intuitively speaking, a statement is truthlike if it is "like the truth" or "similar to the truth" but it does not have to be true or even probable. The idea is to attach to each proposition $\varphi$ of a given basic language $\mathcal{L}$ a new "fuzzy" propositon $\Diamond\varphi$ read as "*approximately-*$\varphi$". This leads to deal with degrees of truth (how close is $\varphi$ to truth = how true is *approximately-*$\varphi$) but, unlike to most systems of many-valued logic, this notion is not compositional (functional) and it is modelled by a logical modality. In our logic, we rely on a system of modal logic related to similarity-based reasoning on *fuzzy* propositions. Technically, we combine many-valued logic (to model fuzziness) and modal logic (to model similarity). Therefore we propose a modal fuzzy logic with semantics based on Kripke structures where the accesibility relations are fuzzy similarity relations measuring how similar are the possible worlds. This will result on a many-valued modal system, a many-valued counterpart of the classical S5 modal system, with many-valued similarity-based Kripke model semantics. In a previous work [9] a modal logic over the Rational Pavelka logic has been defined. Instead, we use a rational modal logic based upon the many-valued Gödel propositional Logic, named *Rational Gödel similarity-based S5 modal logic* (RGS5$_\Diamond$).

The main difference between these approaches is that in SPLP the similarity is defined between symbols in the alphabet of the language, i.e. it is exploited at a syntactic-level, whereas in RGS5$_\Diamond$ the similarity is defined in the set of the interpretations, i.e. it is exploited at a semantic-level. Nevertheless, both approaches can be put into relation.

The paper is organized as follows. After this introduction we survey in Section 2 SPLP and RGS5$_\Diamond$. The third section is devoted to study in detail the above mentioned correspondences between the two approaches. The last section contains some concluding remarks.

## 2 Two similarity-based approaches to approximate reasoning

This section briefly outlines those elements of SPLP and RGS5$_\Diamond$ which we shall take for granted in what follows, and at the same time explains some of the terminology which we shall use throughout the paper. Some common concepts follow.

**Definition 1.** *A* similarity *on a domain $\mathcal{U}$ is a fuzzy relation $\mathcal{R} : \mathcal{U} \times \mathcal{U} \to [0,1]$ in $\mathcal{U}$ such that the following properties hold*

*i)* $\mathcal{R}(x,x) = 1\ \forall x \in \mathcal{U}$ *(reflexivity)*

*ii)* $\mathcal{R}(x,y) = \mathcal{R}(y,x)\ \forall x,y \in \mathcal{U}$ *(symmetry)*

*iii)* $\mathcal{R}(x,z) \geq \mathcal{R}(x,y) \wedge \mathcal{R}(y,z)\ \forall x,y,z \in \mathcal{U}$ *(transitivity)*

*We say that $\mathcal{R}$ is* strict *if the following implication is also verified*

*iv)* $\mathcal{R}(x,z) = 1 \ \Rightarrow\ x = z$.

The mathematical notion of Similarity relation is a many valued extension of the equality, indeed equal elements have similarity degree 1 and completely different elements have similarity degree 0, and it is widely exploited in any context where a weakening of the equality constraint is useful.

### 2.1 Similarity relations on symbols: SPLP

Let $Const$ be a set of propositional constants and let $\mathcal{L}$ be the usual classical propositional language.

We briefly recall that a logic program $P$ on $\mathcal{L}$ is a conjunction of definite clauses of $\mathcal{L}$, denoted as $q \leftarrow p_1, \ldots, p_n$, $n \geq 0$, and a goal is a negative clause, denoted with $\leftarrow q_1, \ldots, q_n$, $n \geq 1$, where the symbol "," that separates the propositional constants has to be interpreted as conjunction, and $p_1, \ldots, p_n, q, q_1, \ldots, q_n \in Const$. A *SPLP-program* is a pair $(P, \mathcal{R})$, where $P$ is a logic program defined on $\mathcal{L}$ and $\mathcal{R}$ is a similarity on $Const$. Given $P$, the least Herbrand model of $P$ is given by $M_P = \{p \in Const \mid P \vDash p\}$, where $\vDash$ denotes classical logical entailment. $M_P$ is equivalent to the corresponding procedural semantics of $P$, defined by considering the SLD Resolution. In the classical case, a mismatch between two propositional constant names causes a failure of the unification process. Then, it is rather natural to admit a more flexible unification in which the syntactical identity is substituted by a Similarity $\mathcal{R}$ defined on $Const$. The modified version of the SLD Resolution, which we shall call *Similarity-based SLD Resolution*, exploits this simple variation in the unification process. The basic idea of this procedure for first order languages has been outlined in [8]. The following definitions formalize these ideas in the case of propositional languages.

