Dott.ssa Francesca A. Lisi

UNIVERSITA' DEGLI STUDI – BARI
Dipartimento di Informatica
Facoltà di Scienze MM. FF. NN.

Learning Rules on top of Ontologies: An Inductive Logic Programming Approach

New challenging application areas like the Semantic Web require the definition of rules on top of ontologies. Hybrid systems for Knowledge Representation and Reasoning (KR&R) that combine (fragments of) Horn clausal logic and Description Logics, notably AL-log, have been invented to provide one such unified framework for dealing with both relational and structural data. Yet acquiring hybrid rules is a demanding and time-consuming task. It can be (partially) automated by applying Machine Learning algorithms, more precisely those ones following the approach known under the name of Inductive Logic Programming (ILP).

In this seminar I shall present a general framework for learning rules on top of ontologies that adopts the methodological apparatus of ILP within the KR&R setting of AL-log. Also I shall briefly discuss an ILP system that implements an instantiation of the framework and the preliminary results of its application in the Semantic Web context.

Planning with Action Languages: Perspectives using CLP(FD) and ASP

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. Action description languages, such as $\mathcal{A}$ and $\mathcal{B}$ (Gelfond and Lifschitz, 1998), are expressive tools introduced for formalizing planning domains and problems. In this work, we investigate two alternative approaches to the problem of encoding action languages using logic programming. Starting from a slight variation of $\mathcal{B}$, we explore the use of *Answer Set Programming (ASP)* and *Constraint Logic Programming over Finite Domains (CLP(FD))* in processing action theory specifications. As regards ASP we present a Prolog translator from action theory into *lparse*'s syntax. Concerning CLP(FD), we describe a program that directly processes a specification in the $\mathcal{B}$ language.

1 Introduction

Building intelligent agents that can be effective in real-world environments has been a goal of researchers from the very first days of Artificial Intelligence (AI). It has long been recognized that such an agent must be able to *acquire*, *represent*, and *reason* with knowledge. As such, a *reasoning component* has been an inseparable part of any agent architecture in the literature.

Although the underlying representations and implementations may vary between agents, the reasoning component of an agent is often responsible for making decisions that are critical to its existence. As described in [10], such a component consists of a knowledge base, a plan database, and a search engine. The knowledge base stores domain models as well as heuristic knowledge, while the plan database consists of plan skeletons. The domain model is used to validate plans while the heuristic information, the planning database, and experts' knowledge help the search engine in quickly deriving plans. In addition, frameworks for planning and reasoning require the ability to deal with real-world knowledge, e.g., domains with actions with durations, actions with bounded resources, and temporally extended goals. Thus, the construction of intelligent agents requires a knowledge representation language capable of expressing various types of knowledge, such as domain and commonsense knowledge, and methodologies for reasoning with such knowledge.

Logic programming languages offer many attributes that make them suitable as knowledge representation languages. Their declarative nature allows for the modular development of provably correct reasoning modules [3]. Recursive definitions can be easily expressed and reasoned upon. Control knowledge and heuristic information can

be declaratively introduced in the reasoning process. Furthermore, many logic programming languages offer a natural support for nonmonotonic reasoning, which is considered essential for commonsense reasoning. These features, along with the presence of fast solvers [2, 15, 19, 9], make logic programming an attractive paradigm for knowledge representation and reasoning.

In the context of knowledge representation and reasoning, one of the most commonly explored application of logic programming has been in the domain of reasoning about actions and change [3]. Planning problems have been effectively encoded using *Answer Set Programming (ASP)* [3]—where distinct answer sets represent different trajectories leading to the desired goal. CLP(FD) has been used less frequently for planning problems (e.g., [13, 21]). Our encoding has some similarities to the one presented in [13] although we rely on CLP instead of CSP and our action language supports statical causal laws and non-determinism, while the work of Lopez and Bacchus is restricted to STRIPS.

A planning problem is based on the notions of *State*, *Action*, and *Fluent*. Recent works on representing and reasoning about actions and change have relied on the use of concise and high-level languages, commonly referred to as *action description languages*, e.g., the languages $\mathcal{A}$ and $\mathcal{B}$ [8]. Action languages allow us to write propositions that describe the effects of actions on states, and to create queries to infer properties of the underlying transition system.

In our experiments, we use a slight variation of $\mathcal{B}$, based on a syntax similar to the one used in [20]. An *action theory* is a specification of a planning problem using the action language.

The purpose of this paper is to investigate and compare two alternative approaches to the problem of encoding action languages using logic programming. In particular, we explore the use of two flavors of logic programming, *Answer Set Programming (ASP)* [14, 12, 17] and *Constraint Logic Programming over Finite Domains (CLP(FD))* [15, 2]. More specifically, regarding CLP(FD), we report a program that directly processes an action theory specification. Concerning the ASP solution, we developed a translator, written in Prolog, that, given an action theory specification, produces a suitable input file for *lparse*—the translation process follows the general guidelines highlighted, for example, in [20, 7].

The ultimate goal of this study is to investigate the relative strengths and weaknesses of the two approaches. While ASP has been widely used in the context of planning, the use of CLP(FD) has been more limited. Apart from the issue of performance, our interest is in exploring declarativeness and potential to handle extensions of traditional action languages.

2 An Action Description Language

Planning domains can be effectively described using high-level *action description languages*, e.g., the languages $\mathcal{A}$ and $\mathcal{B}$ [8]. In our experiments, we use a slight variation of $\mathcal{B}$, based on a syntax similar to the one used in [20]. With a slight abuse of notation, we simply refer to the language used in this paper as $\mathcal{B}$. The readers are referred to [12, 20, 8] for the theoretical foundations of action languages.

A planning problem can be described defining the notions of states, actions, and fluents and the relationships between these notions. In particular

- `Fluents`: they are *atomic formulae*, describing the state of the world, and whose truth value can change over time.
- `State`: a state is a possible configuration of the domain of interest; in particular, a state is described by an assignment of truth values to the fluents.
- `Actions`: they affect the state of the world, and thus allow the transition from a state to another.

Intuitively, starting from an initial state, a planner tries to successively apply actions, transitioning to other states, until a state (*final state*) meeting certain given conditions is reached.

As a concrete syntax, fluents and actions are atomic formulae $p(t_1, \ldots, t_n)$ ($n \geq 0$) from an underlying logic language $\mathcal{L}$. For the sake of simplicity, we assume that the Herbrand universe built using $\mathcal{L}$ is finite (e.g., either there are no function symbols, or the use of functions symbols is restricted to avoid the creation of arbitrary complex terms).

In the definition of an action theory, an assertion of the kind `fluent(f)` declares that `f` is a fluent. Fluent literals are constructed from fluents and their negations (denoted by `mneg(f)`). Declarations of the form `action(a)` are used to describe the possible actions (in this case, `a`). In these declarations, f and a are ground atomic formulae of $\mathcal{L}$.

Following the general principle of *domain predicates* of ASP [19], we assume that the admissible terms that can be used in fluents and actions are classified in different categories, described by facts (e.g., lines (1)–(3) of the example in Figure 1).

The language $\mathcal{B}$ allows one to specify an *action theory*, which relates actions, states, and fluents using predicates of the following forms:

- `executable(a, [list-of-preconditions])`
  asserting that the given preconditions have to be satisfied in the current state in order for the action `a` to be executable.
- `causes(a, f, [list-of-preconditions])`
  encodes a dynamic causal law, describing the effect (the fluent literal `f`) of the execution of action `a` in a state satisfying the given preconditions.
- `caused([list-of-preconditions], f)`
  describes a static causal law—i.e., the fact that the fluent literal `f` is true in a state satisfying the given preconditions.

(where `[list-of-preconditions]` denotes a list of fluent literals). As an example, see lines (7)–(26) of the example in Figure 1.

A specific instance of a planning problem contains also a description of the initial state and of the desired goal:

- `initially(f)`
  asserts that the fluent literal `f` is true in the initial state. In our examples, we assume that the initial state is *complete*—i.e., we have knowledge of the truth value of each fluent in the initial state.
- `goal(f)`
  asserts that the goal requires the fluent literal `f` to be true in the final state.

For an instance, see lines (27)–(34) in Figure 1.

A *trajectory* is a sequence $s_0 a_1 s_1 a_2 \ldots a_n s_n$, where $a_1, \ldots, a_n$ are actions and $s_0, \ldots, s_n$ are states. The actions $a_1, \ldots, a_n$ of a trajectory represent a *plan* if s_0 is the initial state and, for each `goal(f)` in the action theory we have that the fluent f is true in s_n. We assume that the length of the desired plan is given and we disallow parallel actions. Intuitively, we look for plans that go from the initial state to the final state crossing a fixed number of state. Each state transition involves exactly one action execution.

Let us discuss in much detail the above referred example, as reported in Figure 1. There are three authors and four places (lines (1)–(3)). The fluents describe whether the authors are alive, whether they are armed, and whether they are staying at a particular location (lines (2)–(6)). The actions are `shoot` and `move` (lines (7)–(8)).

```
(1)     author(andy). author(ago). author(rico).
(2)     place(udine). place(laquila).
(3)     place(paris). place(lascruces).

(4)     fluent(alive(A)):-author(A).
(5)     fluent(armed(A)):-author(A).
(6)     fluent(stay(A,P)):-author(A),place(P).

(7)     action(shoot(A,P)):-author(A),place(P).
(8)     action(move(A,P)):-author(A),place(P).

(9)     causes(shoot(A,P),mneg(alive(B)),[stay(B,P)]):-
(10)        author(A),author(B),place(P).
(11)    causes(shoot(A,P),mneg(armed(A)),[]) :-
(12)        author(A),place(P).
(13)    causes(move(A,P),stay(A,P),[] ):-
(14)        author(A),place(P).

(15)    caused([stay(A,P1)],mneg(stay(A,P2))):-
(16)        author(A),place(P1),place(P2),diff(P1,P2).
(17)    caused([stay(A,P)],mneg(stay(B,P)) ):-
(18)        author(A),author(B),place(P),diff(A,B).

(19)    executable(shoot(A,P),[armed(A),alive(A)]):-
(20)        author(A),place(P).
(21)    executable(move(A,P2),
(22)            [alive(A),stay(A,P1),mneg(stay(A,P2)),
(23)            mneg(stay(B,P2)),mneg(stay(C,P2))]):-
(24)        author(A),author(B),author(C),
(25)        place(P1),place(P2),
(26)        diff(P1,P2),diff(A,B,C).

(27)    initially(stay(andy,laquila)).
(28)    initially(stay(ago,udine)).
(29)    initially(stay(rico,lascruces)).
(30)    initially(armed(A)):-author(A).
(31)    initially(alive(A)):-author(A).

(32)    goal(mneg(armed(A))):-author(A).
(33)    goal(alive(rico)).
(34)    goal(stay(andy,paris)).
```

Fig. 1. Action Theory: The fighting authors

An author `A` can shoot in the direction of a place `P` if he is armed and alive. An author `A` can move to a place `P2` if he is alive and none of the other authors is currently

located in `P2`. These conditions represent the executability conditions of the actions, reported in lines (19)–(26).

The actions effects are described in lines (9)–(14). If an author `A` shoots in the direction of place `P`, and `B` is in the location `P`, then `B` is no longer alive (this also means that he cannot move away from `P`). If an author `A` shoots, then after this action he will no longer be armed. If, instead, the author `A` moves to the place `P`, then he will stay in `P`.

Lines (15)–(18) handle the static causal law of the action theory. If an author is in a place, then he cannot be located at any other place. If an author is in one place (alive or not) then no other author can stay in the same place.

Initially (lines (27)–(31)) the authors are armed, alive and located at their home universities). The final goal is that nobody is armed, that the author `rico` is alive, and that author `andy` is located in `paris`. A possible plan that meets the requirement is:

1. $\texttt{shoot}(\texttt{rico}, \texttt{paris})$
2. $\texttt{shoot}(\texttt{andy}, \texttt{paris})$
3. $\texttt{shoot}(\texttt{ago}, \texttt{udine})$
4. $\texttt{move}(\texttt{andy}, \texttt{paris})$

which includes the suicide of the oldest author. For other examples see `www.di.univaq.it/~formisano/CLPASP`.

3 ASP

There are several ways to encode an action theory in ASP (e.g., [12, 3]). We implemented a Prolog translator from $\mathcal{B}$ to the *lparse*'s syntax [19]. The translation is relatively straightforward, and we do not enter here in its details. Most of the translation amounts to syntactical rewriting of the action theory specification to suitable ASP rules. The interested reader can obtain the Prolog program of the translator from [5], together with the action theory specifications we used.

The main predicate of such Prolog translator is `main(Input,Len)`. The user must provide the name of an input file containing an action theory specification (`Input`), and the exact length of the desired plan (`Len`). In Figure 2, we show an excerpt of the ASP specification produced by the translator for the action theory describing the problem of the fighting authors (Figure 1). In this case we used `Len=4`. Let us comment on some features of this code:

- line (1) is used to denote the legal time-steps of the plan (from 0 to 4);
- line (4) contains the description of the initial state;
- the constraint in line (5) is used to generate exactly one action occurrence for each time step; similarly, the constraint in line (6) imposes that the action chosen at each time step is executable at that time step;
- a fluent literal ℓ holds at time T if `hold(`ℓ`,T)` is true. The conditions that make `hold(`ℓ`,T)` true are described by the rules in lines (7)–(10); the initial state is defined in line (7), line (8) describes the direct effects of an action, line (9) describes the effect of static causal laws, while line (10) encodes the effect of inertia.
- lines (11)–(13) describe the executability conditions;
- lines (14)–(17) identify the direct effects of each action, while lines (18)–(21) encode the preconditions of the various actions;

- lines (22)–(24) encode the static causal laws;
- lines (25) and (26) represent the goal;
- lines (27)–(31) define some auxiliary rules.

```
(1)     time(0..4).
(2)     fluent(alive(ago)). ... fluent(armed(rico)).
(3)     action(move(ago,laquila)). ... action(shoot(rico,udine)).
(4)     initially(alive(ago)). ... initially(stay(rico,lascruces)).
(5)     1{occ(Act,Ti):action(Act)}1 :- time(Ti), Ti < 4.
(6)     :- occ(Act,Ti), action(Act), time(Ti), not exec(Act,Ti).
(7)     hold(Fl,0) :- initially(Fl).
(8)     hold(Fl,Ti+1) :- time(Ti), literal(Fl), occ(Act,Ti),
        causes(Act,Fl), ok(Act,Fl,Ti), exec(Act,Ti).
(9)     hold(Fl,Ti) :- time(Ti), literal(Fl), caused(Ti,Fl).
(10)    hold(Li,Ti+1) :- time(Ti), literal(Li), hold(Li,Ti),
        opposite(Li,Lu), not hold(Lu,Ti+1).
(11)    exec(move(ago,laquila),Ti) :-
        time(Ti),hold(alive(ago),Ti),hold(stay(ago,udine),Ti),
        hold(mneg(stay(ago,laquila)),Ti),hold(mneg(stay(andy,laquila)),Ti),
        hold(mneg(stay(rico,laquila)),Ti).
(12)    ...
(13)    exec(shoot(ago,laquila),Ti) :-
        time(Ti),hold(armed(ago),Ti),hold(alive(ago),Ti).
(14)    causes(shoot(andy,udine),mneg(alive(andy))).
(15)    causes(shoot(andy,udine),mneg(armed(andy))).
(16)    ...
(17)    causes(move(rico,udine),stay(rico,udine)).
(18)    ok(shoot(andy,udine),mneg(alive(andy)),Ti) :- time(Ti),
        hold(stay(andy,udine),Ti).
(19)    ok(shoot(andy,udine),mneg(armed(andy)),Ti) :- time(Ti).
(20)    ...
(21)    ok(move(rico,udine),stay(rico,udine),Ti) :- time(Ti).
(22)    caused(Ti,mneg(stay(andy,laquila))) :- time(Ti),
        hold(stay(andy,udine),Ti).
(23)    ...
(24)    caused(Ti,mneg(stay(ago,lascruces))) :- time(Ti),
        hold(stay(rico,lascruces),Ti).
(25)    :- not goal.
(26)    goal :- hold(alive(rico),4), hold(mneg(armed(ago)),4),
        hold(mneg(armed(andy)),4), hold(mneg(armed(rico)),4),
        hold(stay(andy,paris),4).
(27)    literal(Fl) :- fluent(Fl).
(28)    literal(mneg(Fl)) :- fluent(Fl).
(29)    opposite(Fl,mneg(Fl)) :- fluent(Fl).
(30)    opposite(mneg(Fl),Fl) :- fluent(Fl).
(31)    :- time(Ti), fluent(Fl), hold(Fl,Ti), hold(mneg(Fl),Ti).
```

Fig. 2. Translation in ASP of the fighting authors

4 CLP(FD)

The proposed CLP(FD) encoding of a planning problem uses the following representation. A plan with exactly N states, p fluents, and m actions is represented by:

- A list, called `States`, containing N lists, each composed of p terms of the type `fluent(fluent_name, Bool_var)`. The variable of the i^{th} term in the j^{th} list is

assigned 1 if and only if the i^{th} fluent is true in the j^{th} state of the trajectory. For example, if we have $N = 3$ and the fluents `f`, `g`, and `h`, we have:

```
States = [[fluent(f,X_f_1),fluent(g,X_g_1),fluent(h,X_h_1)],
          [fluent(f,X_f_2),fluent(g,X_g_2),fluent(h,X_h_2)],
          [fluent(f,X_f_3),fluent(g,X_g_3),fluent(h,X_h_3)]]
```

- A list `ActionsOcc`, containing $N - 1$ lists, each composed of m terms of the form `action(action_name,Bool_var)`. The variable of the i^{th} term of the j^{th} list is assigned 1 if and only if the i^{th} action occurs during the transition from state j to state $j + 1$. For example, if we have $N = 3$ and the actions `a` and `b`, then:

```
ActionsOcc = [[action(a,X_a_1),action(b,X_b_1)],
              [action(a,X_a_2),action(b,X_b_2)]]
```

The planner will make use of this structure in the construction of the plan; appropriate constraints are set between the various boolean variables to capture their relationships (e.g., for each list in `ActionsOcc`, exactly one `action(action_name,Bool_var)` can contain a variable that is assigned the value 1).

We explain below the main parts of the CLP interpreter for the $\mathcal{B}$ language we developed. The interpreter assumes that the action theory is present in the Prolog database—observe that the syntax adopted (see, e.g., Figure 1) is compliant with Prolog's syntax, thus allowing us to directly store the action theory as rules and facts in the Prolog database.

The entry point of the planner is shown in Figure 3. The main predicate is `main(N)` (line (1)). It computes a plan of length `N` for the action theory present in the Prolog database. Lines (2) and (3) collect the lists of fluents (`Lf`) and actions (`La`), the description of the initial state (`Init`) and the required content of the final state (`Goal`).

Lines (4) and (5) calls the predicates for defining the lists `States` and `ActionsOcc`, as explained above. These predicates are defined in lines (12)–(26). Observe that all variables for fluents are declared as boolean variables in line (18); furthermore, in every state transition, exactly one action can be fired (line (23)).

The predicates in lines (6) and (7) handle the knowledge about the initial and the goal states, respectively. Lines (8) and (9) impose the constraints on state transitions and action executability. Line (10) gathers all variables denoting action occurrences, in preparation for the labeling phase (line (11)). Note that the labeling is focused on the selection of the action to be executed at each time step. Please observe that in the code of Figure 3 we omit the parts concerning delivering the results to the user.

In Figure 4, we report the definition of the predicates employed to assign the correct truth values to the fluents in the initial state (predicate `set_initial`) and to the desired fluents in the final state (predicate `set_goal`). The predicate `set_one_static_fluent` is used to perform a declarative closure operation on a state by using the static causal laws of the action theory (i.e., the rules of type `caused` in the action theory). In particular, this is realized by propagating information from the fluents whose truth value has already been established. It is a particular case of predicate `set_one_fluent` explained later, and therefore we do not list here its definition.

The core of the planning process is carried out by the predicates `set_transitions` and `set_executability` (lines (8)–(9)). The code for these predicates is reported in Figure 5. In particular, lines (43)–(50) recursively define the predicate `set_transitions`,

```
(1)     main(N,ACTIONSOCC,STATES):-
(2)         setof(F,fluent(F),Lf), setof(A,action(A),La),
(3)         setof(F,initially(F),Init), setof(F,goal(F),Goal),
(4)         make_states(N,Lf,STATES),
(5)         make_action_occurrences(N,La,ACTIONSOCC),
(6)         set_initial(Init,STATES),
(7)         set_goal(Goal,STATES),
(8)         set_transitions(ACTIONSOCC,STATES),
(9)         set_executability(ACTIONSOCC,STATES),
(10)        get_all_actions(ACTIONSOCC,AllActions),
(11)        labeling([ff],AllActions).

(12)    make_states(0,_,[]):-!.
(13)    make_states(N,List,[S|STATES]) :-
(14)        N1 is N-1, make_states(N1,List,STATES),
(15)        make_one_state(List,S).
(16)    make_one_state([],[]).
(17)    make_one_state([F|Fluents],[fluent(F,VarF)|VarFluents]) :-
(18)        make_one_state(Fluents,VarFluents), VarF in 0..1.

(19)    make_action_occurrences(1,_,[]):-!.
(20)    make_action_occurrences(N,List,[Act|ActionsOcc]) :-
(21)        N1 is N-1, make_action_occurrences(N1,List,ActionsOcc),
(22)        make_one_action_occs(List,Act),
(23)        get_action_list(Act,AList), sum(AList,#=,1).
(24)    make_one_action_occs([],[]).
(25)    make_one_action_occs([A|Acts],[action(A,OccA)|OccActs]) :-
(26)        make_one_action_occs(Acts,OccActs), OccA in 0..1.
```

Fig. 3. Entry Point of the CLP(FD) Planner

which ultimately relies on `set_one_fluent` to constrain all boolean variables related to each possible fluent in each possible state transition (i.e., action execution).

```
(27)    set_initial(List,[InitialState|_]):-
(28)        set_state(List,InitialState),
(29)        complete_state(InitialState,InitialState).
(30)    set_goal(List,States):-
(31)        last(States,FinalState), set_state(List,FinalState).
        set_state([],_).
(32)    set_state([Fluent|Rest],State :-
(33)        (Fluent=mneg(F),!,member(fluent(F,0),State);
(34)        member(fluent(Fluent,1),State)),
(35)        set_state(Rest,State).
(36)    complete_state([],_).
(37)    complete_state( [fluent(Fluent,EV)| Fluents],InitState):-
(38)        (integer(EV), ! ;
(39)        set_one_static_fluent(Fluent,EV,InitState) ),
(40)        complete_state( Fluents ,InitState).
(41)    set_one_static_fluent(Name,EV,State) :-
(42)            .........
```

Fig. 4. Initial and Final State

Let us consider a state transition from state `FromSt` to state `ToSt`, as depicted in Figure 6, where we assume that there are p fluents $1, \ldots, p$ involved. As mentioned earlier, two boolean variables IV_i and EV_i (for $i = 1, \ldots, p$) denote whether the fluent

```
(43)    set_transitions(_,[_]):-!.
(44)    set_transitions([Occ|Occurrences],[S1,S2|Rest]) :-
(45)        set_transition(O,S1,S2,S1,S2),
(46)        set_transitions(Occurrences,[S2|Rest]).
(47)    set_transition(_,[],[],_,_).
(48)    set_transition(Occ,[fluent(F,IV)|R1],[fluent(F,EV)|R2],FromSt,ToSt):-
(49)        set_one_fluent(F,IV,EV,Occ,FromSt,ToSt),
(50)        set_transition(Occ,R1,R2,FromSt,ToSt).
(51)    set_one_fluent(Fl,IV,EV,Occ,FromSt,ToSt) :-
(52)        findall([X,L],causes(X,Fl,L),Pos),
            findall([Y,M],causes(Y,mneg(Fl),M),Neg),
(53)        build_sum_prod(Pos,Occ,FromSt,PFormula,EV,p),
(54)        build_sum_prod(Neg,Occ,FromSt,NFormula,EV,n),
(55)        findall(P,caused(P,Fl),StatPos),
            findall(N,caused(N,mneg(Fl)),StatNeg),
(56)        build_sum_stat(StatPos,ToSt,PStatPos,EV,p),
(57)        build_sum_stat(StatNeg,ToSt,PStatNeg,EV,n),
(58)        append(PFormula,PStatPos,Pos_Fl),
(59)        append(NFormula,PStatNeg,Neg_Fl),
(60)        sum(Pos_Fl,#=,Psum),
(61)        sum(Neg_Fl,#=,Nsum),
(62)        EV #<=> ((Psum + IV - IV * Nsum) #> 0).
(63)    build_sum_prod([],_,_,[],_,_).
(64)    build_sum_prod([[Action,Prec]|Rest],Occ,State,[Flag|PF1],EV,Mode):-
(65)        get_precondition_vars(Prec,State,ListPV),
(66)        length(Prec,NPrec), sum(ListPV,#=,SumPrec),
(67)        member(action(Action,VA),Occ),
(68)        (VA #= 1 #/\ (SumPrec #= NPrec)) #<=> Flag,"
(69)        (Mode == p -> EV #>= Flag; Mode==n -> EV #=< 1-Flag),
(70)        build_sum_prod(Rest,Occ,State,PF1,EV,Mode).
(71)    build_sum_stat([],_,[],_,_).
(72)    build_sum_stat([Cond|Others],State,[Flag|Fo],EV,Mode):-
(73)        get_precondition_vars(Cond,State,List),
(74)        length(List,NL), sum(List,#=,Result),
(75)        Flag #<=> (Result #= NL),
(76)        (Mode==p -> EV #>= Flag; Mode==n -> EV #=< 1-Flag),
(77)        build_sum_stat(Others,State,Fo,EV,Mode).
```

Fig. 5. Establishing Constraints among Fluent and Action Variables

i holds in `FromSt` and in `ToSt`, respectively. Let the variables VA_j, (for $j = 1, \ldots, m$) denote whether the action j occurs in such a state transition. In this situation, let us consider a generic call (line (49)) of the predicate `set_one_fluent` (defined in line (51)). For a given fluent `Fl`, the predicate `set_one_fluent` collects the list `Pos` (resp. `Neg`) of pairs `[Action,Preconditions]` such that `Action` makes `Fl` true (resp. false) in the

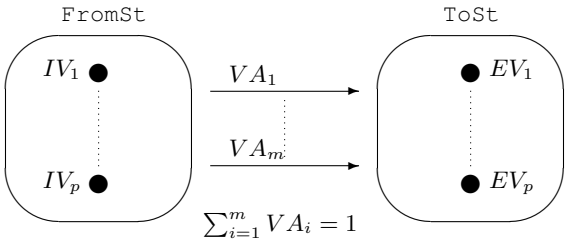


Fig. 6. Action constraints (see procedures `set_transitions`—`build_sum_prod`)

state transition (lines (52)). Similarly, it handles the static causal laws (`caused` assertions), by collecting the lists of conditions that affect the truth value of `Fl` (the variables `StatPos` and `StatNeg`, in lines (55)).

The CLP variables involved are then constrained by the procedures `build_sum_prod` (lines (53)–(54)) and `build_sum_stat` (lines (56)–(57)). Finally, all variables related to the introduction (respectively, removal) of fluents are collected in `Pos_Fl` (respectively `Neg_Fl`). Their sums are collected in variables `Psum` and `Nsum`, respectively. Further constraints in `build_sum_prod` and `build_sum_stat` ensure that these variables assume values that are greater than zero. Moreover, we take care of the inertia law in line (62): if none of the actions affect `Fl`, then $EV = IV$.

Let us focus on the predicate `build_sum_prod`. For the sake of simplicity, let us assume that it is called with `Mode == p` (line (53)). The predicate `build_sum_prod` recursively processes a list of pairs `[Action,Preconditions]`. For each action A_j, if A_j occurs and all of its preconditions (`Prec`) hold, then there is an action effect (`Flag = 1`, line (68)). Moreover, the constraint `Mode == p -> EV #>= Flag` in line (69), imposes that, if an action A_j occurs in a state transition (i.e., the corresponding boolean variable `VA_j` is 1), and such action makes a fluent true, then the boolean variable `EV` associated to it has to be 1. Analogous constraints are imposed in the case `Mode == n`, corresponding to the handling of actions that make a given fluent false (called in line (54)). A few remarks on line (62). A fluent is true in the next state (`EV`) if and only there is an action or a static causal law making it true (`Psum`) or it was true in the previous state (`IV`) and no causal laws make it false.

The modus operandi for imposing the executability conditions for actions is similar to what we have just described—and the corresponding code is listed in Figure 7. By using the predicate `set_executability` (defined in line (78)), all the preconditions of each action are considered. If the action is executed (`VA = 1`, see line (91)), then at least one of the executability conditions of that action holds (see predicates `get_all_preconditions` and `formula`—lines (98)–(105)). Lines (106)–(120) define some auxiliary predicates.

5 Experimentation

We have conducted some experiments to compare the performance of the two approaches to planning. In particular, the CLP(FD) planner has been implemented using SICStus Prolog, and using the `clpfd` library for constraint solving over finite domains [24]. The ASP-based planning approach has been developed using the `lparse` language, and the resulting encodings have been tested using two different systems: SMODELS [19] and CMODELS [11]. The first system is a native ASP solver, based on a variation of the Davis-Putnam procedure. The second system relies on a mapping of the problem of computing models to the problem of testing satisfaction of propositional theories (SAT solving). The advantage of this second approach is the ability to reuse efficient SAT solving technology; our experiments have been conducted using two different underlying SAT solvers—mChaff [16] and Simo [9].

As an example of the planning problems we experimented with, we describe a planning problem in the block world domain with N blocks (blocks $1, \ldots, N$). In the initial

```
(78)    set_executability(ActionsOcc,States) :-
(79)        findall([Act,C],executable(Act,C),Conds),
(80)        group_cond(Conds,GroupedConds),
(81)        set_executability1(ActionsOcc,States,GroupedConds).
(82)    set_executability1([],[_],_).
(83)    set_executability1([AStep|ARest],[State|States],Conds):-
(84)        set_executability_sub(AStep,State,Conds),
(85)        set_executability1(ARest,States,Conds).
(86)    set_executability_sub(_,_,[]).
(87)    set_executability_sub(Step,State,[[Act,C]|CA]) :-
(88)        member(action(Act,VA),Step),
(89)        get_all_preconditions(C,State,NCs,Temps),
(90)        formula(NCs,Temps,Phi),
(91)        (VA #=1) #=> Phi #= 0,
(92)        set_executability_sub(Step,State,CA).
(93)    group_cond([],[]).
(94)    group_cond([[Action,C]|R],[[Action,[C|Cs]]|S]) :-
(95)        findall(L,(member([Action,L],R)),Cs),
(96)        filter(Action,R,Others),
(97)        group_cond(Others,S).
(98)    get_all_preconditions([],_,[],[]).
(99)    get_all_preconditions([C|R],State,[NC|NCs],[T|Temps]) :-
(100)       get_precondition_vars(C,State,Cs),
(101)       length(Cs,NC), sum(Cs,#=,T),
(102)       get_all_preconditions(R,State,NCs,Temps).
(103)   formula([],[],1).
(104)   formula([N|Rest],[V|Val],Phi) :-
(105)       Phi #= Phi1 #/\ (N #> V), formula(Rest,Val,Phi1).
(106)   get_precondition_vars([],_,[]).
(107)   get_precondition_vars([P1|Rest],State,[F|LR]) :-
(108)   get_precondition_vars(Rest,State,LR),
(109)       (P1 = mneg(FN),!,member(fluent(FN,A),State),F #= 1-A;
(110)       member(fluent(P1,F),State)).
(111)   get_all_actions([],[]).
(112)   get_all_actions([A|B],List) :-
(113)       get_action_list(A,List1),get_all_actions(B,List2),
(114)       append(List1,List2,List).
(115)   get_action_list([],[]).
(116)   get_action_list([action(_,V)|Rest],[V|MRest]) :-
(117)       get_action_list(Rest,MRest).
(118)   filter(_,[],[]).
(119)   filter(A,[[A,_]|R],S) :- !, filter(A,R,S).
(120)   filter(A,[C|R],[C|S]) :- !, filter(A,R,S).
```

Fig. 7. Executability Conditions

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 plan made of T actions. An additional restriction must be met: in each state at most three blocks can lie on the table.

Figure 8 displays the specification of an instance of the planning problem with $N = 5$ using the above described action language. In such specification, the blocks are defined in line (1). Lines (2)–(5) define the fluents of the problem. A block may lie on the table or on top of another block (fluent `on(X,Y)`). A block may be clear (if no other block is on top of it) or not. There may be space on the table for other blocks (fluent `space_on_table`). There are two possible moves (lines (6)–(7)). The `causes` rules (lines (8)–(22)) as well as the `caused` assertions (lines (23)–(27)) are of easy reading. Lines (28)–(31) specify the executability conditions for the two possible actions: one can move a block `X` on top of another block `Y` only if both are clear, and a block `X` can be moved to the table only if there is free space left (i.e., there are at most two blocks lying on the table). The initial state and the goal state are described in lines (32)–(40) and (41)–(43), respectively.

Table 1 reports the execution times from the three systems, for different number of blocks and plan lengths (i.e., the number of moves). All the timing results, expressed in seconds, have been obtained by measuring only the CPU usage time needed for computing the first solution, if any. In the case of the ASP solvers, we separately report the time needed by *lparse* to ground the programs. As regards CLP(FD), we used the `runtime` option to measure the execution time. Hence we do not account for the time spent in garbage collection and system calls. All tests have been performed on a PC (P4 processor 2.8 GHz and 512 MB RAM memory) running Linux.

We also experimented with a variant of the problem described above, where a further constraint is imposed: no block x can be placed on top of another block y if $y \geq x$. This constraint can be modeled in the action language, by replacing line `(6)` in Figure 8 with the following one:

```
(6') action(move(X,Y)):- blk(X), blk(Y), X>Y.
```

The results reported in Table 1 show that the ASP solvers can solve the instances (save for the easiest ones) more quickly than CLP(FD). Notice that, the CLP(FD) solution is general and applicable to every action theory described in the proposed action language. We have also tested the two approaches on other classical planning problems, such as for instance, the man-wolf-goat-cabbage problem, the missionaries and cannibals problem, the problem of finding best movements of elevators, Hanoi towers, obtaining similar results. Observe also that, in CLP(FD), a price is paid in making use of a declarative translation from a generic action theory to constraints; for example, a manual and ad-hoc encoding of the same planning problem presented here (using rule `(6')`) for 5 blocks and 13 actions founds the solution in 2ms (see [5] for this code and related running times). We expect similar improvements also from an ad-hoc encoding of planning problems using ASP.

```
(1)     blk(1). blk(2). blk(3). blk(4). blk(5).

(2)     fluent(on_table(X)):- blk(X).
(3)     fluent(clear(X)):- blk(X).
(4)     fluent(on(X,Y)):- blk(X), blk(Y), diff(X,Y).
(5)     fluent(space_on_table).

(6)     action(move(X,Y)):- blk(X), blk(Y), diff(X,Y).
(7)     action(to_table(X)):- blk(X).

(8)     causes(move(X,Y),clear(Z),[on(X,Z)]):- blk(X), blk(Y),
(9)        blk(Z), action(move(X,Y)),diff(X,Z),diff(Y,Z).
(10)    causes(move(X,Y),on(X,Y),[]):- blk(X), blk(Y),
(11)       action(move(X,Y)).
(12)    causes(move(X,Y),mneg(on(X,Z)),[on(X,Z)]):- blk(X),
(13)       blk(Y), blk(Z), action(move(X,Y)), diff(Y,Z).
(14)    causes(move(X,Y),space_on_table,[on_table(X)]):- blk(X),
(15)       blk(Y), action(move(X,Y)).
(16)    causes(to_table(X),on_table(X),[]):-
(17)       blk(X), action(to_table(X)).
(18)    causes(to_table(X),clear(Y),[on(X,Y)]):- blk(X), blk(Y),
(19)       diff(X,Y), action(to_table(X)).
(20)    causes(to_table(X),mneg(space_on_table),[on_table(Y),on_table(Z)]):-
(21)       blk(X), blk(Y), blk(Z), diff(X,Y,Z),
(22)       action(to_table(X)).

(23)    caused([on(X,Y)],mneg(clear(Y))) :- blk(X), blk(Y), diff(X,Y).
(24)    caused([clear(Y)],mneg(on(X,Y))) :- blk(X), blk(Y), diff(X,Y).
(25)    caused([on(X,Y)],mneg(on_table(X))) :- blk(X), blk(Y), diff(X,Y).
(26)    caused([on_table(X)],mneg(on(X,Y))) :- blk(X), blk(Y), diff(X,Y).
(27)    caused([on(X,Y)],mneg(on(Y,X))) :- blk(X), blk(Y), diff(X,Y).

(28)    executable(move(X,Y),[clear(X),clear(Y)]) :-
(29)       blk(X), blk(Y), action(move(X,Y)).
(30)    executable(to_table(X),[clear(X),mneg(on_table(X)),space_on_table]):-
(31)       blk(X), action(to_table(X)).

(32)    initially(clear(5)).
(33)    initially(mneg(clear(X))) :- blk(X), X<5.
(34)    initially(on_table(1)).
(35)    initially(mneg(on_table(X))) :- blk(X), X>1.
(36)    initially(space_on_table).
(37)    initially(on(X,Y)) :- blk(X), blk(Y), Y<5, X is Y+1.
(38)    initially(mneg(on(X,Y))):- blk(X), blk(Y), X<Y.
(39)    initially(mneg(on(X,Y))) :- blk(X), blk(Y), Y<5,
(40)       diff(Y,X), P is Y+1, diff(X,P).

(41)    goal(on(X,Y)) :- blk(X), blk(Y), Y<4, X is Y+2.
(42)    goal(on_table(1)).   goal(on_table(2)).
(43)    goal(space_on_table).
```

Fig. 8. Planning in blocks world

Instance (using `(6)`)		Plan	*lparse*	SMODELS	CMODELS		SICStus
Blocks	Length	exists			mChaff	Simo	
5	5	N	2.31	0.14	0.02	0.02	0.20
5	6	N	2.29	0.17	0.11	0.06	0.11
5	7	Y	2.34	0.21	0.12	0.10	0.08
6	7	N	7.64	0.32	0.16	0.13	0.31
6	8	N	7.65	0.37	0.19	0.15	1.70
6	9	Y	7.69	0.55	0.27	0.43	0.99
7	9	N	22.96	0.64	0.32	0.27	6.23
7	10	N	23.06	0.75	0.39	0.32	38.24
7	11	Y	23.10	2.15	0.57	1.35	17.40
8	11	N	36.71	1.18	0.63	0.53	154.96
8	12	N	36.81	1.92	0.74	0.62	948.31
8	13	Y	37.10	7.98	2.14	10.36	422.51
9	13	N	98.69	2.25	1.09	0.93	–
9	14	N	98.45	5.99	1.46	1.13	–
9	15	Y	100.01	433.28	4.16	23.07	–
Instance (using `(6')`)		Plan	*lparse*	SMODELS	CMODELS		SICStus
Blocks	Length	exists			mChaff	Simo	
4	4	N	0.40	0.05	0.00	0.00	0.08
4	5	N	0.41	0.06	0.06	0.02	0.01
4	6	Y	0.42	0.06	0.06	0.02	0.01
5	11	N	1.67	0.35	0.18	0.33	2.17
5	12	N	1.68	0.59	0.26	0.64	5.92
5	13	Y	1.68	0.75	0.28	0.50	8.07
6	25	N	3.38	–	685.43	–	–
6	26	N	3.38	–	1173.55	–	–
6	27	Y	3.41	–	1181.99	–	–

Table 1. Planning in blocks world (20 minutes time limit)

6 Conclusions and Future Work

In this paper, we described two alternative approaches to solve planning problems using logic programming technology. The first approach, frequently adopted in the literature, relies on the mapping of an action language specification to an answer set program, and on the use of an answer set solver to compute the plans. The second approach relies instead on the use of constraint logic programming (over finite domains), encoding the problem as a collection of boolean variables and finite domain constraints. We described the two encodings in detail and discuss some preliminary performance results. ASP implementations run faster. However, no grounding is needed for the CLP(FD) approach.

The study presented in this paper is in the same spirit of the recent investigation in benchmarking logic programming systems (e.g., [1, 22, 6, 4]); our hope is that this line of investigation will shed some lights on the relative strengths and weaknesses of CLP and ASP, and suggest ways to integrate their use in the context of planning.

Acknowledgments This work is partially supported by PRIN2005 project 2005015491 on constraints.

References

[1] C. Anger, T. Schaub, and M. Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004.

[2] K. Apt. *Principles of Constraint Programming.* Cambridge Unviversity Press, 2003.
[3] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
[4] A. Dovier, A. Formisano, and E. Pontelli. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In Proc. of *ICLP05*, LNCS 3668, pp. 67–82, 2005.
[5] A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP in tackling hard combinatorial problems. Web: `www.di.univaq.it/~formisano/CLPASP`.
[6] 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.
[7] M. Gelfond. Representing Knowledge in A-Prolog. In *Computational Logic: Logic Programming and Beyond*, Springer Verlag, pp. 413–451, 2002.
[8] M. Gelfond and V. Lifschitz. Action Languages. *Electron. Trans. Artif. Intell.* 2:193–210, 1998.
[9] E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-Based Answer Set Programming. In Proc. of *AAAI'04*, pp. 61–66, AAAI/Mit Press, 2004.
[10] A.K. Jonsson et al. Planning in Interplanetary Space: Theory and Practice. In *AIPS*, 2002.
[11] Y. Lierler and M. Maratea. CMODELS-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Proc. of *LPNMR04*, pp. 346–350. Springer Verlag, 2004.
[12] V. Lifschitz. Answer Set Planning. In Proc. of *ICLP99*, MIT Press, pp. 23–37, 1999.
[13] A. Lopez and F. Bacchus. Generalizing GraphPlann by Formulating Planning as a CSP. In Proc. of IJCAI, 2003.
[14] V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm*, pp. 375–398. Springer Verlag, 1999.
[15] K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
[16] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver. In Proc. of *Design Automation Conference*, ACM Press, pp. 530–535, 2001.
[17] I. Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In *Annals of Math and AI*, 25(3–4):241–273, 1999.
[18] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems.* MIT Press, 2001.
[19] P. Simons. Extending and Implementing the Stable Model Semantics. Doctoral dissertation. Report 58, Helsinki University of Technology, 2000.
[20] T. C. Son, C. Baral, and S. McIlraith. Planning with different forms of domain-dependent control knowledge — an answer set programming approach. In Proc. of *LPNMR01*, Springer Verlag, pp. 226–239, 2001.
[21] M. Thielscher. Reasoning about Actions with CHRs and Finite Domain Constraints. In Proc. of *ICLP02*, LNCS 2401, pp. 70–84, 2002.
[22] M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On Benchmarking Constraint Logic Programming Platforms. *Constraints*, 9(1):5–34, 2004.
[23] 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`.
[24] Web references for some CLP(FD) implementations. SICStus Prolog: `www.sics.se/isl/sicstuswww/site/index.html`. B-Prolog: `www.probp.com`. ECLiPSe: `eclipse.crosscoreoptimization.com`. GNU-Prolog: `pauillac.inria.fr/~diaz/gnu-prolog`.

Skolem functions and Hilbert's ϵ-terms in Free Variable Tableau Systems

Domenico Cantone and Marianna Nicolosi Asmundo

Dipartimento di Matematica e Informatica, Università di Catania
Viale A. Doria 6, I-95125 Catania, Italy
e-mail: cantone@dmi.unict.it, nicolosi@dmi.unict.it

Abstract. We apply the framework of the generic $\boldsymbol{\delta}$-rule presented in [4] to show how to map the $\boldsymbol{\delta}^{\epsilon}$-rule, which uses ϵ-terms as syntactical objects to expand existentially quantified formulae, in the context of standard $\boldsymbol{\delta}$-rules based on Skolem terms. Structural results coming out from such mapping process are discussed.

1 Introduction

Elimination of existential quantifiers in tableau proofs is dealt with by the $\boldsymbol{\delta}$-rule expansion rule, which replaces existentially quantified variables with suitable terms.

The $\boldsymbol{\delta}$-rule was introduced in [22], in the context of ground tableaux, by enriching the signature with a countable collection of new parameters of arity 0. Since then, after the introduction of free variables in semantic tableaux, the $\boldsymbol{\delta}$-rule has undergone several liberalizations aimed at increasing the efficiency of the underlying proof system. Most $\boldsymbol{\delta}$-rule variants are based on the Skolemization technique to produce the terms to be substituted for the existentially quantified variables [10, 13, 2, 1, 3].

Skolemization is one of the most widespread techniques for the elimination of existential quantifiers in the context of automated deduction [19]. One important reason for that has to be attributed to the fact that Skolem terms can be easily manipulated in deductions. In fact they can be treated (both at the syntactical and at the semantical levels) analogously to the terms already present in the initial language. However, if a tableau system adopts a $\boldsymbol{\delta}$-rule variant based on intricate constructions of Skolem terms, proving its soundness may become problematic [1, 3].

In consideration of that, in [4] we have presented a generic $\boldsymbol{\delta}$-rule whose soundness is characterized by some conditions which can be easily instantiated to prove the correctness of known $\boldsymbol{\delta}$-rule variants based on Skolemization.

Additionally, such a general framework allows one to compare structurally the various $\boldsymbol{\delta}$-rule variants in such a way as to appraise in a natural way their efficiency. It also gives information on the choice of the Skolem symbols and on the construction of Skolem terms, augmented signatures, and canonical models.

An approach alternative to the one based on Skolemization has been used in [11], where the $\boldsymbol{\delta}^{\epsilon}$-rule, adopting Hilbert's ϵ-terms in place of Skolem terms, has been defined.

In order to use ϵ-terms as syntactical entities in the context of tableau proofs, a suitable semantics, called *substitutive*, has been introduced in [11], which makes ϵ-terms sensitive to the syntactical manipulations usually applied in automated deduction, such as the substitution of terms in a formula.

The relationship between ϵ-terms and Skolem terms, when used for the purpose of eliminating existential quantifiers, has been discussed in [11]. In this paper we show how the $\boldsymbol{\delta}^{\epsilon}$-rule can be mapped, and therefore can be studied, in the context of $\boldsymbol{\delta}$-rules based on the Skolemization technique. Our generic $\boldsymbol{\delta}$-rule serves to such purpose.

We start by defining a $\boldsymbol{\delta}$-rule variant, called $\boldsymbol{\delta}^{\mathbf{sk}}$-rule, whose soundness follows immediately from the fact that it is an instance of our generic $\boldsymbol{\delta}$-rule. Then the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule is identified with the $\boldsymbol{\delta}^{\epsilon}$-rule in the context of a same tableau system. This suggests a mechanism for the $\boldsymbol{\delta}^{\epsilon}$-rule to construct substitutive prestructures from classical first-order structures, similar to the one used by the generic $\boldsymbol{\delta}$-rule to construct canonical structures. As a by-product, we end up with a new soundness proof of the $\boldsymbol{\delta}^{\epsilon}$-rule, alternative to the one presented in [11].