**Definition 2.** *Let $\mathcal{R} : Const \times Const \rightarrow [0,1]$ be a similarity and $p, q \in Const$ be two propositional constants in a propositional language $\mathcal{L}$. We define the* unification-degree *of $p$ and $q$ with respect to $\mathcal{R}$ the value $\mathcal{R}(p,q)$. $p$ and $q$ are* $\lambda$-unifiable *if $\mathcal{R}(p,q) = \lambda$ with $\lambda > 0$, otherwise we say that they are* not unifiable.

**Definition 3.** *Given a similarity $\mathcal{R} : Const \times Const \rightarrow [0,1]$, a program $P$ and a goal $G_0$, a* similarity-based SLD derivation of $P \cup \{G_0\}$, *denoted by*

$$G_0 \Rightarrow_{C_1,\alpha_1} G_1 \Rightarrow \cdots \Rightarrow_{C_k,\alpha_k} G_k$$

*consists of a sequence $G_0, G_1, \ldots, G_k$ of negative clauses, together with a sequence $C_1, C_2, \ldots, C_k$ of clauses from $P$ and a sequence $\alpha_1, \alpha_2, \ldots, \alpha_k$ of values in* [0,1], *such that for all $i \in \{1, \ldots, k\}$, $G_i$ is a resolvent of $G_{i-1}$ and $C_i$ with unification degree $\alpha_i$. The approximation degree of the derivation is $\alpha = inf\{\alpha_1, \ldots, \alpha_k\}$. If $G_k$ is the empty clause $\perp$, for some finite k, the derivation is called a* Similarity-based SLD refutation, *otherwise it is called* failed.

It is easy to see that when the similarity $\mathcal{R}$ is the identity, the previous definition provides the classical notion of SLD refutation. The values $\alpha_i$ can be considered as constraints that allow the success of the unification processes. Then, it is natural to consider the best unification degree that allows us to satisfy all these constraints. In general, an answer can be obtained with different SLD refutations and different approximation degrees, then the maximum $\alpha$ of these values characterizes the best refutations of the goal. In particular, a refutation with approximation-degree 1 provides an exact solution. Let us stress that $\alpha$ belongs to the set $\lambda_1, \lambda_2, \ldots$ of the possible similarity values in $\mathcal{R}$. In the sequel, we assume the Leftmost selection rule whenever Similarity-based SLD Resolution is considered. However, all the presented results can be analogously stated for any selection rule that does not depend on the propositional constant names and on the history of the derivation [1]. Similarity-based SLD Resolution provides a characterization of the fuzzy least Herbrand model $M_{P,\mathcal{R}}$ for $(P, \mathcal{R})$ defined in [7], as stated by the following result.

**Proposition 1.** *Let a similarity $\mathcal{R}$ and a logic program $P$ on a propositional language $\mathcal{L}$ be given. For any $q \in Const$, $M_{P,\mathcal{R}}(q) = \alpha > 0$ if and only if $\alpha$ is the maximum value in* (0,1] *for which there exists a Similarity-based SLD refutation for $P \cup \{\leftarrow q\}$ with approximation degree $\alpha$.*

Intuitively, the degree of membership $M_{P,\mathcal{R}}(q)$ of an atom $q$ is given by the best "tolerance" level $\alpha \in (0,1]$ which allows us to prove $q$ exploiting the Similarity-based SLD Resolution on $P \cup \{\leftarrow q\}$.

Finally, let us remember the following relation between the classical least Herbrand Model of a program $P$ and the fuzzy generalization of this notion.

**Proposition 2.** *Let $P$ be a logic program on a propositional language $\mathcal{L}$ with a strict Similarity $\mathcal{R}$. If we denote the least Herbrand model of $P$ by $M_P$ , then $q \in M_P$ if and only if $M_{P,\mathcal{R}}(q) = 1$.*

### 2.2 Similarity relations on possible worlds: RGS5$_\diamond$

The starting point in this approach is to assume that a possible world or state of a system may resemble more to some worlds than to another ones, and this basic fact may help us to evaluate to what extent a partial description (a proposition) may be close or similar to some other.

In [3] the authors define a many-valued modal logic over Gödel fuzzy logic by introducing only a possibility modal operator $\diamond$, where the intended meaning of $\diamond p$ is *approximately p*. Although $p$ may be a classical proposition, $\diamond p$ is considered to be fuzzy since the current state of the world may be more or less close to $p$. Moreover, since we want to explicitely deal with similarity degrees in the language we will consider as base logic the expansion of Gödel logic with rational truth-constants, called RG in [5].

We consider the language $\mathcal{L}_{G\diamond}$ of Gödel similarity modal logic, built over *Const* with Gödel conectives $\wedge$,$\rightarrow$, $\overline{0}$ and $\diamond$ and truth constants $\overline{r}$ for each $r \in Q \cap [0,1]$.

**Definition 4.** *The Rational Gödel similarity-based S5 modal logic RGS5$_\diamond$ is defined over the language $\mathcal{L}_{G\diamond}$ and is the smallest set of formulas containing every instance of the following axiom schemes and closed under the last two inference rules:*

**Axioms of Rational Gödel logic:**

$(\varphi \rightarrow \psi) \rightarrow ((\psi \rightarrow \chi) \rightarrow (\varphi \rightarrow \chi))$.
$\varphi \rightarrow (\psi \rightarrow \varphi)$.
$(\varphi \wedge \psi) \rightarrow (\psi \wedge \varphi)$.
$(\varphi \wedge (\psi \wedge \chi)) \rightarrow ((\psi \wedge \varphi) \wedge \chi)$.
$(\varphi \rightarrow (\psi \rightarrow \chi)) \equiv ((\varphi \wedge \psi) \rightarrow \chi)$.
$((\varphi \rightarrow \psi) \rightarrow \chi) \rightarrow (((\psi \rightarrow \varphi) \rightarrow \chi) \rightarrow \chi)$.
$\overline{0} \rightarrow \varphi$.
$\varphi \rightarrow (\varphi \wedge \varphi)$.
$\neg\varphi \equiv \varphi \rightarrow \overline{0}$.
$\overline{r} \wedge \overline{s} \equiv \overline{\min\{r,s\}}$.
$\overline{r} \rightarrow \overline{s} \equiv \overline{r \Rightarrow s}$.

**D$_\diamond$:** $\diamond(\varphi \vee \psi) \rightarrow (\diamond\varphi \vee \diamond\psi)$.
**Z$_\diamond^+$ :** $\diamond\neg\neg\varphi \rightarrow \neg\neg\diamond\varphi$.
**F$_\diamond$:** $\neg\diamond 0$.
**R1:** $\overline{r} \equiv \diamond\overline{r}$.
**R2:** $\overline{r} \wedge \diamond\varphi \rightarrow \diamond(\overline{r} \wedge \varphi)$
**T$_\diamond$:** $\varphi \rightarrow \diamond\varphi$.
**B$_\diamond$:** $\varphi \rightarrow \neg\diamond\neg\diamond\varphi$.
**4$_\diamond$:** $\diamond\diamond\varphi \rightarrow \diamond\varphi$.
**RN$_\diamond^+$:** *From $\varphi \rightarrow \psi$ infer $\diamond\varphi \rightarrow \diamond\psi$.*
**MP :** *From $\varphi$ and $\varphi \rightarrow \psi$, infer $\psi$.*

We denote by $\vdash_S$ the notion of derivability inside this logic.