2 Preliminaries

Before going into details, we review some notation and terminology which will be used throughout the paper.

2.1 Signatures and languages

Let $\Sigma = (\mathcal{P}, \mathcal{F})$ be a *signature*, where $\mathcal{P}$ and $\mathcal{F}$ are countable collections of predicate and function symbols, respectively, and let *Var* be a fixed countable collection of individual variables. Then the *language* $\mathcal{L}_\Sigma$ is the collection of all first-order terms and formulae involving besides the standard logical symbols, also individual variables in *Var*, and predicate and function symbols of the signature Σ. For the sake of simplicity, we will assume that the primitive propositional connectives of $\mathcal{L}_\Sigma$ are $\wedge$, $\vee$, and $\neg$. Thus, other connectives will be used just as shorthands: for instance, the formula $\varphi \supset \psi$ is to be considered as a shorthand for $\neg\varphi \vee \psi$.

Terms and formulae construction rules, notions of free and bound variables, of closed formulae (sentences), of (immediate) subformulae, of occurrences of terms and formulae in a given formula are the standard ones. Precise definitions can be found in [10] and in [15].

For any formula φ in the language $\mathcal{L}_\Sigma$, the collection of free variables occurring in φ is denoted by $\mathit{Free}(\varphi)$, whereas the collection of bound variables in φ is denoted by $\mathit{Bound}(\varphi)$.

It is convenient to assume that the individual variables Var are arranged in a sequence $\langle \ldots, \mathsf{x}_{-2}, \mathsf{x}_{-1}, \mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2, \ldots \rangle$, and that the two subsequences

- $\mathit{Var}^- = \langle \mathsf{x}_{-1}, \mathsf{x}_{-2}, \ldots \rangle$, and
- $\mathit{Var}^+ = \langle \mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2, \ldots \rangle$

are singled out, to denote bound and free variables, respectively.

In this paper, we will explicitly refer to the variables in Var only in Section 5, where quasi-key formulae are defined, and in Section 6, where the result of isomorphism between tableau proofs applying the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule and tableau proofs with the $\boldsymbol{\delta}^{\epsilon}$-rule is discussed. But when it is not necessary to insist on such a convention, we will just use the meta-variables x, y, z (possibly subscribed with natural numbers) standing for generic variables in Var.

By $\mathcal{L}^+_\Sigma$ we denote the collection of all formulae in $\mathcal{L}_\Sigma$ whose free variables are contained in Var^+ and whose bound variables in Var^-. In addition, for any given formula φ in $\mathcal{L}^+_\Sigma$, by $\mathcal{L}^+_\varphi$ we denote the collection of all formulae in $\mathcal{L}^+_\Sigma$ which mention only predicate and function symbols occurring in φ.

Without loss of generality, throughout the paper we will consider only formulae in $\mathcal{L}^+_\Sigma$.

For every $\varphi \in \mathcal{L}^+_\Sigma$, we denote by $[\varphi]$ the equivalence class of all the formulae which are equal to φ up to renaming of variables (even bound ones).

An occurrence of a subformula in a formula φ is *positive* if it is in the scope of an *even* number of negation symbols.

A *(variable-) substitution* is a mapping $\sigma : \mathit{Var}^+ \to \mathit{Terms}^+_\Sigma$, where Terms^+_Σ is the collection of all terms on $\Sigma \cup \mathit{Var}^+$. The action of a substitution is recursively extended to terms and formulae of $\mathcal{L}^+_\Sigma$ as usual. We say that a substitution σ is *free* for a formula φ, if the formula φ and the formula $\varphi\sigma$, resulting from applying σ to φ, have exactly the same occurrences of bound variables. For instance, the substitution $\sigma = \{x \leftarrow f(y)\}$ is free for $(\forall z)P(z, x)$ whereas $\sigma' = \{x \leftarrow h(z)\}$ is not. Notice that by the distinction we have done between variables in Var^+ and in Var^-, in this paper we deal with free substitutions.

Let Φ be a set of terms or a set of quantifier-free formulae. A substitution σ is called a *unifier* for Φ if $|\Phi\sigma| = 1$. A unifier σ for Φ is a *most general unifier (MGU)* if, for every unifier τ for Φ, there exists a substitution θ such that $\tau = \sigma\theta$. For example, the substitutions $\tau = \{y \leftarrow x, x \leftarrow f(x)\}$ and $\sigma = \{x \leftarrow f(y)\}$ are both unifiers for $\Phi = \{R(x), R(f(y))\}$, but only σ is an MGU.

2.2 Structures and assignments

A *structure* $\boldsymbol{\mathcal{M}} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}} \rangle$ for a signature $\Sigma = (\mathcal{P}, \mathcal{F})$ consists of a nonempty domain $\boldsymbol{\mathcal{D}}$ and an interpretation $\boldsymbol{\mathcal{I}}$ for the function and predicate symbols in

Σ such that $P^{\mathcal{I}} : \mathcal{D}^{\mathrm{arity}(P)} \to \{\mathbf{true}, \mathbf{false}\}$, for every predicate symbol $P \in \mathcal{P}$, and $f^{\mathcal{I}} : \mathcal{D}^{\mathrm{arity}(f)} \to \mathcal{D}$, for every function symbol $f \in \mathcal{F}$.

An *assignment* $\mathcal{A}$ relative to a structure $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ and to a language $\mathcal{L}_\Sigma$ is a mapping $\mathcal{A} : \mathit{Var} \to \mathcal{D}$. An *x-variant* of an assignment $\mathcal{A}$ is an assignment $\mathcal{A}'$ such that $y^{\mathcal{A}'} = y^{\mathcal{A}}$, for every variable y different from x. We use the notation $\mathcal{A}[x \leftarrow \boldsymbol{d}]$ to denote the x-variant of $\mathcal{A}$ such that $x^{\mathcal{A}} = \boldsymbol{d}$, for any $\boldsymbol{d} \in \mathcal{D}$.

The notions of satisfiability and validity of a (set of) formula(e) are the standard ones. So, for instance, given a structure $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ and an assignment $\mathcal{A}$ relative to $\mathcal{M}$, we write $(\mathcal{M}, \mathcal{A}) \models \varphi$ or $\varphi^{\mathcal{I},\mathcal{A}} = \mathbf{true}$ to express the fact that the formula φ is true when its predicate and function symbols are interpreted by $\mathcal{I}$, its free variables are interpreted according to $\mathcal{A}$, and its logical symbols have their standard meaning. The notation $\mathcal{M} \models \varphi$ is used to indicate that $(\mathcal{M}, \mathcal{A}) \models \varphi$, for every assignment $\mathcal{A}$, whereas $\models \varphi$ denotes that $\mathcal{M} \models \varphi$, for every structure $\mathcal{M}$ over the signature of the language (in this case we say that φ is valid). Again, further details are available in [10].

2.3 Unifying notation

For the sake of simplicity we use Smullyan's unifying notation, which has the advantage of being compact, thus cutting down on the number of cases that must be considered in proofs. Smullyan divides the formulae of the language into four categories: conjunctive, disjunctive, universal, and existential formulae (called $\boldsymbol{\alpha}$-, $\boldsymbol{\beta}$-, $\boldsymbol{\gamma}$-, and $\boldsymbol{\delta}$-formulae, respectively). In particular, $\boldsymbol{\delta}$-formulae are those of the form $(\exists x)\varphi$ and $\neg(\forall x)\varphi$, whereas $\boldsymbol{\gamma}$-formulae are those of the form $(\forall x)\varphi$ and $\neg(\exists x)\varphi$.

Given a $\boldsymbol{\delta}$-formula δ, the notation $\delta_0(x)$ will be used to denote the formula φ, if δ is of the form $(\exists x)\varphi$, or to denote the formula $\neg\varphi$, if δ is of the form $\neg(\forall x)\varphi$. In any case, we will refer to $\delta_0(x)$ as *the instance of* δ and to x as *the quantified variable of* δ.

Likewise, for any $\boldsymbol{\gamma}$-formula γ, $\gamma_0(x)$ denotes the formula φ or $\neg\varphi$, according to whether γ has the form $(\forall x)\varphi$ or $\neg(\exists x)\varphi$, respectively.[1]

Let us define the complement operator by putting:

$$\complement(X) = \begin{cases} Z & \text{if } X = \neg Z \\ \neg X & \text{otherwise.} \end{cases}$$

Then to each $\boldsymbol{\alpha}$- and $\boldsymbol{\beta}$-formula, one can associate its components as shown in Table 1.

Plainly, the following equivalences hold:

$$\models \alpha \equiv \alpha_1 \wedge \alpha_2\,, \quad \models \beta \equiv \beta_1 \vee \beta_2\,, \quad \models \gamma \equiv (\forall x)\gamma_0(x)\,, \quad \models \delta \equiv (\exists x)\delta_0(x)\,.$$

[1] To simplify the exposition, throughout the paper we will feel free to use the same meta-variable x to represent $(Qx)\varphi$ (where Q is a quantifier) and its instance $\varphi(x)$, despite the fact that they stand for different individual variables of Var. In fact, the occurrences of x in $(Qx)\varphi$ stand for a variable in Var^-, whereas the ones in $\varphi(x)$ stand for a variable in Var^+.

α	α_1	α_2
$X \wedge Y$	X	Y
$\neg(X \vee Y)$	$\mathsf{C}(X)$	$\mathsf{C}(Y)$
$\neg(X \supset Y)$	X	$\mathsf{C}(Y)$
$\neg\neg X$	X	–

β	β_1	β_2
$X \vee Y$	X	Y
$\neg(X \wedge Y)$	$\mathsf{C}(X)$	$\mathsf{C}(Y)$
$(X \supset Y)$	$\mathsf{C}(X)$	Y

Table 1. $\boldsymbol{\alpha}$- and $\boldsymbol{\beta}$-components.

$$\frac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta}{\beta_1 \mid \beta_2} \qquad \frac{\gamma}{\gamma_0(x)} \qquad \frac{\delta}{\delta_0(f(\overrightarrow{S}))}$$

Table 2. Tableau rules for a generic calculus.

2.4 Free variable semantic tableaux

Tableaux are proof systems based on a systematic decomposition of the (set of) formula(e) to be proved till a manifest contradiction is found. Decomposition is performed by suitable expansion rules, whereas contradictions are detected by a closure rule. Table 2 presents the rules of a generic free variable tableau calculus (we refer the reader to [7] and [12] for a deeper treatment of the tableau method). The $\boldsymbol{\alpha}$-rule is used to expand conjunctive formulae ($\boldsymbol{\alpha}$-formulae). It decomposes a formula α into its components α_1 and α_2 as described in Table 1, giving rise to a single expansion. If α is of type $\neg\neg X$, then α_2 is not present in the $\boldsymbol{\alpha}$-rule.

Analogously, the $\boldsymbol{\beta}$-rule is employed to decompose disjunctive formulae ($\boldsymbol{\beta}$-formulae). The components β_1 and β_2 of a $\boldsymbol{\beta}$-formula β are determined as illustrated in Table 1. Further, the presence of a vertical bar in the schema of the $\boldsymbol{\beta}$-rule indicates that its application originates a branch splitting.

The $\boldsymbol{\gamma}$-rule decomposes universal formulae, instantiating them with new free variables. To speed-up tableau closure, it is convenient to require that the variable x in the $\boldsymbol{\gamma}$-rule be a free variable in Var^+, new to the current tableau.

Expansion of $\boldsymbol{\delta}$-formulae is performed by the $\boldsymbol{\delta}$-rule which consists in instantiating the existentially quantified formula with a Skolem term $f(\overrightarrow{S})$, where f is a function symbol and $\overrightarrow{S}$ an ordered tuple of terms which have to satisfy suitable requirements.

We postpone the precise definition of $\boldsymbol{\delta}$-rule and Skolem terms to Section 3. There we will characterize the proviso of the $\boldsymbol{\delta}$-rule in such a way as to enforce soundness and encompass the $\boldsymbol{\delta}$-rule variants present in the literature which are based on Skolemization. Here we just recall that Skolem terms are constructed

using function symbols from a countably infinite set **sko**, disjoint from $\mathcal{F}$ and such that it contains countably many distinct function symbols of any arity. We indicate with $\Sigma_{\mathbf{sko}} = (\mathcal{P}, \mathcal{F} \cup \mathbf{sko})$ the augmented signature.

2.5 Tableau proofs

We represent a tableau $\mathcal{T}$ for a set of closed formulae Φ of $\mathcal{L}^+_\Sigma$ as a dyadic ordered tree whose nodes are labelled with formulae of $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$. Any maximal path on $\mathcal{T}$ is a *branch* of $\mathcal{T}$.

Given a set Φ of closed formulae of $\mathcal{L}^+_\Sigma$, a *free variable tableau* for Φ is recursively defined as follows:

- The tree with only one node labelled with **true** is a tableau for Φ (*initial tableau*);
- Let $\mathcal{T}$ be a tableau for Φ, ϑ a branch of $\mathcal{T}$, and φ a formula occurring in $\vartheta \cup \Phi$. Then the tree $\mathcal{T}'$, obtained from $\mathcal{T}$ by extending ϑ through the application of an expansion rule from Table 2 as described by points 1-4 below, is a tableau for Φ.
  1. If φ is an $\boldsymbol{\alpha}$-formula α, the $\boldsymbol{\alpha}$-rule is applied, and the components α_1 and α_2 are appended to ϑ, thus yielding the extended branch $\vartheta; \alpha_1; \alpha_2$.
  2. If φ is a $\boldsymbol{\beta}$-formula β, the $\boldsymbol{\beta}$-rule is applied, and the two extended branches $\vartheta; \beta_1$ and $\vartheta; \beta_2$ are constructed out of ϑ.
  3. If φ is a $\boldsymbol{\gamma}$-formula γ, by the application of the $\boldsymbol{\gamma}$-rule, ϑ is extended to $\vartheta; \gamma_0(x)$, where x is a free variable in Var^+, new to the current tableau.
  4. If φ is a $\boldsymbol{\delta}$-formula δ, the $\boldsymbol{\delta}$-rule is applied, and $\delta_0(f(\overrightarrow{S}))$ is appended to ϑ yielding the extended branch $\vartheta; \delta_0(f(\overrightarrow{S}))$, for a suitable term $f(\overrightarrow{S})$.
- Let $\mathcal{T}$ be a tableau for Φ, ϑ a branch of $\mathcal{T}$, and ψ, ψ' literals on ϑ. If ψ and $\complement(\psi')$ are unifiable with MGU σ, and $\mathcal{T}' = \mathcal{T}\sigma$ is obtained by applying the substitution σ to all the formulae on $\mathcal{T}$, then $\mathcal{T}'$ is a tableau for Φ.

Given a structure $\boldsymbol{\mathcal{M}} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}} \rangle$ for $\Sigma_{\mathbf{sko}}$ and an assignment $\mathcal{A}$ relative to $\boldsymbol{\mathcal{M}}$, a branch ϑ of a tableau $\mathcal{T}$ is *satisfiable* by $\boldsymbol{\mathcal{M}}$ and $\mathcal{A}$, in which case we write $(\boldsymbol{\mathcal{M}}, \mathcal{A}) \models \vartheta$, if $(\boldsymbol{\mathcal{M}}, \mathcal{A}) \models \varphi$ holds, for each φ in the branch ϑ.

A tableau $\mathcal{T}$ for a set of closed formulae Φ of $\mathcal{L}^+_\Sigma$ is *satisfiable* if there exists a structure $\boldsymbol{\mathcal{M}} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}} \rangle$ for $\Sigma_{\mathbf{sko}}$ such that, for every variable assignment $\mathcal{A}$, $(\boldsymbol{\mathcal{M}}, \mathcal{A}) \models \vartheta$ holds for some branch ϑ of $\mathcal{T}$. In such a case, we say that $\boldsymbol{\mathcal{M}}$ is a *model* of $\mathcal{T}$, and also write $\boldsymbol{\mathcal{M}} \models \mathcal{T}$.

A branch ϑ of a tableau $\mathcal{T}$ is *closed* if it contains two literals ψ and ψ' such that $\psi{=}\complement(\psi')$.

A tableau $\mathcal{T}$ is *closed* if all of its branches are closed.

A *tableau proof* of the unsatisfiability of a set Φ of formulae of $\mathcal{L}^+_\Sigma$ is a closed tableau for Φ. A formula φ is a *theorem* of a tableau calculus if the latter produces a closed tableau for $\{\neg\varphi\}$. A tableau calculus is *sound* if every theorem

that it proves is a valid formula, whereas it is *complete* if every valid formula is a theorem of it.

It is useful to index the branches of a tableau as well as the formula occurrences over each of them with incremental natural numbers. In this way the position of the occurrence of a formula φ over a tableau $\mathcal{T}$ is uniquely determined by a pair (n, m) where n is the index of ϑ, the branch of $\mathcal{T}$ where the occurrence of φ lies, and m is the index of the occurrence of φ over ϑ.

Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be variants of the generic tableau calculus introduced above, $\mathcal{T}_1$ a tableau relative to $\mathcal{C}_1$ over the language $\mathcal{L}^+_{\Sigma_1}$, and $\mathcal{T}_2$ a tableau relative to $\mathcal{C}_2$ over the language $\mathcal{L}^+_{\Sigma_2}$.

$\mathcal{T}_1$ is *isomorphic* to $\mathcal{T}_2$ if there exists a *bijection* $\mathsf{f} : \mathcal{L}^+_{\Sigma_1} \to \mathcal{L}^+_{\Sigma_2}$ such that

- $\mathcal{T}_1$ is an initial tableau for a set Φ of closed formulae of $\mathcal{L}^+_{\Sigma_1}$, *if and only* if $\mathcal{T}_2$ is an initial tableau for the set of closed formulae $\Phi' = \{\mathsf{f}(\varphi) : \varphi \in \Phi\}$;
- If $\bar{\mathcal{T}}_1$ is a tableau for Φ relative to $\mathcal{C}_1$ isomorphic to the tableau $\bar{\mathcal{T}}_2$ for Φ' relative to $\mathcal{C}_2$, if ϑ_1 is the branch of index n on $\bar{\mathcal{T}}_1$, and φ the formula occurrence of index m on $\vartheta_1 \cup \Phi$,

  then, $\mathcal{T}_1$ is obtained from $\bar{\mathcal{T}}_1$ by extending ϑ_1 through the application of an expansion rule from the calculus $\mathcal{C}_1$, *if and only if* $\mathcal{T}_2$ is obtained from $\bar{\mathcal{T}}_2$ by extending its branch of index n, ϑ_2, through the application of an expansion rule from $\mathcal{C}_2$, as described in the following.

  1. if φ is an $\boldsymbol{\alpha}$-formula α, the rule of $\mathcal{C}_1$ yielding the extended branch $\vartheta_1; \alpha_1; \alpha_2$ is applied to $\bar{\mathcal{T}}_1$ *if and only if* the corresponding rule of $\mathcal{C}_2$ yielding the extended branch $\vartheta_2; \mathsf{f}(\alpha_1); \mathsf{f}(\alpha_2)$ is applied to $\bar{\mathcal{T}}_2$.
  2. if φ is a $\boldsymbol{\beta}$-formula β, the rule of $\mathcal{C}_1$ constructing the extended branches $\vartheta_1; \beta_1$ and $\vartheta_1; \beta_2$ out of ϑ_1, is applied to $\bar{\mathcal{T}}_1$ *if and only if* the corresponding rule of $\mathcal{C}_2$, yielding the branches $\vartheta_2; \mathsf{f}(\beta_1)$ and $\vartheta_2; f(\beta_2)$ out of ϑ_2, is applied to $\bar{\mathcal{T}}_2$.
  3. the cases where φ is either a $\boldsymbol{\gamma}$- or a $\boldsymbol{\delta}$-formula are treated in a similar way.

- If $\bar{\mathcal{T}}_1$ is a tableau for Φ relative to $\mathcal{C}_1$ isomorphic to the tableau $\bar{\mathcal{T}}_2$ for Φ' relative to $\mathcal{C}_2$, if ϑ_1 is the branch of index n on $\bar{\mathcal{T}}_1$ and ψ, ψ' literals on ϑ_1 of indexes m and m', respectively, such that ψ and $\mathsf{C}(\psi')$ are unifiable with MGU σ,

  then, $\mathcal{T}_1 = \bar{\mathcal{T}}_1\sigma$, *if and only if* $\mathcal{T}_2 = \bar{\mathcal{T}}_2\mathsf{f}(\sigma)$ where $\mathsf{f}(\sigma)$ is an MGU of $\mathsf{f}(\psi)$, $\mathsf{C}(\mathsf{f}(\psi'))$, and $\mathsf{f}(\psi)$, $\mathsf{f}(\psi')$ are the literals of indexes m and m' on the branch ϑ_2 of index n on $\bar{\mathcal{T}}_2$. Notice that if $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, with $\mathsf{f}(\sigma)$ we indicate the substitution $\{\mathsf{f}(x_1) \leftarrow \mathsf{f}(t_1), \ldots, \mathsf{f}(x_n) \leftarrow \mathsf{f}(t_n)\}$.

3 A generic sound δ-rule for Skolemization based δ-rules

In order to show that a $\boldsymbol{\delta}$-rule variant is sound, one has to prove that it preserves the semantics of existential quantification in tableau proofs.

Thus, soundness of a $\boldsymbol{\delta}$-rule variant would be trivially verified if the effect of its application over the $\boldsymbol{\delta}$-formulae occurring on a tableau $\mathcal{T}$ could be identified with their Skolemization, since, as is well-known, Skolemization preserves the semantics of existential quantification [15].

Definition 1 (Skolemization). *Let Φ be a set of formulae containing a formula $\varphi = (\exists x)\delta_0(x)$, whose free variables are among $x_1, \ldots, x_n$. If f is an n-ary function symbol not occurring in Φ, then the formula $\delta_0(x \leftarrow f(x_1, \ldots, x_n))$ is called a* Skolemization *of φ with respect to Φ.* □

However, several $\boldsymbol{\delta}$-rule variants in the literature do not lend themselves to such a direct identification. For instance, some of them allow to reuse the same Skolem symbol in a proof [2]. Others, in addition to the previous feature, present the characteristic of producing a Skolem term containing only a proper subset of the free variables in the formula to be Skolemized [1, 3].

Example 1. Let $\delta_1 = (\exists x)P(x, x_1)$ and $\delta_2 = (\exists x)P(x, x_2)$ be two $\boldsymbol{\delta}$-formulae occurring on a same branch of a tableau based on the $\boldsymbol{\delta}^{+^+}$-rule [2]. Then δ_1 and δ_2 are expanded with the Skolem terms $f_{[\delta_1]}(x_1)$ and $f_{[\delta_2]}(x_2)$, respectively. But since the $\boldsymbol{\delta}^{+^+}$-rule proviso assigns the same Skolem function symbol to formulae identical up to variable renaming, the symbols $f_{[\delta_1]}$ and $f_{[\delta_2]}$ coincide.

Let us suppose that δ_1 is processed before δ_2. Then the expansion of δ_2 is not a valid Skolemization of δ_2 with respect to the formulae occurring on the tableau, since it uses as Skolem symbol a symbol already present in the proof. □

Example 2. Let $\delta_1 = (\exists x)(P(x, x_1) \wedge Q(x_2, g(x_3)))$ be a $\boldsymbol{\delta}$-formula occurring on a tableau constructed according to the $\boldsymbol{\delta}^*$-rule [1]. The Skolem term provided by the $\boldsymbol{\delta}^*$-proviso to instantiate δ is $f_{[\delta]}(x_1)$ (the interested reader is referred to [1] or to [4] for the details of the construction). Since the variables x_2 and x_3, which occur both free in δ, are not present among the arguments of the term $f_{[\delta]}(x_1)$, such expansion is not a valid Skolemization of δ with respect to any set of formulae Φ containing δ. □

Cases like the one described in Example 1 can be treated by constructing the Skolemization of a $\boldsymbol{\delta}$-formula δ with respect to the set $\Phi = \{\delta\}$ rather than with respect to the set of all the formulae in the current tableau. This choice does not penalize more traditional variants of the $\boldsymbol{\delta}$-rule (such as Fitting's $\boldsymbol{\delta}$-rule [10]

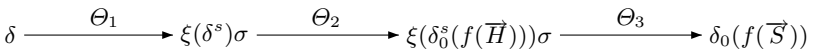


Fig. 1. Characterization of a generic sound $\boldsymbol{\delta}$-rule as a series of satisfiability preserving transformations.

and the $\boldsymbol{\delta}^+$-rule [13]) that require the introduction of a Skolem symbol new to the current tableau at each rule application. In fact it is possible to equip every $\boldsymbol{\delta}$-formula of the language with a countably infinite number of Skolem functions to be "consumed" in the course of the proof.

As witnessed by Example 2, the expedient of constructing the Skolemization of a $\boldsymbol{\delta}$-formula δ with respect to $\Phi = \{\delta\}$ is not enough to guarantee the applicability of the Skolemization to prove the soundness of the generic $\boldsymbol{\delta}$-rule. In fact the employment of $\boldsymbol{\delta}$-rules like the ones presented in [1] and in [4] not always returns the Skolemization of the considered formula. It turns out that Skolemization must be restricted only to a subset of the $\boldsymbol{\delta}$-formulae (called *Skolemizable* formulae) whereas the $\boldsymbol{\delta}$-formulae that are not Skolemizable have to be related to their corresponding Skolemizable formula through suitable satisfiability preserving transformations. In the following, Skolemizable formulae are denoted with δ^s.

According to these considerations, in [4] we have defined a generic $\boldsymbol{\delta}$-rule of the form shown in Table 2, whose soundness can be shown through a series of *satisfiability preserving* transformations, as depicted in Figure 1. We devote the rest of this section to the presentation of such result.

Assuming that our $\boldsymbol{\delta}$-rule instantiates a given $\boldsymbol{\delta}$-formula δ to $\delta_0(f(\overrightarrow{S}))$, then in Figure 1 we have pointed out three such transformations, Θ_1, Θ_2, and Θ_3 (notice that it may be convenient to further split some of them into more basic ones, as will be seen in Section 3.1). In particular,

- Θ_1 transforms the initial $\boldsymbol{\delta}$-formula δ into an intermediate formula $\xi(\delta^s)\sigma$, where δ^s is Skolemizable and occurs only positively in ξ;
- Θ_2 transforms $\xi(\delta^s)\sigma$ into $\xi(\delta_0^s(f(\overrightarrow{H})))\sigma$, where $\overrightarrow{H}$ is a tuple of variables and $\delta_0^s(f(\overrightarrow{H}))$ is a Skolemization of δ^s with respect to $\Phi = \{\delta^s\}$;
- Θ_3 transforms $\xi(\delta_0^s(f(\overrightarrow{H})))\sigma$ into $\delta_0(f(\overrightarrow{S}))$, where $\overrightarrow{S} = \overrightarrow{H}\sigma$.

Thus, with the above approach in mind, we only need to define precisely such satisfiability preserving transformations, which will be done next.

3.1 Skolem terms construction rule

The Skolem term $f(\overrightarrow{S})$ in the $\boldsymbol{\delta}$-rule in Table 2 consists of a function symbol $f \in \mathbf{sko}$ of arity $n \geq 0$ and an n-tuple $\overrightarrow{S}$ of terms in $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$, whose variables belong to Var^+. In general, the constraints that $f(\overrightarrow{S})$ must satisfy may depend on the current tableau $\mathcal{T}$, on the branch ϑ which is about to be expanded, and on the $\boldsymbol{\delta}$-formula δ on ϑ that is about to be instantiated. Notice that the branch ϑ can be encoded by its index in the tableau $\mathcal{T}$ and that the formula δ can be encoded by its position on ϑ. Therefore, if we denote by $Tab^+_{\Sigma_{\mathbf{sko}}}$ the collection of all tableaux on $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$, we can assume that the Skolem term $f(\overrightarrow{S})$ is computed by a certain function

$$\mathcal{S}_{\boldsymbol{\delta}} : Tab^+_{\Sigma_{\mathbf{sko}}} \times \mathbb{N} \times \mathbb{N} \to Terms^+_{\Sigma_{\mathbf{sko}}} ,$$

called *Skolem terms construction rule*.

Next we state how to characterize the Skolem terms construction rule $S_{\boldsymbol{\delta}}$ in order to obtain a sound calculus. We introduce some conditions that guarantee the construction of Skolem terms so as to preserve not only the satisfiability of the $\boldsymbol{\delta}$-formula occurrence which is to be expanded, but also that of the whole tableau. In particular condition **C**0 below describes how the objects needed to construct the Skolem terms are generated, whereas conditions **C**1-**C**7 state the relations that must hold between such objects.

C0**.** We will assume that the $\boldsymbol{\delta}$-rule proviso provides a fixed collection Δ^s of *Skolemizable* $\boldsymbol{\delta}$-formulae of $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$, closed under renaming of variables (even bound ones). In other words, we require that if $\delta^s \in \Delta^s$, then $[\delta^s] \subseteq \Delta^s$.
We will also assume that for each $\delta^s \in \Delta^s$, the $\boldsymbol{\delta}$-rule proviso provides a nonempty set of function symbols $\mathbf{sko}_{[\delta^s]} \subseteq \mathbf{sko}$ such that $\mathbf{sko}_{[\delta^s]}$ contains no function symbol occurring in δ^s and $\text{arity}(f) \geq |Free(\delta^s)|$, for each $f \in \mathbf{sko}_{[\delta^s]}$. Additionally, we will require that the sets $\mathbf{sko}_{[\delta^s]}$ are pairwise disjoint and form a partition of $\mathbf{sko}$. In this way, only formulae that are syntactically identical up to variable renaming may share the same function symbol.
Let δ be a $\boldsymbol{\delta}$-formula at position n in the m-th branch ϑ of a tableau $\mathcal{T}$, which we intend to expand. In order to define $\mathcal{S}_{\boldsymbol{\delta}}(\mathcal{T}, m, n)$, we require that the $\boldsymbol{\delta}$-rule proviso associates to such occurrence of δ the following objects:
- a $\boldsymbol{\delta}$-formula δ^a in the language $\mathcal{L}^+_\delta$, called *abstraction formula of* δ, and a substitution σ;
- a $\boldsymbol{\delta}$-formula δ^s and a formula $\xi = \xi(\delta^s)$ in the language $\mathcal{L}^+_\delta$, respectively called *Skolemization formula* and *transformation formula of* δ;[2]

[2] Given two first-order formulae φ and ψ, we write $\varphi = \varphi(\psi)$ to indicate that the occurrences of ψ in φ play a significant role. Thus, for instance, if ψ' is another formula, by $\varphi(\psi')$ we denote the formula resulting from φ when all occurrences of ψ are replaced by ψ'.

– a function symbol $f \in \mathbf{sko}_{[\delta^s]}$ and an ordered tuple $\overrightarrow{H}$ of variables in *Var* containing all the free variables occurring in δ^s and such that $\text{arity}(f) = |\overrightarrow{H}|$.

The formula δ^a and the substitution σ allow to take care of those $\boldsymbol{\delta}$-rule variants which assign the same function symbol to $\boldsymbol{\delta}$-formulae which are identical up to substitutions [2, 1, 3]. δ^a represents the most general formula with respect to the substitution σ, as specified by the proviso of our generic $\boldsymbol{\delta}$-rule.
Formulae δ^s and ξ allow to encompass those $\boldsymbol{\delta}$-rule variants that construct Skolem terms using only parts of δ^a [1, 3]: the formula ξ has the function of preserving satisfiability in the transformation, whereas δ^s is the formula which is actually Skolemized. $f(\overrightarrow{H})$ is the term used to skolemize δ^s.

We will further assume that such objects satisfy the following conditions:

C1. the substitution σ must be free for the formulae δ^a, $\delta_0^a(f(\overrightarrow{H}))$, $\xi(\delta^s)$, and $\xi(\delta_0^s(f(\overrightarrow{H})))$;

C2. δ^s is a subformula of ξ, occurring *only* positively in it;

C3. the quantified variable of δ^s is the same as the one of δ^a and occurs in ξ only within occurrences of δ^s;

C4. $\models \delta \supset \delta^a\sigma$;

C5. $\models \delta_0^a(x_0) \supset \xi(\delta_0^s(x_0))$;

C6. $\models \delta^a \supset (\forall x_0)(\xi(\delta_0^s(x_0)) \supset \delta_0^a(x_0))$;

C7. $\models (\delta_0^a(f(\overrightarrow{H})))\sigma \supset \delta_0(f(\overrightarrow{H})\sigma)$.

Then we put:

$$\mathcal{S}_{\boldsymbol{\delta}}(\mathcal{T}, m, n) =_{Def} f(\overrightarrow{H})\sigma \,. \tag{1}$$

Soundness of the tableau calculus described in Section 2.4, whose associated Skolem terms construction rule satisfies the above conditions **C**0-**C**7, is proved by showing that tableau satisfiability is preserved by the expansion rules in Table 2 and substitution applications. To this purpose, it is convenient to stratify the language $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$, and then show how we can expand a given structure for $\mathcal{L}^+_{\Sigma}$ to a canonical structure for $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$.

3.2 Stratification of the language $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$

Let $\Sigma = (\mathcal{P}, \mathcal{F})$ be the initial signature of our language. Then we put: $\bar{\mathcal{F}}_0 =_{Def} \mathcal{F}$ and $\Sigma_0 =_{Def} \Sigma$. Moreover, following [2], for $i \geq 1$ we put recursively

$$\begin{aligned} \bar{\mathcal{F}}_i &=_{Def} \bigcup_{\delta^s \in \Delta^s \cap \mathcal{L}^+_{\Sigma_{i-1}}} \mathbf{sko}_{[\delta^s]} \\ \Sigma_i &=_{Def} (\mathcal{P}, \bar{\mathcal{F}}_i) \,. \end{aligned}$$

Without loss of generality, we may assume that $\bigcup_{i=1}^{\infty} \bar{\mathcal{F}}_i = \mathbf{sko}$.[3] Thus we have: $\Sigma_{\mathbf{sko}} = (\mathcal{P}, \bigcup_{i=0}^{\infty} \bar{\mathcal{F}}_i)$.

Relatively to the above stratification of $\Sigma_{\mathbf{sko}}$, it is useful to introduce the following notion of *rank*, for each formula ψ in the language $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$:

$$rank(\psi) =_{Def} \min\{k \in \mathbb{N} : \psi \text{ is in the language } \mathcal{L}^+_{\Sigma_k}\}.$$

3.3 Construction of canonical structures for $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$

Let $\boldsymbol{\mathcal{M}} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}} \rangle$ be a structure for our initial language $\mathcal{L}^+_{\Sigma}$. Let us put $\boldsymbol{\mathcal{I}}_0 =_{Def} \boldsymbol{\mathcal{I}}$ and $\boldsymbol{\mathcal{M}}_0 =_{Def} \boldsymbol{\mathcal{M}}$. Following [2], for $i \geq 1$ we recursively define the expanded structure $\boldsymbol{\mathcal{M}}_i = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}}_i \rangle$ for $\mathcal{L}^+_{\Sigma_i}$ as follows.

For each predicate symbol $P \in \mathcal{P}$ we put $P^{\boldsymbol{\mathcal{I}}_i} =_{Def} P^{\boldsymbol{\mathcal{I}}_{i-1}}$. Likewise, for each function symbol $f \in \bigcup_{j=0}^{i-1} \bar{\mathcal{F}}_j$ we put $f^{\boldsymbol{\mathcal{I}}_i} =_{Def} f^{\boldsymbol{\mathcal{I}}_{i-1}}$.

Finally, for each function symbol $f \in \bar{\mathcal{F}}_i \setminus \bigcup_{j=0}^{i-1} \bar{\mathcal{F}}_j$, we define $f^{\boldsymbol{\mathcal{I}}_i}$ as follows. Let us assume that $f \in \mathbf{sko}_{[\delta^s]}$, for a certain $\boldsymbol{\delta}$-formula $\delta^s \in \Delta^s \cap \mathcal{L}^+_{\Sigma_{i-1}}$, let $k = \text{arity}(f)$, let $\overrightarrow{H}$ be a fixed ordered k-tuple of distinct variables containing all the free variables in δ^s, and let $\overrightarrow{\boldsymbol{b}} \in \boldsymbol{\mathcal{D}}^k$.

We distinguish the following two cases:

(a) if $(\boldsymbol{\mathcal{M}}_{i-1}, \mathcal{A}) \models \delta^s$, for some assignment $\mathcal{A}$ such that $\overrightarrow{H}^{\mathcal{A}} = \overrightarrow{\boldsymbol{b}}$, we put

$$f^{\boldsymbol{\mathcal{I}}_i}(\overrightarrow{\boldsymbol{b}}) =_{Def} \boldsymbol{c} \ , \tag{2}$$

for some $\boldsymbol{c} \in \boldsymbol{\mathcal{D}}$ such that $(\boldsymbol{\mathcal{M}}_{i-1}, \mathcal{A}[x_0 \leftarrow \boldsymbol{c}]) \models \delta^s_0(x_0)$;

(b) otherwise, we put

$$f^{\boldsymbol{\mathcal{I}}_i}(\overrightarrow{\boldsymbol{b}}) =_{Def} \boldsymbol{d} \ , \tag{3}$$

for an arbitrary $\boldsymbol{d} \in \boldsymbol{\mathcal{D}}$.

Finally, we define $\boldsymbol{\mathcal{M}}_{\mathbf{sko}} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}}_{\mathbf{sko}} \rangle$, where $\boldsymbol{\mathcal{I}}_{\mathbf{sko}}|_{\mathcal{P} \cup \mathcal{F}} = \boldsymbol{\mathcal{I}}_0|_{\mathcal{P} \cup \mathcal{F}}$, and $\boldsymbol{\mathcal{I}}_{\mathbf{sko}}|_{\bar{\mathcal{F}}_i} = \boldsymbol{\mathcal{I}}_i|_{\bar{\mathcal{F}}_i}$, for every $i \geq 1$.

Then the soundness of the tableau calculus introduced in Section 2.4, provided that the Skolem terms construction rule is defined as in (1) and conditions **C**0-**C**7 hold, is plainly entailed by the following theorem, whose proof can be found in [4].

Theorem 1. *Assume that the Skolem terms construction rule is defined by (1) and that conditions* ***C****0-****C****7 hold. Let Φ be a set of closed formulae of $\mathcal{L}^+_{\Sigma}$ and let $\boldsymbol{\mathcal{M}} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}} \rangle$ be a structure for $\mathcal{L}^+_{\Sigma}$ such that $\boldsymbol{\mathcal{M}} \models \Phi$. Then $\boldsymbol{\mathcal{M}}_{\mathbf{sko}}$ satisfies any tableau for Φ constructed by the calculus in Section 2.4.*[4]

[3] If $\bigcup_{i=1}^{\infty} \bar{\mathcal{F}}_i \subsetneqq \mathbf{sko}$, then we can replace **sko** by $\bigcup_{i=1}^{\infty} \bar{\mathcal{F}}_i$ in our previous discussion, by also shrinking Δ^s accordingly.

[4] We recall that **sko** and $\boldsymbol{\mathcal{M}}_{\mathbf{sko}}$ have been defined in Sections 2.4 and 3.3, respectively.

4 Hilbert's ϵ-symbol and the δ^{ϵ}-rule

The ϵ-symbol is an operator for the formation of terms that replaces quantifiers in standard predicative logic. ϵ-terms have the shape $\epsilon x.\varphi$ where x is a variable and φ a formula. An ϵ-term $\epsilon x.\varphi$ is interpreted as *an element of the domain satisfying φ if such an element does exist, an arbitrary element of the domain otherwise.*

We refer the reader to [14], for a thorough account of Hilbert's work on the ϵ-symbol, and to [16, 17], for further elaborations and interpretations in the light of more recent developments of mathematical logic.

Apart from foundational issues (ϵ-substitution method, Hilbert's ϵ-theorems, etc.), the ϵ-symbol has also been applied in fields like linguistic, philosophy, and non-classical logics (see for instance [9, 23]).

The use of the ϵ-symbol in automated deduction is rather limited. It has been introduced in the theorem prover Isabelle [18] and, more recently, in the proof verifier AetnaNova [20]. But, to our knowledge, the only effort to use ϵ-terms as syntactical structures to be applied in the context of a standard tableau calculus (with the purpose of effectively substitute Skolem terms in the proofs) is the one reported in [11].

In this section we review the $\boldsymbol{\delta}^{\epsilon}$-rule, the $\boldsymbol{\delta}$-rule variant defined in [11], that expands existentially quantified formulae with ϵ-terms.

Let us introduce some preliminary notions. Let Σ be a signature defined as in Section 2.1. The introduction of the Hilbert's ϵ-symbol among the standard logical symbols results in an enrichment of the language. The rules of formation of terms that are not ϵ-terms and of formulae are the standard ones. So, for instance if f is a symbol in $\mathcal{F}$ of arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms, $f(t_1, \ldots, t_n)$ is a term, and if φ and ψ are formulae, and $\circ$ a binary propositional connective, $\varphi \circ \psi$ is a formula. A totally different discourse holds for ϵ-terms which, in fact, are constructed out of formulae: if φ is a formula and x a variable, then $\epsilon x.\varphi$ is a term. Since ϵ-terms can be used in turn to form other terms and formulae, the ϵ-symbol triggers a mutual recursion between formulae and terms (see [14] for details). We call the resulting language $\mathcal{L}_{\Sigma_\epsilon}$. In this context, notions introduced in Section 2.1 (as, for instance, bound and free variables, substitutions and MGUs) are defined in a similar way, with the only difference that terms are allowed to contain bound variables. Notice in fact that the variable x in the ϵ-term $\epsilon x.\varphi$ is bound.

Even in the present case, assuming that variables in Var^- and Var^+ are employed to denote bound and free variables respectively, we indicate by $\mathcal{L}^+_{\Sigma_\epsilon}$ the collection of all formulae in $\mathcal{L}_{\Sigma_\epsilon}$ whose free variables are in Var^+ and bound variables are in Var^-.

Semantics of $\mathcal{L}^+_{\Sigma_\epsilon}$ is defined, according to [11], through the concept of *pre-structure*. A pre-structure is a triple $\mathfrak{M} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}}, \boldsymbol{\epsilon\text{-}val} \rangle$, where $\boldsymbol{\mathcal{D}}$ and $\boldsymbol{\mathcal{I}}$ are

respectively the domain and the interpretation of a structure $\boldsymbol{\mathcal{M}}$ (see Section 2.2) and $\boldsymbol{\epsilon\text{-}val}$ is a function mapping any ϵ-term $\epsilon x.\varphi$ and variable assignment $\mathcal{A}$ into an element $\boldsymbol{d}$ of the domain.

The evaluation of terms and formulae in a pre-structure $\mathfrak{M}$ is defined as for the case of structures, except for the evaluation of ϵ-terms, where we put $\epsilon x.\varphi^{\boldsymbol{\mathcal{I}},\boldsymbol{\epsilon\text{-}val},\mathcal{A}} \equiv_{Def} \boldsymbol{\epsilon\text{-}val}(\epsilon x.\varphi, \mathcal{A})$. In what follows, we write $\langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}}, \boldsymbol{\epsilon\text{-}val}\rangle, \mathcal{A} \models \varphi$ to indicate that the pre-structure $\mathfrak{M} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}}, \boldsymbol{\epsilon\text{-}val}\rangle$ and the assignment $\mathcal{A}$ satisfy φ.

The $\boldsymbol{\delta^\epsilon}$-rule proviso. The $\boldsymbol{\delta^\epsilon}$-rule is formulated as follows:

$$\frac{\delta}{\delta_0(\epsilon x.\delta_0(x))},$$

where x is the quantified variable in δ and $\epsilon x.\delta_0(x)$ is the ϵ-term associated to δ.

In [11] some restrictions have been applied to the definition of $\boldsymbol{\epsilon\text{-}val}$ with the purpose of providing the ϵ-symbol with a semantics more suitable for the application in automated deduction. Essentially, two different semantics have been introduced in [11]: the *substitutive* semantics, suitable for automated theorem proving, and the *extensional* one, effectively applicable only in interactive theorem provers and requiring the introduction of an additional rule (the ϵ-rule) in the calculus.

Here, we focus on the substitutive semantics, captured by *substitutive pre-structures*, defined as follows.

Definition 2 (Substitutive structure). *A pre-structure $\mathfrak{M}$ for Σ is called* substitutive *if*

1. *given an ϵ-term $\epsilon x.\varphi$ of $\mathcal{L}^+_{\Sigma_\epsilon}$ and two assignments $\mathcal{A}_1, \mathcal{A}_2$ such that $\mathcal{A}_1 \mid_{Free(\exists x\varphi)} = \mathcal{A}_2 \mid_{Free(\exists x\varphi)}$, then $\boldsymbol{\epsilon\text{-}val}(\epsilon x.\varphi, \mathcal{A}_1) = \boldsymbol{\epsilon\text{-}val}(\epsilon x.\varphi, \mathcal{A}_2)$;*
2. *for $y \in Var^+$, $\varphi \in \mathcal{L}^+_{\Sigma_\epsilon}$, assignment $\mathcal{A}$, and term t such that $Free(t) \cap Bound(\epsilon x.\varphi) = \emptyset$, we have*

$$\boldsymbol{\epsilon\text{-}val}(\epsilon x.\varphi\{y \leftarrow t\}, \mathcal{A}) = \boldsymbol{\epsilon\text{-}val}(\epsilon x.\varphi, \mathcal{A}[y \leftarrow t^{\boldsymbol{\mathcal{I}},\boldsymbol{\epsilon\text{-}val},\mathcal{A}}]);$$

3. *for any assignment $\mathcal{A}$ and $\varphi \in \mathcal{L}^+_{\Sigma_\epsilon}$, if $\mathfrak{M}, \mathcal{A} \models (\exists x)\varphi$, then*

$$\mathfrak{M}, \mathcal{A}[x \leftarrow \boldsymbol{\epsilon\text{-}val}(\epsilon x.\varphi, \mathcal{A})] \models \varphi;$$

□

Intuitively, the first condition states that the evaluation of an ϵ-term should depend only on the evaluation of the variables occurring free in it. The second condition expresses the *substitutivity property*, which is very important for

the construction of a calculus, as it reflects at the semantic level the syntactical action of substituting a term in a formula. For instance, it allows to infer $Q(\epsilon y.P(a,y))$ from $(\forall x)Q(\epsilon y.P(x,y))$. Finally, the third condition says that an ϵ-term $\epsilon x.\varphi$ should denote an element of the domain that, assigned to the variable x, satisfies φ.

5 The $\boldsymbol{\delta}^{\mathbf{sk}}$-rule

In this section we introduce the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule, a variant of the $\boldsymbol{\delta}^{*}$-rule, which was first presented in [3] and then revised in [4]. As the $\boldsymbol{\delta}^{*}$-rule, it is based on the *global Skolemization* technique, described in [8] and [6]. The $\boldsymbol{\delta}^{\mathbf{sk}}$-rule proviso does not make use of the concept of relevance and replaces the notion of key formula with the slightly less general one of *quasi-key formula*.