Models are many-valued similarity-based Kripke model $M = \langle W, S, e\rangle$, in which $W \neq \emptyset$ is a set of possible worlds, $S$ is a similarity relation on $W \times W$ and $e$ represents an evaluation assigning to each atomic formula $p_i$ and each interpretation $w \in W$ a truth value $e(p_i, w) \in [0,1]$ of $p_i$ in $w$. $e$ is extended to formulas by means of Gödel logic truth functions by defining

$e(\varphi \wedge \psi, w) = min\{e(\varphi, w), e(\psi, w)\}$,

$e(\varphi \to \psi, w) = e(\varphi, w) \Rightarrow_G e(\psi, w)$
where $\Rightarrow_G$ is the well-known Gödel implication function[4], and
$e(\overline{r}, w) = r$, for all $r \in \mathbb{Q} \cap [0,1]$,
$e(\Diamond\varphi, w) = \sup_{w' \in W} \min\{S(w, w'), e(\varphi, w')\}$.
Based on the completeness results for RG [5] and for $GS5_\Diamond$ [3] we state the following completeness result for $RGS5_\Diamond$: a formula $\varphi$ is provable in $RGS5_\Diamond$, written $\vdash_s \varphi$, iff for every similarity Kripke model $M = \langle W, S, e\rangle$, $e(\varphi, w) = 1$ for every $w \in W$.

## 3 Relationship

Let $P = Facts \cup Rules \subset \mathcal{L}$ be a definite program, where $Facts \subseteq Const$ and
$Rules = \{q_i \leftarrow p_{(i,1)}, p_{(i,2)}, \ldots, p_{(i,n_i)} \mid i, n_i \in \mathbb{N},\ p_{(i,j)}, q_i \in Const\ \forall i, j\}$
are a set of facts and rules, respectively.
Let $\mathcal{R} : Const \times Const \to [0,1]$ be a similarity relation on $Const$.
Define a mapping $* : \mathcal{L} \to \mathcal{L}_{G\Diamond}$ by
$(q \leftarrow p_1, \ldots, p_n)^* = \Diamond p_1 \wedge \ldots \wedge \Diamond p_n \to \Diamond q$
$(p_1, \ldots, p_n)^* = \Diamond p_1 \wedge \ldots \wedge \Diamond p_n$
where $n \geq 1$ and the symbol "," that separates the propositional constants on the left side has to be interpreted as conjunction. Then, we can define
$Rules^* = \{\varphi^* \mid \varphi \in Rules\}$,
$Crisp = \{\overline{p \vee \neg p} \mid p \in Const\}$,
$Sim = \{\overline{\mathcal{R}(p,q)} \to ((p \to \Diamond q) \wedge (q \to \Diamond p)) \mid p, q \in Const\}$
and the following theory in the language $\mathcal{L}_{G\Diamond}$
$P_\Diamond = Facts \cup Rules^* \cup Crisp \cup Sim$.
Notice that the theory $Crisp$ ensures that the all propositional constants of $P$ are treated as Boolean constants in $P_\Diamond$. The aim is to show that one can derive in the similarity logic programming framework introduced in [14] a goal $\leftarrow q'$ from $P$ with a unification degree $\alpha$ if and only if one can derive in RGS5$_\Diamond$ the formula $\overline{\alpha} \to \Diamond q'$ from $P_\Diamond$. This result, besides to be relevant for itself because puts into relation different logical approaches to approximate reasoning, in particular will allows us to prove a formula in a theory $P_\Diamond$ in RGS5$_\Diamond$ by using an Automatic Theorem Prover [10]. So far we have only been able to prove the "only if" direction. The rest of this section is devoted to this task.

**Proposition 3.** *Let $\mathcal{R} : Const \times Const \to [0,1]$ be a similarity relation, $P = Rules \cup Facts$ a definite program on a propositional language $\mathcal{L}$ and $\leftarrow q'$ a goal. If there exists a Similarity-based SLD derivation with approximation degree $\alpha$ for $P \cup \{\leftarrow q'\}$*

$$D = G_0 \Rightarrow_{C_1,\alpha_1} G_1 \Rightarrow \cdots \Rightarrow_{C_k,\alpha_k} G_k$$

*where $G_0 = \leftarrow q'$, $G_k \neq \bot$ and $\alpha = min\{\alpha_1, \ldots, \alpha_k\}$ then*

$$P_\Diamond \vdash_s \overline{\alpha} \to (G_k^* \to \Diamond q')$$

*Proof.* We prove the thesis by induction on the length of $D$. If the length of $D$ is zero, then the thesis is true. Indeed,

$$\vdash_s \Diamond q' \to \Diamond q' \tag{1}$$

$$\vdash_s (\Diamond q' \to \Diamond q') \to (\overline{1} \to (\Diamond q' \to \Diamond q')) \tag{2}$$

Then, from 1, 2 and modus ponens, follows that

$$\vdash_s \overline{1} \to (\Diamond q' \to \Diamond q')$$

[4] $\Rightarrow_G$ is defined as $x \Rightarrow_G y = 1$ if $x \leq y$ and $x \Rightarrow_G y = y$, otherwise

Now, let us suppose that the thesis is true for length of $D$ equal $k$. Let

$$G_0 \Rightarrow_{C_1,\alpha_1} G_1 \Rightarrow \cdots \Rightarrow_{C_k,\alpha_k} G_k \Rightarrow_{C_{k+1},\alpha_{k+1}} G_{k+1}$$

where $G_{k+1} \neq \bot$, be an existing Similarity-based SLD derivation for $P \cup \{\leftarrow q'\}$ of length k+1 with approximation degree $\alpha = min\{\alpha_1, \ldots, \alpha_k, \alpha_{k+1}\}$. Since $min\{\alpha_1, \ldots, \alpha_k\}$ is the approximation degree of $D_1 = G_0 \Rightarrow_{C_1,\alpha_1} G_1 \Rightarrow \cdots \Rightarrow_{C_k,\alpha_k} G_k$ then, by the inductive hypothesis,

$$P_\Diamond \vdash_{\mathcal{S}} \overline{min\{\alpha_1, \ldots, \alpha_k\}} \rightarrow (G_k^* \rightarrow \Diamond q') \tag{3}$$

Let us denote with $q_i$ the head of the input clause $C_{m+1} = q_i \leftarrow p_{(i,1)}, p_{(i,2)}, \ldots, p_{(i,n_i)}$, and with $p$ the leftmost atom of $G_k = (\leftarrow p, \varphi)$ which is the selected atom in $D_1$. Re-typing 3 in an equivalent manner, results that

$$P_\Diamond \vdash_{\mathcal{S}} \overline{min\{\alpha_1, \ldots, \alpha_k\}} \rightarrow ((\Diamond p \wedge \varphi^*) \rightarrow \Diamond q')$$

thus,

$$P_\Diamond \vdash_{\mathcal{S}} \Diamond p \rightarrow (\overline{min\{\alpha_1, \ldots, \alpha_k\}} \rightarrow (\varphi^* \rightarrow \Diamond q')) \tag{4}$$

Since $\alpha_{k+1} = \mathcal{R}(q_i, p)$,

$$P_\Diamond \vdash_{\mathcal{S}} \overline{\alpha_{k+1}} \rightarrow ((q_i \rightarrow \Diamond p) \wedge (p \rightarrow \Diamond q_i))$$

$$P_\Diamond \vdash_{\mathcal{S}} \overline{\alpha_{k+1}} \rightarrow (q_i \rightarrow \Diamond p)$$

$$P_\Diamond \vdash_{\mathcal{S}} (\overline{\alpha_{k+1}} \wedge q_i) \rightarrow \Diamond p \tag{5}$$

$$P_\Diamond \vdash_{\mathcal{S}} \Diamond(\overline{\alpha_{k+1}} \wedge q_i) \rightarrow \Diamond\Diamond p$$

$$P_\Diamond \vdash_{\mathcal{S}} (\overline{\alpha_{k+1}} \wedge \Diamond q_i) \rightarrow \Diamond p$$

$$P_\Diamond \vdash_{\mathcal{S}} \Diamond q_i \rightarrow (\overline{\alpha_{k+1}} \rightarrow \Diamond p) \tag{6}$$

We have to distinguish two cases:

**(i)** If $C_{m+1} \in Rules$ then

$$C_{m+1}^* = \Diamond p_{(i,1)} \wedge \Diamond p_{(i,2)} \wedge \cdots \wedge \Diamond p_{(i,n_i)} \rightarrow \Diamond q_i \in P_\Diamond \tag{7}$$

Thus, by 6, 7, transitivity and modus ponens,

$$P_\Diamond \vdash_{\mathcal{S}} (\Diamond p_{(i,1)} \wedge \Diamond p_{(i,2)} \wedge \cdots \wedge \Diamond p_{(i,n_i)}) \rightarrow (\overline{\alpha_{k+1}} \rightarrow \Diamond p)$$

$$P_\Diamond \vdash_{\mathcal{S}} (\overline{\alpha_{k+1}} \wedge \Diamond p_{(i,1)} \wedge \cdots \wedge \Diamond p_{(i,n_i)}) \rightarrow \Diamond p \tag{8}$$