Analogously to key formulae, quasi-key formulae are the formulae of the language most general with respect to substitutions. In particular, they are the only formulae deserving their own Skolem function symbol. Further, to each formula of the language there corresponds a unique (quasi-)key formula. But while between a formula φ and its key formula φ' a relationship holds such that $\varphi = \varphi'\sigma$ up to bounded variables renaming (where σ is a suitable substitution), the relationship between φ and its quasi-key formula φ_1 is simply $\varphi = \varphi_1\sigma$. Prior to formally define quasi-key formulae, we introduce the notion of *quasi-canonical formula*.

Definition 3. *A formula φ is said to be* quasi-canonical *(with respect to the variable x_0) if there is an $n \geq 0$ such that $Free(\varphi) \setminus \{\mathsf{x}_0\} = \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$, each of these variables occurs in φ just once, and they occur in φ in the order $\mathsf{x}_1, \ldots, \mathsf{x}_n$, from left to right.* □

Every formula φ can be canonized with respect to a designated variable $x \in Var^{+}$, in the sense that there exists a unique corresponding canonical formula φ_2, such that φ and $\varphi_2\sigma$ are equal, where σ is a substitution free for φ which maps variables into variables and such that $x = \mathsf{x}_0\sigma$.

Example 3. The quasi-canonical formula with respect to x corresponding to the formula $\varphi = (\exists y)(\exists z)(R(x, f(y), z, h(w,w)) \wedge Q(u,v))$ is

$$\varphi_2 = (\exists y)(\exists z)(R(\mathsf{x}_0, f(y), z, h(\mathsf{x}_1, \mathsf{x}_2)) \wedge Q(\mathsf{x}_3, \mathsf{x}_4))\,.$$

In this case $\sigma = \{\mathsf{x}_0 \leftarrow x, \mathsf{x}_1 \leftarrow w,\ \mathsf{x}_2 \leftarrow w,\ \mathsf{x}_3 \leftarrow u,\ \mathsf{x}_4 \leftarrow u\}$. □

We define quasi-key formulae to be quasi-canonical formulae that are most general with respect to substitutions.

Definition 4. *A formula φ is said to be a* quasi-key formula *if*

– *it is quasi-canonical with respect to* x_0*, and*
– *for all the formulae* ψ *that are quasi-canonical with respect to* x_0*, if there is a substitution* σ *which is free for* ψ *and such that* $\varphi = \psi\sigma$*, then* $\psi = \varphi$. □

To any formula of the language there uniquely corresponds a quasi-key formula, as the following lemma states. The proof is along the lines of the one given in [3] for key formulae.

Lemma 1. *Let* φ *be a formula in the language* $\mathcal{L}^+_\Sigma$ *and let* $x \in Var^+$ *be any variable. Then there exists a* unique *quasi-key formula* φ_1 *with respect to* x*, denoted by* $QKey(\varphi, x)$*, and a nonempty collection of substitutions free for* φ_1*, denoted by* $SubstKey(\varphi, x)$*, such that for each* $\sigma \in SubstKey(\varphi, x)$ *we have*

– φ *and* $\varphi_1\sigma$ *are identical, and*
– x *does not occur in* $x_i\sigma$*, for* $x \neq x_i$.

□

Example 4. We continue from Example 3. The quasi-key formula with respect to x corresponding to

$$\varphi = (\exists y)(\exists z)(R(x, f(y), z, h(w, w)) \wedge Q(u, v))$$

is

$$\varphi_1 = (\exists y)(\exists z)(R(\mathsf{x}_0, f(y), z, \mathsf{x}_1) \wedge Q(\mathsf{x}_2, \mathsf{x}_3)).$$

A substitution satisfying the conditions of the above lemma is $\sigma' = \{\mathsf{x}_0 \leftarrow x, \mathsf{x}_1 \leftarrow h(w, w,), \mathsf{x}_2 \leftarrow u, \mathsf{x}_3 \leftarrow v\}$. □

The $\boldsymbol{\delta^{sk}}$-rule proviso.

Definition 5. *Let* δ *be a* $\boldsymbol{\delta}$*-formula of the language* $\mathcal{L}^+_\Sigma$*. Let* $\varphi_1 = QKey(\delta_0(x), x)$ *and let* $\sigma \in SubstKey(\delta_0(x), x)$*, where* x *is the quantified variable of* δ *and let* $S_{\varphi_1} = Free(\varphi_1)\backslash\{x\}$*. Then the* $\boldsymbol{\delta^{sk}}$*-rule can be schematically described as follows:*

$$\frac{\delta}{\delta_0(h_{\varphi_1}(\overrightarrow{S}_{\varphi_1})\sigma)} \tag{4}$$

□

The $\boldsymbol{\delta^{sk}}$-rule is sound. Soundness of the $\boldsymbol{\delta^{sk}}$-rule is proved by instantiating our generic $\boldsymbol{\delta}$-rule as follows:

– Δ^s is the collection of all $\boldsymbol{\delta}$-formulae $\delta^s = (\exists x)\, QKey(\delta_0(x), x)$, where δ is a $\boldsymbol{\delta}$-formula of $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$.

– For each $\delta^s \in \Delta^s$, $\mathbf{sko}_{[\delta^s]}$ is a set of function symbols new to δ^s, containing countably many symbols of arity equal to $|Free(\delta^s)|$, one for each quasi-key formula of $[\delta^s]$; additionally, we require that the sets $\mathbf{sko}_{[\delta^s]}$ are pairwise disjoint and form a partition of $\mathbf{sko}$.
– Given a $\boldsymbol{\delta}$-formula δ occurring on a branch θ of a tableau $\mathcal{T}$, we put:
  - $\delta^a = \delta^s = \xi = (\exists x)QKey(\delta_0(x), x)$,
  - σ is the substitution such that $\delta = \delta^a\sigma$;
  - $\overrightarrow{H}$ is a tuple containing all the free variables in δ^s, ordered as they appear on δ^s from left to right, while f is the function symbol in $\mathbf{sko}_{[\delta^s]}$ associated to δ^s.

It can be easily checked that condition **C**0 is satisfied. Concerning conditions **C**1, **C**2, and **C**3, these are fulfilled by the definition of quasi-key formula. Condition **C**4 is satisfied since $\delta = \delta^a\sigma$, whereas conditions **C**5 and **C**6 are trivially satisfied since $\delta^a = \delta^s = \xi$. Finally, condition **C**7 is fulfilled, in view of the fact that $[\delta_0^a(f(\overrightarrow{H}))]\sigma = \delta_0(f(\overrightarrow{H})\sigma)$.

6 Isomorphism between tableau proofs with the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule and with the $\boldsymbol{\delta}^{\boldsymbol{\epsilon}}$-rule

Let us denote by $\mathcal{C}^{\mathbf{sk}}$ and $\mathcal{C}^{\epsilon}$ the tableau calculi obtained from the one defined in Section 2.4 by instantiating the generic $\boldsymbol{\delta}$-rule to the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule, and by replacing the generic $\boldsymbol{\delta}$-rule with the $\boldsymbol{\delta}^{\epsilon}$-rule, respectively. Then the following result holds.

Theorem 2. *Let φ be a formula of $\mathcal{L}^+_{\Sigma}$. Then a tableau $\mathcal{T}^{\mathbf{sk}}$ for φ, relative to the calculus $\mathcal{C}^{\mathbf{sk}}$, is satisfied by a canonical model $\boldsymbol{\mathcal{M}}_{\mathbf{sko}}$ if and only if there exists a tableau $\mathcal{T}^{\epsilon}$ for φ, relative to the calculus $\mathcal{C}^{\epsilon}$, which is isomorphic to $\mathcal{T}^{\mathbf{sk}}$ and is satisfied by a pre-structure $\mathfrak{M}$ for $\mathcal{L}^+_{\Sigma_\epsilon}$.* □

The intuition behind such result is that the application of either the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule or the $\boldsymbol{\delta}^{\epsilon}$-rule within a fixed, generic tableau calculus, gives rise to identical proofs up to the kind of terms (either Skolem terms or ϵ-terms) used to expand the existentially quantified formulae. Notice also that since the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule is an instance of the generic $\boldsymbol{\delta}$-rule, the soundness of the $\boldsymbol{\delta}^{\epsilon}$-rule can be derived as a direct consequence.

For a proof of Theorem 2 the reader is referred to the extended version of the present paper [5]. Here, we limit ourselves to highlight its main ingredients, namely: (a) the construction of a bijection from $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$ to $\mathcal{L}^+_{\Sigma_\epsilon}$, to map tableaux for $\mathcal{C}^{\mathbf{sk}}$ into isomorphic tableaux for $\mathcal{C}^{\epsilon}$, and vice versa; (b) a mechanism to construct a pre-structure $\mathfrak{M}$ for $\mathcal{L}^+_{\Sigma_\epsilon}$ (which we call *canonical*) out of a canonical model $\boldsymbol{\mathcal{M}}_{\mathbf{sko}}$ for $\mathcal{L}^+_{\Sigma_{\mathbf{sko}}}$ and vice versa, to show the equisatisfiability of isomorphic tableaux.

To such purpose, it is useful to stratify the language $\mathcal{L}^+_{\Sigma_\epsilon}$ and show also how to construct a pre-structure for $\mathcal{L}^+_{\Sigma_\epsilon}$, starting from a pre-structure for $\mathcal{L}^+_{\Sigma}$ (which, basically, can be identified with a standard classical structure).

The construction of canonical pre-structures is based on the adaptation of the concept of quasi-key formula to the formulae of $\mathcal{L}^+_{\Sigma_\epsilon}$. Further, the pre-structure resulting from such construction is substitutive; in fact, it can be proved that it satisfies the three conditions introduced in Definition 2.

The stratified language $\mathcal{L}^+_{\Sigma_\epsilon}$ Let $\mathcal{L}^+_{\Sigma}$ be the language of the sentence to be proved, not containing ϵ-terms.

Then we set $\mathcal{L}^+_{\Sigma_{\epsilon_0}} =_{Def} \mathcal{L}^+_{\Sigma}$ and, for $i \geq 1$, we put recursively

$$\mathcal{L}^+_{\Sigma_{\epsilon_{i+1}}} =_{Def} \mathcal{L}^+_{\Sigma_{\epsilon_i}} \cup \{\varphi : \varphi \text{ contains only } \epsilon\text{-terms } \epsilon x.\varphi' \text{ such that } \varphi' \in \mathcal{L}^+_{\Sigma_{\epsilon_i}}\}.$$

Finally, we put $\mathcal{L}^+_{\Sigma_\epsilon} =_{Def} \bigcup_{i=0}^{\infty} \mathcal{L}^+_{\Sigma_{\epsilon_i}}$.

Further, we define a notion of rank for formulae in $\mathcal{L}^+_{\Sigma_\epsilon}$ by putting

$$rank_\epsilon(\varphi) =_{Def} \min\{i \in \mathbb{N} : \varphi \text{ is in the language } \mathcal{L}^+_{\Sigma_{\epsilon_i}}\}.$$

The notion of quasi-key formula, defined in Section 5, can be easily applied to formulae of $\mathcal{L}^+_{\Sigma_\epsilon}$ as well.

Example 5. The quasi-key formula of $\varphi = P(x, h(z), \epsilon y.Q(z, y))$ with respect to the variable x is $\varphi_1 = P(\mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2)$, where $\sigma = \{\mathsf{x}_0 \leftarrow x, \mathsf{x}_1 \leftarrow h(z), \mathsf{x}_2 \leftarrow \epsilon y.Q(z, y)\}$ is a substitution such that $\varphi = \varphi_1\sigma$. □

Canonical pre-structures. A canonical pre-structure $\mathfrak{M} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}}, \boldsymbol{\epsilon\text{-}val} \rangle$ for the language $\mathcal{L}^+_{\Sigma_\epsilon}$ is defined from a first-order structure $\boldsymbol{\mathcal{M}} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}} \rangle$ for $\mathcal{L}^+_{\Sigma}$, by constructing the function $\boldsymbol{\epsilon\text{-}val}$ according to the stratification of the language as shown in below.

For every $\boldsymbol{\delta}$-formula δ of $\mathcal{L}^+_{\Sigma_\epsilon}$, let $[\mathcal{A}]_\delta$ be the equivalence class of all the assignments $\mathcal{A}_1, \mathcal{A}_2$ such that $\mathcal{A}_1 \mid_{Free(\delta)} = \mathcal{A}_2 \mid_{Free(\delta)}$. We introduce a family of functions $\{\boldsymbol{\epsilon\text{-}key}_i\}_{i\geq 1}$ to interpret all the quasi-key formulae of $\mathcal{L}^+_{\Sigma_\epsilon}$, in such a way that each $\boldsymbol{\epsilon\text{-}key}_i$ maps each pair $\langle \epsilon x.\delta_0, [\mathcal{A}]_\delta \rangle$, where $\epsilon x.\delta_0$ is the ϵ-term relative to a quasi-key formula δ_0 of rank $i-1$, to an element of $\boldsymbol{\mathcal{D}}$. Further, we define the family of functions $\{\boldsymbol{\epsilon\text{-}val}_i\}_{i\geq 0}$, to extend the interpretation to all the formulae of the language.

– For every $\boldsymbol{\delta}$-formula δ of $\mathcal{L}^+_{\Sigma_\epsilon}$ and assignment $\mathcal{A}$, we put

$$\boldsymbol{\epsilon\text{-}val}_0(\epsilon x.\delta_0, \mathcal{A}) =_{Def} \boldsymbol{d},$$

where $\boldsymbol{d}$ is an arbitrary element of the domain $\boldsymbol{\mathcal{D}}$.

(1.) $P(x_1, x_2)$	(1.) $P(x_1, x_2)$
(2.) $(\forall z)(\exists x)\neg P(x, g(z))$	(2.) $(\forall z)(\exists x)\neg P(x, g(z))$
(3.) $(\exists x)\neg P(x, g(x_3))$	(3.) $(\exists x)\neg P(x, g(x_3))$
(4.) $\neg P(f(g(x_3)), g(x_3))$	(4.) $\neg P(\epsilon x.\neg P(x, g(x_3))), g(x_3))$

Fig. 2. Two isomorphic tableaux relative to the calculi $\mathcal{C}^{\mathbf{sk}}$ and $\mathcal{C}^{\epsilon}$, for the formula in Example 6.

– For every quasi-key formula δ_0 of rank i, with $i \geq 0$, if $\langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}}, \boldsymbol{\epsilon\text{-}val}_i \rangle, \mathcal{A}' \models \delta$, for any $\mathcal{A}'$ in $[\mathcal{A}]_\delta$, we put

$$\boldsymbol{\epsilon\text{-}key}_{i+1}(\epsilon x.\delta_0, [\mathcal{A}]_\delta) =_{Def} \boldsymbol{c}\,,$$

where $\boldsymbol{c}$ is an element of $\boldsymbol{\mathcal{D}}$ such that $\langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}}, \boldsymbol{\epsilon\text{-}val}_i \rangle, \mathcal{A}'[x \leftarrow \boldsymbol{c}] \models \delta_0$, otherwise we put

$$\boldsymbol{\epsilon\text{-}key}_{i+1}(\epsilon x.\delta_0, [\mathcal{A}]_\delta) =_{Def} \boldsymbol{d}\,,$$

where $\boldsymbol{d}$ is an arbitrary element of the domain.

– For every formula δ of $\mathcal{L}^+_{\Sigma_\epsilon}$ and assignment $\mathcal{A}$, we define $\boldsymbol{\epsilon\text{-}val}_{i+1}(\epsilon x.\delta_0, \mathcal{A})$ by distinguishing the following cases:

- if $rank_\epsilon(\delta) < i$, then we put

$$\boldsymbol{\epsilon\text{-}val}_{i+1}(\epsilon x.\delta_0, \mathcal{A}) =_{Def} \boldsymbol{\epsilon\text{-}val}_i(\epsilon x.\delta_0, \mathcal{A})\,;$$

- if $rank_\epsilon(\delta) = i$, φ is the quasi-key formula of δ, and σ is a substitution such that $\delta_0 = \varphi\sigma$, then we put

$$\boldsymbol{\epsilon\text{-}val}_{i+1}(\epsilon x.\delta_0, \mathcal{A}) =_{Def} \boldsymbol{\epsilon\text{-}key}_{rank_\epsilon(\varphi)+1}(\epsilon x.\varphi, [\mathcal{A}_1]_{(\exists x\varphi)})\,,$$

where $\mathcal{A}_1 = \mathcal{A}[x_j \leftarrow (x_j\sigma)^{\boldsymbol{\mathcal{I}}, \boldsymbol{\epsilon\text{-}val}_i, \mathcal{A}}]_{x_j \in Free(\exists x\varphi)}$;

- otherwise, namely when $rank_\epsilon(\delta) > i$, we put

$$\boldsymbol{\epsilon\text{-}val}_{i+1}(\epsilon x.\delta_0, \mathcal{A}) =_{Def} \boldsymbol{d}\,,$$

where $\boldsymbol{d}$ is an arbitrary element of the domain.

Finally, we put $\boldsymbol{\epsilon\text{-}val}(\epsilon x.\delta_0, \mathcal{A}) =_{Def} \boldsymbol{\epsilon\text{-}val}_{rank_\epsilon(\delta)+1}(\epsilon x.\delta_0, \mathcal{A})$.

Example 6. Let us consider the unsatisfiable formula $\varphi = (\forall x)(\forall y)(P(x, y) \wedge (\forall z)(\exists x)\neg P(x, g(z))$. In Figure 2 we describe two isomorphic tableaux for φ. The first one is relative to the calculus $\mathcal{C}^{\mathbf{sk}}$, the second to $\mathcal{C}^{\epsilon}$. For space reasons we have not reported the application of the $\boldsymbol{\alpha}$-rule and the first two applications

of $\boldsymbol{\gamma}$-rule, resulting in the instantiation of the formula with the free variables x_1, x_2.

Both tableaux can be expanded to closed tableaux. The first one through the application of the substitution $\tau_1 = \{x_1 \leftarrow f(g(x_3)), x_2 \leftarrow g(x_3)\}$, the second one by means of $\tau_2 = \{x_1 \leftarrow \epsilon.x\neg P(x, g(x_3)), x_2 \leftarrow g(x_3)\}$.

The two tableaux are identical up to the terms used for the instantiation of the $\boldsymbol{\delta}$-formula. $(\exists x)\neg P(x, g(x_3))$ has as quasi-key formula $\varphi_1 = \neg P(\mathsf{x}_0, \mathsf{x}_1)$ and as corresponding substitution $\sigma = \{\mathsf{x}_0 \leftarrow x, \mathsf{x}_1 \leftarrow g(x_3)\}$. The Skolem term $f(g(x_3))$ used by the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule in the first proof is equal to $f(\mathsf{x}_1)\sigma$, where f is the Skolem symbol associated to φ_1. The ϵ-term $\epsilon x.\neg P(x, g(x_3))$ used by the $\boldsymbol{\delta}^{\epsilon}$-rule in the second proof is equal to $\epsilon x.\neg P(x, \mathsf{x}_1)\sigma$, where $\epsilon x.\neg P(x, \mathsf{x}_1)$ is the ϵ-term associated to φ_1 when x_0 instantiates the quantified variable x. □

7 Conclusions

The result of isomorphism between tableaux adopting the $\boldsymbol{\delta}^{\epsilon}$-rule and tableaux applying the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule witnesses that the Skolem terms constructed by the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule and the ϵ-terms produced by the $\boldsymbol{\delta}^{\epsilon}$-rule can be considered as being essentially the same thing.

Such result is a point of conjuncture of two tendencies. The first one, relative to $\boldsymbol{\delta}$-rule variants based on Skolemization, relies on the construction of Skolem terms in a more natural way, reflecting the meaning of the instantiation. The second one, relative to the $\boldsymbol{\delta}^{\epsilon}$-rule, is based on the treatment of ϵ-terms as common terms of the language, where suitable constraints are imposed in the corresponding semantical structures.

The identification of Skolem terms and ϵ-terms has already been carried out in a purely proof theoretic way in [8], in the context of a free variable version of the predicate calculus, applicable in the field of automated deduction.

Our definition of the $\boldsymbol{\delta}^{\mathbf{sk}}$-rule (and, more generally, of our generic $\boldsymbol{\delta}$-rule) is also based on the constructions presented in [8], but keeps into account model-theoretic aspects particularly important in the context of semantic tableaux.

The two approaches to the elimination process of existential quantifiers in the context of semantic tableaux through the introduction of Skolem terms or ϵ-terms needs further investigation. In particular, we intend to compare variants of the $\boldsymbol{\delta}$-rule based on Skolemization with the $\boldsymbol{\delta}^{\epsilon}$-rule, endowed with extensional semantics (more appropriate for interactive theorem provers and calculi with equality). We also plan to further generalize the $\boldsymbol{\delta}$-rule framework introduced in [4], so as to embrace also $\boldsymbol{\delta}$-rules based on ϵ-terms.

References

1. Matthias Baaz and Christian G. Fermüller. Non-elementary speedups between different versions of tableaux. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *Automated Rea-*

soning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX'95, volume 918 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 1995.

2. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. The even more liberalized delta-rule in free variable semantic tableaux. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Third Kurt Gödel Colloquium*, volume 713 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 1993.
3. Domenico Cantone and Marianna Nicolosi Asmundo. A further and effective liberalization of the *delta*-rule in free variable semantic tableaux. In Ricardo Caferra and Gernot Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2000.
4. Domenico Cantone and Marianna Nicolosi Asmundo. A sound framework for δ-rule variants in free variable semantic tableaux. In Reinhold Letz, editor, *Proc. of the Fifth International Workshop on First-Order Theorem Proving (FTP 2005)*, Research Report 13/2005, pages 51–69. Research Report 13/2005, Universität Koblenz-Landau, Institut für Informatik, 2005. A revised version of the paper will appear in the special issue of the *Journal of Automated Reasoning* on *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2005)*.
5. Domenico Cantone and Marianna Nicolosi Asmundo. *Skolem functions and Hilbert's ϵ-terms in free variable tableau systems (extended version)*. Available at `http://www.dmi.unict.it/~cantone/papers/CNA06ext.pdf`, 2006.
6. Domenico Cantone, Marianna Nicolosi Asmundo, and Eugenio G. Omodeo. Global Skolemization with grouped quantifiers. In Moreno Falaschi, Marisa Navarro, and Alberto Policriti, editors, *Proc. of 1997 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'97*, pages 405–414, 1997.
7. Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors. *Handbook of Tableau Methods*. Kluwer, Dordrecht, 1999.
8. Martin Davis and Ronald Fechter. A free variable version of the first-order predicate calculus. *J. Log. Comput.*, 1(4):431–451, 1991.
9. Melvin C. Fitting. A modal logic ε-calculus. *Notre Dame J. Formal Logic*, 16(1):1–16, 1975.
10. Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer-Verlag New York, Inc., 2nd edition, 1996.
11. Martin Giese and Wolfgang Ahrendt. Hilbert's ϵ-terms in automated theorem proving. In *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '99*, volume 1617 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 1999.
12. Reiner Hähnle. Tableaux and related methods. In Robinson and Voronkov [21], chapter 3, pages 100–178.
13. Reiner Hähnle and Peter H. Schmitt. The liberalized delta-rule in free variable semantic tableaux. *J. Autom. Reasoning*, 13(2):211–221, 1994.
14. A. C. Leisenring. *Mathematical Logic and Hilbert's Epsilon-Symbol*. MacDonald, 1969.
15. Reinhold Letz. First-order tableau methods. In Robinson and Voronkov [21], pages 125–196.
16. Grigori Mints. Thoralf Skolem and the epsilon substitution method for predicate logic. *Nordic Journal of Philosophical Logic*, 1:133–146, 1996.
17. Georg Moser. The epsilon substitution method. Master's thesis, University of Leeds, 2000.
18. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
19. Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [21], pages 335–367.
20. Eugenio G. Omodeo, Domenico Cantone, Alberto Policriti, and Jacob T. Schwartz. A computerized Referee. To appear in Marco Schaerf and Oliviero Stock, editors, *Reasoning, Action and Interaction in AI Theories and Systems — Festschrift in Honor of Luigia Carlucci Aiello*, *Lecture Notes in Artificial Intelligence*, Springer, 2006.
21. John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.
22. Raymond M. Smullyan. *First-Order Logic*. Springer-Verlag, Berlin, 1968.
23. Klaus von Heusinger. Definite descriptions and choice functions. In Seiki Akama, editor, *Logic, Language and Computation*, pages 61–91. Kluwer, Dordrecht, 1997.

Towards an efficient relational deductive system for propositional non-classical logics*

Andrea Formisano** and Marianna Nicolosi Asmundo***

Abstract. We describe a relational framework that uniformly supports formalization and automated reasoning in various propositional modal logics. The proof system we propose is a relational variant of the classical Rasiowa-Sikorski proof system. We introduce a compact graph-based representation of formulae and proofs supporting an efficient implementation of the basic inference engine, as well as of a number of refinements.

1 Introduction

In the last decades, relational formalization of many non-classical propositional logics has been systematically and rigorously studied (see, for instance, [26, 28]). Long-standing research and well established results indicate that standard relational structures can be considered as a common core supporting representations of many non-classical propositional logics. In such an algebraic framework several alternatives for automation of (relational) reasoning can be considered as viable. For instance, to mention some of the approaches proposed in literature, we have tableaux systems [17], Gentzen-style systems [23], systems *à la* Rasiowa-Sikorski [29], display calculus [13], and equational proof systems [10].

Once a relational rendering of a modal theorem is obtained through a translation process, possibly together with the relational counterparts of the modal axioms, a relational proof system can be exploited in order to mechanize modal reasoning. This approach can be seen as alternative and complementary to the common *ad hoc* direct inference methods (cf., e.g., [24]).

This paper mainly focuses on Rasiowa-Sikorski systems. The basic constituent of a Rasiowa-Sikorski system is a collection of inference rules. Similarly to the case of tableaux systems [5], given a theorem to be proved, (the search for) a proof is developed through repeated applications of a set of *decomposition rules*. Through these rules the solution of a problem is reduced to the solutions of (syntactically) simpler sub-problems. The proof is completed whenever a success condition becomes satisfied. Success situations are detected by means of so called *closure rules* that, together with the decomposition rules, characterize the proof system. Dually with respect to tableaux systems, in a Rasiowa-Sikorski system the goal consists in proving a formula to be tautological, instead

* This work is partially supported by INTAS Project — *Algebraic and deduction methods in non-classical logic and their applications to computer science* and by TARSKI Project — *Theory and Applications of Relational Structures as Knowledge Instruments COST Action 274*. (An extended version of this paper will appear in J. Appl. Non-Classical Logics.)

** Università di L'Aquila, formisano@di.univaq.it

*** Università di Catania, nicolosi@dmi.unict.it

of unsatisfiable. For this reason Rasiowa-Sikorski systems are sometimes called *dual-tableaux* systems.

Let us consider any non-classical logic $\mathcal{L}$. Whenever a translation method for formulae of $\mathcal{L}$ into the relational calculus is known, proving a modal theorem φ of $\mathcal{L}$ amounts to prove validity of its relational formulation $t_{\mathcal{L}}(\varphi)$. Different translations are needed to deal with different logics since the proper (modal) axioms characterizing the specific logic have to be reflected in the relational framework. This usually corresponds to identify a collection $\mathcal{C}$ of constant relations to be interpreted as relational counterparts of modal accessibility relations. Proper modal axioms are then rendered by imposing relational axioms which restrain the admitted interpretations of constants in $\mathcal{C}$. Moreover, in the context of a Rasiowa-Sikorski system, properties of relations in $\mathcal{C}$ can also be dealt with by enriching the collection of inference rules of the system. Specific decomposition rules and refined closure rules are then added to the basic inference system.

A goal of the research described here consists in the realization of a relational Rasiowa-Sikorski system suitable to uniformly support modal reasoning for any relationally expressible propositional logic. The duality results linking tableaux and dual-tableaux constitute one of the starting points of this research. In developing the system we are going to describe, we fruitfully adapted and combined several techniques and proof-strategies independently developed for tableaux systems, as well as a compact representation for relational expressions akin to Decision Diagrams. Such techniques, to the best of our knowledge, have never been combined together in the realization of a (relational) deductive framework.

2 A relational logic for propositional non-classical logics

In this section we briefly describe a generic relational logic based on algebras of relations constituting a common framework in which the relational rendering of many non-classical propositional logics can be embedded and homogeneously treated (several examples of such logics are given in Sec. 2.2).

In such a homogeneous framework, all these renderings share basic features: formulae and accessibility relations are represented as binary relations; relational constructs represent extensional and intensional operations; and the deductive apparatus corresponds to a relational calculus.

Let us start by recalling some preliminary notions. Let $\boldsymbol{\mathcal{D}}$ be a non empty set. The full algebra of binary relations over $\boldsymbol{\mathcal{D}}$ is the structure $Re(\boldsymbol{\mathcal{D}}) = (\wp(\boldsymbol{\mathcal{D}} \times \boldsymbol{\mathcal{D}}), \overline{\ }, \cap, \cup, U, Z, ;, \dagger, ^{\smile}, I, D)$. The elements of $Re(\boldsymbol{\mathcal{D}})$ are the subsets of $\boldsymbol{\mathcal{D}} \times \boldsymbol{\mathcal{D}}$. Further, $(\wp(\boldsymbol{\mathcal{D}} \times \boldsymbol{\mathcal{D}}), \overline{\ }, \cap, \cup, U, Z)$ is a Boolean algebra and $(\wp(\boldsymbol{\mathcal{D}} \times \boldsymbol{\mathcal{D}}), ;, ^{\smile}, I)$ is a monoid with involution [1]. Notice that, for the sake of simplicity, we are considering as primitive, or *basic*, all the relational constructs $\overline{\ }, \cap, \cup, ;, \dagger$, and $^{\smile}$ as well as the constants U, Z, I, D. An alternative possibility could consist in introducing a minimal set of primitive constructs (such as $\overline{\ }, \cap, ;, ^{\smile}, I$, for instance) and in defining the remaining ones in term of these.

Occasionally, we will enrich such a basic structure by considering a collection of relational constants (representing as we will see, accessibility relations) and

with a set of *non-standard* relational constructs not definable in terms of the basic ones. A distinction between basic and non-standard relational constructs can be established by classifying a construct as basic if its semantics can be expressed through a first-order sentence in three variables [34]. We will call *non-standard* those constructs that are not expressible in three variables. Recall that establishing the 3-variable expressibility of a sentence is, in general, an undecidable problem [34, 20]. Nevertheless, techniques can be designed to treat particular classes of formulas [3]. Let $\mathcal{L}$ be a non-classical logic and $Rel\mathcal{L}$ its relational rendering. In what follows we introduce a common syntax and semantics for $Rel\mathcal{L}$ whereas in Sec. 3 we develop the corresponding deductive apparatus.

2.1 Syntax and semantics of $Rel\mathcal{L}$

Let $\mathcal{V}$ be a set of individual variables (denoted by u, w, x, y, z, or occasionally by e, t, possibly subscribed), $\mathcal{R}$ a set of relational variables (P,Q,R,...), and $\mathcal{C}$ a set of relational constants. A *relational expression* is any term generated from the symbols in $\mathcal{R} \cup \mathcal{C} \cup \{U, Z, I, D\}$ and the relational constructs. The collection of all the relational expressions is denoted by $\mathcal{E}$. As usual, given a binary relational construct $\circ$, its dual $\diamond$ is defined as $P \diamond Q =_{\text{Def}} \overline{\overline{P} \circ \overline{Q}}$ (and similarly for monadic constructs different from complementation).

A *relational equation* is a writing of the form $R{=}Q$, with $R, Q \in \mathcal{E}$. Shorthand notations for equalities of special kind are also possible, e.g.: $P{\sqsubseteq}Q \leftrightarrow_{\text{Def}} P{-}Q{=}Z$.

A *relational formula* is a writing of the form $x\varphi y$ where $\varphi \in \mathcal{E}$ is a relational expression and $x, y \in \mathcal{V}$ are individual variables. If $\varphi \in \mathcal{R} \cup \mathcal{C} \cup \{U, Z, I, D\}$, then $x\varphi y$ is said to be an *atomic* relational formula. Any atomic relational formula $x\varphi y$ and its complement $x\overline{\varphi}y$ are *literals*. A formula is *compound* if it is not a literal. The *leading* construct of a compound formula $x\varphi y$ is the dual of $\circ$ if $\varphi = \overline{\psi \circ \phi}$ or $\varphi = \overline{\circ\psi}$. While, it is $\circ$ if $\varphi = \psi \circ \phi$ or $\varphi = \circ\psi$.

Relational formulae are interpreted in semantic structures of the kind $\boldsymbol{\mathcal{M}} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{I}} \rangle$ where $\boldsymbol{\mathcal{D}}$ is a non-empty set and $\boldsymbol{\mathcal{I}}$ is a function assigning:

- A binary relation over $\boldsymbol{\mathcal{D}}$ to each relation symbol in $\mathcal{R} \cup \mathcal{C} \cup \{U, Z, I, D\}$. In particular, U is interpreted as the universal relation, I as the identity, Z as $\overline{U}$, and D as $\overline{I}$. Relational variables are interpreted as right-ideal relations. (A relation R is right-ideal if $R;U = R$.)
- The intended interpretation to the primitive and defined constructs. (Considering, as usual, $\cup$ and $\dagger$ to be the duals of $\cap$ and ;, respectively.)

We call such structures, models of $Rel\mathcal{L}$. An evaluation (or assignment) A in $\boldsymbol{\mathcal{M}}$ is a function $A : \mathcal{V} \to \boldsymbol{\mathcal{D}}$. A relational formula $x\varphi y$ of $Rel\mathcal{L}$ is satisfied by $\boldsymbol{\mathcal{M}}$ and A if $(x^A, y^A) \in \varphi^{\boldsymbol{\mathcal{I}}}$. In this case we write $\boldsymbol{\mathcal{M}}, A \models x\varphi y$. The formula $x\varphi y$ is true in $\boldsymbol{\mathcal{M}}$ if and only if for every evaluation A in $\boldsymbol{\mathcal{M}}$, it holds that $\boldsymbol{\mathcal{M}}, A \models x\varphi y$. A formula $x\varphi y$ of $Rel\mathcal{L}$ is valid if it is satisfied in every model of $Rel\mathcal{L}$.

2.2 Relational formalizations of modal logics

In this section we recall the relational formalizations of various propositional logics [26]. For each logic we outline a syntax-directed translation into the algebra

Table 1. Lattice-based modalities: translations and axioms for constant relations

possibility $\Diamond$	$t(\Diamond\chi) =_{\text{Def}} \overline{\leqslant_1; \overline{S_\Diamond; \leqslant_2; t(\chi)}}$	$\leqslant_1^\smile; R_\Diamond; \leqslant_1^\smile \sqsubseteq R_\Diamond$ $\leqslant_2; S_\Diamond; \leqslant_2 \sqsubseteq S_\Diamond$	$R_\Diamond \sqsubseteq S_\Diamond; \leqslant_1^\smile$ $S_\Diamond \sqsubseteq \leqslant_2; R_\Diamond$
necessity $\Box$	$t(\Box\chi) =_{\text{Def}} \overline{R_\Box; \overline{t(\chi)}}$	$\leqslant_1; R_\Box; \leqslant_1 \sqsubseteq R_\Box$ $\leqslant_2^\smile; S_\Box; \leqslant_2^\smile \sqsubseteq S_\Box$	$R_\Box \sqsubseteq \leqslant_1; S_\Box$ $S_\Box \sqsubseteq R_\Box; \leqslant_2^\smile$
sufficiency $\boxdot$	$t(\boxdot\chi) =_{\text{Def}} \overline{R_\boxdot; \leqslant_2; t(\chi)}$	$\leqslant_1; R_\boxdot; \leqslant_2 \sqsubseteq R_\boxdot$ $\leqslant_2^\smile; S_\boxdot; \leqslant_1^\smile \sqsubseteq S_\boxdot$	$R_\boxdot \sqsubseteq \leqslant_1; S_\boxdot$ $S_\boxdot \sqsubseteq R_\boxdot; \leqslant_1^\smile$
dual sufficiency $\diamondsuit$	$t(\diamondsuit\chi) =_{\text{Def}} \overline{\leqslant_1; \overline{S_\diamondsuit; t(\chi)}}$	$\leqslant_1^\smile; R_\diamondsuit; \leqslant_2^\smile \sqsubseteq R_\diamondsuit$ $\leqslant_2; S_\diamondsuit; \leqslant_1 \sqsubseteq S_\diamondsuit$	$R_\diamondsuit \sqsubseteq S_\diamondsuit; \leqslant_2^\smile$ $S_\diamondsuit \sqsubseteq \leqslant_2; R_\diamondsuit$

of relations enriched with a specific collection $\mathcal{C}$ of constants. Such constants are often subject to a set of axioms restraining their admitted interpretations.

Mono-modal logics. This is the basic translation of (propositional) modal formulae into relational terms. In this case $\mathcal{C} = \{\mathsf{r}\}$. The propositional connectives and the necessity operator are so translated:

$$t(\neg\,\psi) =_{\text{Def}} \overline{t(\psi)} \qquad t(\psi\,\&\,\chi) =_{\text{Def}} t(\psi) \cap t(\chi)$$
$$t(\Diamond\,\psi) =_{\text{Def}} \mathsf{r}\,;\, t(\psi) \qquad t(p_i) =_{\text{Def}} p_i'$$

where $p_i' \in \mathcal{R}$ uniquely corresponding to the propositional variable p_i (similar rules are introduced for the other customary propositional connectives).

Multi-modal logic. These logics correspond to multi-modal frames consisting of a relational system where $\mathcal{C}$ enjoys closure properties with respect to relational constructs. Modalities are then of the form $[R]$ and $\langle R\rangle$, where $R \in \mathcal{E}$ [26]. The translation of modal operators is the same as in the case of mono-modal logic. The differences between operators are articulated in terms of the properties of the corresponding accessibility relations.

Lattice-based modal logics. Lattice-based modal logics have the operations of disjunction and conjunction and, moreover, each of them includes a modal operator which can be either a possibility $\Diamond$, or necessity $\Box$, or sufficiency $\boxdot$, or dual sufficiency operator $\diamondsuit$ (see [8]). Notice that, since negation is not available in these logics, both in the possibility–necessity and in the sufficiency–dual-sufficiency pair, neither operator is expressible in terms of the other. We can also consider mixed languages with any subset of these operators. For all of these logics we have $\{\leqslant_1, \leqslant_2\} \subseteq \mathcal{C}$. Moreover, $\leqslant_1$ and $\leqslant_2$ are assumed to be reflexive and transitive and such that $\leqslant_1 \cap \leqslant_2 = I$. The translations of disjunction and conjunction are:

$$t(\psi \vee \chi) =_{\text{Def}} \overline{\leqslant_1; \overline{\leqslant_2; (\overline{t(\psi)} \cup \overline{t(\chi)})}} \qquad t(\psi\,\&\,\chi) =_{\text{Def}} t(\psi) \cap t(\chi).$$

As regards the alternative modalities, Table 1 summarizes their translations. Each modality is modeled in the relational framework by means of a pair of constant relations subject to suitable axioms (also displayed in Table 1).

Temporal logics. We consider here the relational formalization of temporal logics given in [27]. The modalities referring to states in the future are: $\mathtt{G}\phi$ (interpreted as "ϕ will be always true in the future"); $\mathtt{F}\phi$ ("ϕ will be true sometime in the future"); $\phi\,\mathtt{U}\,\chi$ (Until: "there will be an instant in the future when χ is

Table 2. Classification of basic relational formulae

Conjunctive formulae,	$\alpha = xR \cap Sy$	$\alpha_1 = xRy$	$\alpha_2 = xSy$	$(\wedge)$
α-formulae	$\alpha = x\overline{R \cup S}y$	$\alpha_1 = x\overline{R}y$	$\alpha_2 = x\overline{S}y$	$(\neg\vee)$
Disjunctive formulae,	$\beta = xR \cup Sy$	$\beta_1 = xRy$	$\beta_2 = xSy$	$(\vee)$
β-formulae	$\beta = x\overline{R \cap S}y$	$\beta_1 = x\overline{R}y$	$\beta_2 = x\overline{S}y$	$(\neg\wedge)$
	$\beta = x\overline{\overline{R}}y$	$\beta_1 = xRy$		$(\neg\neg)$
δ^α-formulae:	$\delta^\alpha = xR;Sy$	$\delta_0^{\alpha_1} = xRz$	$\delta_0^{\alpha_2} = zSy$	$(\exists\wedge)$
	$\delta^\alpha = x\overline{R \dagger S}y$	$\delta_0^{\alpha_1} = x\overline{R}z$	$\delta_0^{\alpha_2} = z\overline{S}y$	$(\neg\forall\vee)$
	where z is an existentially quantified variable ($\delta^\alpha \equiv (\exists z)(\delta_0^{\alpha_1}(z) \wedge \delta_0^{\alpha_2}(z))$)			
γ^β-formulae	$\gamma^\beta = x\overline{R;S}y$	$\gamma_0^{\beta_1} = x\overline{R}z$	$\gamma_0^{\beta_2} = z\overline{S}y$	$(\neg\exists\wedge)$
	$\gamma^\beta = xR \dagger Sy$	$\gamma_0^{\beta_1} = xRz$	$\gamma_0^{\beta_2} = zSy$	$(\forall\vee)$
	where z is a universally quantified variable ($\gamma^\beta \equiv (\forall z)(\gamma_0^{\beta_1}(z) \vee \gamma_0^{\beta_2}(z))$)			
κ-formulae	$\kappa = xR^\smile y$	$\kappa_1 = yRx$		
	$\kappa = x\overline{R^\smile}y$	$\kappa_1 = y\overline{R}x$		

true and from now *until* then ϕ will be true"); $\mathtt{X}\phi$ ("ϕ will be true in the *next* instant in time"). Analogous modalities are introduced for states in the past.

Relational translations of temporal formulae are expressed by considering an accessibility relation r that links time instants:

$$t(\mathtt{G}\phi) =_{\text{Def}} \overline{\mathsf{r}; \overline{t(\phi)}} \qquad t(\mathtt{F}\phi) =_{\text{Def}} \mathsf{r}; t(\phi)$$
$$t(\phi\,\mathtt{U}\,\chi) =_{\text{Def}} t(\phi)\,\mathtt{U}\,t(\chi) \qquad t(\mathtt{X}\phi) =_{\text{Def}} t((\phi\ \&\ \phi)\,\mathtt{U}\,\phi)$$

Notice that in translating the modal operator $\mathtt{U}$ we introduced a new relational construct (denoted, for simplicity, by the same symbol). Observe that this construct is *non-standard* (in the sense of Sec. 2, page 3) since it cannot be defined in terms of the basic relational constructs [27]. This is the intended interpretation of $\mathtt{U}$: $P\mathtt{U}Q$ designates the binary relation consisting of all pairs $\langle u, v\rangle$ such that there exists t such that $\langle u, t\rangle$ belongs to the accessibility relation $\mathsf{r}^{\Im}$, $\langle t, v\rangle$ belongs to $Q^{\Im}$, and for all w, if $\langle u, w\rangle \in \mathsf{r}^{\Im}$ and $\langle w, t\rangle \in \mathsf{r}^{\Im}$ then $\langle w, v\rangle \in P^{\Im}$. We will discuss more on this aspect in Sec. 4.3.

Other propositional modal logics. Other modal logics for which it is possible to give a relational formalization are, among others: the logics of knowledge and information described in [6], the logics with specification operators [25], the logics with Humberstone operators [18], the logics with sufficiency operators [7].

2.3 Classification of relational formulae

In order to illustrate the deductive system and the related proof techniques and heuristics in a more concise and clear way, we introduce a classification of relational formulae w.r.t. their leading construct. A *basic* (resp. *non-standard*) relational formula is a formula having a leading construct which is basic (resp. non-standard). Basic relational formulae can be further classified by taking inspiration from Smullyan's uniform notation for first-order logic [33]. In fact, they can be grouped into five categories according to the first-order characterization of the relational constructs (cf. Table. 2).

Table 3. Basic decomposition rules

$$\frac{x\alpha y}{x\alpha_1 y | x\alpha_2 y} \qquad \frac{x\beta y}{\begin{array}{c} x\beta_1 y \\ [x\beta_2 y] \end{array}} \qquad \frac{x\delta^{\alpha} y}{x\delta_0^{\alpha_1} z, x\delta^{\alpha} y | z\delta_0^{\alpha_2} y, x\delta^{\alpha} y} \qquad \frac{x\gamma^{\beta} y}{\begin{array}{c} x\gamma_0^{\beta_1} w \\ w\gamma_0^{\beta_2} y \end{array}} \qquad \frac{x\kappa y}{x\kappa_1 y}$$

If the leading construct is extensional (i.e., $\cup$, $\cap$, $\overline{\ }$) then, in analogy with Smullyan's notation, the relational formula is classified as an α-formula if it involves intersection as leading construct (or an analogous construct of "conjunctive nature", such as difference or complemented union). Conversely, a formula is classified as β-formula if its leading construct is union (or another one of "disjunctive nature"). Formulae of the form $x\overline{\overline{R}}y$ are classified as β-formulae.

Observe that, the first-order formulation of a formula with leading operator ; (such as $xR; Sy$) is of the form $(\exists z)(xRz \wedge zSy)$, where a conjunction occurs under the scope of an existential quantifier. Dually, formulae with † as leading operator (such as $xR \dagger Sy$) present first-order equipollents of the form $(\forall z)(xRz \vee zSy)$, where a disjunction is universally quantified. From such perspective, ; and † could be seen as compound operators (existential quantification+conjunction, universal quantification+disjunction). Thus, in analogy with Smullyan's uniform notation, where existentially (resp. universally) quantified formulae are classified as δ-formulae (resp. γ-formulae), we classify relational formulae with leading operator ; (resp. †) as δ^{α}-formulae (resp. γ^{β}-formulae).[1] Complemented Peircean constructs are classified analogously. Formulae of kind κ deal with conversion $\smile$.

3 A Rasiowa-Sikorski proof system for relational logics

Proof development in usual Rasiowa-Sikorski systems proceeds by systematically decomposing the (disjunction of) formula(e) to be proved till a tautological condition is detected through a closure rule (examples of such systems are described in [31, 28]). Analogously to [28], the relational proof system we present relies upon a collection of decomposition rules for basic relational formulae and upon a closure rule which takes care of axiomatic sequences such as xRy, $x\overline{R}y$, and xUy (described in Sec. 3.2). The basic decomposition rules of Table 3 have been defined according to the classification of formulae given in Sec. 2.3.

The α- and β-rules are used to decompose conjunctive and disjunctive formulae, respectively. (The square brackets in the β-rule indicate that the second component, namely $x\beta_2 y$ may or may not be present.) The δ^{α}-rule decomposes δ^{α}-formulae, whereas γ^{β}-formulae are expanded by the γ^{β}-rule. The individual variable z introduced in the δ^{α}-rule is chosen among the individual variables occurring in the proof. On the other hand, w in the γ^{β}-rule is an individual

[1] In what follows we feel free to use notations as δ-formula in place of δ^{α}-formula (and similarly for δ-expression, δ-rule, and so on), to denote formulae, expressions, etc., having ; as leading construct.

variable new to (the current branch of) the derivation. Note that such decomposition rules reflect the duality of Rasiowa-Sikorski systems with respect to tableaux systems [12].

The decomposition rules in Table 3 constitute a common core of any relational Rasiowa-Sikorski system. As mentioned, there are logics whose relational translation may involve intensional operators not expressible by means of basic relational constructs. For instance, in temporal logics we introduced a new construct, namely $\mathtt{U}$ as counterpart of the Until modal operators (pag. 4). In cases such that, the decomposition rules from Table 3 do not suffice. In order to manipulate these new constructs some *ad hoc* decomposition rules have to be introduced. In particular, [27] proposes the following rule for $\mathtt{U}$:

$$\frac{xP\mathtt{U}Qy}{xrt, xP\mathtt{U}Qy \mid tPy, xP\mathtt{U}Qy \mid x\overline{\mathsf{r}}u, u\overline{\mathsf{r}}t, uPy, xP\mathtt{U}Qy}$$

where t is chosen among the individual variables occurring in the proof and u is an individual variable new to the current branch of the derivation.[2]

The introduction of this rule can be justified by considering its frame-based semantics. In fact, we have this (binary) first-order formulation:

$$(\forall x\, y)(x\, P\mathtt{U}Q\, y) \equiv (\forall x\, y)(\exists t)(x\mathsf{r}t \wedge tQy \wedge (\forall u)(x\mathsf{r}u \wedge u\mathsf{r}t) \supset uPy).$$

which translates in the above decomposition rule.

3.1 The proof construction

A Rasiowa-Sikorski derivation $\mathcal{D}$ for a disjunction of relational formulae S is represented as a binary ordered tree whose nodes are labeled by disjunctions of formulae.[3] We call *branch* of $\mathcal{D}$ any maximal path in $\mathcal{D}$. More formally:

Definition 1. *Let S be a disjunction of relational formulae of Rel$\mathcal{L}$. A Rasiowa-Sikorski derivation $\mathcal{D}$ for S is recursively defined as follows.*

The tree with only one node labeled with S, is a derivation for S. Let $\mathcal{D}$ be a derivation for S, θ a branch of $\mathcal{D}$, N the leaf-node of θ. Then, the tree obtained from $\mathcal{D}$ by applying a decomposition rule, as illustrated by items 1-6 below, is a derivation for S. Let φ be a formula in N.

1. *If φ is a β-formula $x\beta y$, add $(N \setminus \{x\beta y\}) \cup \{x\beta_1 y, x\beta_2 y\}$ as a successor of N;*
2. *If φ is a κ-formula, $x\kappa y$, add $(N \setminus \{x\kappa y\}) \cup \{y\kappa_1 x\}$ as successor of N;*
3. *If φ is an α-formula $x\alpha y$, add $(N \setminus \{x\alpha y\}) \cup \{x\alpha_1 y\}$ and $(N \setminus \{x\alpha y\}) \cup \{x\alpha_2 y\}$ as left and right successors of N, respectively;*
4. *If φ is a γ^β-formula $x\gamma^\beta y$, add $(N \setminus \{x\gamma^\beta y\}) \cup \{x\gamma_0^{\beta_1} w, w\gamma_0^{\beta_2} y\}$ as successor of N, where w is a variable new to θ;*
5. *If φ is a δ^α-formula $x\delta^\alpha y$, add $N \cup \{x\delta_0^{\alpha_1} z\}$ and $N \cup \{z\delta_0^{\alpha_2} y\}$ as left and right successors of N, where z is chosen among the variables occurring in $\mathcal{D}$;*
6. *If φ is a non-standard formula, extend θ according to the corresponding specific decomposition rule.*

[2] Notice that the above decomposition rules originate three branches. Nonetheless, it is easy to surrogate such triple-branching by a binary tree.

[3] By abuse of notation, we often identify a node N with the disjuncts of its label.

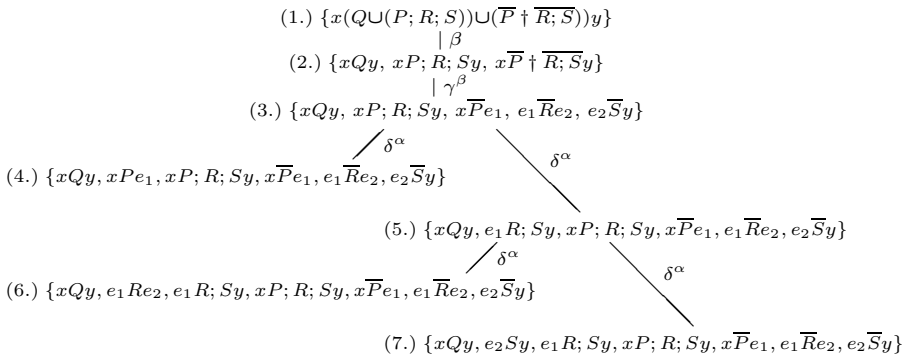


Fig. 1. Rasiowa-Sikorski proof for Example 1

3.2 Closure phase

The closure of a branch is determined by applying a closure rule. For instance, we mentioned that a branch is declared closed whenever its leaf-node contains a pair of the form xRy and $x\overline{R}y$. The following definition summarizes, in more generality, such closure condition by handling the properties of equality.

Definition 2. *A node N is* closed *if one of the following conditions holds:*

1. *N contains the formulae $x_1Dx_2,\ldots,x_{h-1}Dx_h$, $y_1Dy_2,\ldots,y_{\ell-1}Dy_\ell$, $x_1\overline{R}z$, x_hRy_ℓ, with $z=y_\ell$ (for some $h,\ell\geq 1$). In case R is a right-ideal atomic relation, then it is not required that $z=y_\ell$ holds.*
2. *N contains the formulae $x_1Ix_h, x_1Dx_2,\ldots,x_{h-1}Dx_h$ (for some $h\geq 1$).*

Moreover, in order to take care of symmetry of equality, the distinction between x_iDx_j and x_jDx_i in these conditions is considered immaterial. A node is atomically closed *if R is an atomic relation.*

Notice that the basic closure conditions of usual relational Rasiowa-Sikorski systems, cf. [27], are instances of the above ones when putting $h=\ell=1$.

A derivation $\mathcal{D}$ for a disjunction of formulae S of $Rel\mathcal{L}$ is *satisfied* by a model $\boldsymbol{\mathcal{M}}=\langle\boldsymbol{\mathcal{D}},\boldsymbol{\mathcal{I}}\rangle$ of $Rel\mathcal{L}$ if, for every variable assignment A and branch θ of $\mathcal{D}$, $\boldsymbol{\mathcal{M}}$ and A satisfy θ (and we write $(\boldsymbol{\mathcal{M}},A)\models\theta$). A model $\boldsymbol{\mathcal{M}}$ and an assignment A satisfy a branch θ if they satisfy each node of θ. A node N is satisfied by $\boldsymbol{\mathcal{M}}$ and A if at least one formula φ in N is satisfied by $\boldsymbol{\mathcal{M}}$ and A. A branch θ in a derivation is said to be (*atomically*) *closed* if its leaf-node is (atomically) closed. A derivation $\mathcal{D}$ is (atomically) closed if all its branches are (atomically) closed. A closed derivation for S is a *proof* of S.

Example 1. Consider the valid formula $x(Q\cup(P;R;S))\cup(\overline{P}\dagger\overline{R;S})y$. Fig. 1 illustrates a Rasiowa-Sikorski proof for it. The labels of the arcs indicate the decomposition rules applied. In particular, node (2.) has been obtained from node (1.) by applying the β-rule twice. Similarly, to obtain (3.), the γ^β-rule has been applied twice. Nodes (4.) and (6.) are closed because of the pairs of literals

xPe_1 and $x\overline{P}e_1$, and e_1Re_2 and $e_1\overline{R}e_2$, respectively. Finally, the pair e_2Sy and $e_2\overline{S}y$ closes node (7.).

4 An efficient representation of formulae and proofs

Trees are the most natural structure to represent proofs in analytic systems such as tableaux and Rasiowa-Sikorski (cf. Fig. 1). Indeed, they reflect the idea of recursively decompose the formula to be proved till elementary contradictions (in case of tableaux) or tautologies (in case of Rasiowa-Sikorski) are found. Nevertheless the frequent presence of redundant parts in the trees, makes the proof search procedure highly inefficient in practice [4]. Therefore, the use of more suitable data structures is required to construct efficient concrete provers.

In what follows we show how to represent relational formulae and Rasiowa-Sikorski proofs by means of labeled acyclic graphs and describe a series of refinements on such basic framework. Such a representation is akin to Binary Decision Diagrams and has several desirable properties when Boolean formulae are processed: it maximizes structure sharing since common sub-formulae are represented (and processed) once; it has unique (up to the ordering imposed on labels of nodes) reduced canonical forms; and satisfiability and tautology checking are easily performed on reduced canonical forms. As we will see, in order to treat Peircean constructs (which involve implicit use of quantification), we need to circumvent the limited expressive power of basic BDD-style representations.

4.1 From trees to graphs: structure sharing

Let us start by considering a simplified scenario where only Boolean constructs (union, intersection, complement, etc) are involved. Thus, only α- and β-rules can be applied to construct a proof for a given relational formula xRy: proving that $R{=}U$ amounts to fully decompose xRy by α- and β-steps to obtain a fully expanded proof $\mathcal{D}$ and then to close each of its leaves by applying the closure rule of Sec. 3.2. A similar situation arises while using BDDs in checking for satisfiability of Boolean functions [2]. For this reason, the use of graph-based representations (such as BDDs or Shannon graphs) have been proposed in the context of tableaux systems (see, among others, [30, 32, 15]). We adopt a similar representation for relational expressions.

Given a relational formula involving only Boolean constructs, by α- and β-decompositions, we obtain its representation as a (labeled) rooted directed acyclic graph. Let us call such graph RDG (standing for relational decision graph). Each non-leaf node n of an RDG has two outgoing edges labeled $-$ and $+$, respectively. Let us denote the corresponding sub-graphs by n^- and n^+, resp. Moreover, let $r(n)$ denote the formula labeling n. An example of RDG of an atomic formula (xSy, in this case) is depicted in Fig. 2 (where $\mathbf{0}$ and $\mathbf{1}$ are new symbols labeling the only two leaf-nodes). In general, an RDG of a given relational formula is built up in syntax directed bottom-up process, from the RDGs of its sub-formulae, by "replacement of leaves". For instance, given the

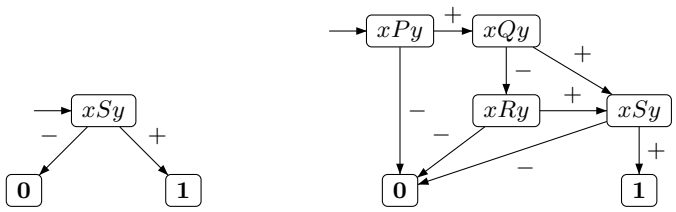


Fig. 2. The RDGs for xSy and $xP \cup ((Q \cap R) \cup S)y$

RDGs for xRy and xQy, an RDG for $x(Q \cap R)y$ can be obtained by replacing the **0**-leaf of the former with the root of the latter and merging the two **1**-leaves. This corresponds to apply α-decomposition. Similarly, an RDG for $x(Q \cap R) \cup Sy$ is obtained replacing the **1**-leaf of the RDG for $x(Q \cap R)y$, with the root node of the RDG for xSy (and merging the remaining identical leaves). This corresponds to apply β-decomposition.

The major advantage of adopting such a representation is the maximal structure sharing it intrinsically provides. Indeed, in building up the RDG of a formula, whenever a sub-formula occurs multiply, its sub-RDG is built only once.

An RDG obtained in this way fulfills these conditions: *a)* there are two leaves only (i.e., **0** and **1**); *b)* there are not two distinct non-leaf nodes n_1, n_2 such that the sub-graphs rooted at n_1 and n_2 are isomorphic. On the other hand, at this stage, we do not preclude nodes n such that $n^- = n^+$. (Hence, such RDG are not guaranteed to be *reduced* in the sense of [2].) Moreover, we do not impose any order on the (formulae labeling the) nodes of the RDG. We will deal with these aspects in the sequel, while introducing a normalization procedure for RDGs.

We are left to deal with complementation of relations: given an RDG for xRy, an RDG for $x\overline{R}y$ can be obtained by exchanging the two leaf-nodes.

Given an RDG, a **1**-path is a path from the root node to the **1**-leaf. Any **1**-path p can be denoted as a sequence $n_1, s_1, n_2, s_2, \ldots, n_k, s_k, \mathbf{1}$ (for $k \geq 0$), where each symbol s_i is the label of the i^{th} edge of p. The set $r(p)$ of relational formulae occurring on a **1**-path $p = n_1, s_1, n_2, s_2, \ldots, n_k, s_k, \mathbf{1}$ is defined as:

$$r(p) = \{r(n_i) : s_i = +\} \cup \{\overline{r(n_j)} : s_j = -\}.$$

A **1**-path p is closed if $r(p)$ is closed in the sense of Def. 2. Checking if a relational formula xRy, involving only Boolean constructs, is valid amounts to verifying that every **1**-path p of an RDG for xRy is closed.

4.2 Dealing with Peircean constructs

The procedure described so far does not handle Peircean constructs. As mentioned, γ- and δ-decomposition rules correspond to universal and existential quantifications. Hence dealing with them imposes going beyond the expressive power of basic BDD-like representations.

Among the various approaches proposed in literature, we refine and adapt to our context an interesting extension of un-ordered BDD proposed, for instance, in [30, 32]. In particular, in [30] the authors propose a variant of a free-variable tableaux system where universal quantification is handled by means of

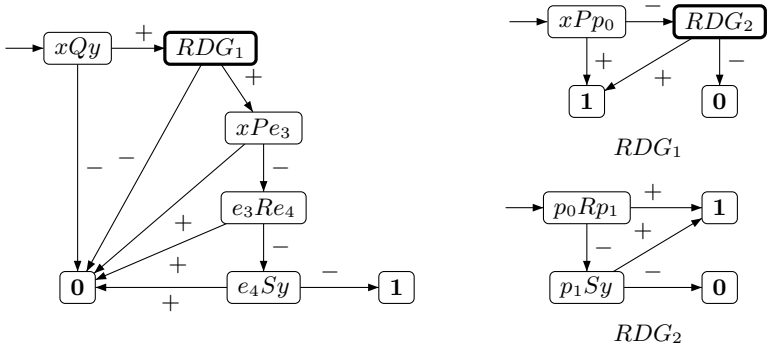


Fig. 3. RDGs for Example 2

a graph-nesting mechanism, while existential quantification is handled through a preliminary transformation into Skolemized negative normal form.

Since, w.r.t. tableaux systems, we are facing a dual problem, (i.e., to prove validity in place of unsatisfiability), we use such nesting mechanism to process δ-formulae instead of γ-formulae. Notice also that we do not need to manipulate first-order formulae in their full generality, this because relational equalities characterize a proper sub-language of first-order logic obeying significant restrictions [34]. This simplifies the treatment and originates a more profitable and efficient usage of the graph-nesting technique. The basic idea consists in allowing an RDG to label a node in another RDG. Whenever a δ-decomposition is applied during the construction of an RDG, another RDG is constructed for the δ-formula at hand. Such an ancillary RDG will be "nested" as a single node (let us call δ-node a node of this kind) in the main RDG.

Example 2. Consider the formula of Example 1. A (nested) RDG for it is depicted in Fig. 3 (where for the sake of readability, we duplicated the **0**-leaf and the **1**-leaf. Thicker lines indicate δ-nodes). Notice how the nesting mechanism is exploited to translate the sub-expression $P;R;S$: the formula $x(P;(R;S))y$ corresponds to RDG_1, while RDG_2 is the auxiliary graph for $p_0(R;S)y$. In the main graph, two γ-decompositions (of the formula $x(\overline{P} \dagger \overline{R;S})y$) introduce the fresh individual variables e_3 and e_4. (Compare this RDG with the tree in Fig. 1.)

In order to reflect the extension mechanism illustrated in Def. 1, any application of a δ-rule must leave open the possibility of further δ-decompositions of the same δ-formula. In each of these decompositions any of the individual variables occurring in the branch being extended may be used. Intuitively speaking, in trying to close a **1**-path p, we proceed by extending p with an instantiation of one of its nested RDGs. Such an instantiation involves one of the individual variables occurring in p. To prepare for this instantiation+extension step, when a δ-node is created, a template for the corresponding (nested) RDG is built and a "placeholder" is used as parameter (cf., p_0 and p_1 in Fig. 3). Such a placeholder will be replaced during the extension step, by the individual variable selected from those occurring in the path.

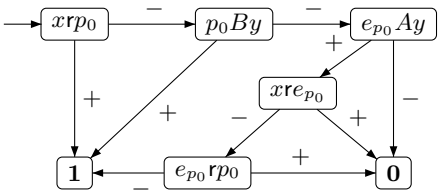


Fig. 4. RDG for the construct U

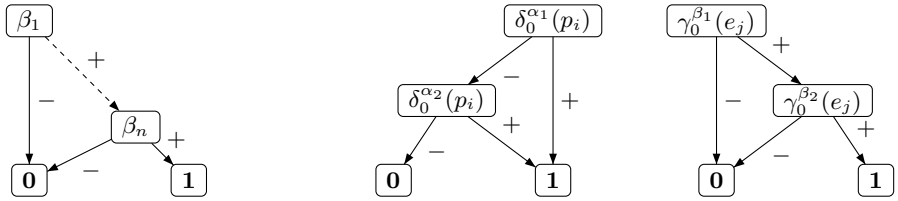


Fig. 5. Initial graph proof for $S = \beta_1 \cup \ldots \cup \beta_n$ and δ^α- and γ^β-decompositions

As regards γ-formulae, we proceed similarly to β-decomposition, with the only difference that a freshly generated individual-variable symbol is introduced (cf, Def. 1). This clearly corresponds to Skolemization and if a γ-decomposition occurs within nested RDGs (i.e. within the scope of one or more δ-expressions), the Skolem term will be parametric in all the corresponding placeholders.

Recall that, the term to be introduced by a γ-decomposition has to be chosen among a finite set of individual variables. (In the case of tableaux systems such term may be selected among any of the compound terms of the language of the current branch. Such a language could be infinite.)

The remaining primitive Peircean construct, conversion, is eliminated during the generation of the RDG by rewriting any formula of the form $xR^\smile y$ as yRx.

4.3 Non-standard relational constructs

Thanks to the use of nesting, it is possible to generalize RDGs in order to treat any kind of construct, provided that its semantics can be given in terms of a formula of binary predicate logic. As an example, let us consider the Until operator of temporal logic and the corresponding relational construct U. It can be characterized by the first-order formula $x\,A\mathtt{U}B\,y \equiv \exists t(x\mathsf{r}t \wedge tBy \wedge (\forall u)(uAy \vee x\overline{\mathsf{r}}u \vee u\overline{\mathsf{r}}t))$, where r denotes a constant relation modeling world accessibility. The structure of the formula suggests the structure of an RDG: we treat disjunctions (resp. conjunctions) similarly to α-decompositions (resp. β-decompositions). The RDG sought for is shown in Fig. 4. Whenever, in building up an RDG for a given formula, the construct U has to be decomposed, a δ-node having the graph in Fig. 4 as nested RDG is created.

Such a mechanism corresponds to use a sort of "macro expansion" of non-standard constructs. In fact, the part of the graph related to the existentially quantified variable t presents a structure similar to the graph generated by a δ-decomposition (the only difference is in the occurrences of individual variables).

Conversely, the part of the graph related to the universally quantified variable u presents a structure similar to the graph generated by a γ-decomposition. These analogies allows us to reuse the very same machinery developed for basic Peircean constructs to handle nested RDG originated by non-standard constructs.

5 Soundness and Completeness

In this section we briefly outline the ideas behind the proofs of soundness and completeness, limiting ourselves to consider the graph-based version of the system with basic rules only. A detailed treatment, including the missing proofs, can be found in [9]. In Sec. 4 a syntax directed bottom-up process is outlined to construct graph-based representations of Rasiowa-Sikorski proofs. This choice is basically due to efficiency reasons, since it allows maximization of structure sharing. Here, in order to make proofs of the soundness and completeness statements simpler, we introduce a top-down recursive construction of the very same graph-based proof.

Definition 3. *Let S be a disjunction of formulae of Rel$\mathcal{L}$. A derivation $\mathcal{G}$ for S is recursively defined as follows. The graph in Fig. 5 (with $\beta_1, \ldots, \beta_n$ the disjuncts in S) is a derivation for S. Moreover, let $\mathcal{G}$ be a derivation for S, then the graph $\mathcal{G}'$, obtained from $\mathcal{G}$ by applying one of the decomposition steps described in Sec. 4.1 and 4.2, as shown below, is a derivation for S. Let n be any node in $\mathcal{G}$ labeled by φ.*

1. *If φ is a β-formula, replace n with its β-decomposition;*
2. *If φ is an α-formula, replace n with its α-decomposition;*
3. *If φ is a κ-formula, substitute φ with its component $y\kappa_1 x$;*
4. *If φ is a γ^β-formula, replace n with its γ^β-decomposition, with e_j a variable new to every $\mathbf{1}$-path containing n;*
5. *If φ is a δ^α-formula, add its δ^α-decomposition as successor of n, with the placeholder p_i replaced by any e_i, chosen among the variables in $\mathcal{G}$.*

Soundness. The relational Rasiowa-Sikorski system with graph-based proof representation is sound if every disjunction of relational formulae with a closed derivation (graph) is valid. The statement can be proved by showing that preservation of validity is an invariant property throughout the recursive graph construction illustrated in Def. 3, and that the closure rule from Def. 2 "closes" only $\mathbf{1}$-paths p with tautological $r(p)$.

Completeness. The system is complete if for every valid disjunction S of relational formulae it is able to produce a finite closed RDG for S. As usual, the statement is proved by exhibiting a fair proof-search procedure (such as the one introduced in Sec. 6) that, in a deterministic way, builds a *saturated* RDG for a disjunction of relational formulae S. A derivation $\mathcal{G}$ for S is *propositionally saturated* if it is constructed out of the initial graph-proof for S by applying only decomposition steps $1 - 4$ from Def. 3, till all its nodes are labeled by either atomic formulae or by δ^α-formulae. A δ^α-formula occurrence $x\delta^\alpha y$ in $\mathcal{G}$

Table 4. Rewriting system for the normalization process

1)	$RDG(r(n), n^-, n^+) \rightsquigarrow n^-$	if $n^- = n^+$
2)	$RDG(r(n_1), RDG(r(n_2), n_2^-, n_2^+), n_1^+) \rightsquigarrow RDG(r(n_1), n_2^-, n_1^+)$	if $r(n_1) = r(n_2)$
3)	$RDG(r(n_1), n_1^-, RDG(r(n_2), n_2^-, n_2^+)) \rightsquigarrow RDG(r(n_1), n_1^-, n_2^+)$	if $r(n_1) = r(n_2)$
4)	$RDG(r(n_1), RDG(r(n_2), n_2^-, n_2^+), n_1^+) \rightsquigarrow$ $RDG(r(n_2), RDG(r(n_1), n_2^-, n_1^+), RDG(r(n_1), n_2^+, n_1^+))$	if $r(n_1) \succ r(n_2)$
5)	$RDG(r(n_1), n_1^-, RDG(r(n_2), n_2^-, n_2^+)) \rightsquigarrow$ $RDG(r(n_2), RDG(r(n_1), n_1^-, n_2^-), RDG(r(n_1), n_1^-, n_2^+))$	if $r(n_1) \succ r(n_2)$
6)	$RDG(r(n), n^-, n^+) \rightsquigarrow n^-$	if $r(n)$=xUy
7)	$RDG(r(n), n^-, n^+) \rightsquigarrow n^+$	if $r(n)$=xZy
8)	$RDG(r(n), n^-, n^+) \rightsquigarrow n^-$	if $r(n)$=xIx
9)	$RDG(r(n), n^-, n^+) \rightsquigarrow n^+$	if $r(n)$=xDx

is *fully expanded* if, for every variable e_i in the **1**-paths of $\mathcal{G}$ containing $x\delta^\alpha y$ with a positive sign, the corresponding δ^α-decomposition graph (see Fig. 5) instantiated to e_i, is propositionally saturated and attached as a right successor to $x\delta^\alpha y$. A derivation $\mathcal{G}$ for S is saturated if it is propositionally saturated and each δ^α-formula occurring in it is fully expanded. Every **1**-path on a saturated derivation is said to be saturated. It can be proved that if p is a saturated open **1**-path, then there exists a model not satisfying $r(p)$ [9].

6 Towards an efficient implementation

In this section we describe a number of techniques we adopted in implementing an automated Rasiowa-Sikorski deduction system. The system we are going to delineate has been implemented in SICStus Prolog.

Ordering. Given a relational formula, the procedure outlined in Sec. 4 produces an RDG without imposing any order on the (atomic formulae labeling the) nodes. Furthermore, it may be the case that the same formula labels two distinct nodes in a path. We introduce now a normalizing procedure that, by proceeding top-town, imposes a given order $\succ$ on the nodes of the RDG. Such a procedure is described as a term rewriting system (in the spirit of [16, 37]). The rewriting process will continues until no further rewriting is applicable. An useful piece of notation: let the term $RDG(r(n), n^-, n^+)$ denote the RDG rooted in the node n. The rewriting system is constituted of the rules $1) - 5)$ in Table 4, where $\succ$ is any total ordering on the collection of relational atomic formulae. Notice that all these rules preserve the closure property of the RDGs. Moreover, if all of the **1**-paths of the initial RDG can be closed because of pairs of complementary formulae, then the normalization process yields an RDG made of a single **0**-leaf. This immediately certifies the initial formula to be tautological. The described rewriting system can be enriched by adding additional rules (i.e. ,6) $-$ 9) in Table 4) to handle basic constants U, Z, I, D. In this way, the rewriting process simplifies the RDG by removing those **1**-paths which are tautological because of atomic formulae of the forms xUy, xIx, etc.

To impose node ordering and maximal structure-sharing (even between isomorphic sub-graphs of different nested graphs), the normalization process is recursively applied to each nested RDG.

Remark 1. Two refinements often adopted in tableaux systems are the use of lemmas and the imposition of some sort of regularity constraint on the application strategy for decomposition rules [21]. These techniques have as counterpart, in our system, the adoption of a graph-based representation where ordering and maximal structure sharing are imposed.

Equality and symmetric or reflexive relations. The adoption of a two-phases approach (i.e., a procedure to build-up the RDG coupled with a normalizing phase), instead of developing a procedure to obtain an ordered RDG directly from the formula, permits a simpler treatment of those relations subject to specific properties or axioms. Examples are reflexivity and symmetry.

Let us start by considering any relational expression R which is known to denote a symmetric binary relation, i.e., such that $\forall x \forall y(xRy \rightarrow yRx)$. In this case the label xRy be rewritten as yRx preserving the validity of the whole relational expression. This fact justifies the introduction of the following rewriting rule in the normalization procedure:

$$RDG(xRy, n^-, n^+) \rightsquigarrow RDG(yRx, n^-, n^+) \text{ if } x \succ y$$

where we consider the total order $\succ$ as extended to the collection of all individual variables. On the other hand, whenever a relation R is such that $\forall x(xRx)$ holds, we can apply the following rewriting rule:

$$RDG(xRy, n^-, n^+) \rightsquigarrow n^- \text{ if } x = y$$

Clearly, these rules applies in the case of the constants I and D, as well. Nevertheless, in the case of D more fruitful rewriting rules can be introduced:

$$RDG(xDy, n^-, n^+) \rightsquigarrow RDG(yDx, n^-, n^+) \text{ if } x \succ y$$
$$RDG(xDy, n^-, n^+) \rightsquigarrow RDG(xDy, n^-, n') \text{ if } y \succ x$$

where n' is obtained from n^+ by substituting each occurrence of y with x. For instance, if $x_1 \succ \cdots \succ x_h \succ y_1 \succ \cdots \succ y_\ell$, such rules (in combination with the others) rewrite the set $\{x_1 D x_2, \ldots, x_{h-1} D x_h, y_1 D y_2, \ldots, y_{\ell-1} D y_\ell, x_1 \overline{R} y_1, x_h R y_\ell\}$ into $\{x_h \overline{R} y_\ell, x_h R y_\ell\}$.

Remark 2. It is important to notice that, in virtue of this refinement of the rewriting system, we can sensibly simplify the closure rule. Actually, we can modify the closing conditions in Def. 2 by imposing $h = \ell = 1$.

Proof-search procedure. Once an RDG is normalized, two situations may arise. If the RDG is made of a single node, then establishing validity of the initial formula xRy is immediate. Conversely, it might be the case that such RDG contains a number of non-trivial **1**-paths. In order to declare xRy valid, each of them has to be closed. Checking all the **1**-paths involves visiting the whole (finite) RDG. For any **1**-path $p : n_1, s_1, n_2, s_2, \ldots, n_k, s_k, \mathbf{1}$ let $l(p)$ be the set of formulae of $r(p)$ restricted to the nodes of p that are not δ-nodes. (Notice that, by the bottom-up procedure used to obtain the RDGs, $l(p)$ is a set of literals.) Moreover, let $\delta(p)$ be the set of δ-nodes n_i, in p, such that $s_i = +$. For any **1**-path two situation are possible: *i)* $l(p)$ satisfies a closure condition. In this case p is declared closed. *ii)* $l(p)$ does not satisfy any closure condition. In this case closure may still be achievable by performing some extensions using one or more nested RDG. Hence, the search procedure proceeds by selecting

a δ-node m in $\delta(p)$ and an individual variable e among those occurring in the path. Let p_m be the placeholder of m. The **1**-path p is extended with a copy of the ancillary RDG of m, with e replaced for the placeholder p_m. The extension happens substituting the ending **1**-leaf of p with the root of the selected RDG. Such a step introduces a number of new **1**-paths (all of them having p as prefix). At this point the process (recursively) proceeds trying to close these new **1**-paths. The process continues, possibly by applying further extension steps, until, either all **1**-paths are declared closed; or no more extensions are possible; or some termination condition is verified.

Clearly, suitable strategies and heuristics should be exploited to guide the proof procedure. In particular, two questions must be answered each time an extension step has to be performed: *a)* which δ-node is selected? And *b)* which individual variable is selected? Since more than one δ-node may be necessary in order to close a **1**-path, any *fair* selection rule can be adopted. As regards the selection of individual variables for repeated use of the same δ-node, we rely once more on the order $\succ$ and each time an extension step is performed the next unused variable according to $\succ$ is used.

It must be noted that Skolem terms may occur in nested RDG. Such terms are parameterized by placeholders names. Hence each time an extension step is performed, new individual variables (may) be introduced in the (extended) **1**-path. Consequently, the search procedure is not guaranteed to terminate. This phenomenon cannot be avoided, since establishing the validity of a relational formula is, in general, undecidable. To avoid infinite computation we adopt bounded depth-first iterative deepening [19].

Given a specific RDG, an effective approach in efficiently implementing the proof-search procedure outlined so far consists in generating the Prolog code which encodes the proof-search itself. In particular, each node n of the RDG originates a Prolog clause which, during execution, calls the clauses corresponding to n^+ and n^-. Clearly, the clause related to the **1**-leaf of the RDG also encodes the extension mechanism. Hence the proof of a modal theorem follows these phases: *a)* the given formula is translated in relational form (Sec. 2.2); *b)* an RDG for such form is generated (Sec. 4); *c)* the RDG is normalized (Sec. 6); *d)* the normalized RDG is compiled into a Prolog program; *e)* the search for the proof is done by executing such program.

Some concluding remarks about the graph representation: we conclude this section by observing that our approach presents some differences w.r.t. the one in [30] in at least two further aspects that have significant impact in the realization of the proof-search procedure. First, we do not make use of free-variables, even if, in principle, this is not precluded. Hence our system turns out to be closer to ground tableaux systems that to free-variable ones. As a consequence, we do not use any unification procedure to implement closure rules. Second, we profitably impose ordering of nodes. This sensibly reduces the size of the generated RDGs to be processed by a subsequent proof-search procedure.

7 Conclusions and future works

The research we reported upon in this paper shows that the duality results holding between (ground) tableaux systems and Rasiowa-Sikorski systems, naturally provide strong support to cross-fertilization between the two streams of research. Actually, we explored just a small portion of the immense work and well established research done in the context of tableaux systems, considering a rather smooth application of such technology to the development of Rasiowa-Sikorski systems. Besides tableaux technology, techniques and heuristics can be borrowed from related fields such as automation of propositional logic, equational reasoning, term rewriting systems, semantic unification, theory reasoning, to mention some. In particular, in this initial work, we adopted a representation akin to Decision Diagrams, in origin designed to implement propositional logics. Moreover, we reported on the applicability of techniques developed for equality handling and based on term rewriting rules.

Several steps in this direction of research have to be yet completed. For instance, the theoretical basis of our system design are inspired to ground versions of tableaux, but an interesting theme for further study would consist in investigating the use of *free variables* in relational Rasiowa-Sikorski systems. In general, free-variable tableaux systems can be implemented so to ensure greater efficiency, w.r.t. ground tableaux (see [11] for an efficient treatment of the unification procedure). It is reasonable that the use of free variables in Rasiowa-Sikorski systems could be advantageous as well.

The system described in this paper can be seen as an alternative to other, more traditional methodologies such as the ones reviewed in [14]. Comparisons in term of proof complexity and efficiency are needed in order to identify those cases where our deduction method behaves better (or worse) than others.

Most of the modal logics of interest are decidable. Decidability results constitute a solid theoretical background for the development of attractive specific inference tools for such logics. A challenging theme of research consists in detecting under which conditions the decidability results for a non-classical logic can be reflected in some decidability property of its relational counterpart. A possibility could consist in identifying particular classes of deduction rules, together with a specific strategy in rule application, that ensures decidability in the relational framework.

Finally, an extensive comparison has to be done with alternative approaches and tools designed for relational and modal reasoning. Among the systems supporting various forms of relational manipulation and reasoning we would like to mention, RALF, RELVIEW, RALL, δRA, and ARA. References for most of such systems can be found in [36]. (Among the tools based on Rasiowa-Sikorski systems, we mention RelDT [22].) As regards systems expressively dedicated to modal reasoning, references for most of them can be found in [35].

Acknowledgements We thank Martin Giese, Joanna Golińska, Rajeev Goré, and Ewa Orłowska for useful discussions on the topics of this paper.

References

[1] C. Brink, W. Kahl, and G. Schmidt. *Relational Methods in Computer Science*. Advances in Computing Science. Springer, 1997.

[2] R. E. Bryant. Graph-based algorithms for boolean function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] D. Cantone, A. Formisano, E. Omodeo, and C. Zarba. Compiling dyadic first-order specifications into map algebra. *Theoretical Computer Science*, 293(2):447–475, 2003.

[4] M. D'Agostino. Are tableaux an improvement on truth tables? *Journal of Logic, Language and Information*, 1:235–252, 1992.

[5] M. D'Agostino, D. M. Gabbay, R. Hänle, and J. Posegga, editors. *Handbook of tableau methods*. Kluwer Academic Publishers, Dordrecht, 1999.

[6] S. Demri and E. Orłowska. *Incomplete Information: Structure, Inference, Complexity*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2002.

[7] I. Düntsch and E. Orłowska. Beyond modalities: Sufficiency and mixed algebras. In E. Orłowska and A. Szalas, editors, *Relational Methods for Computer Science Applications*, pages 277–299, Heidelberg, 2001. Physica-Verlag.

[8] I. Düntsch, E. Orłowska, A. M. Radzikowska, and D. Vakarelov. Relational representation theorems for some lattice-based structures. *Journal on Relational Methods in Computer Science*, 1:132–160, 2004.

[9] A. Formisano and M. Nicolosi Asmundo. An efficient relational deductive system for propositional non-classical logics. Technical Report 8/06, Dipartimento di Informatica, Università di L'Aquila, 2006.

[10] A. Formisano, E. G. Omodeo, and M. Temperini. Instructing equational set-reasoning with otter. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated reasoning: First International Joint Conference, IJCAR 2001. Proceedings*, volume 2083 of *LNCS*, pages 152–167. Springer, 2001.

[11] M. Giese. *Proof Search without Backtracking for Free Variable Tableaux*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, July 2002.

[12] J. Golińska-Pilarek and E. Orłowska. Tableaux and dual tableaux: Transformation of proofs. unpublished, 2006.

[13] R. Goré. Cut-free display calculi for relation algebras. In *CSL*, volume 1258 of *LNCS*, pages 198–210, 1996. Selected Papers of the Annual Conference of the Association for Computer Science Logic.

[14] R. Goré. Tableau methods for modal and temporal logics, 1999. In [5].

[15] J. Goubault. Proving with BDDs and control of information. In A. Bundy, editor, *Proceedings of the 12$^{\text{th}}$ International Conference on Automated Deduction*, volume 814 of *LNCS*, pages 499–513, Nancy, France, 1994. Springer.

[16] J. F. Groote and J. van de Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, editors, *Logic for programming and automated reasoning: 7$^{\text{th}}$ International Conference, LPAR 2000*, volume 1955 of *LNCS*, pages 161–178. Springer, 2000.

[17] M. C. B. Hennessy. A proof system for the first-order relational calculus. *Journal of Computer and System Sciences*, 20(1):96–110, 1980.

[18] L. Humberstone. Inaccessible worlds. *Notre Dame Journal of Formal Logic*, 24:346–352, 1983.

[19] R. E. Korf. Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[20] M. Kwatinetz. *Problems of Expressibility in Finite Languages*. Doctoral diss., Univ. of California, Berkeley, 1981.
[21] R. Letz. First-order tableau methods, 1999. In [5].
[22] W. MacCaull and E. Orlowska. A logic of typed relations and its applications to relational databases. Technical report, Department of Mathematics, Statistics and Computer Science, St. Francis Xavier University, Antigonish, Canada, 2003.
[23] R. D. Maddux. A sequent calculus for relation algebras. *Annals of Pure and Applied Logic*, 25:73–101, 1983.
[24] H. J. Ohlbach, A. Nonnengart, M. de Rijke, and D. M. Gabbay. Encoding two-valued non-classical logics in classical logic. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 21, pages 1403–1486. Elsevier Science B.V., 2001.
[25] E. Orłowska. Proof system for weakest prespecification. *Information Processing Letters*, 27(6):309–313, 1988.
[26] E. Orłowska. Relational semantics for non-classical logics: Formulas are relations. In J. Wolenski, editor, *Philosophical Logic in Poland.*, pages 167–186. Kluwer, 1994.
[27] E. Orłowska. Temporal logics — in a relational framework. In L. Bolc and A. Szalas, editors, *Time and Logic — A Computational Approach.*, pages 249–277. Univ. College London Press, 1995.
[28] E. Orłowska. Relational proof systems for modal logics. In H. Wansing, editor, *Proof Theory of Modal Logic*, volume 2, pages 55–77. Kluwer, 1996.
[29] E. Orłowska. Relational formalisation of nonclassical logics. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science*, Advances in Computing Science, chapter 6, pages 90–105. Springer, Wien, New York, 1997.
[30] J. Posegga and P. H. Schmitt. Implementing semantic tableaux, 1999. In [5].
[31] H. Rasiowa and R. Sikorski. *The Mathematics of Metamathematics*. Polish Scientific Publishers, 1963.
[32] K. Schneider, R. Kumar, and T. Kropf. Accelerating tableaux proofs using compact representations. *Formal Methods in System Design*, 5(1-2):145–176, 1994.
[33] R. M. Smullyan. *First-Order Logic*. Dover Publications, New York, second corrected edition, 1995. First published 1968 by Springer.
[34] A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*, volume 41 of *Colloquium Publications*. American Mathematical Society, 1987.
[35] Web site of AiML. `www.cs.man.ac.uk/~schmidt/tools`.
[36] Web site of RelMiCS. `www2-data.informatik.unibw-muenchen.de/relmics/html`.
[37] H. Zantema and J. van de Pol. A rewriting approach to binary decision diagrams. *Journal of Logic and Algebraic Programing*, 49(1-2):61–86, 2001.

OntoDLV: An Object-Oriented Disjunctive Logic Programming System

Nicola Leone, Francesco Ricca

Department of Mathematics, University of Calabria, Rende (CS) 87036, Italy

Abstract. The paper presents OntoDLV a system based on an extension of Disjunctive Logic Programming (DLP) which combines the expressive power of DLP with the modeling capabilities of the object-oriented languages. In particular, the OntoDLV language supports the most important object-oriented constructs including classes, objects, (multiple) inheritance, and types.
OntoDLV is built on top of DLV (a state-of-the art DLP system), and provides a graphical user interface that allows to specify, update, browse, query, and reason on knowledge bases. Two strong points of the system are the powerful type-checking mechanism, and the advanced interface for visual querying.

1 Introduction

Disjunctive Logic Programming under the Stable Model Semantics [1] (DLP) is an expressive logic programming language, which has been proposed in the area of nonmonotonic reasoning and logic programming.

The high expressive power of DLP [2] can be profitably exploited when one has to deal with problems of high complexity, and this makes DLP an advanced formalism for Knowledge Representation and commonsense Reasoning (KR&R)[1].

Moreover, the availability of a couple of efficient DLP systems, like DLV [3], GnT [4] and, more recently, the disjunctive version of Cmodels [5] make DLP a powerful tool for developing advanced knowledge-based applications [6, 7].

Despite its high expressiveness, there are several problems that DLP cannot encode in a natural way. For instance, it misses constructs for representing complex real-world entities[8], like classes, objects, compound objects, and taxonomies. Moreover, DLP systems are missing tools for supporting the programmers, like type-checkers and easy-to-use graphical environments, to manage the large and complex domains to be dealt with in real-world applications.

The recent applications of DLP in the emerging areas of Knowledge Management (KM) and Information Integration [9] have evidenced the practical need to enhance DLP languages and systems to overcome the above drawbacks.

This paper describes the OntoDLV system, a first step towards overcoming the above limitations. It is a cross-platform development environment for knowledge modeling and advanced knowledge-based reasoning. The OntoDLV system

allows for the development of complex applications and allows one to perform advanced reasoning tasks in a user friendly visual environment. The OntoDLV system seamlessly integrates the DLV system [3] exploiting the power of a stable and efficient DLP solver.

A strong point of the system is its powerful language, extending DLP by object-oriented features. In particular, the language includes, besides the concept of **relation**, the object-oriented notions of **class**, **object** (class instance), **object-identity**, **complex object**, **(multiple) inheritance**, and the concept of modular programming by means of **reasoning modules**.

A *class* can be thought of as a collection of individuals that belong together because they share some features. An individual, or *object*, is any identifiable entity in the universe of discourse. Objects, also called class instances, are unambiguously identified by their object-identifier (oid) and belong to a class. A class is defined by a name (which is unique) and an ordered list of attributes, identifying the properties of its instances. Each attribute has a name and a type, which is, in truth, a class. This allows for the specification of *complex objects* (objects made of other objects).

Classes can be organized in a specialization hierarchy (or data-type taxonomy) using the built-in *is-a* relation (*multiple inheritance*).

Relationships among objects are represented by means of *relations*, which, like classes, are defined by a (unique) name and an ordered list of attributes (with name and type)[1].

Importantly, OntoDLP supports two kind of classes and relations: *(base) classes and (base) relations*, corresponding to basic facts (that can be stored in a database); and *derived classes and derived relations* corresponding to facts that can be inferred by logic programs.

As in DLP, logic programs are sets of logic rules and constraints. However, OntoDLP extends the definition of logic atom by introducing class and relation predicates, and complex terms (allowing for a direct access to object properties). In this way, the OntoDLP rules merge, in a simple and natural way, the declarative style of logic programming with the navigational style of the object-oriented systems. In addition, OntoDLP logic programs are organized in *reasoning modules*, taking advantage of the benefits of modular programming.

Noteworthy, the strongly-typed nature of OntoDLP allowed for the implementation of a number of **type-checking** routines that verify the correctness of a specification on the fly, resulting in an help for the programmer.

Moreover, OntoDLV offers several important facilities driving the development of both the knowledge base and the reasoning modules. Using OntoDLV, developers and domain experts can create, edit, navigate and query object-oriented knowledge bases by an easy-to-use **visual environment**, enriched by a graphic **query interface** à la QBE.

In short, the contribution of the paper is twofold:

– We describe a new language, named OntoDLP, for Knowledge Representation and Reasoning, extending DLP with relevant constructs of the object-

[1] Note that, unlike objects, relation instances are not identified by means of oid's.

oriented paradigm, like Classes, Types, Objects and Inheritance. We illustrate syntax, semantics, and knowledge modeling features of OntoDLP by examples.
– We design and implement a system supporting OntoDLP, named OntoDLV. The system offers all features of OntoDLP, it provides a user friendly Graphical User Interface, and a powerful type checking mechanism, which supports the user in a fast development of error-free ontologies. OntoDLV is endowed also with a visual query interface, allowing to combine navigation and querying for powerful information extraction.

The system is already employed in practice in a couple of applications for text classification and information extraction (see Section 5).

2 The OntoDLP Language

The role of a knowledge representation language is to capture domain knowledge and provide a commonly agreed upon understanding of a domain. The specification of a common vocabulary defining the meaning of terms and their relations, usually modeled by using primitives such as concepts organized in taxonomy, relations, and axioms is commonly called an ontology.

In this section we describe the OntoDLP language, a knowledge representation and reasoning language which allows one to define and to reason on ontologies.

An ontology in OntoDLP can be specified by means of *classes*, and *relations*. Classes are organized in an *inheritance* (ISA) hierarchy, while the properties to be respected are expressed through suitable *axioms*, whose satisfaction guarantees the consistency of the ontology. *Reasoning modules* allow us to express rich forms of reasoning on the ontologies.

For a better understanding, we will describe each construct in a separate section and we will exploit an example (the *living being ontology*), which will be built throughout the whole section, thus illustrating the features of the language.

It is worth noting that OntoDLP is actually an extension of Disjunctive Logic Programming (DLP)[2], which has been enriched by concepts from the object-oriented paradigm; from now on, we assume the reader to be familiar with DLP syntax and semantics. For a comprehensive introduction to DLP the reader can refer to [1, 11].

2.1 Classes

One of the most powerful abstraction mechanism for the representation of a knowledge domain is *classification*, i.e. the process of identifying object categories (*classes*), on the basis of the observation of common properties (*class attributes*).

A *class* can be thought of as a collection of individuals that belong together because they share some properties.

[2] We actually use DLP with aggregates functions [10].

Suppose we want to model the *living being* domain, and we have identified four classes of individuals: *persons*, *animals*, *food*, and *places*. Those classes can be defined in OntoDLP as follows:

class *person.* ***class*** *animal.* ***class*** *food.* ***class*** *place.*

The simplest way to declare a class is, hence, to specify the class name, preceded by the keyword **class**. However, when we recognize a class in a knowledge domain, we also identify a number of properties or attributes which are defined for all the individuals belonging to that class.