Then, by 8, 4, transitivity and modus ponens,

$$P_\Diamond \vdash_{\mathcal{S}} (\overline{\alpha_{k+1}} \wedge \Diamond p_{(i,1)} \wedge \cdots \wedge \Diamond p_{(i,n_i)}) \rightarrow (\overline{min\{\alpha_1, \ldots, \alpha_k\}} \rightarrow (\varphi^* \rightarrow \Diamond q'))$$

and

$$P_\Diamond \vdash_{\mathcal{S}} \overline{min\{\alpha_1, \ldots, \alpha_{k+1}\}} \rightarrow ((\Diamond p_{(i,1)} \wedge \cdots \wedge \Diamond p_{(i,n_i)} \wedge \varphi^*) \rightarrow \Diamond q')$$

**(ii)** If $C_{m+1} \in Facts$ then

$$C_{m+1} = q_i \in P_\Diamond \tag{9}$$

Then, by 5, 4, transitivity and modus ponens,

$$P_\Diamond \vdash_{\mathcal{S}} (\overline{\alpha_{k+1}} \wedge q_i) \rightarrow (\overline{min\{\alpha_1, \ldots, \alpha_k\}} \rightarrow (\varphi^* \rightarrow \Diamond q'))$$

Thus,

$$P_\Diamond \vdash_{\mathcal{S}} q_i \rightarrow (\overline{\alpha_{k+1}} \rightarrow (\overline{min\{\alpha_1, \ldots, \alpha_k\}} \rightarrow (\varphi^* \rightarrow \Diamond q')))$$

$$P_\diamond \vdash_s q_i \rightarrow ((\overline{\alpha_{k+1}} \wedge \overline{min\{\alpha_1, \ldots, \alpha_k\}}) \rightarrow (\varphi^* \rightarrow \diamond q'))$$

$$P_\diamond \vdash_s q_i \rightarrow (\overline{min\{\alpha_1, \ldots, \alpha_{k+1}\}} \rightarrow (\varphi^* \rightarrow \diamond q'))$$

and, by 9 and modus ponens,

$$P_\diamond \vdash_s \overline{min\{\alpha_1, \ldots, \alpha_{k+1}\}} \rightarrow (\varphi^* \rightarrow \diamond q').$$

□

**Corollary 1.** *Let $\mathcal{R} : Const \times Const \rightarrow [0,1]$ be a similarity relation, $P = Rules \cup Facts$ a definite program on a propositional language $\mathcal{L}$ and $\leftarrow q'$ a goal. If there exists a Similarity-based SLD refutation with approximation degree $\alpha$ for $P \cup \{\leftarrow q'\}$*

$$D = G_0 \Rightarrow_{C_1,\alpha_1} G_1 \Rightarrow \cdots \Rightarrow_{C_{k-1},\alpha_{k-1}} G_{k-1} \Rightarrow_{C_k,\alpha_k} \bot$$

*where $G_0 = \leftarrow q'$ and $\alpha = min\{\alpha_1, \ldots, \alpha_k\}$, then*

$$P_\diamond \vdash_s \overline{\alpha} \rightarrow \diamond q'$$

*Proof.* In $D$ necessarily $G_{k-1} = \leftarrow p$ for some $p \in Const$, $C_k = q_i \in Facts$ and $\alpha_k = \mathcal{R}(p, q_i)$. Thus, since $G_{k-1} \neq \bot$, by Proposition 3 follows that

$$P_\diamond \vdash_s \overline{min\{\alpha_1, \ldots, \alpha_{k-1}\}} \rightarrow (\diamond p \rightarrow \diamond q')$$

Moreover, from $\alpha_k = \mathcal{R}(p, q_i)$ results that

$$P_\diamond \vdash_s (\overline{\alpha_k} \wedge q_i) \rightarrow \diamond p$$

Then, by transitivity and modus ponens,

$$P_\diamond \vdash_s (\overline{\alpha_k} \wedge q_i) \rightarrow (\overline{min\{\alpha_1, \ldots, \alpha_{k-1}\}} \rightarrow \diamond q')$$

$$P_\diamond \vdash_s q_i \rightarrow (\overline{\alpha_k} \rightarrow (\overline{min\{\alpha_1, \ldots, \alpha_{k-1}\}} \rightarrow \diamond q'))$$

$$P_\diamond \vdash_s q_i \rightarrow ((\overline{\alpha_k} \wedge \overline{min\{\alpha_1, \ldots, \alpha_{k-1}\}}) \rightarrow \diamond q')$$

$$P_\diamond \vdash_s q_i \rightarrow (\overline{min\{\alpha_1, \ldots, \alpha_k\}} \rightarrow \diamond q')$$

Thus, since $q_i \in Facts$ implies that $q_i \in P_\diamond$, by modus ponens follows that

$$P_\diamond \vdash_s \overline{min\{\alpha_1, \ldots, \alpha_k\}} \rightarrow \diamond q'$$

□

## 4 Related works and conclusions

In the paper we have established some syntactic relationships between two approaches to similarity reasoning, namely SPLP and RGS5$_\diamond$. It remains as a future task to check whether the original aim of this paper of having a full relationship can be devised, also regarding predicate languages. On the other hand, in the literature there are other approaches to similarity-based reasoning, both in the fuzzy logic programming framework, like [11], as well as within other logical formalisms [4]. It will also be an interesting future work to study possible links among all these formalisms.

## References


1. R.K. Apt, "Logic Programming", in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, Amsterdam, 1990, pp. 492-574.

2. L. Biacino, G. Gerla, M. Ying, "Approximate reasoning based on similarity", in: Math. Logic Quart., 46, 2000.

3. X. Caicedo, R. Rodriguez, "A Gödel modal logic", in: Proc. of Logic, Computability and Randomness 2004. Cordoba, Argentina.

4. F. Esteva, P. Garcia, L. Godo, "On syntactical and semantical approaches to similarity-based approximate reasoning", in: Proc. Joint 9th IFSA and 20th NAFIPS 2001, July 25-28, Vancouver (Canada) pp. 1598-1603.

5. F. Esteva, L. Godo, C. Noguera, "On Rational Gödel and Nilpotent Minimum logics", in: Proc. of the Xth International Conference IPMU 2004, Perugia (Italia), July 4-9, pp. 561-568.

6. F. Formato, G. Gerla and M.I. Sessa, "Similarity-based unification", in: Fundamenta Informaticae 40, 2000, pp. 1-22.

7. G. Gerla and M.I. Sessa, "Similarity in Logic Programming", in: G. Chen, M. Ying, K.-Y. Cai (Ed.s), Fuzzy Logic and Soft Computing, Kluwer Acc. Pub., Norwell, 1999, pp. 19-31.

8. G. Gerla, M.I. Sessa, "Similarity Logic and Similarity PROLOG", in: Proc. Joint EUROFUSE-SIC99 International Conference, 25-28 May 1999, Budapest, Hungary, pp.367-372.

9. L. Godo, R. Rodriguez, "A Fuzzy Modal Logic for Similarity Reasoning", in: Fuzzy Logic and Soft Computing, Guoqing Chen, Mingsheng Ying and Kai-Yuan Cai eds., Kluwer, 1999, pp. 33-48.

10. V. Loia, S. Senatore, M.I. Sessa, "Similarity-based SLD Resolution and its implementation in an Extended Prolog System", in: Proceedings of 10th IEEE International Conference on Fuzzy Systems, Melbourne, Australia, December 2-5 2001. IEEE PRESS.

11. J.Medina, M. Ojeda-Aciego and P. Vojtáš. "Similarity-based unification: a multi-adjoint approach". *Fuzzy Sets and Systems* 146 (2004) 4362.

12. E.H. Ruspini, "On the semantics of fuzzy logic", in: Int. Journal of Approximate Reasoning, 5, 1991, pp. 45-88.

13. M.I. Sessa, "Translations and Similarity-based Logic Programming", in: Soft Computing 5(2), 2001.

14. M.I. Sessa, "Approximate Reasoning by Similarity-based SLD Resolution", in: Theoretical Computer Science, 275, 2002, pp. 389-426.

15. M.S. Ying, "A Logic for Approximated Reasoning", in: The Journal of Symbolic Logic, 59, 1994, pp. 830-837.

16. L.A. Zadeh, "Similarity Relations and Fuzzy Orderings", in: Information Sciences 3, 1971, pp. 177-200.