A class attribute can be specified in OntoDLP by means of a pair *(attribute-name : attribute-type)*, where *attribute-name* is the name of the property and *attribute-type* is the class the attribute belongs to.

For instance, we can enrich the specification of the class *person* by the definition of some properties which are common to each person: the name, age, father, mother, and birthplace.

Note that many properties can be represented by using alphanumeric strings and numbers. To this end, OntoDLP features the built-in classes *string* and *integer*, respectively representing the class of all alphanumeric strings and the class of non-negative numbers.

Thus, the class person can be better modeled as follows:

class *person(name:string, age:integer, father:person, mother:person, birthplace:place).*

Note that this definition is "recursive" (both father and mother are of type *person*). Moreover, the possibility of specifying user-defined classes as attribute types allows for the definition of complex objects, i.e. objects made of other objects. It is worth noting that attributes model the properties that *must* be present in all class instances; properties that *might* be present or not should be modeled, as will be shown later, by using relations[3].

In the same way, we could enrich the specification of the other above mentioned classes in our domain by adding some attributes. For instance, we could have a name for each *place*, *food* and *animal*, an age for each animal etc.

class *place(*name:string*).*

class *food(*name:string, origin:place*).*

class *animal(*name:string, age:integer, speed:integer*).*

Thus, each class definition contains a set of attributes, which is called *class scheme*. The class scheme represents, somehow, the "structure" of (the data we have about) the individuals belonging to a class.

Next section illustrates how we represent individuals in OntoDLP.

[3] In other words, an attribute $(n : k)$ of a class c is a total function from c to k; while partial functions from c to k can be represented by a binary relation on (c, k).

2.2 Objects

Domains contain individuals which are called *objects* or *instances*.

Each individual in OntoDLP belongs to a class and is univocally identified by using a constant called *object identifier* (oid) or *surrogate*.

Objects are declared by asserting a special kind of logic facts (asserting that a given instance belongs to a class). For example, we declare that "Rome" is an instance of the class *place* as follows:

rome : place(name:"Rome").

Note that, when we declare an instance, we immediately give an oid to the instance (in this case is *rome*), and a value to the attributes (in this case the *name* is the string "Rome").

The oid *rome* can now be used to refer to that place (e.g. when we have to fill an attribute of another object). Suppose that, in the *living being* domain, there is a person (i.e. an instance of the class person) whose name is "John". John is 34 years old, lives in Rome, his father and his mother are identified by *jack* and *ann* respectively. This instance can be declared as follows:

john:person(name:"John", age:34, father:jack, mother:ann, birthplace:rome).

In this case, "*john*" is *the object identifier* of this instance, while "*jack*", "*ann*", and "*rome*" are suitable oids respectively filling the attributes *father*, *mother* (both of type person) and *birthplace* (of type place).

The language semantics (and our implementation) guarantees the referential integrity, both *jack*, *ann* and *rome* have to exist when *john* is declared.

2.3 Inheritance

Another relevant abstraction tool in the the field of knowledge representation is the *specialization/generalization* mechanism, allowing to organize concepts of a knowledge domain in a taxonomy. This is obtained in the object-oriented languages by using the well-known mechanism of inheritance.

Inheritance is supported by OntoDLP, and class hierarchies can be specified by using the special binary relation *isa*.

For instance, one can exploit inheritance to represent some special categories of persons, like *students* and *employees*, having some extra attribute, like a school, a company etc. This can be done in OntoDLP as follows:

class *student* ***isa*** {*person*} (*code:string, school:string tutor:person*).

class *employee* ***isa*** {*person*}(*salary:integer, skill:string, company:string, tutor:employee*).

In this case, we have that *person* is a more generic concept or *superclass* and both *student* and *employee* are a specialization (or *subclass*) of *person*. Moreover, an instance of *student* will have both the attributes: code, school, and tutor, which are defined locally, and the attributes: name, age, father, mother, and birthplace, which are defined in *person*. We say that the latter are "inherited" from the superclass *person*. An analogous consideration can be made for the attributes of *employee* which will be name, age, father, mother, birthplace, salary, skill, company, and tutor.

An important (and useful) consequence of this declaration is that each proper instance of both *employee* and *student* will also be automatically considered an instance of *person* (the opposite does not hold!).

For example, consider the following two instances of *student* and *employee*:

al:student(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome, code:"100", school:"Cambridge", tutor:hanna).
jack:employee(name:"Jack", age:54, father:jim, mother:mary, birthplace:rome, salary:1000, skill:"Java programmer", company:"SUN", tutor:betty).

They are automatically considered also instances of person as follows:

al:person(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome).
jack:person(name:"Jack", age:54, father:jim, mother:mary, birthplace:rome).

Note that it is not necessary to assert the above two instances, both *al* and *jack* are automatically considered instances of *person*.

In OntoDLP there is no limitation on the number of superclasses (i.e. multiple inheritance is allowed). Thus, a class can be a specialization of any number of classes, and, consequently, it inherits all the attributes of its superclasses.

As an example, consider the following declaration:

class *stud-emp* ***isa*** {student, employee}(*workload:integer*).

So, the class *stud-emp* (exploiting multiple inheritance) is a subclass of both *student* and *employee*. Note that, the attribute tutor is defined in both *student*, with type *student*, and *employee* with type *employee*[4].

In this case, the attribute *tutor* will be taken only once in the scheme of *stud_emp*, but it is not intuitive what type will be taken for it.

This tricky situation is dealt with by applying a simple criterion. The type of the "conflicting" attribute *tutor* will be *employee*, which is the "intersection" (somehow in the sense of instance sharing) of the two types of the tutor attribute (*person* and *employee*). This choice is reasonably safe, and guarantees that all instances of *stud_emp* are correct instances of both *student* and *employee* [5].

[4] We acknowledge that is quite unnatural that the tutor of a student employee is an employee. Actually we made this choice to show an important feature of the language.

[5] The criterion adopted in OntoDLP for solving type conflicts due to multiple inheritance was introduced with the COMPLEX language, see [12]

We complete the description of inheritance recalling that there is also another built-in class in OntoDLP, which is the superclass of all the other classes and is called *object* (or $\top$).

2.4 Relations

A fundamental feature of a knowledge representation language is the capability to express relationships among the objects of a domain. This can be done in OntoDLP by means of *Relations*.

Relations are declared like classes: the keyword **relation** (instead of **class**) precedes a list of attributes.

As an example, the relation *friend*, which models the friendship between two persons, and the relation *lived* containing information about the places where a person lived can be declared as follows:

relation *friend(pers1:person, pers2:person).*
relation *lived(per:person, pla:place, period:string).*

Like classes, the set of attributes of a relation is called *scheme* while the cardinality of the scheme is called arity. The scheme of a relation defines the structure of its tuples (this term is borrowed from database terminology).

In particular, to assert that a person, say "john", lived in Rome for two years we write the following logic fact:

lived(per:john, pla:rome, period:"two years").

We call this assertion a tuple of the relation *lived*. Thus, tuples of a relation are specified similarly to class instances, that is, by asserting a set of facts (but tuples are not equipped with an oid).

2.5 Derived Classes and Derived Relations

The notions of class and relation introduced above correspond, from a data-base point of view, to the the *extensional* part of the OntoDLP language. In fact, their instances and tuples are defined explicitly asserting some logic facts. However, there are many cases in which some property or some class of individuals can be "derived" (or inferred) from the information already stated in an ontology. In the database world, the *views* allows to specify this this kind of knowledge which is usually called "intensional". In OntoDLP there are two different "intensional" constructs: *derived classes* and *derived relations*.

As an example, suppose we want to define the class of peoples which are less than 21 years old and have less than two friends (we name this class *youngAndShy*). Note that, this information is implicitly present in the ontology, and the "intensional" class *youngAndShy* can be defined as follows:

derived class *youngAndShy(friendsNumber: integer)* {
$X : youngAndShy(friendsNumber : N)$:– $X : person(age : Age)$,
$Age < 21$, $\#count\{F : friend(pers1 : X, pers2 : F)\} < 2$. }

Note that, in this case the instances of the class *youngAndShy* are "borrowed" from the class person, and are inferred by using a logic rule.

In general, the derived classes neither have proper instances nor proper oid's (they group already defined objects), and cannot be organized in taxonomies by using the *isa* relation.

In an analogous way we specify *derived relations*. As an example, consider the derived relations *ancestor*:

relation *ancestor(anc:person,disc:person).* {
$ancestor(anc : A, disc : X) :- X : person(father : A).$
$ancestor(anc : A, disc : X) :- X : person(mother : A).$
$ancestor(anc : A, disc : X) :- X : person(father : Y),$
$\qquad ancestor(anc : A, disc : Y).$
$ancestor(anc : A, disc : X) :- X : person(mother : Y),$
$\qquad ancestor(anc : A, disc : Y).$ }

The above definition states that A is an ancestor of X if: (i) A is either the father of X or the mother of X; or (ii) A is either an ancestor of the father of X or an ancestor of the mother of X.

Note that this definition is recursive (it is a kind of transitive closure).

In general, *derived classes* and *derived relations* are both more natural and more expressive than relational database views, in-fact they allow the use of the navigational style of object-oriented programming combined with a more powerful language that allows recursion and negation as failure[6].

2.6 Axioms and Consistency

The structural representation of a knowledge domain is obtained in OntoDLP by specifying classes and relations. In general, this information is not enough to obtain a correct description of the domain. Often, it is necessary to impose constraints asserting additional conditions which hold in the domain.

These assertions are modeled in OntoDLP by means of *axioms*.

An *axiom* is a consistency-control construct modeling sentences that are always true (at least, if everything we specified is correct). They can be used for several purposes, such as constraining the information contained in the ontology and verifying its correctness.

As an example suppose we declared the relation colleague, which associates persons working together in a company, as follows:

relation *colleague (emp1:employee, emp2:employee).*

It is clear that the information about the company of an employee (recall that there is an attribute company in the scheme of the class employee) must be consistent with the information contained in the tuples of the relation colleague. To enforce this property we assert the following axioms:

[6] It is worth noting that the programs which define derived classes and derived relations must be normal and stratified (see e.g. [3]).

(1) ::– $colleague(emp1 : X1, emp2 : X2)$, `not` $colleague(emp1 : X2, emp2 : X1)$
(2) ::– $colleague(emp1 : X1, emp2 : X2)$,
$X1 : employee(company : C)$, `not` $X2 : employee(company : C)$.

The above axioms states that, (1) the relation colleague is symmetric, and (2) if two persons are colleagues and the first one works for a company, then also the second one works for the same company.

Note that OntoDLP axioms do not derive new knowledge, but they are only used to model sentences that must be always true, like integrity constraints[7].

Observe that axioms are syntactically distinguished by constraints because they are declared by using the symbol ::– instead of :– .

The usefulness of axioms is rather clear, as they allows one to enforce the consistency of the specified ontology.

Consequently, if an axiom is violated, then we say that the ontology is inconsistent (that is, it contains information which is, somehow, contradictory or not compliant with the intended perception of the domain).

2.7 Reasoning modules

Given an ontology, it can be very useful to reason about the data it describes.

Reasoning modules are the language components endowing OntoDLP with powerful reasoning capabilities to OntoDLP. Basically, a *reasoning module* is a disjunctive logic program conceived to reason about the data described in an ontology. Reasoning modules in OntoDLP are identified by a name and are defined by a set of (possibly disjunctive) logic rules and integrity constraints.

Syntactically, the name of the module is preceded by the keyword *module* while the logic rules are enclosed in curly brackets (this allows one to collect all the rules constituting the encoding of a problem in a unique definition identified by a name). Moreover, it is possible to define *derived* predicates having a "local scope" without giving a scheme definition. This gives the possibility to exploit a form of modular programming, because it becomes possible to organize logic programs in a simple kind of library.

We now show an example demonstrating that the reasoning power of OntoDLP can be exploited for solving complex real-world problems.

Given our living being ontology, we want to compute a project team satisfying the following restrictions (i.e. we want to solve an instance of *team building problem*):

- the project team has to be constituted of a fixed number of employees;
- the availability of a given number of different skills has to be ensured inside the team;
- the sum of the salaries of the team members cannot exceed a given budget;
- the salary of each employee in the team cannot exceed a certain value.

[7] The difference between axioms and constraints is that axioms are specifically conceived to work with the knowledge contained in an ontology, while constraints are conceived in order to enforce some property in a logic program.

Suppose that the ontology contains the class *project* whose instances specify the information about the project requirements, i.e. the number of team employees, the number of different skills required in the project, the available budget, the maximum salary of each team employee:

class $project(numEmp : integer, numSk : integer, budget : integer, maxSal : integer)$.

We can solve the above team building problem with the following module:

$\boldsymbol{module}(teamBuilding)\{$

$(r) \qquad inTeam(E,P) \vee outTeam(E,P) :- E : employee(), P : project().$

$(c_1) :- P : project(numEmp : N),\ \texttt{not}\ \#\texttt{count}\{E : inTeam(E,P)\} = N.$
$(c_2) :- P : project(numSk : S),\ \texttt{not}\ \#\texttt{count}\{Sk :\ E : employee(skill : Sk), inTeam(E,P)\} \geq S.$
$(c_3) :- P : project(budget : B), \texttt{not}\ \#\texttt{sum}\{Sa, E :\ E : employee(salary : Sa), inTeam(E,P)\} \leq B.$
$(c_4) :- P : project(maxSal : M),\ \texttt{not}\ \#\texttt{max}\{Sa :\ E : employee(salary : Sa), inTeam(E,P)\} \leq M.$
$\}$

Intuitively, the disjunctive rule r guesses whether an employee is included in the team or not, generating the search space, while the constraints c_1, c_2, c_3, and c_4 model the project requirements, cutting off the solutions that do not satisfy the constraints.

Concluding, reasoning modules isolate a set of logic rules and constraints conceptually related, they exploit the expressive power of disjunctive logic programming allowing to perform complex reasoning tasks on the information encoded in an ontology.

2.8 Querying

An important feature of the language is the possibility of asking queries in order to extract knowledge contained in the ontology, but not directly expressed. As in DLP a query can be expressed by a conjunction of atoms, which, in OntoDLP, can also contain complex terms.

As an example, we can ask for the list of persons having a father who is born in Rome as follows:

X:person(father:person(birthplace:place(name: "Rome")))?

Note that we are not obliged to specify all attributes; rather we can indicate only the relevant ones for querying. In general, we can use in a query both the predicates defined in the ontology and the derived predicates in the reasoning modules.

3 The OntoDLV System

OntoDLV is a complete tool that allows one to specify, navigate, query and perform reasoning on OntoDLP ontologies. We refrain describing the implementation details of OntoDLV in this paper[8]. Rather, we illustrate the overall OntoDLV architecture, and present the main features of the system by describing the main components of the graphical user interface of OntoDLV.

3.1 System Architecture

The system architecture of OntoDLV, depicted in Figure 1, is composed of eight modules, namely, GUI, Parser, Data Handler, Type Checker, Intelligent Rewriter, Output Handler and Message Handler, DLV, and two libraries: Lucene and, JGraph.

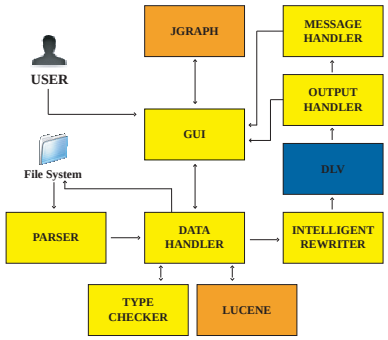


Fig. 1. The OntoDLV architecture

The user exploits the system through an easy-to-use visual environment called GUI (Graphical User Interface). The GUI combines a number of specialized visual tools for authoring, browsing and querying a OntoDLP ontology. In particular, the GUI features a graph-based ontology viewer and a graphical query environment, which are based on JGraph, an open-source library.

The Parser has the job to analyze and load the content of a OntoDLP text file in the data structures supplied by the Data Handler. The Data Handler provides all the methods needed to access and manipulate the ontology components. In particular, data indexing and full-text search are based on the open-source library Lucene. The admissibility of an ontology is ensured by the Type Checker module which implements a number of type checking routines. The Intelligent Rewriter module translates OntoDLP ontologies, reasoning modules and queries to an equivalent Disjunctive Logic Program which runs on the DLV system. The Intelligent Rewriter features a number of optimization and caching techniques in order to reduce the time used by interacting with DLV. Reasoning results

[8] For a better description of the implementation and for an in-depth specification of the rewriting procedure see [15].

and possible error messages are handled by the Output Handler and by the Message Handler modules respectively, and are displayed by the user interface accordingly.

3.2 Implementation and Usage

The OntoDLV system has been implemented in Java and is based on an efficient and optimized implementation of the rewriting module. Moreover, the OntoDLV system exploits the DLV system, a state-of-the-art DLP solver that has been shown to perform efficiently on both hard and "easy" (having polynomial complexity) problems[9].

The DLV system is a highly portable software written in ISO C++, available for various operating systems (UNIX, Mac OSX and Windows). Thus, the OntoDLV system runs under a variety of operating systems.

The OntoDLV system was designed to be simple for a novice to understand and use, and powerful enough to support experienced users.

The GUI presents several panels offering access to several facilities combining the browsing environment with the editing environment.

The class/subclass hierarchy is displayed both in an indented text and a graph-based form.

The user can browse the ontology by double-clicking the items in the panels. The structure of each ontology entity (classes, relations, and instances) can be displayed in the middle of the screen by switching among several tabbed panels.

In the editing phase, the user enters the domain information by filling in the blanks of intuitive forms and selecting items from lists (exploiting an simple mechanism based on drag-and-drop). An up-to-date list of messages informs the user about the occurrence of errors (e.g. type checking messages, etc.) in the ontology under development. When the user clicks on an error message item the system promptly shows the entity involved in it.

Reasoning and querying can be performed by selecting the appropriate panel. The interface also allows the reasoning modality (both brave reasoning and cautious reasoning are supported) to be selected, and the reasoning modules needed to solve the specified reasoning task to be enabled/disabled. Importantly, queries can also be created by exploiting both a text-editing tool and a visual querying interface à la QBE which allows one to write queries without wondering about the syntax in a drag-and-drop-based environment. A sort of "reverse-engineering" procedure allows to smoothly switch between the text editing and the visual editing environment.

Finally, query results are presented to the user in an appealing way, while details about the interaction with DLV are hidden by the system.

The OntoDLV system, together with the system manual describing all the features available, can be downloaded at *http://www.mat.unical.it/ontodlv*.

[9] This feature is crucial for the implementation of the OntoDLV system, in fact ontologies are translated in an equivalent DLP program which is solved by DLV in polynomial time (under data complexity)

4 Related Work

A number of languages and systems somehow related to OntoDLP have been proposed in the literature. The most closely related system is COMPLEX [12], supporting the Complex-Datalog language, an extension of (non-disjunctive) Datalog with some concepts from the object-oriented paradigm. OntoDLV and COMPLEX share a similar object-oriented model, however the language of the latter is less expressive than OntoDLP. In fact, COMPLEX supports normal (non-disjunctive) stratified programs only (its expressive power is confined to P), which are strictly less expressive than OntoDLP language expressing even Σ_2^P-complete properties [2].

Another popular logic-based object-oriented language is F-Logic [13], implemented in the Flora-2 system [14], which includes most aspects of object-oriented and frame-based languages. F-logic was conceived as a language for intelligent information systems based on the logic programming paradigm. Comparing OntoDLP with F-Logic, we note that the latter has a richer set of object oriented features (e.g. class methods, and multi-valued attributes), but it misses some important constructs of OntoDLP like disjunctive rules, which increase the knowledge modeling ability of the language. Concerning system-related aspects, an important advantage of OntoDLV (w.r.t. Flora-2) is the presence of a graphical development environment, which simplifies the interaction with OntoDLV for both the end user and the knowledge engineer.

A couple of other formalisms for specifying ontologies have been recently proposed by W3C, namely, RDF/RDFS and OWL. The Resource Description Framework (RDF) [16] is a knowledge representation language for the Semantic Web. It is a simple assertional logical language which allows for the specification of binary properties expressing that a resource (entity in the Semantic Web) is related to another entity or to a value. RDF has been extended with a basic type system; the resulting language is called RDF Vocabulary Description Language (RDF Schema or RDFS). Basically, RDF(S) allows for expressing knowledge about the resources (identified via URI), and features a rich data-type library (richer than OntoDLP), but, unlike OntoDLP, it does not provide any way to extract new knowledge from the asserted one (RDFS does not support any "rule-based" inference mechanisms nor query facilities).

The Ontology Web Language (OWL)[17] is an ontology representation language built on top of RDFS. The ontologies defined in this language consist of *concepts* (or classes) and *roles*(binary relations also called class properties). OWL has a logic based semantics, and in general allows to express complex statements about the domain of discourse (OWL is undecidable in general)[17]. The largest decidable subset of OWL, called OWL-DL, coincides, basically, with $SHOIN(D)$, an expressive Description Logic (DL)[18]. OWL is based on classical logic (there is a direct mapping from $SHOIN$ to First Order Logic (FOL)) and, consequently, is quite different form OntoDLP, which is based on DLP. Compared to OntoDLP, OWL misses, for instance, *default negation*, *nonmonotonic disjunction*, and *inference rules*.

In sum, the strong point of OntoDLP, w.r.t. to other ontology representation languages, is the natural way in which it combines the most common ontology definition constructs with a powerful logic programming language, including rules, nonmonotonic disjunction, and default negation.

5 Conclusion

In this paper, we have presented the OntoDLP language, an extension of disjunctive logic programming with relevant object-oriented constructs, including classes, objects, (multiple) inheritance, and types. By using an example, we have described the syntax of the language, and shown its usage for ontology representation and reasoning.

Importantly, we have provided also a concrete implementation of OntoDLP: the OntoDLV system. OntoDLV is built on top of DLV (the state-of-the art DLP system). It implements all features of OntoDLP, it also provides an advanced visual-interface, and a powerful type-checking mechanism, supporting the user for fast ontologies specification and errors detection.

The OntoDLV system is a valid support for the development of knowledge-based applications. Indeed, even if OntoDLP has been released very recently, it is already employed, playing a central role, in a couple of advanced applications for information extraction and text classification: HiLEx [19] and OLEX [21].

Ongoing work concerns the enhancement of OntoDLV by extending its language with new features such as optional and multi-valued attributes, a more powerful forms of both intentional classes, and reasoning modules.

Acknowledgements

This work was partially supported by M.I.U.R. under projects "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione", and "ONTO-DLV: Un ambiente basato sulla Programmazione Logica Disgiuntiva per il trattamento di Ontologie" n.2521.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22** (1997) 364–418
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL (2005) To appear.
4. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). LNCS 2923

5. Lierler, Y.: Cmodels for Tight Disjunctive Logic Programs. In: W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany. Ulmer Informatik-Berichte, Universität Ulm, Germany (2005) 163–166
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2002)
7. Lobo, J., Minker, J., Rajasekar, A.: Foundations of Disjunctive Logic Programming. The MIT Press, Cambridge, Massachusetts (1992)
8. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Nonmonotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar on Nonmonotonic Reasoning, Answer Set Programming and Constraints (2005)
9. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
10. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: IJCAI 2003, Acapulco, Mexico,(2003) 847–852
11. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: Logic-Based Artificial Intelligence. Kluwer (2000) 79–103
12. Greco, S., Leone, N., Rullo, P.: COMPLEX: An Object-Oriented Logic Programming System. IEEE TKDE **4** (1992)
13. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. JACM **42** (1995) 741–843
14. Yang, G., Kifer, M., Zhao, C.: "flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web.". In: CoopIS/DOA/ODBASE. (2003) 671–688
15. Ricca, F., Leone, N.: Disjunctive Logic Programming with Types and Objects: The DLV$^+$ System. KBS Research Reports INFSYS RR-1843-05-10 Institut für Informationssysteme Technische Universität Wien - Austria
16. W3C: The resource description framework. (2006) http://www.w3.org/RDF/./.
17. Smith, M.K., Welty, C., McGuinness, D.L.: OWL web ontology language guide. W3C Candidate Recommendation (2003) http://www.w3.org/TR/owl-guide/.
18. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation, and Applications. CUP (2003)
19. Ruffolo, M., Leone, N., Manna, M., Sacca', D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
20. Cumbo, C., Iiritano, S., Rullo, P.: Reasoning-based knowledge extraction for text classification. In: Discovery Science. (2004) 380–387
21. Curia, R., Ettorre, M., Iiritano, S., Rullo, P.: Textual Document Per-Processing and Feature Extraction in OLEX. In: Proceedings of Data Mining 2005, Skiathos, Greece (2005)
22. Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. NGC **9** (1991) 401–424

A Disjunctive Logic Programming

In this section, we provide a brief introduction to the syntax and semantics of Disjunctive Logic Programming; for further background see [11, 1].

Syntax. A *disjunctive rule* R is a formula:

$$a_1 \vee \cdots \vee a_n \ :- \ b_1, \cdots, b_k, \ \texttt{not} \ b_{k+1}, \cdots, \ \texttt{not} \ b_m.$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atomsand $n \geq 0$, $m \geq k \geq 0$. A literal is either an atom a or its default negation $\texttt{not} \ a$. Given a rule r, let $H(r) = \{a_1, ..., a_n\}$ denote the set of head literals, $B^+(r) = \{b_1, ..., b_k\}$ and $B^-(r) = \{\texttt{not} \ b_{k+1}, ..., \texttt{not} \ b_m\}$ the set of positive and negative body literals, resp., and $B(r) = B^+(r) \cup B^-(r)$. the set of body literals.

A rule r with $B^-(r) = \emptyset$ is called *positive*; a rule with $H(r) = \emptyset$ is referred to as *integrity constraint*. If the body is empty we usually omit the $:-$ sign.

A *disjunctive logic program* $\mathcal{P}$ is a finite set of rules; $\mathcal{P}$ is a *positive* program if all rules in $\mathcal{P}$ are positive (i.e., `not` -free). An object (atom, rule, etc.) containing no variables is called *ground* or *propositional*.

Semantics. The semantics of a disjunctive logic program is given by its stable models [22], which we briefly review in this section.

Given a program $\mathcal{P}$, let the *Herbrand Universe* $U_\mathcal{P}$ be the set of all constants appearing in $\mathcal{P}$ and the *Herbrand Base* $B_\mathcal{P}$ be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in $\mathcal{P}$ with the constants of $U_\mathcal{P}$.

Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in R to elements of $U_\mathcal{P}$. Similarly, given a program $\mathcal{P}$, the *ground instantiation* $\mathcal{P}$ of $\mathcal{P}$ is the set $\bigcup_{R \in \mathcal{P}} Ground(r)$.

For every program $\mathcal{P}$, we define its stable models using its ground instantiation $\mathcal{P}$ in two steps: First we define the stable models of positive programs, then we give a reduction of general programs to positive ones and use this reduction to define stable models of general programs.

A set L of ground literals is said to be *consistent* if, for every atom $\ell \in L$, its complementary literal $\texttt{not} \ \ell$ is not contained in L. An interpretation I for $\mathcal{P}$ is a consistent set of ground literals over atoms in $B_\mathcal{P}$.A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its complementary literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I. Interpretation I is *total* if, for each atom A in $B_\mathcal{P}$, either A or $\texttt{not} \ .A$ is in I (i.e., no atom in $B_\mathcal{P}$ is undefined w.r.t. I). A total interpretation M is a *model* for $\mathcal{P}$ if, for every $R \in \mathcal{P}$, at least one literal in the head is true w.r.t. M whenever all literals in the body are true w.r.t. M. X is a *stable model* for a positive program $\mathcal{P}$ if its positive part is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. an interpretation X is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by (i) deleting all rules $R \in \mathcal{P}$ whose negative body is false w.r.t. X and (ii) deleting the negative body from the remaining rules.

A stable model of a general program $\mathcal{P}$ is a model X of $\mathcal{P}$ such that X is a stable model of $\mathcal{P}^X$.

Semantic Information Elicitation from Unstructured Medical Records

Massimo Ruffolo[1,3], Vittoria Cozza[2], Lorenzo Gallucci[1,2] Marco Manna[4], Mariarita Pizzonia[2]

[1] Exeura s.r.l.
[2] DEIS - Department of Electronics, Computer Science and Systems
[3] ICAR-CNR - Institute of High Performance Computing and Networking of the Italian National Research Council
[4] Department of Mathematics
University of Calabria, 87036 Arcavacata di Rende (CS), Italy
e-mail: ruffolo@icar.cnr.it
e-mail: manna@mat.unical.it
e-mail: {cozza,pizzonia}@deis.unical.it
e-mail: {ruffolo,gallucci}@exeura.it
WWW home page: http://www.exeura.it

Abstract. Semantic elicitation of relevant information entities from semi- and unstructured documents is an important problem in many application fields. This paper describes HiLeXa system implementing a very powerful semantic approach to information extraction from semi- and unstructured documents obtained combining knowledge representation formalisms, like ontology languages, and two-dimensional languages exploiting a two-dimensional spatial representation of documents. The HiLeX system constitutes a new generation technology capable of capturing and eliciting relevant information regarding a specific domain. It is founded on OntoDLP, an extension of disjunctive logic programming for ontology representation and reasoning. In the HiLeX system the semantics of the information to be extracted is represented by using OntoDLP ontologies and the extraction patterns are expressed by means of regular and two-dimensional expressions. By converting the extraction patterns to OntoDLP reasoning modules, the HiLeX system can actually extract information from HTML pages as well as from flat text documents using the same patterns. In this paper the extraction of clinical information and events, regarding patients, diseases, therapies and drugs, from electronic textual medical records is shown. Extracted information are represented in XML and can be stored in structured form using relational database or ad-hoc ontologies to enable further analysis.

1 Introduction

To extract automatically relevant information entities from unstructured electronic sources is an important problem of information management in many application fields. Information contained in semi- and unstructured documents, is usually arranged according to syntactic, semantic and presentation rules of a given natural

language. While this is useful to humans, the lack of machine readability make impossible to manage the huge amount of information available on the Internet and in the document repositories of all kind of organizations. Information extraction allows the acquisition of information from semi- and unstructured documents and their storage in a structured machine-readable form useful for further analysis and exchanges. Existing information extraction systems are unable to handle the actual knowledge that the information conveys because the lack of semantic-awareness [2,1,11,13,7]. Being able to exploit the semantics is extremely important in order to recognize and extract automatically relevant information (entities) from semi- and unstructured documents.

In this paper is presented the application of HiLeX, a language independent system for semantic information extraction, to the extraction of information about clinical processes (regarding entities like patients, diseases, treatments, drugs, therapies, etc.) from electronic medical records having a flat textual form [8]. Extracted information are represented in XML and can be stored in relational database or in OntoDLP ontologies.

The system implements a logic based approach to information extraction which combines both syntactic and semantic knowledge for the expression of very powerful extraction patterns. In particular, the paper shows the extraction of information from a set of clinical records, in Italian language, regarding patients affected by lungs cancer.

The paper is organized as follow: in section 2 are described the HiLeX system, the ontologies and the patterns defined to extract medical entities of interest; in section 3 are shown the electronic medical records in input to the HiLeX system and the organization of the structured records obtained as output of the extraction process.

2 HiLeX system overview

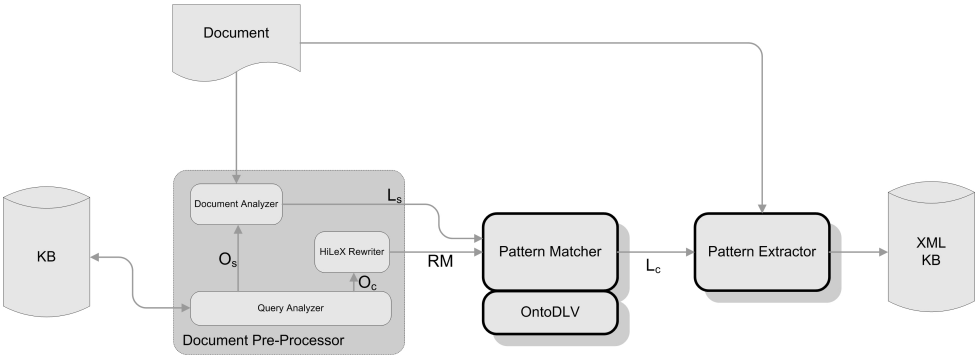


Fig. 1. HiLeX architecture

The HiLeX system is based on the exploitation of the OntoDLP [4] a powerful logic-based ontology representation language which extends Disjunctive Logic Pro-

gramming (DLP) [9] with object-oriented features, such as relations, classes, object instances. Also, notions coming from object-oriented world are present, such as complex-objects, (multiple) inheritance and the concept of modular programming (by means of *reasoning modules*). This makes OntoDLP a complete ontology representation language supporting sophisticated reasoning capabilities. The OntoDLP language is implemented in the OntoDLV system, a cross-platform development environment for knowledge modeling and advanced knowledge-based reasoning. The OntoDLV system [4] permits to easily develop real world complex applications and allows advanced reasoning tasks in a user friendly visual environment. OntoDLP seamlessly integrates the DLV [6,12] system exploiting the power of a stable and efficient Answer Set Programming solver (for further background on DLV and DLP^{+}see [4,5]).

OntoDLP allows the formal representation of the semantics of information to be extracted (by means of suitable ontologies) and the encoding of the *logic two-dimensional representation* of unstructured documents. Moreover, OntoDLP reasoning modules (which are specialized OntoDLP logic programs) allow the exploitation of the the bottom-up reasoning capability, and thus the implementation of the logic-based pattern matching method yielding the actual semantic information extraction.

The semantic information extraction approach, implemented in the HiLeX system, can be viewed as a process composed of four main steps: knowledge representation, document preprocessing, pattern matching and pattern extraction. To understand how HiLeX works in the following the main system modules are shortly described. A more detailed description of the HiLEx system is given in [14].

2.1 Knowledge Base

The Knowledge Base (KB) stores core and domain ontologies describing the semantics of information to extract, extraction patterns and the logic *two-dimensional representation* of semi- and unstructured documents. The KB provide an API containing methods aimed at handling ontology querying and at assisting pattern specification and matching.

The elements of information to be extracted (entities) are modeled starting from the OntoDLP class *element* which is defined as follows:

```
class element (type: expression_type, expression: string,
   label: string).
```

The three attributes have the following meaning:

- `expression`: holds a string representing the pattern specified by regular expressions or by the HiLeX two-dimensional language, according to the `type` property. Patterns contained in these attributes are used to recognize the information elements in a document.
- `type`: defines the type of the expression (i.e. `regexp_type`, `hilex_type`).
- `label`: contains a description of the element in natural language.

The element class is the common root of both `core ontology` and `domain ontologies`. Every pattern encoding information to be extracted is represented by an instance of a class belonging to these ontologies.

The internal representation of extraction patterns is obtained by means of a two-dimensional language, founded on picture languages [3,10], and allowing the definition of very expressive target patterns. Each pattern represents a two-dimensional composition of portions annotated w.r.t. the elements defined in the ontology.

The two-dimensional language exploit the two-dimensional representation of an unstructured document, constituting the main notion, which the semantic information extraction approach implemented in the HiLeX system, is founded on. Following this idea elements are located inside rectangular regions of the input document called `portions` each univocally identified through the Cartesian coordinates of two opposite vertices. Document portions, and the enclosed elements, are represented in OntoDLP by using the class *point* and the relation *portion*

```
class point (x: integer, y: integer).
relation portion (p: point, q: point, elem: element).
```

Each instance of the relation `portion` represents the relative rectangular document region. It relates the two points identifying the region, expressed as instances of the class `point`, and an ontology element, expressed as instance of the class `element`. The set of instances of the `portion` relation constitute the *logic two-dimensional representation* of an unstructured document. This OntoDLP encoding allows to exploit the two-dimensional document representation for pattern matching.

In the following the structure of core and domain ontologies are described in greater detail.

The Core Ontology The core ontology represents general information elements valid in all the possible application domain. It is composed of three parts. The first part represents general elements describing a language (like, e.g., alphabet symbols, Part-of-Speech, regular forms such as date, e-mail, etc.). The second part represents elements describing presentation styles (like, e.g., font types, font styles, font colors, background colors, etc.). The third part represents structural elements describing tabular and textual structures (e.g. table cells, table columns, table rows, paragraphs, item lists, texture images, text lines, etc.). The core ontology is organized in the class hierarchy shown below:

```
class linguistic_element isa {element}.
    class character isa {linguistic_element}.
        class number_character isa {character}.
        ...
    class regular_form isa {linguistic_element}.
        class float_number isa {regular_form}.
        ...
    class separator isa {linguistic_element}.
    ...
class presentation_element isa {element}.
    class font_type isa {presentation_element}.
    ...
class structural_element isa {element}.
```

```
class table_cell isa {structural_element}.
...
```

Examples of instances of these classes are:

```
float_number: number (type: regexp_type,
                          expression:"(\\d{1,3}(?>.\\d{3})*,\\d+)").
```

When in a document a regular expression (pattern) characterizing a particular concept is recognized, a document portion is generated and annotated w.r.t. the corresponding class instance.

Domain Ontologies The Domain ontologies contain information elements of a specific knowledge domain. The distinction between core and domain ontologies allows to describe knowledge in a modular way. When a user need to extract data from a document regarding a specific domain, he can use only the corresponding domain ontology. The modularization improve the extraction process in terms of precision and overall performances.

A strong research effort has been taken, in the recent past, to provide an uniform representation of medical knowledge useful in health care information systems. Interesting results have been obtained in the field of medical knowledge representation, where many ontologies, such as *UMLS*, *ICD9-CM*, have been developed on different medical topics. In this work a domain ontology, inspired to *ICD9-CM*, has been implemented in OntoDLP to model knowledge and patterns about oncological domain. In the following is shown a piece of the ontology regarding the care of lung's cancer.

```
class cura_tumore_domain_element isa {tumore_domain_element}.
    class modalita isa {cura_tumore_domain_element}.
        class modalita_terapia isa {modalita}.
    class terapia_domain_element isa {cura_tumore_domain_element}.
        class chemioterapia isa {terapia_domain_element}.
        class radioterapia isa {terapia_domain_element}.
        class intervento_domain_element isa {terapia_domain_element}.
            class intervento_chirurgico isa {intervento_domain_element}.
    class medicinale_domain_element isa {cura_tumore_domain_element}.
        class farmaco isa {medicinale_domain_element}.
            class farmaco_chemioterapico isa {farmaco}.
        class posologia isa {medicinale_domain_element}.
```

The medical domain ontology that deals with therapies for the lung's cancer is shown graphically in figure 2. In particular, in the ontology are represented patterns expressing the three main modalities of lung's cancer care (therapies) the surgery, the x-ray and the chemotherapy. The HiLεX patterns allowing the extraction of chemotherapy (drugs dosage), expressed by using the construct of HiLεXtwo-dimensional language, *sequenceOf* are showed below:

```
farmaco_001: farmaco_chemioterapico (type: regexp_type,
```

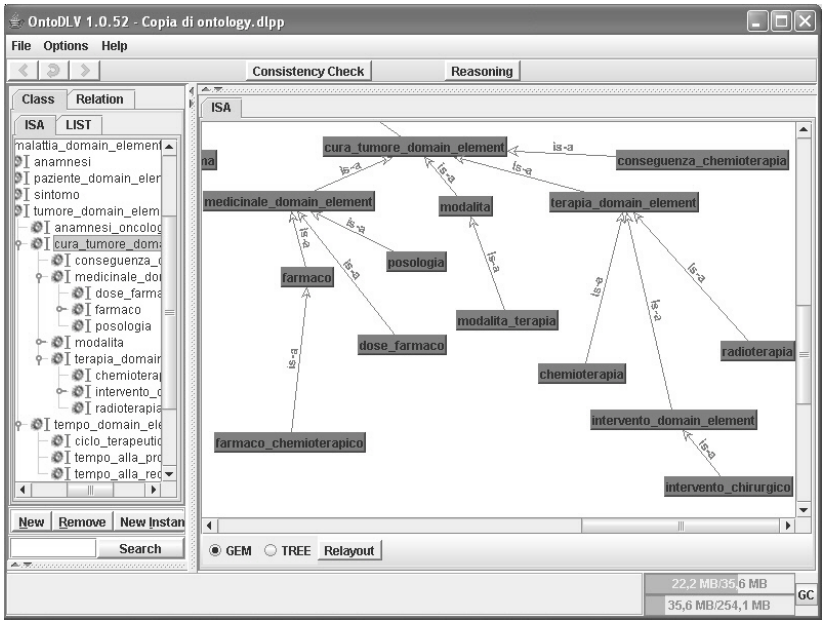


Fig. 2. Ontology snapshot

```
    expression: "cddp|cisplatino", label: " ").
farmaco_002: farmaco_chemioterapico (type: regexp_type,
    expression:"gemcitabina|gem|gemzar", label: " ").

posologia_01: posologia (type: hilex_type,
    expression:"sequenceOf(arg: [@number, @unita_misura, @number,
                @periodo], dir:horizontal, sep: sep002)", label: " ").

chemioterapia_01: chemioterapia (type: hilex_type,
    expression:"sequenceOf (arg: [@farmaco_chemioterapico,
                @posologia], dir: horizontal, sep: sep002)", label: " ").
```

The *sep002* instance define a separators among concepts. It helps to easily express that one or more white space can be found between the concepts *farmaco_chemioterapico* and *posologia*. It is worthwhile noting that each pattern allows to obtain a more complex concept as a two-dimensional composition of more simple information elements expressed by means of ontology concepts. The notation *@farmaco_chemioterapico* expresses a generic instance of the concept *farmaco_chemioterapico* that represents a set of possible chemotherapy drugs.

2.2 Document Preprocessor

The document preprocessor takes as input an unstructured document (i.e. the flat text medical records), and a set of class and instance names representing the

information that the user wishes to extract. After the execution the document preprocessor returns the *two-dimensional logic representation* of the document and a set of reasoning modules constituting the input for the pattern matcher. The document preprocessing is performed by the three sub-modules described in the following.

Query analyzer. This submodule takes as input a set of classes and instances and explores the ontology in order to identify patterns for the extraction process. The output of the query analyzer are two sets of couples (*class instance name*, *pattern*). The first set (O_s) contains couples in which instances are characterized by patterns represented by regular expressions (simple elements), whereas in the second set (O_c) patterns are expressed using the HiLeX pattern representation language (complex elements). The set O_s is the input for the document analyzer submodule and the set O_c is the input for the rewriter submodule.

Document Analyzer. The input of this submodule is an unstructured document and the set of couples O_s. The document analyzer is able to recognize regular expressions, applying pattern matching mechanisms, to detect simple elements constituting the document and for each of them generates the relative *portion*. At the end of the analysis this module provides the *logic two-dimensional document representation* L_s which is a uniform abstract view of different document formats. In order to perform adequately even on large input documents the Document Analyzer is built as a reconfigurable set of specialized *processing units*, managed by a framework named CPF (Concurrent Processing Framework). Each unit is highly focused on a small part of the analysis process (e.g. a regular expression recognizer takes document fragments as input and can produce objects named *matches*), works on its input set members one at a time and is able to yield same results not dependant on the particular permutation of members sequence seen as input (i.e. a processing unit can work on its input set in any order). Since unit operation scheduling can be changed freely, while keeping correct results, the CPF is allowed to employ an execution strategy customized for a particular execution environment, in order to achieve top performances.

HiLeX Rewriter. The input for this submodule is the set of couples O_c containing the extraction patterns expressed by means of the HiLeX two-dimensional language. Each pattern is translated in a set of logical rules implemented in a OntoDLP reasoning modules (RM) which are to be executed by the OntoDLV system. The translation allows the actual semantic information extraction from unstructured documents performed by the pattern matcher module.

2.3 Pattern Matcher

The pattern matcher is founded on the OntoDLV system. It takes as input the logic two-dimensional document representation (L_s) and the set of reasoning modules (RM) containing the translation of the HiLeX patterns in term of logic rules and recognize new complex elements. The output of this step is the *augmented logic*

two-dimensional representation (L_c) of an unstructured document in which new document regions, containing more complex elements (e.g. phrases containing information about chemotherapy, diagnosis, etc.) are identified. The logic-based pattern matching mechanism implemented in this module exploits the translation of extraction patterns performed by the HiLeX rewriter submodule.

It is noteworthy that patterns are very synthetic and expressive. Moreover, patterns are general in the sense that they are independent from the document format. This last peculiarity implies that the extraction patterns, presented above, are more robust w.r.t. variations of the page structure than extraction patterns defined in the previous approaches.

2.4 Pattern Extractor

This module takes in input the augmented logic representation of a document (L_c) and allows the acquisition of requested information entities. Acquired entities are represented in XML and can be stored in a OntoDLV ontology, a relational database, an XML database. So, extracted information can be used in other applications and more powerful query and reasoning task are possible on them. The extraction process causes the annotation of the documents w.r.t. the ontologies concepts. This feature can enable, for example in document management contexts the semantic classification.

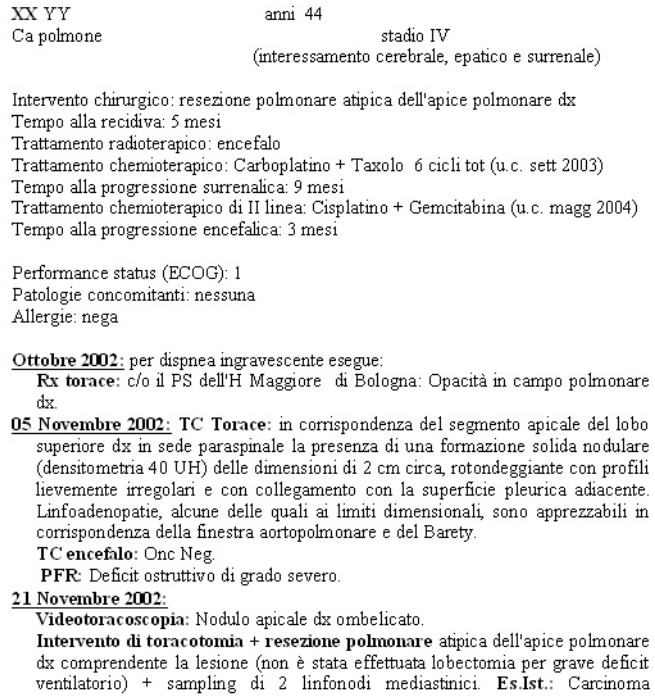
3 Information Extraction from unstructured medical records

In this section the clinical data extraction problem, from electronical medical records (EMR) in textual format, and the data structuring by means of XML, is faced. In the experiments has been extracted entities regarding hospital and ward name; patient identifier, sex an age; diagnosis and its date; oncological familiar analysis; chemotherapy; time to tumor progression and tumor recurrence; side effects of the chemotherapy. In the experiments has been used 100 EMR, written in Italian language, belonging to patients with lung's cancer. A cross-validation to proof the efficiency of the information extraction approach as been performed on them. In figure 3 is shown an EMR piece.

EMRs are weakly-structured documents (having usually 3 pages) since can be find in them, frequently, a standard structure. For example, the personal data of the patient are in the top of the document, the timetable of clinical events (medical exams, surgical operations, diagnosis, chemotherapy, etc.) is introduced by a date, and so on.

The semantic information extraction form the EMR is required because it's quite important for the doctors to have, easily and rapidly, information on the state of a patient, diagnosis, surgical operations and chemotherapies. These information should be available for all the medical staff in the same hospital, but also for more hospitals in different geographic areas.

The semantic of the medical information and the extraction patterns are represented as shown in the previous sections. The extracted entities are stored in a



XX YY anni 44
Ca polmone stadio IV
(interessamento cerebrale, epatico e surrenale)

Intervento chirurgico: resezione polmonare atipica dell'apice polmonare dx
Tempo alla recidiva: 5 mesi
Trattamento radioterapico: encefalo
Trattamento chemioterapico: Carboplatino + Taxolo 6 cicli tot (u.c. sett 2003)
Tempo alla progressione surrenalica: 9 mesi
Trattamento chemioterapico di II linea: Cisplatino + Gemcitabina (u.c. magg 2004)
Tempo alla progressione encefalica: 3 mesi

Performance status (ECOG): 1
Patologie concomitanti: nessuna
Allergie: nega

Ottobre 2002: per dispnea ingravescente esegue:
Rx torace: c/o il PS dell'H Maggiore di Bologna: Opacità in campo polmonare dx.
05 Novembre 2002: TC Torace: in corrispondenza del segmento apicale del lobo superiore dx in sede paraspinale la presenza di una formazione solida nodulare (densitometria 40 UH) delle dimensioni di 2 cm circa, rotondeggiante con profili lievemente irregolari e con collegamento con la superficie pleurica adiacente. Linfoadenopatie, alcune delle quali ai limiti dimensionali, sono apprezzabili in corrispondenza della finestra aortopolmonare e del Barety.
TC encefalo: Onc Neg.
PFR: Deficit ostruttivo di grado severo.
21 Novembre 2002:
Videotoracoscopia: Nodulo apicale dx ombelicato.
Intervento di toracotomia + resezione polmonare atipica dell'apice polmonare dx comprendente la lesione (non è stata effettuata lobectomia per grave deficit ventilatorio) + sampling di 2 linfonodi mediastinici. **Es.Ist.:** Carcinoma

Fig. 3. Input: clinical record (flat text)

common and understandable XML format. A piece of XML record in output from the extraction process is the XML element presented below:

```
<chemioterapia>
    <patient_id> 8723746 <patient_id>
    <date> 13/01/2006 </date>
    <farmaco>
        <name> Platinex </name>
        <posologia> 75 mg/mq 1 g  </posologia>
    </farmaco>
    <farmaco>
        <name> Gemzar </name>
        <posologia> 1200 mg/mq 1,8 gg </posologia>
    </farmaco>
</chemioterapia>
```

In order to extract the type of surgery on the patient, complex operations, for example the removal or the resection of a particular part of the body, implemented through the composition of lower level HiLeX expressions, have been expressed. To extract the item related with x-ray, after expressing the concept of x-ray and several ways to express it, it has been taken into consideration the way to execute such therapy or the purpose of such therapy (palliative or analgesic scope).

4 Conclusions and future works

The semantic approach to information extraction presented in this work is novel, powerful and expressive, as well as concrete (it is implemented in the HιLεX system), and constitutes an enhancement in the field of information extraction. Unlike previous approaches, the same extraction patterns can be used to extract information, according to their semantics, from different kind of semi- and unstructured documents (HTML, flat text).

This work shows, also, how semantic information extraction allows the acquisition of relevant entities of a domain. Using HιLεX doctors can acquire and structure information about clinical process without change in the usual clinical practices. The definition of suitable extraction patterns allows to obtain main EMR information when it are written in flat text format. In particular, information about the kind of patient surgery under some particular conditions (age, familiar oncological anamnesis), the therapy for the patient, the effects on the patient because such therapy, the surgical operations and its results, the time a disease takes to propagate and finally the time the cancer takes to develop itself again after the last surgery, can be captured.

Using the obtained structured data (a structured EMR for each patient) the oncological ward of the hospital can monitor the state of disease of patients daily. The structured EMR can also be used to exchange information on the patient with others clinical center where, for example, patient could be hosted in the future. Moreover, obtained EMR can be exchanged among medical researchers to try discovering, for example by means of data mining methods, the effect of innovative diagnostic approaches and/or therapeutic procedures and the adverse reactions to some drugs.

Currently, consolidation of the approach is ongoing and its theoretical foundations are under investigation and improvement. Future work will be focused on the consolidation and extension of the HιLεX two-dimensional language, the investigation of the computational complexity issues from a theoretical point of view, the extension of the approach to PDF as well as other document formats, the exploitation of natural language processing techniques aimed at improve information extraction from flat text documents.

References

1. R. Baumgartner, S. Flesca, and G. Gottlob. Declarative information extraction, web crawling, and recursive wrapping with lixto. In *LPNMR '01: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 21–41, London, UK, 2001. Springer-Verlag.
2. R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with lixto. In *The VLDB Journal*, pages 119–128, 2001.
3. S.-K. Chang. The analysis of two-dimensional patterns using picture processing grammars. In *STOC '70: Proceedings of the second annual ACM symposium on Theory of computing*, pages 206–216, New York, NY, USA, 1970. ACM Press.
4. T. Dell'Armi, N. Leone, and F. Ricca. Il linguaggio dlp+. Internal report, Exeura s.r.l, June 2004.

5. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.
6. W. Faber and G. Pfeifer. Dlv homepage, since 1996.
7. R. Feldman, Y. Aumann, M. Finkelstein-Landau, E. Hurvitz, Y. Regev, and A. Yaroshevich. A comparative study of information extraction strategies. In A. F. Gelbukh, editor, *CICLing*, volume 2276 of *Lecture Notes in Computer Science*, pages 349–359. Springer, 2002.
8. C. Friedman, G. Hripcsak, W. DuMouchel, S. B. Johnson, and P. D. Clayton. Natural language processing in an operational clinical environment. In *Natural Language Engineering*, volume 1, pages 83–108, 1995.
9. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.
10. D. Giammarresi and A. Restivo. Two-dimensional languages. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, Beyond Words, pages 215–267. Springer-Verlag, Berlin, 1997.
11. S. Kuhlins and R. Tredwell. Toolkits for generating wrappers – a survey of software toolkits for automated data extraction from web sites. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science (LNCS)*, pages 184–198, Berlin, Oct. 2003. International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7–10, 2002, Springer.
12. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. 2004.
13. B. Rosenfeld, R. Feldman, M. Fresko, J. Schler, and Y. Aumann. Teg: a hybrid approach to information extraction. In D. Grossman, L. Gravano, C. Zhai, O. Herzog, and D. A. Evans, editors, *CIKM*, pages 589–596. ACM, 2004.
14. M. Ruffolo, N. Leone, M. Manna, D. Sacc, and A. Zavatto. Exploiting asp for semantic information extraction. In *Answer Set Programming*, 2005.

Applying ASP Inferential Engines to the Filtering, Decoration and Validation of Data from Web Sources

Massimo Marchi[1], Giacomo Fiumara[2], and Alessandro Provetti[2]

DSI - University of Milan,
Via Comelico, 39/41. I-20135 Milan, Italy,
marchi@dsi.unimi.it
Dept. of Physics, University of Messina,
Sal. Sperone, 31. S. Agata di Messina. I-98166 Messina, Italy,
{*fiumara,ale*}*@unime.it*
http://mag.dsi.unimi.it/

Abstract. We propose a software architecture for semantics-based annotation of data extracted from Web sources. Data can be extracted from arbitrary Web sources thanks to two wrapping engines: the *LiXto* suite, which supports semi-automated data extraction and XML formatting, and *Dynamo*, which is based on the novel approach of HTML meta-tag decoration. The XML tags produced by the wrappers are then fed to an Answer Set Programming inferential engine which applies filters or annotates tags with extra information. Unlike with XSLT-based tag manipulation, inference-based annotations can exploit meta-information (e.g., about the source of the data) or incomplete information to draw conclusions.

1 Introduction

This article illustrates the architecture for Web data gathering and annotation that we have developed by combining novel and existing modules. The key functionalities of our application, i.e., data extraction from HTML pages and annotation of XML tags with new data, are defined in terms of *default rules,* with Datalog-style syntax and Gelfond-Lifschitz Answer Set semantics [1]. Our architecture can be outlined as follows. Web sources, i.e., Web sites posting *dynamic* data (news, webcasts, blogs etc.) are routinely consulted and relevant information is selected, downloaded and saved into XML *tags*. The gathering process is performed by two modules, *LiXto*[1]and *Dynamo*.

The *LiXto* Suite can *wrap* generic HTML pages by inspecting their internal HTML schema. *Dynamo* is a new system [2,3] that we are currently developing around an alternative concept. It can wrap data by discovering special meta-tags that have been (deliberately) embedded within the HTML code[2].

[1] *LiXto* by now is a released software by *LiXto* GmbH: *http://www.lixto.com/*

[2] The deployment of *Dynamo* requires collaboration from the Web source to be polled. However, in some cases a pre-processing phase from the *Dynamo* side can make up

The XML tags generated by *LiXto* and *Dynamo* are then translated into sets of Datalog-syntax [4] *facts;* such facts are then added to the (non-ground) logic programming rules that describe a *tag manipulation policy*. The resulting ASP program is then fed to the Lparse grounder [5] and to the smodels inferential engine; deduction can start. We have defined tag manipulation so as to obtain the following effects (often in combination):

1. some element of the tag, is dropped, e.g., because it is deemed incorrect, uninteresting or superseded by other tags[3]
2. new tags are added, to annotate the present data with extra informations about, e.g., the source, its reliability or some framework information that help in understanding/classifying the data.

The idea here is that the new tags added through inference will bring out some *semantical* consideration that would not be found by simply accessing the text of the Web source. As a result, the Web data will be transformed into a decorated XML version that mirrors the available data *as well as* the particular interpretation that has been applied. It should be noticed that there is no assumption on the shape of the original Web data, i.e., both the XML encapsulation and the subsequent transformation can be applied to arbitrary XML sources. Finally, the resulting XML tags are made available to Web services (WS) through standard channeling methods. In this article we restrict to considering RSS channeling.

1.1 The rôle of Logic Programming

It is important to notice that in the project presented here each relevant component of the proposed architecture, including the *LiXto* suite, is related to Logic Programming. Indeed, the project presented here is part of our long-term research effort (see, e.g., [6]) on *declarative policies* in the context of Web services.

Interestingly, even the wrapping of Web sources is done using the tools developed by the *LiXto* project, which in turn is based on *Elog*, an extension of DATALOG which can described as Positive logic programs + built-in regular expressions. A formal account of the logical interpretation of XML transformations is given by Gottlob and Koch in [7]. *Dynamo*, on the other hand, is a simple Java servlet; it does not need sophisticated pattern matching and manipulation: data polled from Web sources already come in a standard format (albeit an HTML one). To do so, a novel bijective mapping from XML tags to Datalog-syntax facts has been defined; details are in [8].

The automated reasoning and tag manipulation task is carried out in Answer Set Programming (ASP), which can be seen as an extension to Datalog, sometimes called Datalog^{n}ot, that deals with default reasoning. For lack of space,

for the lack of some or all meta-tags. Please refer to [2,3] for a discussion on this issue.

[3] In some cases, the whole tag could be dropped on the basis of consistency considerations. This is the case when tags are coming from Dynamo sources, who may not comply with the agreed data semantics.

we refer the reader to the pioneer works of Gelfond and Lifschitz [9] and to the survey in [1] for a detailed introduction to ASP.

Although our work does not address the Semantic Web as it is commonly understood, i.e., in terms of managing or publishing data annotated with a Semantic Web language such as RDF or OWL, our understanding, supported by some preliminary experiments, is that the inferential part can be used to apply *annotation policies* that transform Web data into RDF/OWL tags.

This article is structured as follows. Section 2 gives an overview of the *LiXto* project and explains the main features of the *LiXto* suite that were used in our project. In section 3 we describe the structure of *Dynamo* and its main features. Our architecture is introduced and explained in detail in Section 4. Section 5 describes the application example we have worked on to test and validate our blueprint. Finally, Section 6 summarizes the work done so far and discusses the current development lines.

2 The *LiXto* architecture

The *LiXto Suite* is a data extraction and transformation software kit for retrieving and converting information from regular documents, usually found on the World Wide Web. It is mainly composed of two applications:

1. the Visual Wrapper (VW), and
2. Transformation Server (TS).

that we are going to describe in more detail next. It should be remarked that the Transformation Server is indeed an application that treats arbitrary XML data and thus it is interesting in its own, i.e., for managing sources of native XML data.

2.1 The *LiXto* Visual Wrapper

The *Visual Wrapper* (VW) is a visual, interactive tool for generating *wrappers*. A wrapper is understood as a program that allows for automatic and flexible extraction of information from regular documents such as Web pages. Wrappers are designed to continually extract relevant information from dynamic Web pages and *organize* it into XML trees.

The VW is defined by Gottlob et al. [10]:

> The VW allows a user to create wrappers by visually selecting relevant patterns directly on browser-displayed pages. It allows for extraction of target patterns based on surrounding landmarks, on the contents itself, on HTML attributes, on the order of appearance and on ontological or syntactic concepts. Extraction is not limited to tokens of some document object model, but also possible from flat strings. The VW also allows for more advanced features such as disjunctive pattern definitions, following links to other pages during extraction and recursive wrapping.

Therefore, *LiXto* can implement data manipulation tasks that are beyond pattern recognition, i.e., are *data-driven* and need to adapt to the input. For further details on how the information extraction works and on its computational complexity, please refer to the presentations in [10,11].

2.2 The *LiXto* Transformation Server

The *Transformation Server* (TS) is a software that supports the design and execution of applications –called *pipes*– for processing XML data flows. The TS extracts data from Web sources and organizes them into XML trees, by mean of wrappers, designed with the Visual Wrapper described above.

Subsequently, *LiXto* TS allows the application designer to format, transform, merge and deliver XML data to various devices (e.g. HTML pages, XML pages, email, SMS). XML data manipulation is done by specialized and interacting modules (*components*) that the TS user creates, configures and connects (following a *pipeline* paradigm) in a completely visual environment.

There exist several specialized components e.g. those for wrapping, standardization, integration or delivery purposes. In particular, it is worth mentioning the so-called *Shell* component which allows for executing external programs on XML data. We have exploited the shell component as a gateway to the inferential engine described next.

3 *Dynamo*

Dynamo is a new, experimental architecture for automated data collection and XML delivery of data from traditional albeit dynamic HTML Web sites. The data of interest are routinely polled from the actual sources by standard HTTP querying. In order to be able to extract relevant informations from plain HTML documents Bossa [2] defined a set of annotations in form of meta-tags, which can be inserted inside an HTML document in order to give it semantic structure and highlight informational content.

The meta-tags are enclosed in HTML comment tags, so they remain transparent to Web browsers and do not alter the original HTML structure of the document. Once HTML documents are processed by *Dynamo*, raw data and annotations are extracted and organized into a simple XML format which is stored and used as a starting point for document querying and transformation. The first deployment of of *Dynamo*[4]extracts news from two Web sites, namely *theserverside.com* and *java.net*, and publishes RSS1 and RSS2 feeds which, as described, are produced on the fly starting from XML data.

Finally, we show an example of what *Dynamo*produces by showing an HTML fragment taken from *theserverside.com* after the process of insertion of meta-tags (lightly simplified due to the formatting guidelines):

[4] The application described here is the subject of [2] and is running on the site: *http://dynamo.dynalias.org/*. The *Dynamo*source code and more informations can be accessed thereof.

```
<!-- <channel:image  url="http://www.theserverside.com/[...]/feed-logo.jpg"
  title='The Enterprise Java Community[...]'   link="http://www.theserverside.com" /> -->
<!-- <channel:extension  uri="http://purl.org/dc/elements/1.1/"  prefix="dc" localName="language" >
 en-us
</channel:extension> -->
<!-- <channel:title> --> The Enterprise Java Community[...]
  <!-- <channel:link> -->  http://www.theserverside.com<!-- </channel:link> -->
<!-- </channel:title> -->
<!-- <channel:description>-->Enterprise Java Community is a developer community[...]
<!-- </channel:description> -->
<!-- <channel:extension uri="http://purl.org/dc/elements/1.1/" prefix="dc" localName="date"> -->
<!-- </channel:extension> -->
<td colspan="2">
<h1><!-- <item:title index="1"> -->wingS 2.0 web framework released<!-- </item:title> --></h1>
<div class="iteminfo">
Posted by:
<!-- <item:link index="1"> -->
  <a href="/user/userthreads.tss?user_id=194346" title="view Joseph's recent threads [...]">
<!-- </item:link> -->Joseph Ottinger</a>on
<!-- <item:extension index="1" uri="http://purl.org/dc/elements/1.1/"  prefix="dc" localName="date" >-->
    December 08, 2005 @ 08:25 AM
<!-- </item:extension> --></div>
<p>
<!-- <item:description index="1"> -->
The <a href="http://www.j-wings.org/" target="_blank">wingS project</a>
has just released version 2.0 of its framework with lots of major improvements.
<br><br>
wingS is a component based web framework resembling the Java Swing API with its MVC paradigm and [...]
<!-- </item:description index="1"> -->
<br><br>
Version 2.0 comes with a completely rewritten rendering subsystem focusing on optimal
stylability via CSS [...]. Various
<a href="http://www.j-wings.org/[...]/Demo"    target="_blank">demo applications
   are available on-line</a>.
wingS is released under the LGPL license.
</p>
```

and the same fragment converted to XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<resource url="http://dynamo.dynalias.org/tss.jsp" rssId="tss.xml" timestamp="1139592119233">
    <channel>
      <title>Enterprise Java Community: Your Enterprise Java Community</title>
      <link>http://dynamo.dynalias.org/tss.jsp</link>
      <description>Enterprise Java Community is a developer community, containing up-to-date
        news, discussions, patterns, resources, and media</description>
      <image>
        <title>TheServerSide.com</title>
        <url>http://www.theserverside.com/[...]/feed-logo.jpg</url>
        <link>http://www.theserverside.com</link>
      </image>
      <extensions>
        <dc:language xmlns:dc="http://purl.org/dc/elements/1.1/">
          en-us</dc:language>
      </extensions>
    </channel>
    <item id="1139592119233-1" index="1">
      <title>wingS 2.0 web framework released</title>
```

```
      <link>http://feeds.feedburner.com/techtarget/tsscom/home?m=476</link>
      <description>
        The wingS project has just released version 2.0 of its framework with lots of major
        improvements. [...]
      </description>
      <extensions>
        <dc:date xmlns:dc="http://purl.org/dc/elements/1.1/">
          Fri, 10 Feb 2006 09:58:49 EST </dc:date>
      </extensions>
    </item>
    [...]
</resource>
```

4 The proposed architecture

In this section we describe the core activity of our architecture: the inference-based manipulation of XML tags. The internal schema of our architecture is shown in Figure 1.

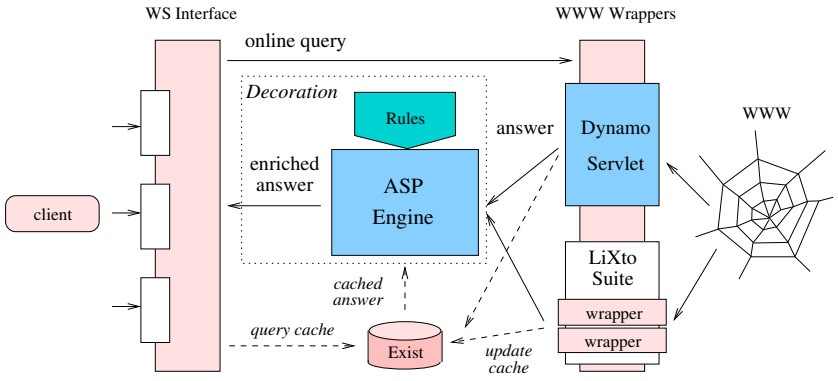


Fig. 1. The internal schema.

First, a simple Web Service collects data requests from its clients. The architecture serves each request by activating the *LiXto* WS. On the contrary, the *Dynamo* servlet is always active and performs routine polls. For the sake of performance, it is also possible to perform *off-line queries* by consulting the internal cache. Such cache is populated by *LiXto* and *Dynamo* throughout continuous scanning of the Web sites that are being monitored. The so-extracted data are then encoded as XML tags. Next, the *decoration* phase starts.

Decoration takes place in several steps. First, a Perl program called *xml2asp* [5] translates XML data into Datalog facts. Second, the obtained facts and the

[5] Both *xml2asp* and *asp2xml* are described in [8] and available from *http://mag.dsi.unimi.it/~carlo/#projects*

rules, i.e., the Datalog-syntax rules that describe tag manipulation, are fed to the ASP inferential engine. In our case, the ASP engine consists of the well-known *lparse* grounder and *smodels* solver [12]. The *smodels* output, i.e., the answer set, is then filtered to retrieve the relevant facts describing the decorated XML tag. Another Perl program, called *asp2xml* will then re-create and actual tag, which finally will be served to the clients by some more-or-less standard Web service. A detailed description of our architecture is depicted in Figure 2.

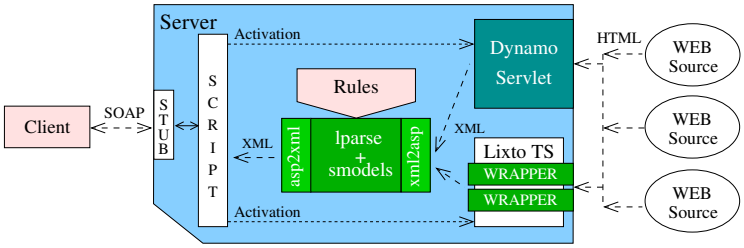


Fig. 2. Our architecture in more detail.

5 An Application example

We are implementing a service that allows a user to monitor hotels availability in big cities and on certain dates, e.g., during conferences. International travelers are accustomed to the *stars* rating system, where stars are proportional to the level of services and comforts available at the hotel. When the stars rating is absent, our service tentatively classifies the hotel facilities on the basis of the price range w.r.t. the city or even w.r.t. the particular location.

5.1 Wrapping an on-line hotel booking service

The Kelkoo portal[6]for e-commerce allows users, among other things, to search for available hotels, in a given resort and for a given period, by querying many (in this case, more than a dozen) hotel websites. Searching is based on the following parameters:

– location/resort;

[6] Kelkoo has developed several country-specific portals where customers can compare prices of competing e-commerce sites. In this Section we describe our work with the *kelkoo.co.uk* portal. However, the wrapper and the decorator could be almost effortlessly adapted to other Kelkoo national portals and, with some reprogramming, to other e-commerce sites.

– arrival date;
– departure date;
– room type;
– number of adults, and
– number of children.

The result page displays available hotels as an HTML table, one hotel per row. Using the *LiXto Visual Wrapper*, we implemented a wrapper to extract, for each hotel, the following information:

– name,
– address,
– brief description of the facilities,
– room type, and
– price.

and organize it into an XML tree. The following is a fragment of an XML output of the wrapper:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <rootPattern>
    <Hotel>
      <Name>HOTEL DE CHAMPAGNE</Name>
      <Address>
            Rue de Faubourg Saint Denis,
            Paris, Paris, 75010 France
      </Address>
      <Region>Paris</Region>
      <Room>
        <Description>Double</Description>
        <Price>34.56</Price>
        <Currency>GBP</Currency>
      </Room>
      <HotelDescription>
          Our hotel is located in the heart of Paris,
          two steps away from the Gare du Nord.
      </HotelDescription>
    </Hotel>
    ...
  </rootPattern>
</document>
```

5.2 Annotating Kelkoo data

The *LiXto transformation server* provides a visual tool for creating pipelines of activities. The pipe meant to "decorate" the information fetched by the wrapper for *kelkoo.co.uk* (see Section 5.1) is depicted in Figure 3 below.

The *Source* component runs the wrapper described in Section 5.1 on a kelkoo.co.uk result page, triggered by a proper querying URL whose parameters' values (see Section 5.1) are user-definable.

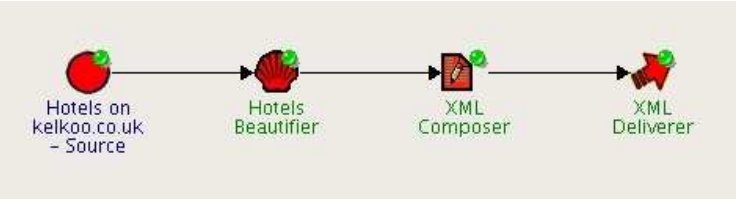


Fig. 3. A smart pipeline for on-line hotel booking on kelkoo.co.uk.

Next, the *Shell* component is configured to invoke (through the Operating System) the *xml2asp* translator, the inferential engine and finally the *asp2xml* back-translator. The ASP program used to annotate these data is in charge of adding the `<Stars>` tags which we infer from the hotel room fares. This is an example rule:

```
newNode(Hotel,"Stars",5):-
    tag(Hotel,"Hotel"),
    node(Hotel),
    tag(Room,"Room"),
    node(Room),
    parent_of(Hotel,Room),
    tag(Price,"Price"),
    node(Price),
    parent_of(Room,Price),
    tag(Currency,"Currency"),
    node(Currency),
    parent_of(Room,Currency),
    isNumber(Price,Qty),
    tag(Region,"Region"),
    node(Region),
    parent_of(Hotel,Region),
    threshold_price(Region,Reg_Currency,Reg_Thld),
    convert_local(Reg_Thld,Reg_Currency,Loc_Thld),
    convert_local(Price,Currency,Loc_Price),
    Loc_Thld < Loc_Price.
```

The auxiliary predicates needed to reason about the tag structure are defined as follows:

```
parent_of(X,Y) :- node(X),
                  node(Y),
                  firstchild(X,Y).

parent_of(X,Y) :- node(X),
                  node(Z),
                  firstchild(X,Z),
                  node(Y),
                  has_brother(Z,Y).
```

```
has_brother(X,Y) :- node(X),
                    node(Y),
                    nextsibling(X,Y).

has_brother(X,Y) :- node(X),
                    node(Y),
                    node(Z),
                    nextsibling(X,Z),
                    has_brother(Z,Y).
```

Applying the rules above leads to the derivation of some *newnode* atoms which will be included in the answer set returned by *smodels*. The shell component will take the answer set and pass it, modulo dropping some irrelevant atom, to the *asp2xml* back-translator. As a result, a the output tag will show an additional `<Stars>` tag, whose value, between 1 and 5, expresses the inferred class for a hotel, as in the following fragment:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <rootPattern>
    <Hotel>
      <Name>HOTEL DE CHAMPAGNE</Name>
      <Address>
            Rue de Faubourg Saint Denis,
            Paris, Paris, 75010 France
      </Address>
      <Region>Paris</Region>
      <Room>
        <Description>Double</Description>
        <Price>34.56</Price>
        <Currency>GBP</Currency>
      </Room>
      <HotelDescription>
            Our hotel is located in the heart of Paris,
            two steps away from the Gare du Nord.
      </HotelDescription>
      <Stars>3</Stars>
    </Hotel>

    ...

  </rootPattern>
</document>
```

After the decoration phase, it is possible, again through *Composer* module of *LiXto* TS to force some re-arrangement of the XML tag. In any case, the Composer module handles the resulting tag(s) over to the *LiXto Deliverer* component which takes care of forwarding them to the clients. In the example described above, we chose simply to save the resulting tags in a database. To facilitate further reuse, and beside to test the viability of the solution, we chose to employ the native XML database Exist [13] for the accumulation of the tags created

by our system. A much larger experimental tests, involving thousands of polling cycles a day, is currently under way to validate the approach in a realistic Web scenario.

For the sake of simplicity, this example shows only the simplest decoration activity, i.e., the adding of new tags. For the example above, the full inferential power (and therefore complexity) of Answer Set Programming is not needed. However, it is easy to imagine situations where a decision about whether to add, change or remove tags within a given, extracted tag will be based i) on lack of present data, which could be addressed by stratified negation as failure or ii) by reasoning by cases, which could be addressed by the generation of alternative answer sets.

6 Conclusions

We have described an architecture that allows analysis and manipulation of Web data based on default inferences with ASP, which is the core concept of our system. Indeed, the key functionalities of our application, i.e., data extraction from HTML pages and annotation of XML tags with new data, are defined in terms of ASP programs [1]. In particular, Eiter et al. [14] have discussed the application of ASP to reasoning about data from the Web.

Thanks to *LiXto* suite, the input data could be extracted –albeit with some human intervention– practically from any Web source. Also, thanks to *Dynamo*, old-fashioned HTML Web sources can participate to our data collection. Again thanks to *LiXto*, it remains easy to set up a Web service that supplies the result information over the Web. The core of the application, however, is the execution of sophisticated non-textual filtering operations, based on *inferences* about the source, the text itself or other issues. Our approach makes three kinds of advanced tag manipulation possible:

1. a tag, obtained from *LiXto* or *Dynamo*, is filtered, i.e. left out of the final result whenever it is deemed irrelevant;
2. a tag, again obtained from *LiXto* or *Dynamo*, gets annotated with extra tags, which would not be otherwise available by text analysis, however sophisticated and
3. filtering and decorations are carried out as a (default) reasoning activity, in the framework of Answer Set Programming.

Even though more test cases are needed for a careful assessment, we believe that the architecture proposed here can become a useful platform for the development of tools that act as bridges between the Web as we know it and more sophisticated forms of interactions. We believe that this is the case for the Semantic Web, since our system permits to program and execute the OWL (or RDF) marking up of data.

Hence, our approach could greatly simplify the process of *importing* Web data into the semantic Web. However, it should be stressed that in any case

the import would remain a semi-automatic activity, where human judgment will remain essential in two phases. Let us discuss them now.

The first phase consists in the selection of the Web source and the finding of the required data on the page (document). This phase is assisted and made easier by the *LiXto* visual wrapper, but seems unlikely to become fully automated. Note that in case of a *collaborative* Web source, the human intervention is even more important as meta-tags are to be inserted in the original Web HTML documents. The second phase of human intervention consists in the writing of the *beautifier rules*. The rules associated to a given Web source in effect represent the semantics of the source itself, and can embed its data inside the Semantic Web. In the follow up of this work we intend to join this research effort with that, reported in [6], aiming at a representation of default assumptions directly as a signature over RDF tags. In such a way, the RDF statements *produced* by our system would carry along an indication of the type of default assumptions that, supports them.

Acknowledgments

This project was started with contributions from S. Bossa and C. Bernardoni's graduation projects. Thanks to R. Baumgartner, G. Gottlob and M. Ornaghi. We are grateful to *LiXto* GmbH for granting us an academic license of *LiXto* suite.

References

1. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. The Logic Programming Paradigm: a 25-Year Perspective, Springer-Verlag (1999) 75–398
2. Bossa, S.: Polling of arbitrary web sources. Graduation project in Computer Science (in Italian), Univ. of Messina. (2005)
3. Bossa, S., Fiumara, G., Provetti, A.: A ligthweight architecture for rss polling of arbitrary web sources. Submitted for pubblication. Available from *http://mag.dsi.unimi.it/* (2006)
4. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Transactions on Database Systems (TODS) **22(3)** (1997) 364–418
5. Solvers: Web location of some of the most known asp solvers. (
   Aspps: *http://cs.engr.uky.edu/ai/aspps/*
   CMODELS: *http://www.cs.utexas.edu/users/tag/cmodels.html*
   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/*
   PSmodels: *http://www.tcs.hut.fi/Software/smodels/priority/*)
6. Bertino, E., Provetti, A., Salvetti, F.: Reasoning about rdf statements with default rules. In: Rule Languages for Interoperability, W3C (2005)
7. Gottlob, G., Koch, C.: Monadic datalog and the expressive power of languages for web information extraction. Journal of the ACM **51** (2004)

8. Bernardoni, C.: Beyond *LiXto*: Automated reasoning for the annotation of web data sources. Graduation project in Computer Science (in Italian), Univ. of Milan. (2005)
9. Lifschitz, V., Gelfond, M.: The stable model semantics for logic programming. Proc. of 5th ILPS conference (1988) 1070–1080
10. Gottlob, G., Baumgartner, R., Flesca, S.: Visual web information extraction with lixto. Proc. of VLDB (2001)
11. Gottlob, G., Koch, C., Baumgartner, R., Herzog, M., Flesca, S.: The lixto data extraction project - back and forth between theory and practice. In Deutsch, A., ed.: PODS, ACM (2004) 1–12
12. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: a system for answer set programming. Proc. of the 8th International Workshop on Non-Monotonic Reasoning (ILPS) (2000) 1070–1080
13. Meier, W.: exist: An open source native xml database. In Chaudhri, A.B., Jeckle, M., Rahm, E., Unland, R., eds.: Web, Web-Services, and Database Systems. Volume 2593 of Lecture Notes in Computer Science., Springer (2002) 169–183
14. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: A generic approach for knowledge-based information-site selection. In Fensel, D., Giunchiglia, F., McGuinness, D.L., Williams, M.A., eds.: KR, Morgan Kaufmann (2002) 459–469

Finding Instances of Deduction and Abduction in Clinical Experimental Transcripts

M. Amalfi, K. Lo Presti, A. Provetti[1], and F. Salvetti[2,3]

Dip. di Fisica, Università degli Studi di Messina.
Sal. Sperone 31. S. Agata di Messina, I-98166 Italy
http://informatica.unime.it/
ale@unime.it
Dept. of Computer Science, University of Colorado at Boulder.
430 UCB, Boulder, CO 80309, USA
Umbria Inc.
1655 Walnut St., Boulder CO 80302, USA
http://umbrialistens.com
franco.salvetti@umbrialistens.com

Abstract. This article describes the design and implementation of a prototype that analyzes and classifies transcripts of interviews collected during an experiment that involved lateral hemisphere-brain-damage patients. The patients' utterances are classified as instances of categorization, prediction and explanation (abduction) based on surface linguistic cues. The agreement between our automatic classifier and human annotators is measured. The agreement is statistically significant, thus showing that the classification can be performed in an automatic fashion.

1 Introduction

This article describes a software, with a Prolog-based automated reasoner at its core, that we have designed and implemented in the context of our work on measuring utterances that are evidence of inferential ability, viz., prediction and explanation, trying to match a human annotator. The starting point has been a clinical experiment that involved three groups: patients who have been diagnosed brain damage in the left hemisphere (henceforth left-patients), patients who have been diagnosed brain damage in the right hemisphere (henceforth right-patients) and a control group. The experiment was carried out at the Boston VA hospital by professional clinicians; the results were available in writing. The experiment consisted in showing to the subject the picture in Figure 1 (without the A..D frames on) and asking them

> "I have a picture here. As you see, there's a lot going on. Look it over and tell me about it."

Patients' answers were transcripted and analyzed to find evidence that the subject had verbalized some form of reasoning that allowed them to either *explain* certain details of the pictured situation or to *predict* what would happen immediately next. Finding evidence of mental inference in these documents

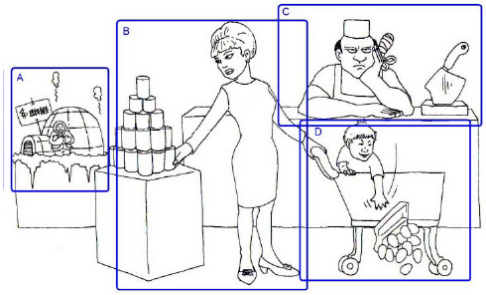


Fig. 1. Picture shown to the patients (with frames added)

is sometimes a challenging task. Also, opinions on what counts as a prediction/explanation may differ widely. However, the experiment was conceived in a way to circumscribe the domain of discourse to the few characters and objects present in the picture. Thus, automated text analysis could be carried out by relatively standard Artificial Intelligence techniques such as pattern matching and backward-chaining reasoning with a Prolog interpreter.

We have noticed that the sub-scenes, framed and labeled A to D in Figure 1 but not in the experiment, suggest more-or-less straightforward *predictions,* e.g., the pile of cans, (frame B) will fall down, and *explanations,* e.g., the boy (frame D) threw off the eggs. This experiment lends itself to automated annotation by text analysis.

To the best of our knowledge, this is the first implemented system that supports automated annotation over this type of clinical test.

A cognitive science interpretation of this experiment is found, among others, in [1]. Their interpretation can be summarized as follows. Left hemisphere-brain-damage patients exhibit verbal evidence of reasoning which, compared to those of a control group, suggests that while capable of performing reasoning, they have less problems in producing sentences with evidence of prediction than sentences with evidence of explanation. Vice versa, right-hemisphere brain damage patients exhibit verbal evidence of reasoning which, compared to those of a control group, suggests that even though abstract reasoning and the ability to make inferences may be impaired, they have fewer problems producing sentences with evidence of explanation than sentences with evidence of prediction.

In this article we are not concerned with the cognitive interpretation of the data, nor with the validation of a particular hypothesis relating lateral brain damage to specific types of reasoning impairment. Rather, we would like to validate our approach to automated text classification by showing, via statistical analysis, that the results are comparable with those of human annotators. Our architecture is designed to avoid commitment to any particular model of rationality but could serve as a tool for validating Cognitive Science theories.

Indeed, one could say that the relatively simple software architecture described here is effective for textual analysis only when simple sentences having

a limited lexicon are considered. However, the advantage of using an automated tool will be can evident when similar experiments will be administered to large populations and human annotation will become uneven or even impossible. Although our software, described in Section 3 is not suitable for large-scale activities *as is,* standard computational complexity analysis yields that our approach can indeed scale up to several hundreds transcripts.

1.1 Relating reasoning to speech

Abductive reasoning is a well-studied topic in knowledge representation and automated reasoning, e.g., in automated diagnosis. So do predictive reasoning and categorization. However, the characterization of abductive phrases as opposed to deduction or categorization phrases requires some preliminary agreement on what constitutes evidence of abduction. As will be explained in the next sections, all human annotators taking part in the validation of our system were given a standard set of rules on what should count as evidence of abduction. The definition of the *is_explan* and *is_predict* predicates in Section 3.3 can be taken as a case-by-case definition of the guidelines[1]. It is interesting to assess how the *abductive* reasoning considered here relates to the general use of term.

In systematic treatment of abductive reasoning, Magnani [2] introduces the term *manipulative abduction.* Manipulative abduction happens when we are thinking *through doing* and not only, in a pragmatic sense, *about doing.* In Magnani's classification an abduction is always related to one of these *conjectural template:*

1. curious and anomalous phenomena;
2. dynamical aspects;
3. artificial apparatus or
4. epistemic acting.

The latter conjectural template is the one where our examples fit more easily. Epistemic acting involves interesting features:

- simplification of the reasoning task;
- treatment of incomplete and inconsistent information;
- control of sense data;
- external artifactual models and
- natural objects and phenomena.

We appreciate that this experiment circumscribes the abduction activity in a simple and directed way. That is, abduction is reduced to *simplification of the reasoning task,* since other forms would not apply here. Nonetheless, we found that to have a clear distinction between abduction and the other two forms of reasoning a temporal dimension is still necessary, namely to separate

[1] The guidelines for panels of human annotators, however, are available from *http://mag.dsi.unimi.it/inference-finder/*

phrases about the past from phrases about the present or the immediate future. For instance, consider the phrase *butcher is angry,* which is found in several transcripts. Prima facie, it fits both the *curious and anomalous phenomena* and the *natural objects and phenomena* features. Yet, we can only suppose that the butcher is angry because he cut his finger.

1.2 The underlying Cognitive Model

It is fair to notice that from the point of view of cognitive science the results given by our automated annotator are qualified by a set of framework assumptions about what constitutes evidence of reasoning (deductive or not). The framework assumptions can be summarized as follows:

- it is possible to find evidence of cognitive process by analyzing utterances;
- in particular, it is possible to evaluate an individual's inferential ability through the analysis of his transcript, and
- the experiment has been effective in circumscribing the domain of discourse to a fixed set of reasoning instances that can be looked upon in the transcripts.

2 Related Research

Deductive reasoning is at the heart of logic-based knowledge representation and reasoning. Recently, it is becoming a topic of interest in Cognitive Science and even in Neuroscience. The reader may refer, for instance, to the survey in [3]. However, the most research efforts seem to use different techniques than the traditional logic-based AI methods. To the best of our knowledge, the work which is closest in spirit to our line of research is the set of experiments that Bucciarelli and her co-authors have designed and performed to verify the predictions of the well-known Johnson-Lairds Mental Model Theory (MMT) by [4].

According to MMT, humans make inferences by constructing mental models that are internal (mental) representations of some external state of affairs [5]. This theory postulates that reasoning depends on understanding the meaning of premises, and then using this meaning and general knowledge to construct mental models of the possibilities under description.

In a recent work [6], Bucciarelli and her collaborators studied the ability of comprehend logical connectives both as abstract verbal entities and inside a complex pragmatic context. To validate the MMT basic predictions that

- hardness of a certain mental task depends on the number of models that need to be considered, and
- reasoning about falsity is harder than reasoning about truth,

[6] describes an experiment where participants are given questions that involve evaluating the truth-value of non-atomic phrases.

Bucciarelli et al. results strongly suggest that the right hemisphere plays a great role in deductive reasoning. In particularly, right-patients seem at loss w.r.t. control patients when it comes to reasoning with problems, reporting them and paraphrasing them. However, such disadvantage is not uniform across the spectrum of tests. Indeed, although syllogisms over a unique model are harder for the right-patients than for the control group, syllogisms over multiple models proved hard for all the participants. That finding confirms the MMT forecast that problems with multiple models are difficult *tout court.*

An alternative explanation of Bucciarelli's results, which was put forward by the same authors, is that while the analogical component plays a great role in the reasoning process, the oral component is equally as important in reasoning with syllogisms and periphrasis; indeed on those aspect the the scores of right patients did not differ much from those of the control group.

3 The software architecture

Our objective is that of implementing a software for the recognition of categorizations, explanations and predictions that gives results as close as possible to those of human annotators. To do so, our first step has been to reduce categorizations, explanations and predictions to the occurrence of distinct <subject/verb/object> triples within a transcript [7]. The three types of inference are distinguished by the particular choice of verb tense. Categorization, explanation and prediction are defined as occurrence of key-words in the document. To this purpose, we have formulated a definition of the three concepts in terms of occurrence of words in the sentences.

Figure 1 shows the areas of interest of the picture. Each frame is associated to one or more possible inferences. The *categorization* has been considered as the presence of a word (or its synonym) that individualizes an object, a situation or a behavior represented in the picture. Thanks to this definition, the program can detect categorizations by searching and counting the co-occurrences of one or more key-words at sentence level.

Verbal acts of *explanation* and of *prediction* have been equated to the occurrence of a triple of words inside a sentence. The triples are intended as subject-verb-object (SVO), in this order, which are considered as sufficient evidence of a explanation/prediction act.

For instance, if $<$ *butcher, cut, finger* $>$ is found in a sentence the program should give a score $+1$ to the interviewed patient (and to his/her group) for having explained why the *butcher* has his finger bandaged. Similarly, finding *woman, demolish* and *cans* in a sentence will result in one point score given for prediction.

Clearly, what triple should be considered good evidence of prediction and/or explanation reasoning depends on the experiment, i.e., on the picture being shown to the subjects of the experiment. Hence, the triples to be searched for are part of the input to the program. It should be noticed at this point that the program is independent from the experiment: to adapt it to the annotation

of a set of transcripts pertaining a different experiment (same modality, different picture) one needs to change exactly three predicate definitions: *is_term*, *is_explan* and *is_predict*. The first definition can be obtained automatically as the output of a lexical analyzer whereas the latter two need a careful analysis of the experiment.

Let us now describe in detail the architecture and the data representation adopted in this project[2]. The system consists of two main components that have been designed and implemented for the experiment described above.

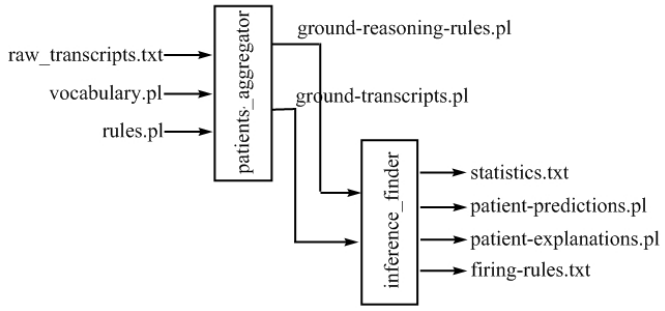


Fig. 2. Overview of the architecture

Their input-output description follows:

1. program *patients-aggregator* takes as input data:
   - the patients' transcript;
   - local vocabulary (since the experiment consider phrases related to the picture in Figure 1, the number of words of interest is finite and rather limited, i.e., 149 words as categorizations. So, it has been possible to represent all tokens of interest by means of Prolog facts) and
   - the general-purpose deduction rules for the Prolog inferential engine.

   It produces:
   - rules data about categorizations, explanations and predictions;
   - patients' data.

   Next, patients-aggregator scans the transcripts and creates a suitable Prolog representation of the phrases. Then, it also produces the schematic rules that the subsequent Prolog interpretation will use to discover the instances of prediction and explanation in the transcripts.
2. program *inference-finder* is written in Prolog; it takes as input the data generated by patients-aggregator and it produces:
   - statistics;
   - patient predictions and explanations and
   - firing rules [3].

[2] The software (source and binary codes), the results and documentation is available from our group page: *http://mag.dsi.unimi.it/inference-finder/*

[3] Prediction and explanation rules verified for each patient.

3.1 Representing interview transcripts

The first program, called *patients-aggregator,* takes as input the 3 files with the interviews and it finds out the individual words and delimits sentences[4].

Every transcript begins with "Patient #" followed by the patient's identification, the program will write the relative id, type and transcript into file *ground-transcripts.pl*.

For instance, the following is the transcript phrase *Patient #000001*

> The butcher / had a / thumb / the butcher / had a bandaid in his thumb / he cuts / his knife / his thumb / he cut / his thumb / with the knife / while cutting cheese / he looks angry / and upset / The boy / in the carriage / has dropped / the eggs / eggs / will fall / on floor / and breaks / break. / The woman / is pulling out / the can / from the bottom. / The can / will fall / in the floor. Frozen food / is on sale / The picture shows the igloo / and the eskimo. / The igloo / is / on sale / too. That's about it.

and patients-aggregator finds out the following rules:

patient(000001).
patient_type(000001, lh).
phrase([000001, [the, butcher, had, a, thumb, the, ..., and, breaks, break],
[the, woman, is, pulling, out, the, can, from, the, bottom],
[the, can, will, fall, in, the, floor],
[frozen, food, is, on, sale, the, picture, shows, the, igloo, and, the, eskimo],
[the, igloo, is, on, sale, too],
[that, s, about, it]]).

Patients-aggregator will write also the file *ground-reasoning-rules.pl* (a file witch contains schematic rules with variables witch describe the reasoning rules).

3.2 Format of the output

For each patient, our Prolog analyzer produces as output:

1. Patient's identifier,
2. Patient's proposed classification (*lh, rh* or *control)*
3. words count,
4. categorizations count,
5. explanations count, and
6. predictions count.

[4] Special characters e.g., "," "/," "-" and "(" mark the end of words while the usual "." "?," and "!" mark the end of sentences. This simple way for tokenization and segmentation is here considered sufficient for the purposes of this research.

3.3 Representing rules

The following rules were used to describe possible categorizations for the example at hand.

```
is_term(lady).
is_term(butcher).
is_term(can).
...
```

This is an example of an explanation rule we used:

```
is_explan( 'butcher cuts finger',
           [butcher, he, man, boy, mister, kid, guy],
           [cut, sore, wrapped, sliced, cuts, have, ...],
           [finger, fingers, knife, bandage, ...]).
```

whereas this is an example of a prediction rule:

```
is_predict( 'woman demolishes cans',
            [she, lady, mother, woman],
            [pull, pulls, get, demolish, pulling, ...],
            [can, bottom, display, cans, stack, ...]).
```

By means of lists, we have extended the triples so as to provide a moderate tolerance to stemming, i.e, in the examples *cuts* is also accepted. These extensions, however, have to be discussed on a one-by-one basis to account for verb tenses, which can be very strong indicators in differentiating abduction from prediction. The following Prolog predicates are for parsing the interviews, applying the recognition rules and counting the *hits*.

Categorizations:

```
category1( _, [ ], 0).
category1( IdPatient, [Term|TermList], N):-
           phrase([IdPatient|Interview]),
           count_occurrences(Term, Interview, N2),
           category1(IdPatient, TermList, N1),
           sum(N1, N2, N).
```

Explanations:

Ph = phrase where we search in the order subject-verb-complement
NameExp = name of explanation
SubL = subjects list
VerbL = verbs list
CompL = complements list
explain(Ph, SubL, VerbL, CompL) = search in the order subject-verb-complement

```
explain( Ph, NameExp):-
         is_explan(NameExp, SubL, VerbL, CompL),
         search_sub(Ph, SubL, VerbL, CompL);
         fail.
```

Predictions:

Ph = phrase where we search in the order subject-verb-complement
NamePre = name of prediction
SubL = subjects list
VerbL = verbs list
CompL = complements list
predict(Ph, SubL, VerbL, CompL) = search in the order subject-verb-complement

```
predict( Ph, NamePre):-
        is_predict(NamePre, SubL, VerbL, CompL),
        search_sub(Ph, SubL, VerbL, CompL);
        fail.
```

Ph = one or more phrases over which subject is searched
[SubLH|SubLT] = subjects list
VerbL = verbs list
CompL = complements list
phrase_remainder = search subject into phrase
search_verb(Ph_Remain, VerbL, CompL) = search verb into ph. remainder after subject is found
search_sub(Ph, SubLT, VerbL, CompL) = calls itself with SubLT

```
search_sub( _, [ ], _, _):-
            fail.
search_sub( Ph, [SubLH|SubLT], VerbL, CompL):-
            phrase_remainder(SubLH, Ph, Ph_Remain),
            search_verb(Ph_Remain, VerbL, CompL);
            search_sub(Ph, SubLT, VerbL, CompL).
```

List_H = element that we search (Subject/Verb/Complement)
[Ph_H|Ph_T] = phrase over which we search element List_H
Ph_Tail = phrase remainder after element List_H

```
phrase_remainder( _, [ ], _):-
                  fail.
phrase_remainder( List_H, [Ph_H|Ph_T], Ph_Tail):-
                  List_H==Ph_H, Ph_Tail=Ph_T;
                  phrase_remainder(List_H, Ph_T, Ph_Tail).
```

Let us now see in detail how the classifier works vis-a-vis the results of a human panel of evaluators.

4 Results and comparisons

All the interviews considered in this work were annotated by two independent panels of human annotators, here called B (for Boulder) and M (for Messina). Each panel was made of two graduate students of Computer Science, who received similar instructions and very precise instructions on how to annotate interviews.

4.1 Annotating individual phrases

Table 1 summarizes the number of instances of inferential reasoning that were found for this interview.

Results		
-	expl.	pred.
B panel	3	4
M panel	3	1
Program	1	2

Table 1. Annotations on patient #000001 transcript

Let us now see some actual instance. To wit, the M panel found these instances of explanation:

1. *he cuts / his knife / his thumb,*
2. *he looks angry* and
3. *the boy / in the carriage / has dropped / the eggs / eggs / will fall / on floor*

The classifier has been able to find only the first instance of explanation with the *'butcher cuts finger'* rule, which was matched against the following phrase:

> [the, butcher, had, a, thumb, the, butcher, had, a, bandaid, in, his, thumb, he, cuts, his, knife, his, thumb, ...]

Considering prediction, The M panel found only this instance:

1. *the woman / is pulling out / the can / from the bottom. / The can / will fall / in the floor.*

whereas the classifier found two instances of explanations:

1. *woman demolish cans,* and
2. *pile crashing down*

In other words, the classifier split the phrase above into two.

4.2 Annotations by the B panel

In the same interview, the B panel found the following instances of explanation:

1. *he cut / his thumb / with the knife*
2. *while cutting cheese*
3. *has dropped / the eggs*

The B panel and the M found the same number of explanations and they are in agreement on only two explanations, the 1st and the 3rd.
In refer to the predictions, the B panel found four instances of them:

1. *eggs / will fall / on floor*
2. *and breaks / break*
3. *The woman / is pulling out / the can / from the bottom*
4. *The can / will fall / in the floor*

The first two were not found by the M panel, while the last two are indicated by the M panel as the same prediction.

5 Validation of the results

The overall number of annotations obtained during our experimental annotation is illustrated in Table 2.

Instances Found		
-	expl.	pred.
B panel	99	71
M panel	87	35
Program	71	32

Table 2. Overall no. of instances found

In the following we describe a statistical analysis of the annotations that supports a more sophisticated understanding of the results.

5.1 The Kappa index

The Kappa index [8], introduced by Cohen [9], has been proposed as a measure of the specific agreement for category among two observers. Kappa measures the accord among the answers of two observers (or the same observer in different moments), that appraises couples of objects or diagnostic categories.

This index captures and corrects the so-called *accidentals agreements.* An agreement is called accidental when two observers reach the same conclusion even though they did so by employing completely different sets of criteria to distinguish between the presence/absence of relevant conditions. In such cases the *raw* agreement index would not reflect a real agreement. The idea underlying the Kappa index is that the actual accord between two observers is as the difference between the raw agreement and the agreement we would have under the hypothesis that between the two there is no accord and thus their answers may coincide only by chance.

To define K formally, first we introduce the following notation. For any two possible classifications i j, let p_{ij} by the proportion of cases that where classified as i by the first observer and j by the second. Clearly, $p_i i$ is the proportions of cases that have been agreed upon to be of type i. Also, let $p_{i.}$ (resp. $p_{.i}$) be the proportion of cases that the first (the second) observer classified as i (sometimes called marginal frequencies). Now, let

P_o be the proportion of frequencies observed of accords among the two evaluators, and

P_e it is the proportion of accords expected under the *void hypothesis:* accord is determined by the product of the marginal frequencies $p_{i.}$ and $p_{.i}$.

The value of K is given by the ratio between the excess agreement $(P_o - P_e)$ and the maximum obtainable agreement $(1 - P_e)$:

$$K = \frac{P_o - P_e}{1 - P_e} \tag{1}$$

For ordinal variables, which are the case with our scores, the weighted-kappa index, is defined. Weighted-Kappa assigns less weight to agreements as categories are further apart.

Let m be the overall number of categories and let w_{ij} be the weight assigned to the [dis]agreement of the ith and jth categories, with $w_{ii} = 1$, $w_{ij} \geq 0$ $(\forall i \neq j)$ and $w_{ij} = w_{ji}$. Then

$$w_{ij} = 1 - \frac{(i-j)^2}{(m-1)^2} \tag{2}$$

$$P_o^w = \sum_{i=1}^{m} \sum_{j=1}^{m} w_{ij} p_{ij} \tag{3}$$

$$P_e^w = \sum_{i=1}^{m} \sum_{j=1}^{m} w_{ij} p_{i.} p_{j.} \tag{4}$$

$$K^w = \frac{P_o^w - P_e^w}{1 - P_e^w}. \tag{5}$$

If there is a complete agreement, then K (resp. K^w) will be equal to 1. If the observed agreement is greater than or equal to the agreement attended only by chance obtained then the K index will result near zero or even slightly negative. Values of K above 0.6 suggest that there is a substantial agreement; values below 0.21 indicate a weak agreement. Table 3 from [10] shows an interpretation of the values.

To sum it up, K^w is the right type of index to assess the quality of our program vis-à-vis human analysis of some experimental results. Measuring the degree of agreement among two rules and the program, we have the following results:
Between the M panel and the system a substantial agreement is found; whereas group B has only fair agreement both with group M and with the program.

Kappa	Strength of Agreement
< 0.00	Poor
0.00–0.20	Slight
0.21–0.40	Fair
0.41–0.60	Moderate
0.61–0.80	Substantial
0.81–1.00	Almost Perfect

Table 3. The K benchmarks

Explanations			
-	B	M	Program
B panel	1	0,497	0,386
M panel	-	1	0,635
Program	-	-	1

Table 4. The K-weighted degree of agreement on explanations

Predictions			
-	B	M	Program
B panel	1	0,336	0,363
M panel	-	1	0,531
Program	-	-	1

Table 5. The K-weighted degree of agreement on predictions

5.2 Interpretation of the results

The statistics described above show a good agreement between the program scores and those given by the M panel. Vice versa, the B panel results have a relatively low agreement K^w with both the program and the M panel. The B panel consistently finds more instances of reasoning (of any type) than the M panel and the system. These differences can be explained by the fact that the mental model of the M panel annotators is reflected in the program. These results are very satisfying from an Artificial Intelligence perspective: they show that, e.g., from the point of view of the B panel, the classification given by the M panel and that given by the system are hardly distinguishable.

6 Conclusions

We have described the design and implementation of a prototype that analyzes and classifies transcripts of interviews collected during a cognitive science experiment that concerned assessing reasoning bias in lateral-brain damage patients. Our Prolog-based software takes a static description of reasoning rules and matches them on patients' transcripts. Hence, patients' utterances were classified as instances of categorization, prediction and explanation (abduction) based on surface linguistic cues. The agreement between our automatic classifier and human annotators is measured. The agreement is statistically significant, w.r.t. the inherent limitations of the experiment thus showing that the classification can be performed in an automatic fashion. The statistical results support our claim that our software can be safely applied to automate the analysis of experimental results of the type described earlier. Our program can be useful as a provider of second opinions to reveal possible overlooks or mistakes in the diagnostic analysis.

From a Cognitive science point of view, our project may be considered limited by the fact that is can analyze only verbal (transcripted) responses to experiments. Vice versa, from an A.I. point of view the pattern matching mechanism, though rather basic vis-à-vis current natural language processing techniques is implemented fairly elegantly and efficiently in Prolog.

We are currently working to incorporate such techniques, (e.g., regular expressions) into our program. It would be interesting to apply our classifier to the transcripts of the experiment in [6] since their experiment seems within the reach of the techniques we have employed. Another promising direction of research consist in attaching to the token words some *semantics* obtained by automated reference to Wordnet[5].

acknowledgments

Thanks to Enea Zaffanella for carefully reviewing our submitted version and suggesting several improvements. Thanks to Monica Bucciarelli for suggestions on how to extend this work.

[5] http://wordnet.princeton.edu

References

1. Salvetti, F.: Reasoning and brain damage: prediction and explanation in left- and right-hemisphere patients. Proc. of the XXVth Conf. of the Cognitive Science Society (CogSci) (2006)
2. Magnani, L.: An abductive theory of scientific reasoning. Proc. of the int. workshop on computational models of scientific reasoning and applications (2002)
3. Goel, V.: Cognitive Neuroscience of Deductive Reasoning. Cambridge Univ. Press (2004)
4. Johnson-Laird, P.N.: Mental models. Cambridge University Press, Cambridge, UK (1983)
5. Johnson-Laird, P.N.: Human reasoning and rationality. Int'l Sym. on Fondation and the Ontological Quest: Prospects for the New Millennium (2001)
6. Sacco, K., Bucciarelli, M., Adenzato, M.: Mental models and the meaning of connectives: A study on children,adolescents and adults. Proc. of the XXIIIth Conf. of the Cognitive Science Society (2001) 875–880
7. LoPresti, K.: An automated Classifier for recognizing inferential capacity biases. Graduation Project, Univ. of Messina (2005)
8. Carletta, J.: Assessing agreement on classification tasks: the kappa statistic. Computational Linguistics **22(2)** (1996) 249–254
9. Cohen, J.: A coefficient of agreement for nominal scales. Educational and Psychological Measurement **20(1)** (1960) 37–46
10. Soliani, L.: Statistics Manual for Research and Profession (In Italian). (2005)

An Event-Condition-Action Logic Programming Language

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

[1] CENTRIA, Universidade Nova de Lisboa, Portugal,
jja|banti@di.fct.unl.pt
[2] Dipartimento di Informatica, Università di Pisa, Italy,
brogi@di.unipi.it

Abstract. Event-Condition-Action (ECA) languages are an intuitive and powerful paradigm for programming reactive systems. Usually, important features for an ECA language are reactive and reasoning capabilities, the possibilities to express complex actions and events and a declarative semantics. In this paper, we introduce ERA, an ECA language based on the framework of logic programs updates that, together with these features, also exhibits capabilities to integrate external updates and perform self updates to its knowledge (data and classical rules) and behaviour (reactive rules).

1 Introduction

Event Condition Action (ECA) languages are an intuitive and powerful paradigm for programming reactive systems. The fundamental construct of ECA languages are *reactive rules* of the form:

$$\textbf{On } Event \ \ \textbf{If } Condition \textbf{ Do } Action \tag{1}$$

which mean: when $Event$ occurs, if $Condition$ is verified, then execute $Action$. ECA systems receive inputs (mainly in the form of *events*) from the external environment and react by performing actions that change the stored information (internal actions) or influence the environment itself (external actions). There are many potential and existing areas of applications for ECA languages such as active and distributed database systems [33, 10], Semantic Web applications [2, 3, 28], distributed systems [19], Real-Time Enterprize and Business Activity Management and agents [16].

To be useful in a wide spectrum of applications an ECA language has to satisfy several properties. First of all, events occurring in a reactive rule can be complex, resulting by the occurrence of several basic ones. A general and widely used way for defining complex events is to rely on some event algebras [15, 4], i.e. to introduce operators that define complex events as the result of compositions of more basic events that occur at the same or at different instants. Also the actions that are triggered by reactive rules might be complex operations involving several (basic) actions that have to be performed concurrently or in a given order and under certain conditions. The possibility to define events and actions in a compositional way (meaning in terms of sub-events and sub-actions that have been already defined), would permit a simpler and more elegant

programming style by breaking complex definitions into simpler ones and by making it possible to use the definition of the same entity in different fragments of code.

An ECA language would also benefit from a declarative semantics that would valorize the simplicity of the basic concepts of the ECA paradigm. Moreover, an ECA language must in general be coupled with a knowledge base, which, in our opinion, should be richer than a simple set of facts, and allow for the specification of data and classical rules, i.e. rules that specify knowledge about the environment, besides the ECA rules that specify reactions to events. Together with the richer knowledge base, an ECA language should exhibit inference capabilities in order to extract knowledge from such data and rules.

Clearly ECA languages deal with systems that evolve. However, in existing ECA languages this evolution is mostly limited to the evolution of the (extensional) knowledge base. But in a truly evolving system, that is able to adapt to changes in the considered domain, there can be evolution of more than the extensional knowledge base: derivation rules of the knowledge base (intensional knowledge), as well as the reactive rules themselves may change over time. We believe another capability that should be considered is that of *evolving* in this broader sense. Here, by evolving capability we mean that a program should be able to automatically integrate external updates and to autonomously perform self updates. The language should allow to update both the knowledge (data and classical rules) and the behaviour (reactive rules) of the considered ECA program due to external and internal changes.

To the best of our knowledge, no existing ECA language provides all the above mentioned features, in particular, none provides the evolving capability (for a detailed discussion see section 6). The purpose of this paper is to define an ECA language based on logic programming that satisfies all these features. Logic programming (LP) is a flexible and widely studied paradigm for knowledge representation and reasoning based on rules. In the last years, in the area of LP, an amount of efforts has been developed to provide a meaning to updates of logic programs by other logic programs. The output of this research are frameworks that provide meaning to sequence of logic programs, also called Dynamic Logic Programs (DyLPs) [5, 7, 9, 14, 17, 24, 29, 34, 32], where the initial program is seen as the initial knowledge base and the subsequent programs are the various updates (see Section 2). DyLPs and the update languages defined on top of them [6, 18, 23, 8] conjugate a declarative semantics and reasoning capabilities with the possibility to specify (self) evolutions of the program. However, unlike ECA paradigms, these languages do not provide mechanisms for specifying the execution of external actions nor they provide mechanism for specifying complex events or actions.

To overcome the limitations of both ECA and LP update languages, we elaborate an ECA language defined starting from DyLPs called ERA (after Evolving Reactive Algebraic programs) incorporating complex events, external and complex actions with the inference and evolving capabilities of updates languages.

Besides *reactive rules* of the form (1), ERA also allows LP rules supporting default negation [26] and the possibility to express negation in the head of rules [25]. Regarding reactive rules, $Event$ is a basic or complex event expressed by an algebra similar to the Snoop algebra [4]. Basic events are passed to a program as external inputs. These inputs are represented by sets of data and rules called *input programs*. The $Condition$ part in

ERA is a set of literals. Like events, actions can be basic or complex. Basic actions can be external (modify the environment) or internal (add or retract data, *rules*, and *reactive rules*). Complex actions are obtained by applying algebraic operators on basic actions that specifies whether two actions have to be executed concurrently or sequentially, or that, if some condition hold an action is executed, otherwise another action is executed instead. ERA also allows *inhibition rules* of the form:

$$\textbf{When } B \textbf{ Do } not\ Action \tag{2}$$

that intuitively mean: when C is satisfied, do not execute $Action$. Inhibition rules are mainly used to update the behaviour of reactive rules. If the inhibition rules above is asserted (either by an external or a self update) all the reactive rules with $Action$ in the head are updated with the extra condition that C must *not* be satisfied in order to execute $Action$.

A semantics for ERA is defined by means of an *inference system* (that specifies what conclusions are derived by a program) and of an *operational semantics* (that specifies the effects of actions). The former is derived from the refined semantics for DyLPs [5]. The latter is defined by a transition system inspired by existing work on process algebras. [27, 22, 30].

The rest of the paper is structured as follows: In section 2 we briefly introduce the syntax and semantics of DyLPs, and establish general notation. We start section 3 by informally illustrating an example of an ECA system, and then we present the syntax of ERA programs formally elaborating (a part of) the example. Section 4 is dedicated to the definition of the semantics of ERA. Section 5 completes the formal elaboration of the example presented in section 3. In section 6 we discuss related works, confront them with ERA and, finally, we draw conclusions and sketch future work.

2 Background and Notation

In what follows, we use the standard LP notation and, for the knowledge base, *generalized logic programs* (GLP) [25]. Arguments of predicates (here also called atoms) are enclosed within parenthesis and separated by commas. Names of arguments with capitalized initials stand for variables, names with uncapitalized initials stand for constants.

A GLP over an alphabet (a set of propositional atoms) $\mathcal{L}$ is a set of rules of the form $L \leftarrow B$, where L (called the head of the rule) is a literal over $\mathcal{L}$, and B (called the body of the rule) is a set of literals over $\mathcal{L}$. As usual, a literal over $\mathcal{L}$ is either an atom A of $\mathcal{L}$ or the negation of an atom $not\ A$. In the sequel we also use the symbol not to denote complementary default literals, i.e. if $L = not\ A$, by $not\ L$ we denote the atom A.

A (two-valued) *interpretation* I over $\mathcal{L}$ is any set of literals in $\mathcal{L}$ such that, for each atom A, either $A \in I$ or $not\ A \in I$. A set of literals S is true in an interpretation I (or that I satisfies S) iff $S \subseteq I$. In this paper we will use programs containing variables. As usual in these cases a program with variables stands for the propositional program obtained as the set of all possible ground instantiations of the rules. Two rules τ and η are *conflicting* (denoted by $\tau \bowtie \eta$) iff the head of τ is the atom A and the head of η is $not\ A$, or viceversa.

A Dynamic Logic Program $\mathcal{P}$ over an alphabet $\mathcal{L}$ is a sequence $P_1, \ldots, P_m$ where the P_is are GLPs defined over $\mathcal{L}$. Given a DyLP $P_1 \ldots P_n$ and a set of rules R we denote by $\mathcal{P} \setminus R$ the sequence $P_1 \setminus R, \ldots, P_n \setminus R$ where $P_i \setminus R$ is the program obtained by removing all the rules in R from P_i. The *refined stable model semantics* of a DyLP, defined in [5], assigns to each sequence $\mathcal{P}$ a set of refined models (that is proven there to coincide with the set of stable models when the sequence is formed by a single normal [21] or generalized program [25]). The rationale for the definition of a refined model M of a DyLP is made according with the *causal rejection principle* [17, 23]: If the body of a rule in a given update is true in M, then that rule rejects all rules in previous updates that are conflicting with it. Such rejected rules are ignored in the computation of the stable model. In the refined semantics for DyLPs a rule may also reject conflicting rules that belong to the same update. Formally the set of rejected rules of a DyLP $\mathcal{P}$ given an interpretation M is:

$$Rej^S(\mathcal{P}, M) = \{\tau \in P_i : \exists\, \eta \in P_j\ i \leq j,\ \tau \bowtie \eta\ \wedge\ B(\eta) \subseteq M\}$$

An atom A is false by default if there is no rule, in none of the programs in the DyLP, with head A and a true body in the interpretation M. Formally:

$$Default(\mathcal{P}, M) = \{not\ A\ :\ \nexists\ A \leftarrow B \in \bigcup P_i\ \ \wedge B \subseteq M\}$$

If $\mathcal{P}$ is clear from the context, we omit it as first argument of the above functions.

Definition 1. *Let $\mathcal{P}$ be a DyLP over the alphabet $\mathcal{L}$ and M an interpretation. M is a refined stable model of $\mathcal{P}$ iff*

$$M = least\left(\left(\bigcup P_i \setminus Rej^S(M)\right) \cup Default(M)\right)$$

where $least(P)$ denotes the least Herbrand model of the definite program [26] obtained by considering each negative literal $not\ A$ in P as a new atom.

In the following, a conclusion over an alphabet $\mathcal{L}$ is any set of literals over $\mathcal{L}$. An inference relation $\vdash$ is a relation between a DyLP and a conclusion. Given a DyLP $\mathcal{P}$ with a unique refined model M and a conclusion B, it is natural to define an inference relation $\vdash$ as follows: $P_S \vdash B \ \Leftrightarrow B \subseteq M$ (B is derived iff B is a subset of the unique refined model). However, in the general case of programs with several refined models, there could be several reasonable ways to define such a relation. A possible choice is to derive a conclusion B iff B is a subset of the intersection of all the refined models of the considered program ie, $P_S \vdash B \ \Leftrightarrow\ B \subseteq M\ \forall\ M \in \mathcal{M}(\mathcal{P})$ where $\mathcal{M}(\mathcal{P})$ is the set of all refined models of $\mathcal{P}$. This choice is called *cautious reasoning*. Another possibility is to select one model M (by a selecting function Se) and to derive all the conclusions that are subsets of that model ie, $\mathcal{P} \vdash B \ \Leftrightarrow\ B \subseteq Se(\mathcal{M}(\mathcal{P}))$. This choice is called *brave reasoning*. In the following, in the context of DyLPs, whenever an inference relation $\vdash$ is mentioned , we assume that $\vdash$ is one of the relations defined above.

Let E_S be a sequence of programs (ie, a DyLP) and E_i a GLP, by $E_i.E_S$ we denote the sequence with head E_i and tail E_S. If E_S has length n, by $E_S..E_{n+1}$ we denote the sequence whose first n^{th} elements are those of E_S and whose $(n+1)^{th}$ element

is E_{n+1}. For simplicity, we use the notation $E_i.E_{i+1}.E_S$ and $E_S..E_i..E_{i+1}$ in place of $E_i.(E_{i+1}.E_S)$ and $(E_S..E_i)..E_{i+1}$ whenever this creates no confusion. Symbol $null$ denotes the empty sequence. Let E_S be a sequence of n GLPs and $i \leq n$ a natural number, by E_S^i we denote the sequence of the first i^{th} elements of E_S. Let $\mathcal{P} = \mathcal{P}'..P_i$ be a DyLP and E_i a GLP, by $\mathcal{P} \uplus E_i$ we denote the DyLP $\mathcal{P}'..(P_i \cup E_i)$.

3 Syntax of ERA programs

Before the definition of the syntax of ERA programs, we present a motivating examples that, besides illustrating several features of the languages that are not present in other ECA languages (such as the ones pointed out in the introduction), helps on informally introducing the syntax.

Example 1. Consider an (ECA) system for managing several electronic devices of a building, particulary the phone lines and the fire security system. The system receives inputs such as signals of sensors and messages from employees and system administrators, and can activate devises like electric doors or fireplugs, redirect phone calls and send emails. For instance, sensors alert the system whenever an abnormal quantity of smoke is found. If a (basic) event ($alE(S)$) corresponding to a warning from a sensor S occurs, the system opens all the fireplugs Pl in the floor where S is located. This behaviour is encoded by a reactive rule reacting to a basic event with a basic action $openA(Pl)$. The situation is different when the signals are given by several sensors. If two signals from sensors located in different rooms occur without a $stop\_alertE$ event occurring in the meanwhile, the system starts a more sophisticated action $fire\_alarmA$ that applies a security protocol. All the doors are unlocked (by the action $opendoorsA$) to allow people to leave the building. At the same time, the system sends a registered phone call to a firemen station (by the action $firecallA$). Then the system cuts the electricity in the building (the fireplugs work with an independent electric system). To open the doors and call the firemen station are actions that can be executed simultaneously, but the last one (cutting the electricity) has to be executed after the electric doors have been opened (otherwise they would remain closed). To implement such a behaviour we clearly need an ECA language capable to define complex events ($alert2E$) and actions $fire\_alarmA$ starting from basic ones.

Knowledge representation and reasoning also plays an important role. Suppose, for instance, the system receives an event notifying that a meeting of a big working group has been organized. The system must advise by email all the employee in the working group. The big working group is formed by subgroups possibly themselves divided into other subgroups. The system has stored data on which subgroup an employee belongs to; it must be able to represent that, if an employee belongs to a subgroup, he also belongs to its supergroups and so to infer which are the ones in the big working group.

Finally, let us consider evolution. After some false alarms, the administrators decide to update the behaviour of the system. From then onwards, when a sensor rises an alarm, only the fireplugs in the room where the signal is located will be open. Moreover, an employee can communicate to the system to start redirecting phone calls to him and to stop redirection by turning back to the previous behaviour (whatever it was). Such changes could be done by handily modifying the system. ERA, as we shall see,

offers the possibility to update reactive rules instead of rewriting. This second approach could be very useful in large systems possibly developed and modified by several programmers and administrators. In particular, it could be very useful when updates are performed by users that are not aware of the existing rules governing the system (see example in section 5).

Expressions in ERA are formed by atoms with a LP-like syntax (cf. section 2 for details). In the sequel, we use names of atoms ending by E to represent events, and ending by A to represent actions. E.g. atom $openA(D)$ is the action of opening a device D, $openE(D)$ is the basic event occurring whenever D is opened, while $open(D)$ is the internal representation of the fact that D is open.

Expressions in an ERA program are divided in *rules* (themselves divided into *active, inference and inhibition rules*,) and *definitions* (themselves divided into *event* and *action definitions*). We already seen in section 1 the form of reactive rules. Atoms in the condition part of a reactive rule are separated by commas. When the $Condition$ part of a reactive rule is the empty set of literals, to simplify notation we omit the operator **if**. For instance, the reactive rule

$$\textbf{On } alE(S) \ \textbf{If } flr(S, Fl), firepl(Pl), flr(Pl, Fl) \ \textbf{Do } openA(Pl). \qquad (3)$$

stands for "on (event) alarm $alE(S)$, if sensor S and fireplug Pl are at the same floor Fl, then open Pl".

To allow for representing and reasoning about knowledge, ERA uses *inference rules* with a LP form. For instance, the rule:

$$ingroup(Emp, G) \leftarrow ingroup(Emp, S), sub(S, G).$$

specifies that an employee Emp belongs to the working group G if he belongs to S which is a subgroup of G and that every working group is a subgroup of itself. Together with facts of the form $subgroup(s, g)$ and $ingroup(emp, g)$, these rules allow to infer which employees belong to a given working group[1].

The reactive rule (3) executes an external action. While external actions are related to the specific application of the language, internal actions always have one of the following forms: $rise(e), assert(\tau), retract(\tau), define(d)$. Action $rise(e_b)$, where e_b is a basic event, means "e_b occurs in the next input programs", while the remaining internal actions actually modify the program. Actions $assert(\tau)$ and $retract(\tau)$, where τ is (any kind of) rule, mean, respectively, "update the current program with τ" and "delete τ from the current program" (see below for the discussion of action $define(d)$). For instance, the following reactive rules update the program each time a device D is opened or closed.

$$\textbf{On } openE(D) \ \textbf{Do } assert(open(D)). \ \textbf{On } closeE(D) \ \textbf{Do } assert(not\ open(D)).$$

Sometimes we need to combine basic events to obtain complex ones. This is done by an event algebra whose operators are: $\triangle \mid \ \bigtriangledown \ \mid A \mid not$. Given the events e_1, e_2, e_3,

[1] The rules above uses recursion, on the predicate $ingroup/2$, a feature that is beyond the capabilities of many ECA commercial system, like e.g. SQL-triggers [33].

event $e_1 \triangle e_2$ occurs at instant i iff e_1 *and* e_2 occur at instant i; event $e_1 \bigtriangledown e_2$ occurs at instant i iff e_1 *or* e_2 occur at instant i. Event $A(e_1, e_2, e_3)$ occurs at instant i iff event e_1 occurred at some previous instant m, e_2 did not occurred at m nor at any instant between m and i and e_3 occurs for the first time since m at instant i. Given an event e, event $not\ e$ occurs at instant i iff e *does not* occur at instant i. It is also possible to define a new event e_{def} by associating an event e to the atom e_{def}. This is done by expressions called *event definitions* of the form e_{def} **is** e where e is obtained by the event algebra above.

For instance, let $falsE$ be the basic event that *never* occurs. Event $e_1; e_2$ means "e_1 occurred in the past and now e_2 occurs" and its definition is:
$e_1; e_2$ **is** $A(e_1, e_2, falsE)$. It is also possible to use defined events in the definition of other events. For instance, event $e_1 + e_2$ means " e_1 and e_2 occurred concurrently or in any order" and its definition is: $e_1 + e_2$ **is** $(e_1 \triangle e_2) \bigtriangledown (e_1; e_2) \bigtriangledown (e_2; e_1)$. Still referring to example 1,event $alert2E(S_1, S_2)$ occurs when two signals $alE(S_1), alE(S_2)$ occur simultaneously or in different instants without event $stop\_alertE$ occurring in the meanwhile. Formally:

$$alert2E(S_1, S_2) \textbf{ is } A(alE(S_1), alE(S_2), stop\_alertE) \bigtriangledown (alE(S_1) \triangle alE(S_2))$$

As for events, often we need to combine basic actions to obtain complex ones. In ERA this is done by an *action algebra* whose operators are: $\triangleright | \ \| \ | \ IF$. Given the actions a_1, a_2 and a literal C, to execute $a_1 \triangleright a_2$ means to *sequentially* execute first a_1, then a_2; to execute $a_1 \| a_2$ means to *concurrently* execute a_1 and a_2. To execute $IF(C, a_1, a_2)$ means that " *if* C is derived, *then* execute a_1 *else* execute a_2. As for events, it is possible to define new actions. This is done by expressions called *action definitions* of the form a_{def} **is** a where a is obtained by the action algebra above and a_{def} the name of the new actions.

For instance, complex action $fire\_alarmA$ is obtained by first executing concurrently the actions $opendoorsA$ (open all the electric doors in the building) and $firecallA$ (call the firemen station) and after that to execute action $electricityA(off)$ (turn the electricity off) and it is defined as follows:

$$fire\_alarmA \textbf{ is } (opendoorsA \| firecallA) \triangleright electricityX(off)$$

The basic action $define(d)$ introduced above, where d is an event (resp. action) definition e_{def} **is** e (resp. a_{def} **is** a) is used to introduce new definitions or to replace the old definitions for e_{def} (resp. a_{def}) with the new one.

As anticipated in the introduction, *inhibition rules* are rules of the form (2) where B is a set of literals (normal literals or events) that are used to inhibit previous reactive rules. For instance, the inhibition rule

$$\textbf{When } alE(S), room(S, R), not\ room(Pl, R) \textbf{ Do } not\ openA(Pl).$$

updates rule (3). When this rule is asserted, whenever $alE(S)$ occurs, any fire plug Pl which is not in the room R where the sensor S is located is not opened, even if Pl is on the same floor of S. A program developed in ERA initially consists of the *ERA program* P_1, i.e. an initial set of rules and definitions. Program P_1 is then updated by other ERA

programs $P_2, \ldots, Pn$ consisting of facts and rules asserted and new definitions. Such a sequences, called *ERA dynamic programs* determine, at each instant, the behaviour of the system. For this reason the semantics of ERA is given in section 4 wrt ERA dynamic programs. Formally, the complete syntax of ERA is:

Definition 2. *Let $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{def}$, $\mathcal{A}_X$ and $\mathcal{A}_{def}$ be sets of atoms called,* condition alphabet *and set of, respectively,* basic events, event names, external actions *and* action names *and let L, e_b, e_{def}, a_x and a_{def} be generic elements of, respectively, $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{def}$, $\mathcal{A}_X$ and $\mathcal{A}_{def}$. The set of* positive events *$\mathcal{E}$ over $\mathcal{E}_B$, and $\mathcal{E}_{def}$ is the set of atoms e_p of the form:*

$$e_p ::= e_b \mid e_1 \triangle e_2 \mid e_1 \bigtriangledown e_2 \mid A(e_1, e_2, e_3) \mid e_{def}.$$

where e_1, e_2, e_3 are generic elements of $\mathcal{E}$. An event over *$\mathcal{E}$ is any literal over $\mathcal{E}$. A* negative event *over $\mathcal{E}$ is any literal of the form $not\ e_p$.*

A basic action *a_b over $\mathcal{E}, \mathcal{L}, A_X, \mathcal{A}_{def}$ is any atom of the form:*

$$a_b ::= a_x \mid rise(e_b) \mid assert(\tau) \mid retract(\tau) \mid define(d).$$

where τ (resp. d) is any ERA rule (resp. definition) over $\mathcal{L}^{ERA}$.

The set of actions *$\mathcal{A}$ over $\mathcal{E}, C, A_X, \mathcal{A}_{def}$ is the set of atoms a of the form:*

$$a ::= a_b \mid a_1 \triangleright a_2 \mid a_1 \| a_2 \mid IF(C, a_1, a_2) \mid a_{def}.$$

where a_1 and a_2 are arbitrary elements of $\mathcal{A}$ and C is any literal over $\mathcal{E} \cup \mathcal{L}$.

The ERA alphabet *(or simply the alphabet) $\mathcal{L}^{ERA}$ over $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{def}$, $\mathcal{A}_X$ and $\mathcal{A}_{def}$ is the triple $\mathcal{E}, \mathcal{L}, \mathcal{A}$. In the following, let e and a be arbitrary elements of, respectively, $\mathcal{E}$ and $\mathcal{A}$, B any set of literals over $\mathcal{E} \cup \mathcal{L}$ and $Condition$ any set of literals over $\mathcal{L}$.An* ERA expression *is either an ERA definition or an ERA rule. An* ERA definition *is either an event definition or and action definition.An* event definition *over $\mathcal{L}^{ERA}$ is any expression of the form e_{def} **is** e.An* action definition *over $\mathcal{L}^{ERA}$ is any expression of the form a_{def} **is** a.An* ERA rule *is either an inference, active or inhibition rule over $\mathcal{L}^{ERA}$. An* inference rule *over $\mathcal{L}^{ERA}$ is any rule of the form $L \leftarrow B$.A* reactive rule *over $\mathcal{L}^{ERA}$ is any rule of the form **On** e **If** $Condition$ **Do** a.An* inhibition rule *over $\mathcal{L}^{ERA}$ is any rule of the form **When** B **Do** $not\ a$.An* ERA program *over $\mathcal{L}^{ERA}$ is any set of ERA rules and definitions over $\mathcal{L}^{ERA}$. An* ERA dynamic program *is any sequence of ERA programs.*

4 Semantics of ERA

Having defined a syntax for programming ECA systems in ERA (called, from now onwards, *ERA systems*) we now provide a semantics to such systems. An ERA system receives inputs in the form of *input programs*. Formally, an input program E_i, over an alphabet $\mathcal{L}^{ERA}$, is any set whose elements are either ERA expressions over $\mathcal{L}^{ERA}$ or facts of the form e_b where e_b is an element of $\mathcal{E}_B$ (i.e. a basic event). At any instant i, an ERA systems receives a, possibly empty, input program[2] E_i. The sequence of input programs $E_1, \ldots E_n$ denotes the sequence of input programs received at instants $1, \ldots, n$.

[2] ERA adopts a discrete concept of time, any input program is indexed by a natural number representing the instant at which the input program occurs.

A basic event e_b *occurs* at instant i iff the fact e_b belongs to E_i. Since, by operator A, a complex event may be obtained combining basic events occurring at different instants, in order to detect the occurrence of such complex events in general it is necessary to store the sequence of all the received input programs. Formally, an *ERA system* $\mathcal{S}$ is a triple of the form $(\mathcal{P}, E_P, E_i.E_F)$ where $\mathcal{P}$ is an ERA dynamic program, E_P is the sequence of all the previously received input programs and $E_i.E_F$ is the sequence of the current (E_i) and the future (E_F) input programs. As it will be clear from sections 4.1 and 4.2, the sequence E_F does not influence the system at instant i and hence no "look ahead" capability is required. However, since a system is capable (via action $rise$) of autonomously *rising* events in the future, future input programs are included in the system as "passive" elements that are modified as effects of actions (see rule (5)). Providing a semantics to ERA systems means to specify, at each instant, which conclusions are derived, which actions executed and what are the effects of those actions. Given a conclusion B, and an ERA system $\mathcal{S}$, notation $S \vdash_e B$ denotes that S derives B (or that B is inferred by S). The definition of $\vdash_e$ is to be found in section 4.1.

At each instant, an ERA system $\mathcal{S}$ *concurrently* executes all the actions a_k such that $\mathcal{S} \vdash_e a_k$. As a result of these actions an ERA system *transits* into another ERA system. While the execution of basic actions is "instantaneous", complex actions may involve the execution of several basic actions in a given order and hence require several transitions to be executed. For this reason, the effects of actions are defined by transitions of the form $< \mathcal{S}, A > \mapsto^G < \mathcal{S}', A' >$ where $\mathcal{S}, \mathcal{S}'$ are ERA systems, A, A' are sets of actions and G is a set of basic actions. The basic actions in G are the first step of the execution of set of actions A, while the set of actions A' represents the remaining steps to complete the execution of A. For this reason A' is also called the *set of residual actions* of A. The transition relation $\mapsto$ is defined by a transition system in section 4.2. At each instant an ERA system receives an input program, derives a new set of actions A_N and starts to execute these actions together with the residual actions not yet executed. As a result, the system evolves according to the transition relation $\rightarrow$. Formally:

$$\frac{A_N = \{a_k \in \mathcal{A} : \ \mathcal{S} \ \vdash_e a_k\} \ \wedge \ < \mathcal{S}, (A \cup A_N) > \mapsto^G < \mathcal{S}', A' >}{< \mathcal{S}, A > \rightarrow^G < \mathcal{S}', A' >} \tag{4}$$

4.1 Inferring conclusions

The inference mechanism of ERA is derived from the inference mechanism for DyLPs. In section 2, we provide two distinct ways (called resp. cautious and brave reasoning) to define an inference relation $\vdash$ between a DyLP and a conclusion on the base of the refined semantics. From inference relation $\vdash$, in the following we derive a relation $\vdash_e$ that infers conclusions from an ERA system.

Let $\mathcal{S} = (\mathcal{P}, E_P, E_i.E_F)$ be an ERA system over the alphabet $\mathcal{L}^{ERA} : \ (\mathcal{E}, \mathcal{L}, \mathcal{A})$, with $E_P = E_1, \ldots E_{i-1}$. For any $m < i$, let $\mathcal{S}^m$ be the ERA system $(\mathcal{P}, E^{m-1}, E^m.null)$. Sequence E_F represents future input programs and is irrelevant for the purpose of inferring conclusions in the present, and sequence E_P stores previous events, and is only used for detecting complex events. The relevant expressions are hence those in $\mathcal{P}$ and E_i. As a first step we reduce the expressions of these programs to LP rules. An event definition, associates an event e to a new atom e_{def}. This is encoded by the rule

$e_{def} \leftarrow e$. Action definitions, instead, specify what are the effects of actions and hence are not relevant for inferring conclusions. Within ERA, actions are executed iff they are inferred as conclusions. Hence, reactive (resp. inhibition) rules are replaced by LP rules whose heads are actions (resp. negation of actions) and whose bodies are the events and conditions of the rules. Formally: let $\mathcal{P}^R$ and E_i^R be the DyLP and GLP obtained by $\mathcal{P}$ and E_i by deleting every action definition and by replacing:

every rule	**On** e **If** $Condition$ **Do** $Action$.	with $Action \leftarrow Condition, e$.
every rule	**When** B **Do** $not\ Action$	with $not\ Action \leftarrow B$.
every definition	e_{def} **is** e.	with $e_{def} \leftarrow e$.

As the reader will notice, events are reduced to ordinary literals. Since events are meant to have special meanings, we encode these meanings by extra rules. Intuitively, operators $\triangle$ and $\triangledown$ stands for the logic operators $\wedge$ and $\vee$. This is encoded by the set of rules $ER(\mathcal{E})$ defined as follows:

$$ER(\mathcal{E}) : \triangle(e_1, e_2) \leftarrow e_1, e_2.\ \ \triangledown(e_1, e_2) \leftarrow e_1.\ \ \triangledown(e_1, e_2) \leftarrow e_2.\ \ \forall\, e_1, e_2, e_3\ \in \mathcal{E}$$

Event $A(e_1, e_2, e_3)$ occurs at instant i iff e_2 occurs at instant i and some conditions on the occurrence of e_1, e_2 and e_3 where satisfied in the previous instants. This is formally encoded by the set of rules $AR(\mathcal{S})$ defined as follows[3]:

$$AR(\mathcal{S}) = \left\{ \begin{array}{ll} \forall\, e_1, e_2, e_3\ \in \mathcal{E} & A(e_1, e_2, e_3) \leftarrow e_2 : \exists\, m < i\ t.c. \\ \mathcal{S}^m\ \vdash_e e_1 \wedge \mathcal{S}^m\ \nvdash_e e_3 \wedge\ \forall j\ \ t.c.\ m < j < i : & \mathcal{S}^j\ \ \nvdash_e e_2 \wedge \mathcal{S}^j\ \nvdash_e e_3 \end{array} \right\}$$

The sets of rules E_i^R, $ER(\mathcal{E})$ and $AR(\mathcal{S})$ are added to $\mathcal{P}^R$ and conclusions are derived by the inference relation $\vdash$ applied on the obtained DyLP[4]. Formally:

Definition 3. *Let $\vdash$ be an inference relation defined as in section 2, $\mathcal{S}$, $\mathcal{P}^R$, E_i^R, $ER(\mathcal{E})$, $AR(\mathcal{S})$ be as above and K be any conclusion over $\mathcal{E} \cup \mathcal{L} \cup \mathcal{A}$. Then:*

$$(\mathcal{P}, E_P, E_i.E_F)\ \vdash_e K \quad \Leftrightarrow \quad \mathcal{P}^R \uplus (E_i^R \cup ER(\mathcal{E}) \cup D(\mathcal{P}) \cup AR(\mathcal{S})) \vdash K$$

Note that we do not specify rules for operator not . These rules are not needed since event (literal) $not\ e_p$ is inferred by default negation whenever there is no prove for e_p. In section 3 we provided the intuitive meanings of the various operators. The following theorem formalizes these intuitive meanings.

Theorem 1. *Let $\mathcal{S}$ be as above, e_b, a basic event, e_p a positive event, e_{def} an event name and e_1, e_2, e_3 three events, the following double implications hold:*

$$\begin{array}{lll} \mathcal{S}\ \vdash_e e_1 \triangle e_2 & \Leftrightarrow \mathcal{S}\ \vdash_e e_1\ \wedge\ \mathcal{S}\ \vdash_e e_2. & \mathcal{S}\ \vdash_e e_b\ \ \Leftrightarrow\ \ e_b \in E_i \\ \mathcal{S}\ \vdash_e e_1 \triangledown e_2 & \Leftrightarrow \mathcal{S}\ \vdash_e e_1\ \vee\ \mathcal{S}\ \vdash_e e_2. & \mathcal{S}\ \vdash_e not\ e_p \Leftrightarrow\ \mathcal{S}\ \nvdash_e e_p. \\ \mathcal{S}\ \vdash_e A(e_1, e_2, e_3) & \Leftrightarrow\ \exists\, m < i\ t.c.\ \mathcal{S}^m\ \vdash_e e_1\ \wedge \mathcal{S}^m\ \nvdash_e e_3\ \wedge\ \forall j\ t.c. & \\ & m < j < i :\ \mathcal{S}^j\ \ \nvdash_e e_2 \quad \wedge \mathcal{S}^j\ \nvdash_e e_3\ \wedge\ \mathcal{S}\ \vdash_e e_2. & \\ \mathcal{S}\ \vdash_e e_{def} & \Leftrightarrow \mathcal{S}\ \vdash_e e\ \wedge\ e_{def}\ \textbf{\textit{is}}\ e\ \in \mathcal{P} & \end{array}$$

[3] The definition of $AR(\mathcal{S})$ involves relation $\vdash_e$ which is defined in terms of $AR(\mathcal{S})$ itself . This mutual recursion is well-defined since, at each recursion, $AR(\mathcal{S})$ and $\vdash_e$ are applied on previous instants until eventually reaching the initial instant (ie, the basic step of recursion)

[4] The program transformation above is functional for defining a declarative semantics for ERA, rather then providing an efficient tool for an implementation. Here specific algorithms for event-detection clearly seems to provide a more efficient alternative.

4.2 Execution of actions

We are left with the goal of defining what are the effects of actions. This is accomplished by providing a transition system for the relation $\mapsto$ that completes, together with transition (4) and the definition of $\vdash_e$, the semantics of ERA. As mentioned above, these transitions have the form: $< \mathcal{S}, A > \mapsto^G < \mathcal{S}', A' >$.

The effects of basic actions on the current ERA program are defined by the *updating function* $up/2$. Let $\mathcal{P}$ be and ERA dynamic program A a set of, either internal or external, basic actions. The output of function $up/2$ is the updated program $up(P, A)$ obtained in the following way: First delete from $\mathcal{P}$ all the rules retracted according to A, and all the (event or action) definitions d_{def} **is** d_{old} such that action $define(d_{def}$ **is** $d_{new})$ belongs to A; then update the obtained ERA dynamic program with the program consisting of all the rules asserted according to A and all the new definitions in A. Formally:

$$\begin{aligned} DR(A) &= \{d : define(d) \in A\} \ \cup \ \{\tau : \ assert(\tau) \in A\} \cup D(A) \\ R(\mathcal{P}, A) &= \{\tau : \ retract(\tau) \in A\} \cup \ \{d_{def} \textbf{ is } d_{old} \ \in \mathcal{P} : \ d_{def} \textbf{ is } d_{new} \in D(A)\} \\ up(\mathcal{P}, A) &= (\ \mathcal{P} \setminus R(\mathcal{P}, A)\ ) .. DR(A) \end{aligned}$$

Let e_b be any basic event and a_i an external action or an internal action of one of the following forms: $assert(\tau), retract(\tau), define(d)$. On the basis of function $up/2$ above, we define the effects of (internal and external) basic actions. At each transition, the current input program E_i is evaluated and stored in the sequence of past events and the subsequent input program in the sequence E_F becomes the current input program (see 1st and 3rd rules below). The only exception involves action $rise(e_b)$ that adds e_b in the subsequent input program E_{i+1}. When a set of actions A is completely executed its set of residual actions is $\emptyset$. Basic actions (unlike complex ones) are completely executed in one step, hence they have no residual actions. Formally:

$$\begin{aligned} < (\mathcal{P}, E_P, E_i.E_F), \emptyset > \ &\mapsto^{\emptyset} \ < \mathcal{P}, E_P..E_i, E_F, \emptyset > \\ < (\mathcal{P}, E_i.E_{i+1}.E_S), \{rise(e_b)\} > \ &\mapsto^{\emptyset} \ < (\mathcal{P}, E_P..E_i, (E_{i+1} \cup \{e_b\}).E_F), \emptyset > \\ < (\mathcal{P}, E_P, E_i.E_F), \{a_i\} > &\mapsto^{\{a_i\}} < (up(\mathcal{P}, \{a_i\}), E_P..E_i, E_F), \emptyset > \end{aligned} \quad (5)$$

Note that, although external actions do not affect the ERA system, as they do not affect the result of $up/2$, they are nevertheless *observable*, since they are registered in the set of performed actions (cf. 3rd rule above). Unlike basic actions, generally the execution of a complex action involves several transitions. Action $a_1 \triangleright a_2$, where a_1, a_2 are arbitrary actions, consists into first executing all basic actions for a_1, until the set residual actions is $\emptyset$, then to execute all the basic actions for a_2. We use the notation $A_1 \triangleright a_2$, where A_1 is a set of actions, to denote that action a_2 is executed after all the actions in the set A_1 have no residual actions. Action $a_1 \| a_2$, instead, consists into *concurrently* executing all the basic actions forming both actions, until there are no more of residual actions to execute. Similarly, the execution of a set of actions A consists in the concurrent execution of all its actions a_k until the set of residual actions is empty. Semantically, the concurrent execution of basic actions is defined by function $up/2$ which is defined over an ERA dynamic program and an *set* of basic actions. The execution of $IF(C, a_1, a_2)$ is equal to the execution of a_1 if the system infers C, otherwise it is equal

to the execution of a_2. Given an ERA system $\mathcal{S} = (\mathcal{P}, E_P, E_i.E_F)$ with $\mathcal{P} : P_1 \dots P_n$, let $D(\mathcal{S})$ the set of all the action definitions d such that, for some j, $d \in P_j$ or $d \in E_i$. The execution of action a_{def}, where a_{def} is defined by one ore more action definitions, corresponds to the concurrent executions of all the actions a_ks such that a_{def} **is** a_k belongs to $D(\mathcal{S})$. Formally:

$$\frac{< \mathcal{S}, \{a_1, a_2\} > \mapsto^{G} < \mathcal{S}', A' >}{< \mathcal{S}, \{a_1 \| a_2\} > \mapsto^{G} < \mathcal{S}', A' >} \quad \frac{< \mathcal{S}, \{a_1\} > \mapsto^{G_1} < \mathcal{S}', A'_1 >}{< \mathcal{S}, \{a_1 \triangleright a_2\} > \mapsto^{G_1} < \mathcal{S}', \{A'_1 \triangleright a_2\} >}$$

$$\frac{< \mathcal{S}, A_1 > \mapsto^{G_1} < \mathcal{S}', A'_1 >}{< \mathcal{S}, \{A_1 \triangleright a_2\} > \mapsto^{G_1} < \mathcal{S}', \{A'_1 \triangleright a_2\} >} \quad \frac{< \mathcal{S}, \{a_2\} > \mapsto^{G_2} < \mathcal{S}'', A''_2 >\}}{< \mathcal{S}, \{\emptyset \triangleright a_2\} > \mapsto^{G_2} < \mathcal{S}'', A''_2 >}$$

$$\frac{\mathcal{S} \vdash_e C \wedge < \mathcal{S}, \{a_1\} > \mapsto^{G_1} < \mathcal{S}', A' >}{< \mathcal{S}, \{IF(C, a_1, a_2)\} > \mapsto^{G_1} < \mathcal{S}', A'_1 >} \quad \frac{\mathcal{S} \nvdash_e C \wedge < \mathcal{S}, \{a_2\} > \mapsto^{G_2} < \mathcal{S}'', A''_2 >}{< \mathcal{S}, \{IF(C, a_1, a_2)\} > \mapsto^{G_2} < \mathcal{S}'', A''_2 >}$$

$$\frac{A = \{a_1, \dots, a_n\} \; < (\mathcal{P}, E_P, E_i.E_{i+1}.E_F), \{a_k\} > \mapsto^{G_k} < (\mathcal{P}^k, E_P..E_i, E^k_{i+1}.E_F), A'_k\} >}{< (\mathcal{P}, E_P, E_i.E_{i+1}.E_F), A > \mapsto^{\bigcup G_k} < (up(\mathcal{P}, \bigcup G_k), E_P..E_i, \bigcup E^k_{i+1}.E_F), \bigcup A'_k >}$$

$$\frac{A = \{a_k : \; a_{def} \textbf{ is } a_k. \in D(\mathcal{S})\} \wedge \; < \mathcal{S}, A > \mapsto^{G} < \mathcal{S}', A' >}{< \mathcal{S}, \{a_{def}\} > \mapsto^{G} < \mathcal{S}', A' >}$$

5 Example of Usage: A monitoring system

We now present the ERA program P_1 (partially) formalizing example 1.

(a)	**On** $alE(S)$	**If** $flr(S, Fl), firepl(Pl), flr(Pl, Fl)$ **Do** $openA(Pl)$.
(b)	**On** $openE(D)$	**Do** $assert(open(D))$.
(c)	**On** $closeE(D)$	**Do** $assert(not\ open(D))$.
(d)	$alert2E(S_1, S_2)$	**is** $A(alE(S_1), alE(S_2), stop\_alertE) \bigtriangledown (alE(S_1) \bigtriangleup alE(S_2))$.
(e)	$fire\_alarmA$	**is** $(opendoorsA \| firecallA) \triangleright electricityA(off)$.
(f)	**On** $alert2E(S_1, S_2)$	**If** $not\ same\_room(S_1, S_2)$ **Do** $fire\_alarmA$.
(g)	$same\_room(S_1, S_2)$	$\leftarrow room(S_1, R_1), room(S_2, R_1)$.
(j)	**On** $seminarE(S)$	**If** $compose(S, M), sendto(S, E@)$ **Do** $sendA(M, E@)$.
(k)	$sendto(S, E@)$	$\leftarrow group(S, G), ingroup(Emp, G), mail(Emp, E@)$.
(h)	$ingroup(Emp, G)$	$\leftarrow ingroup(Emp, S), sub(S, G)$.

We already discussed expressions $(a - e)$ in section 3. Reactive rule (f) triggers action $fire\_alarmA$ when event $alert2E(S_1, S_2)$ occurs (ie, two sensors rise an alarm) under the condition that the senors are located in different rooms (this condition is inferred by rule g). When event $seminarE(S)$ occurs (a seminar S is announced) reactive rule (j) triggers action $sendA(M, E@)$ ie, a message M is sent to the email address $E@$ of any employee Emp in the working group G to which seminar S is dedicated (rules $(k-h)$).

An external update is done by an input program containing internal actions ($assert$, $retract$ and $define$) as facts. For instance, the behaviour of P_1 is updated by the following input program:

$$E_i : \quad assert(R_1). \quad assert(R_2). \quad assert(R_3). \quad define(d_1). \quad define(d_2).$$

where $R_1 - R_3$ and $d_1 - d_2$ are the following expressions.

R_1 : **When** $alE(S), room(S,R), not\ room(Pl,R)$ **Do** $not\ openA(Pl)$.
R_2 : **On** $redirectE(Emp,Num)$ **Do** $redirectA(Emp,Num)$.
R_3 : **On** $stop\_redirectE(Emp,Num)$ **Do** $stop\_redirectA(Emp)$.
$d_1 : redirectA(Emp,Num)$ **is** $assert(\tau_1) \triangleright assert(\tau_2)$.
$d_2 : stop\_redirectA(Emp,Num)$ **is** $retract(\tau_1) \| retract(\tau_2)$.

and τ_1 and τ_2 are the following rules:

τ_1 : **When** $phonE(Call), dest(Call,Emp)$ **Do** $not\ forwX(Call,N)$.
τ_2 : **On** $p\ honE(Call)$ **If** $dest(Call,Emp)$ **Do** $forwX(Call,Num)$.

We already discussed the effect of inhibition rule R_1. Reactive rules $R_2 - R_3$ encode the new behaviour of the system when an employee Emp asks the system to start (resp. to stop) redirecting to the phone number Num any phone call $Call$ to him. This is achieved by sequentially asserting (resp. retracting) rules τ_1, τ_2. The former is an inhibition rule that inhibits any previous rule reacting to a phone call for Emp (ie, to the occurrence of event $phonE(Call)$) by forwarding the call to a number N. The latter is a reactive rule forwarding the call to number Num. Note that the two rules have to be asserted sequentially in order to prevent mutual conflicts. To revert to the previous behaviour it is sufficient to retract the two rules as done by action $stop\_redirectA$.

6 Conclusions and related work

We identified a set of desirable features for an ECA language, namely: a declarative semantics, the capability to express complex events and actions in a compositional way, and that of receiving external updates, and perform self updates to data, inference rules and reactive rules. For this purpose we defined the logic programming ECA language ERA, based on the concept of DyLPs and provided it with a declarative semantics based on the refined semantics of DyLPs (for inferring conclusions) and a transition system (for the execution of actions).

There exists several alternative proposals of ECA formalisms. However, most of these approaches are mainly procedural like, for instance, AMIT [31], RuleCore [1] and JEDI [19] or at least not fully declarative [33]. A declarative situation calculus-like characterizations of active database systems is given in [10], although the subject of complex actions is not treated there. Examples of Semantic Web-oriented ECA languages are XChange [13] and Active XML [3]. XChange has an LP-like semantics, allows to define reactive rules with complex actions and (unlike Active XML) complex events. However, neither of the two supports a construct similar to action definitions for defining actions. This possibility is allowed by the Agent-Oriented Language DALI [16]. On the other hand, DALI does not support complex events. The ideas and methodology for defining complex actions are inspired to works on process algebras like CCS [27], CSP [22] and TCC [30]. Rather then proposing high level ECA languages, these works design abstract models for defining programming languages for parallel execution of processes. Other related frameworks are Dynamic Prolog [12] and Transaction

Logic Programming (TLP) [11]. These works focus on the problem of updating a deductive database by performing transactions. In particular, TLP shares with ERA the possibilities to specify complex (trans)actions in terms of other, simpler, (trans)actions. However, TLP (and Dynamic Prolog) does not support complex events, nor does it cope with the possibility to receive external inputs during the computation of complex actions. None of the languages cited so far shows updates capabilities analogous to the ones evidenced in ERA. A class of proposals close in spirit to ERA, are LP update languages like EPI [18], Evolp [6], LUPS [8] and Kabul [23]. All these languages show the possibility to update rules. Within EPI and LUPS, it is possible to update derivation rules but not reactive rules. Evolution of reactive rules is a feature of both Kabul and Evolp. However, nor Evolp, Kabul, nor any of the cited LP update languages supports external nor complex actions or complex events.

The language ERA still requires a relevant amount of research. Preliminary investigations evidenced interesting properties of the operators of the action algebra like associativity, commutativity etc. Moreover, the transition system for ERA defined in section 4 is deterministic. Beside the opportunities opened by action definitions, the problem arises of restricting the class of allowed definitions in order to guarantee that transition relation (4) is always defined. Our idea is to apply a criterium of *left guardedness* to action definitions (see [27]). In this work we opted for an inference system based on the refined semantics for DyLPs. With limited efforts, it would be possible to define an inference system on the basis of another semantics for DyLPs such as [7, 9, 14, 17, 24, 29, 34, 32]. In particular, we intend to develop a version of ERA based on the well founded semantics of DyLPs [9]. Well founded semantics [20] is a polynomial approximation of the answer set semantics that could be particulary suitable for applications involving large databases requiring the capability to quickly process vast amount of information. Finally, implementations of the language and its possible extensions are in order.

References

1. Rulecore. *http://www.rulecore.com*.
2. Amazon web site. http//:www.amazon.com/, 2001.
3. S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active xml: Peer-to-peer data and web services integration. *citeseer.ist.psu.edu/article/abiteboul02active.html*, 2002.
4. Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. In *ADBIS*, pages 190–204, 2003.
5. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1), 2005.
6. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *JELIA'02*, LNAI, 2002.
7. 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.
8. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.
9. 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.

10. Chitta Baral and Jorge Lobo. Formal characterization of active databases. In *Logic in Databases*, pages 175–195, 1996.
11. A. J. Bonner and M. Kifer. Transaction logic programming. In David S. Warren, editor, *ICLP-93*, pages 257–279, Budapest, Hungary, 1993. The MIT Press.
12. Anthony J. Bonner. A logical semantics for hypothetical rulebases with deletion. *Journal of Logic Programming*, 32(2), 1997.
13. François Bry, Paula-Lavinia Patranjan, and Sebastian Schaffert. Xcerpt and xchange - logic programming languages for querying and evolution on the web. In *ICLP*, pages 450–451, 2004.
14. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *ICLP-99*, Cambridge, November 1999. MIT Press.
15. Jan Carlson and Björn Lisper. An interval-based algebra for restricted event detection. In *FORMATS*, pages 121–133, 2003.
16. Stefania Costantini and Arianna Tocchio. The dali logic programming agent-oriented language. In *JELIA*, pages 685–688, 2004.
17. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of semantics based on causal rejection. *Theory and Practice of Logic Programming*, 2:711–767, 2002.
18. Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. A framework for declarative update specifications in logic programs. In *IJCAI*, 2001.
19. G.Cugola, E. D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *20th International Conference on Software Enginieering*, 1998.
20. 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.
21. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *5th International Conference on Logic Programming*. MIT Press, 1988.
22. C.A.R. Hoare. *Communication and Concurrency*. Prentice-Hall, 1985.
23. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
24. J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In *LPKR'97: workshop on Logic Programming and Knowledge Representation*, 1997.
25. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *KR-92*, 1992.
26. J. W. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, 2 edition, 1987.
27. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
28. S.Abiteboul, C.Culet, L. Mignet, B.Amann, T.Milo, and A. Eyal. Active views for electronic commerce. In *25th Very Large Data Bases Coference Proceedings*, 1999.
29. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR-99*, Berlin, 1999. Springer.
30. Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5/6), 1996.
31. Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Amit - the situation manager. *The International Journal on Very Large Data Bases archive*, 13, 2004.
32. J. Sefranek. A kripkean semantics for dynamic logic programming. In *Logic for Programming and Automated Reasoning(LPAR'2000)*. Springer Verlag, LNAI, 2000.
33. J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.
34. 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.

A Similarity Measure for the $\mathcal{ALN}$ Description Logic

Nicola Fanizzi and Claudia d'Amato

Dipartimento di Informatica, Università degli Studi di Bari
Campus Universitario, Via Orabona 4, 70125 Bari, Italy
{fanizzi|claudia.damato}@di.uniba.it

Abstract. This work presents a similarity (and a derived dissimilarity) measure for Description Logics that are the theoretical counterpart of the standard representations for ontological knowledge. The focus is on the definition of a similarity measure for $\mathcal{ALN}$ concept descriptions, based both on the syntax and on the semantics of the descriptions elicited from the current state of the world. An extension of the measure is proposed for involving individuals and then for evaluating their (dis-)similarity, which makes it suitable for several (inductive) tasks.

1 Assessing the Similarity in Concept Languages

In the Semantic Web perspective [3], similarity plays an important role in several tasks, such as classification, clustering, retrieval and knowledge integration. Nevertheless, we are still at an initial phase in the definition of measures for assessing the similarity or the dissimilarity of concepts as described in the standard ontology languages [5].

Various distance measures for concept representations have been proposed in the literature (see a survey in [20]); they can be essentially categorized in three different types. *Path distance* measures have been defined as a function of the distance between terms in the hierarchical structure underlying an ontology [6]. The *feature matching* approach [24] uses both common and discriminant features among concepts and/or concept instances to compute the semantic similarity. Finally, there are methods founded on the *information content* [19, 10] where a similarity measure for concepts within a hierarchy is defined in terms of the variation of the information content conveyed by the concepts and the one conveyed by their immediate common super-concept. This is a measure of the variation of the information from a description level to a more general one.

Other measures compute the similarity among concepts belonging to different ontologies (e.g. see [25]). In [21] a similarity function determines similar classes by using a matching process making use of synonym sets, semantic neighborhood, and discriminating features that are classified into parts, functions, and attributes (see a recent survey in [23]). However, for the moment, this topic is beyond the scope of our work.

As pointed out in [5], most of the measures proposed so far are applicable to the assessment of the similarity of atomic concepts (within a hierarchy) rather

than on composite ones or they refer to very simple ontologies, built only using simple relations such as *is-a* and *part-of* (typical of lexical ontologies). Nevertheless, the standard ontology languages (e.g., OWL [18]) are founded in Description Logics (henceforth DLs) since they borrow the typical DLs constructors. Thus, it becomes necessary to investigate the similarity of more complex concept descriptions expressed in DLs. However, it has been observed that the structure of the descriptions becomes much less important when richer representations are adopted, due to the expressive operators that can be employed.

An approach intended for information retrieval purposes on DLs knowledge bases [16], aims at finding commonalities among concepts or among assertions, employing the *Least Common Subsumer* (LCS) operator [7] that computes the most specific generalization of the input concepts (with respect to subsumption). 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 measures defined in the cited works are suitable for very simple languages and not for the composite descriptions that can be obtained using the operators of DLs. Hence the semantics of these descriptions derives almost straightforwardly from their simple structures. We decided to focus our attention on measures which are essentially founded on semantics. Initially, we have defined dissimilarity measures between concept descriptions that virtually may work for any representation [9], being based exclusively on semantics. But this falls short when individuals come into play. Indeed, in the tasks which represent the final aim of our investigation on these measures, such as clustering, classification and retrieval, it is necessary to compute distances between individuals and concepts or between individuals. By recurring the notion of *most specific concept* (MSC) of an individual with respect to an ABox [1], measures based both on the concept structure and their semantics can be extended to such cases.

On the grounds of these ideas, we could define measures which are suitable for composite DLs descriptions and in particular for $\mathcal{ALC}$ [8, 10]. These measures elicit the underlying semantics by querying the knowledge base for assessing the information content of concept descriptions with respect to the knowledge base, as proposed also in [2]. In the perspective of defining a measure for more expressive ontology languages endowed with more constructors, with this work we intend to investigate and extend these ideas to languages endowed with numeric restrictions, starting from $\mathcal{ALN}$.

The remainder of this paper is organized as follows. In Sect. 2 the representation language $\mathcal{ALN}$ is presented. The similarity measure is illustrated and discussed in Sect. 3, with the extension to the cases involving individuals. Final remarks and possible applications and developments of the measure are examined in Sect. 4.

2 Background: The $\mathcal{ALN}$ Description Logic

$\mathcal{ALN}$ is a DLs language which allows for the expression of universal features and numeric constraints [1]. It has been adopted because of the tractability of the main related reasoning services [11]. Furthermore it has already been adopted also in other frameworks for learning in hybrid representations such as CARIN-$\mathcal{ALN}$ [22] or IDLP [13]. In order to keep this paper self-contained, syntax and semantics for the reference representation is briefly recalled with the characterization of the descriptions in terms of concept graphs.

2.1 Syntax and Semantics

In DLs, primitive *concepts* $N_C = \{A, \ldots\}$ are interpreted as subsets of a certain domain of objects and primitive *roles* $N_R = \{R, S, \ldots\}$ are interpreted as binary relations on such a domain. In $\mathcal{ALN}$, composite concept descriptions are built using atomic concepts and primitive roles by means of the constructors presented in Table 1. The meaning of such descriptions is defined by means of an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is the *domain* of the interpretation and the functor $\cdot^{\mathcal{I}}$ (the *interpretation function*) maps concept and role descriptions to their extension: $\forall C \in N_C : C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and $\forall R \in N_R : R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

Table 1. Constructors and related interpretations for $\mathcal{ALN}$.

Name	Syntax	Semantics
top concept	$\top$	$\Delta^{\mathcal{I}}$
bottom concept	$\bot$	$\emptyset$
primitive concept	A	$A^{\mathcal{I}} \subseteq \Delta$
primitive negation	$\neg A$	$\Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$
concept conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
value restriction	$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y\ (x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
at-most restriction	$\leq n.R$	$\{x \in \Delta^{\mathcal{I}} \mid \ \lvert\{y \in \Delta^{\mathcal{I}} \mid (x,y) \in R^{\mathcal{I}}\}\rvert \leq n\}$
at-least restriction	$\geq n.R$	$\{x \in \Delta^{\mathcal{I}} \mid \ \lvert\{y \in \Delta^{\mathcal{I}} \mid (x,y) \in R^{\mathcal{I}}\}\rvert \geq n\}$

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. Differently from ILP, each (non primitive) concept has a single definition. Moreover, the DLs definitions are assumed not to be recursive, i.e. concepts cannot be defined in terms of themselves.

The A-box $\mathcal{A}$ 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 $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$. Note that, differently from the examples in the ILP setting, the concept description C can be more complex than LP facts. For instance they could assert a universal property

of the an individual: $(\forall R.(A \sqcap \neg B))(a)$ that is, role R relates a exclusively to individuals[1] that are instances of the concept $A \sqcap \neg B$.

Example 2.1. Examples of $\mathcal{ALN}$ descriptions are [2]:

$$\begin{aligned}
\mathsf{Single} &\equiv \mathsf{Person} \sqcap \leq 0.\mathsf{marriedTo} \\
\mathsf{Polygamist} &\equiv \mathsf{Person} \sqcap \forall \mathsf{marriedTo}.\mathsf{Person} \sqcap \geq 2.\mathsf{marriedTo} \\
\mathsf{Bigamist} &\equiv \mathsf{Person} \sqcap \forall \mathsf{marriedTo}.\mathsf{Person} \sqcap = 2.\mathsf{marriedTo} \\
\mathsf{MalePolygamist} &\equiv \mathsf{Male} \sqcap \mathsf{Person} \sqcap \forall \mathsf{marriedTo}.\mathsf{Person} \sqcap \geq 2.\mathsf{marriedTo}
\end{aligned}$$

The notion of *subsumption* between DLs concept descriptions can be given in terms of the interpretations defined above:

Definition 2.1 (subsumption). *Given two concept descriptions C and D, C* subsumes *D iff it holds that $C^{\mathcal{I}} \supseteq D^{\mathcal{I}}$ for every interpretation $\mathcal{I}$. This is denoted denoted by $C \sqsupseteq D$. The induced equivalence relationship, denoted $C \equiv D$, amounts to $C \sqsupseteq D$ and $D \sqsupseteq C$.*

Note that this notion is merely semantic and independent of the particular DLs language adopted. It is easy to see that this definition also applies to the case of role descriptions.

A related inference used in the following is *instance checking*, that is deciding whether an individual is an instance of a concept [12, 1]. Conversely, it may be necessary to solve the *realization problem* that requires finding the concepts which an individual belongs to, especially the most specific one:

Definition 2.2. *Given an ABox $\mathcal{A}$ 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: $D \sqsupseteq C$.*

2.2 Structural Characterizations

Semantically equivalent (yet syntactically different) descriptions can be given for the same concept. However they can be reduced to a canonical form by means of equivalence-preserving rewriting rules, e.g. $\forall R.C_1 \sqcap \forall R.C_2 \equiv \forall R.(C_1 \sqcap C_2)$ (see [17, 1]). The normal form employs the notation needed to access the different parts (*sub-descriptions*) of a concept description C:

- $\mathsf{prim}(C)$ denotes the set of all (negated) concept names occurring at the top level of the description C;
- $\mathsf{val}_R(C)$ denotes conjunction of concepts $C_1 \sqcap \cdots \sqcap C_n$ in the value restriction of role R, if any (otherwise $\mathsf{val}_R(C) = \top$);
- $\mathsf{min}_R(C) = \max\{n \in \mathbb{N} \mid C \sqsubseteq (\geq n.R)\}$ (always a finite number);
- $\mathsf{max}_R(C) = \min\{n \in \mathbb{N} \mid C \sqsubseteq (\leq n.R)\}$ (if unlimited then $\mathsf{max}_R(C) = \infty$).

[1] It holds even in case no such $R-$filler is given.
[2] Here $(= n.R)$ is an abbreviation for $(\leq n.R \sqcap \geq n.R)$.

Definition 2.3 ($\mathcal{ALN}$ normal form). *A concept description C is in $\mathcal{ALN}$ normal form iff $C = \top$ or $C = \bot$ or*

$$C = \prod_{P \in \mathsf{prim}(C)} P \sqcap \prod_{R \in N_R} (\forall R.C_R \sqcap \geq n.R \sqcap \leq m.R)$$

where $C_R = \mathsf{val}_R(C)$, $n = \mathsf{min}_R(C)$ and $m = \mathsf{max}_R(C)$.

The complexity of normalization is polynomial [1]. Besides, subsumption can be checked in polynomial time too [11]. Note also that we are considering the case of subsumption with respect to empty terminologies that suffices for our purposes. Otherwise deciding this relationship may be computationally more expensive.

Although subsumption between concept descriptions is merely a semantic relationship, a more syntactic relationship can be found for a language of moderate complexity like $\mathcal{ALN}$ that allows for a structural characterization of subsumption [14].

Proposition 2.1 (subsumption in $\mathcal{ALN}$). *Given two $\mathcal{ALN}$ concept descriptions C and D in normal form, it holds that $C \sqsupseteq D$ iff all the following relations hold between the sub-descriptions:*

- $\mathsf{prim}(C) \subseteq \mathsf{prim}(D)$
- $\forall R \in N_R\colon\ \mathsf{val}_R(C) \sqsupseteq \mathsf{val}_R(D)$
- $\mathsf{min}_R(C) \leq \mathsf{min}_R(D) \wedge \mathsf{max}_R(C) \geq \mathsf{max}_R(D)$

Hence subsumption checking is accordingly polynomial like $O(n \log n)$, where n is the size of concept C. In the following we will refer to concepts descriptions in normal form unless a different case is explicitly stated.

The tree-structured representation of concept description are defined as follows [17]:

Definition 2.4 (description tree). *A* description tree *for a concept C in $\mathcal{ALN}$ normal form is a tree $\mathcal{G}(C) = (V, E, v_0, l)$ with root v_0 where:*

- *each node $v \in V$ is labelled with a finite set $l(v) \subseteq N_C \cup \{\neg A \mid A \in N_C\} \cup \{\geq n.R \mid n \in \mathbb{N},\ R \in N_R\} \cup \{\leq n.R \mid n \in \mathbb{N},\ R \in N_R\}$*
- *each edge in E is labelled with $\forall R$, where $R \in N_R$*

Proposition 2.2 (equivalence). *An $\mathcal{ALN}$ description C is semantically equivalent to an $\mathcal{ALN}$ description tree $\mathcal{G}(C)$ of size polynomial in the size of C, which can be constructed in polynomial time.*

Example 2.2. The concept description $D \equiv \forall R.(P \sqcap \forall S.Q) \sqcap \forall S.(Q \sqcap \leq 1S)$ is equivalent to the tree depicted in Fig. 1.

Instance checking can be characterized in terms of homomorphisms between trees and graphs standing for the ABoxes [17]:

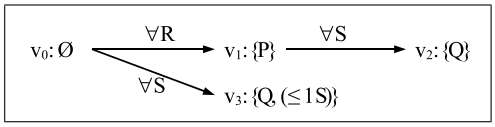


Fig. 1. The concept $D \equiv \forall R.(P \sqcap \forall S.Q) \sqcap \forall S.(Q \sqcap \leq 1S)$ as a description tree.

Definition 2.5 (A-box description graph). *Let $\mathcal{A}$ be an $\mathcal{ALN}$ A-box, a be an individual occurring in $\mathcal{A}$ ($a \in \mathsf{Ind}(\mathcal{A})$) and $C_a = \bigsqcap_{C(a)\in\mathcal{A}} C$. Let $\mathcal{G}(C_a) = (V_a, E_a, a, l)$ denote the description tree of C_a. $\mathcal{G}(\mathcal{A}) = (V, E, l)$ is a* A-box description graph *with:*

- $V = \bigcup_{a\in\mathsf{Ind}(\mathcal{A})} V_a$
- $E = \{aRb \mid R(a,b) \in \mathcal{A}\} \cup \bigcup_{a\in\mathsf{Ind}(\mathcal{A})} E_a$
- $l(v) = l_a(v) \quad$ *for all* $v \in V_a$

In a machine learning perspective, subsumption and instance checking can be used to translate an individual of the domain (an instance of the target concept) into a set of features suitable for propositional algorithms. Indeed, DLs that allow for efficient subsumption procedures, such as $\mathcal{ALN}$, are to be preferred.

3 Measure Definition

Using the structural notion of $\mathcal{ALN}$ normal form and the world-state as represented by the knowledge base, a similarity measure for the space of (equivalent) descriptions $\mathcal{L} = (\mathcal{ALN}\,|_{\equiv})$ can be defined as follows:

Definition 3.1 ($\mathcal{ALN}$ similarity measure). *The function $s : \mathcal{L} \times \mathcal{L} \mapsto [0,1]$ is inductively defined as follows. Given $C, D \in \mathcal{L}$:*

$$s(C,D) := \lambda \left[s_P(\mathsf{prim}(C), \mathsf{prim}(D)) + \frac{1}{|N_R|} \sum_{R\in N_R} s(\mathsf{val}_R(C), \mathsf{val}_R(D)) + \frac{1}{|N_R|} \sum_{R\in N_R} s_N((\mathsf{min}_R(C), \mathsf{max}_R(C)), (\mathsf{min}_R(D), \mathsf{max}_R(D))) \right]$$

where $\lambda \in]0,1]$ ($\lambda \leq 1/3$),

$$s_P(\mathsf{prim}(C), \mathsf{prim}(D)) := \frac{|\bigcap_{P_C\in\mathsf{prim}(C)} P_C^{\mathcal{I}} \cap \bigcap_{Q_D\in\mathsf{prim}(D)} Q_D^{\mathcal{I}}|}{|\bigcap_{P_C\in\mathsf{prim}(C)} P_C^{\mathcal{I}} \cup \bigcap_{Q_D\in\mathsf{prim}(D)} Q_D^{\mathcal{I}}|}$$

and if $\min(M_C, M_D) > \max(m_C, m_D)$ *then*

$$s_N((m_C, M_C), (m_D, M_D)) := \frac{\min(M_C, M_D) - \max(m_C, m_D) + 1}{\max(M_C, M_D) - \min(m_C, m_D) + 1}$$

else

$$s_N((m_C, M_C), (m_D, M_D)) := 0$$

The rationale for the measure is the following. Due to the relative simplicity of the language, the definition of operators working on $\mathcal{ALN}$ may be given structurally, as seen in the Sect. 2. Thus, we define the measure by recursively decomposing the normal form of the concept descriptions under comparison, and measure, per each level and separately, the similarity of the sub-concepts: primitive concepts, value restrictions, and number restrictions. We decided to combine the contribution of each similarity at a given level supposing a fixed[3] rate λ. Actually, in order to have the function s ranging over $[0, 1]$, λ should be less or equal to $1/3$.

The similarity of the primitive concept sets is computed as the ratio of the number of common individuals (belonging to both primitive conjuncts) with respect to the number of the individuals belonging to either conjunct. For those sub-concepts that are related through a role – say R – the similarity of the concepts made up by the fillers is computed recursively by applying the measure to $\mathsf{val}_R(\cdot)$. Finally, the similarity of the numeric restrictions is simply computed as a measure of the overlap between the two intervals. Namely it is the ratio of the amounts of individuals in the overlapping interval and those the larger one, whose extremes are minimum and maximum. Note that some intervals may be unlimited above: $\max = \infty$. In this case we may approximate with an upper limit N greater than $|\Delta| + 1$.

Note that the baseline of this measure is the extension of primitive concepts. Since such extensions cannot be known beforehand due to the *Open World Assumption* (OWA), we make an epistemic adjustment by assuming that it is approximated by retrieving[4] the concept instances based on the current world-state (i.e. according to the ABox $\mathcal{A}$):

$$P^I \leftarrow \{a \in \mathsf{Ind}(\mathcal{A}) \mid I \models_{\mathcal{A}} P(a)\}$$

The interpretation is not decisive because of the *unique names assumption* (UNA) holding for the individual names. Then, we may say that the *canonical interpretation*[5] [1] is considered for counting the retrieved individuals.

Furthermore, it can be foreseen that, per each level, before summing the three measures assessed on the three parts, these figures be normalized. Moreover, a lowering factor $\lambda_R \in]0, 1[$ may be multiplied so to decrease the impact of the sets of individuals related to the top-level ones through some role R.

Example 3.1 (computing the similarity). We show how the distance is practically computed on the ground of an ABox which can be supposed to have been completed according to the TBox descriptions (e.g. $\mathsf{Female} \sqsubseteq \neg\mathsf{Male}$).

[3] Actually we could assign different rates to the similarity of primitive concepts and numerical restrictions and the similarity of concepts for the role fillers.

[4] Formally, given the ABox $\mathcal{A}$ and a concept C, the retrieval service returns the individuals a such that $\mathcal{A} \models C(a)$.

[5] An interpretation where individual names occurring in the ABox stand for themselves.

Let such an ABox be

$$\mathcal{A} = \left\{ \begin{array}{l} \mathsf{Person(Meg)}, \neg\mathsf{Male(Meg)}, \mathsf{hasChild(Meg,Bob)}, \mathsf{hasChild(Meg,Pat)}, \\ \mathsf{Person(Bob)}, \mathsf{Male(Bob)}, \mathsf{hasChild(Bob,Ann)}, \\ \mathsf{Person(Pat)}, \mathsf{Male(Pat)}, \mathsf{hasChild(Pat,Gwen)}, \\ \mathsf{Person(Gwen)}, \neg\mathsf{Male(Gwen)}, \\ \mathsf{Person(Ann)}, \neg\mathsf{Male(Ann)}, \mathsf{hasChild(Ann,Sue)}, \mathsf{marriedTo(Ann,Tom)}, \\ \mathsf{Person(Sue)}, \neg\mathsf{Male(Sue)}, \\ \mathsf{Person(Tom)}, \mathsf{Male(Tom)} \end{array} \right\}$$

and let two descriptions be:

$$C \equiv \mathsf{Person} \sqcap \forall \mathsf{marriedTo}.\mathsf{Person} \sqcap \leq 1.\mathsf{hasChild}$$
$$D \equiv \mathsf{Male} \sqcap \forall \mathsf{marriedTo}.(\mathsf{Person} \sqcap \neg\mathsf{Male}) \sqcap \leq 2.\mathsf{hasChild}$$

Their similarity in the knowledge base is computed as follows (let $\lambda = 1/3$):

$$s(C,D) = \tfrac{1}{3} \cdot [s_P(\mathsf{prim}(C), \mathsf{prim}(D)) + \tfrac{1}{2} \textstyle\sum_{R \in N_R} s(\mathsf{val}_R(C), \mathsf{val}_R(D)) + \\ + \tfrac{1}{2} \textstyle\sum_{R \in N_R} s_N((\mathsf{min}_R(C), \mathsf{max}_R(C)), (\mathsf{min}_R(D), \mathsf{max}_R(D)))]$$

Now, we compute the three parts separately:

$$\begin{aligned} s_P(\mathsf{prim}(C), \mathsf{prim}(D)) &= s_P(\{\mathsf{Person}\}, \{\mathsf{Male}\}) = \\ &= \frac{|\{\mathsf{Meg}, \mathsf{Bob}, \mathsf{Pat}, \mathsf{Gwen}, \mathsf{Ann}, \mathsf{Sue}, \mathsf{Tom}\} \cap \{\mathsf{Bob}, \mathsf{Pat}, \mathsf{Tom}\}|}{|\{\mathsf{Meg}, \mathsf{Bob}, \mathsf{Pat}, \mathsf{Gwen}, \mathsf{Ann}, \mathsf{Sue}, \mathsf{Tom}\} \cup \{\mathsf{Bob}, \mathsf{Pat}, \mathsf{Tom}\}|} \\ &= \frac{|\{\mathsf{Bob}, \mathsf{Pat}, \mathsf{Tom}\}|}{|\{\mathsf{Meg}, \mathsf{Bob}, \mathsf{Pat}, \mathsf{Gwen}, \mathsf{Ann}, \mathsf{Sue}, \mathsf{Tom}\}|} = 3/7 \end{aligned}$$

For the number restrictions on role $\mathsf{hasChild}$:

$$\begin{aligned} s_N((m_C, M_C), (m_D, M_D)) &= s_N((0,1),(0,2)) = \\ &= \frac{\min(1,2) - \max(0,0) + 1}{\max(1,2) - \min(0,0) + 1} = \frac{1 - 0 + 1}{2 - 0 + 1} = 2/3 \end{aligned}$$

For the number restrictions on role $\mathsf{marriedTo}$:

$$s_N((m'_C, M'_C), (m'_D, M'_D)) = 1$$

As regards the value restrictions on $\mathsf{marriedTo}$, $\mathsf{val}_{\mathsf{marriedTo}}(\mathsf{C}) = \mathsf{Person}$ and $\mathsf{val}_{\mathsf{marriedTo}}(\mathsf{D}) = \mathsf{Person} \sqcap \neg\mathsf{Male}$, hence:

$$s(\mathsf{Person}, \mathsf{Person} \sqcap \neg\mathsf{Male}) = 1/3 \cdot (s_P(\{\mathsf{Person}\}, \{\mathsf{Person}, \neg\mathsf{Male}\}) + 1 + 1)$$

and

$$\begin{aligned} s_P(\{\mathsf{Person}\}, \{\mathsf{Person}, \neg\mathsf{Male}\}) &= \frac{|\{\mathsf{Meg}, \mathsf{Bob}, \mathsf{Pat}, \mathsf{Gwen}, \mathsf{Ann}, \mathsf{Sue}, \mathsf{Tom}\} \cap \{\mathsf{Meg}, \mathsf{Gwen}, \mathsf{Ann}, \mathsf{Sue}\}|}{|\{\mathsf{Meg}, \mathsf{Bob}, \mathsf{Pat}, \mathsf{Gwen}, \mathsf{Ann}, \mathsf{Sue}, \mathsf{Tom}\} \cup \{\mathsf{Meg}, \mathsf{Gwen}, \mathsf{Ann}, \mathsf{Sue}\}|} \\ &= \frac{|\{\mathsf{Meg}, \mathsf{Gwen}, \mathsf{Ann}, \mathsf{Sue}\}|}{|\{\mathsf{Meg}, \mathsf{Bob}, \mathsf{Pat}, \mathsf{Gwen}, \mathsf{Ann}, \mathsf{Sue}, \mathsf{Tom}\}|} = 4/7 \end{aligned}$$

As there are no value restrictions on $\mathsf{hasChild}$, the similarity is maximal ($\mathsf{val}_{\mathsf{hasChild}}(\mathsf{C}) = \mathsf{val}_{\mathsf{hasChild}}(\mathsf{D}) = \top$).

Summing up:

$$\begin{aligned} s(C,D) &= \frac{1}{3}\left[\frac{3}{7} + \frac{1}{2}\left(\frac{1}{3}\left(\frac{4}{7}+1+1\right) + \frac{1}{3}\left(1+1+1\right)\right) + \frac{1}{2}\left(1+\frac{2}{3}\right)\right] \\ &= \frac{1}{3}\left[\frac{3}{7} + \frac{13}{14} + \frac{5}{6}\right] = \frac{92}{126} \simeq .7301 \end{aligned}$$

□

3.1 Discussion

It can be proven that s is really a similarity measure. (or *similarity function* [4]), according to the formal definition:

Definition 3.2 (similarity function). *Let S be a space of elements. A* similarity measure *f is a real-valued function defined on the set $S \times S$ that fulfills the following properties:*

1. $f(a,b) \geq 0 \quad \forall a,b \in S$ (positive definiteness)
2. $f(a,b) = f(b,a) \quad \forall a,b \in S$ (symmetry)
3. $\forall a,b \in S: \; f(a,b) \leq f(a,a)$

Proposition 3.1. *The function s is a similarity measure for the space $\mathcal{L}$.*

Proof. We have to prove the three properties:

1. *It is straightforward to see that s is positive definite since it is defined recursively as a sum of non-negative values.*
2. *s is also symmetric because of the commutativity of the operations involved, namely sum, minimum, and maximum (note that the value of s_N in Def. 3.1 does not change by exchanging C with D).*
3. *We must show that $\forall C,D \in \mathcal{L}: \; s(C,D) \leq s(C,C)$. This property can be proved by structural induction on D. The base cases are those related to primitive concepts and number restrictions, the inductive ones are those related to value restrictions and conjunctions:*
   - *if D is primitive then $s(C,D) = \lambda[s_P(\mathsf{prim}(C), \mathsf{prim}(D)) + s_1 + s_2] \leq$ $\lambda[\frac{|\bigcap_{P_C \in \mathsf{prim}(C)} P_C^{\mathcal{I}} \cap \bigcap_{Q_D \in \mathsf{prim}(D)} Q_D^{\mathcal{I}}|}{|\bigcap_{P_C \in \mathsf{prim}(C)} P_C^{\mathcal{I}} \cup \bigcap_{Q_D \in \mathsf{prim}(D)} Q_D^{\mathcal{I}}|} + 1 + 1] \leq$ $\lambda[\frac{|\bigcap_{P_C \in \mathsf{prim}(C)} P_C^{\mathcal{I}} \cap \bigcap_{P_C \in \mathsf{prim}(C)} P_C^{\mathcal{I}}|}{|\bigcap_{P_C \in \mathsf{prim}(C)} P_C^{\mathcal{I}} \cup \bigcap_{P_C \in \mathsf{prim}(C)} P_C^{\mathcal{I}}|} + 1 + 1] = \lambda[1+1+1] = s(C,C).$*
   - *if D is a number restriction the proof is analogous to the previous one, observing that* $0 \leq \frac{\min(M_C,M_D) - \max(m_C,m_D)}{\max(M_C,M_D) - \min(m_C,m_D)} \leq \frac{\min(M_C,M_C) - \max(m_C,m_C)}{\max(M_C,M_C) - \min(m_C,m_C)} \leq 1$

- *if D is a value restriction, then supposing by induction hypothesis that the property holds for descriptions whose depth is less than D's depth. This is the case of the sub-concept* $\mathsf{val}_R(D)$.
  Thus $s(\mathsf{val}_R(C), \mathsf{val}_R(D)) \leq s(\mathsf{val}_R(C), \mathsf{val}_R(C))$ *from which we may conclude that the property holds.*
- *if D is a conjunction of two simpler concepts, say* $\exists D_1, D_2 \in \mathcal{L} : D = D_1 \sqcap D_2$*, then assuming by induction hypothesis that the property holds for descriptions whose depth is less than D's depth such as* $D_{1,2}$*. This means that* $\forall i \in \{1, 2\} : s(C, D_i) \leq s(C, C)$*. It can be proven that* $\forall i \in \{1, 2\} : s(C, D) \leq s(C, D_i)$*. Hence the property holds.*

□

From a computational point of view, in order to control the computational cost of these functions, we may assume that the retrieval of the primitive concepts may be computed beforehand on the ground of the current knowledge base and then the similarity measure (or a derived dissimilarity measure) can be computed bottom-up through a procedure[6] based on dynamic programming.

3.2 Dissimilarity Measures Involving Individuals

Many machine learning algorithms (especially bottom-up ones) often require measuring the similarity between individuals. Also top-down ones are often based on a notion of *coverage* (instance checking) assessing the likelihood that an individual may belong to a concept by means of logic inferences or (somehow more simply) employing a notion of similarity between an individual and a concept description.

A dissimilarity measure can be easily derived from s in the following way:

Definition 3.3 ($\mathcal{ALN}$ dissimilarity measure). *The* dissimilarity function $d : \mathcal{L} \times \mathcal{L} \mapsto [0, 1]$ *is defined as follows. Given* $C, D \in \mathcal{L}$*:*

$$d(C, D) = 1 - s(C, D)$$

The notion of *Most Specific Concept* has been exploited for lifting individuals to the concept level [7]. On performing experiments related to a similarity measure exclusively based on concept extensions [9], we noticed that, resorting to the MSC, for adapting that measure to the individual to concept case, just falls short: indeed the MSCs may be too specific and unable to include other (similar) individuals in their extensions.

By comparing concept descriptions reduced to the normal form, we have given a more structural definition of dissimilarity. However, since MSCs are computed from the same ABox assertions, reflecting the current knowledge state, this guarantees that structurally similar representations will be obtained for semantically similar concepts. In fact, in this way, all equivalent concepts written using the

[6] This procedure has been implemented for instance-based learning algorithms as well as the measures proposed in [8, 10].

same subconcepts but using different descriptions, can be expressed in the same form.

Let us recall that, given the ABox, it is possible to compute the most specific concept of an individual a w.r.t. the ABox, $\mathsf{MSC}(a)$ (see Def. 2.2) or at least its approximation $\mathsf{MSC}^k(a)$ up to a certain description depth k. In the following we suppose to have fixed this k to the depth of the ABox, as shown in [16]. In some cases these are equivalent concepts but in general we have that $\mathsf{MSC}^k(a) \sqsupseteq \mathsf{MSC}(a)$.

Given two individuals a and b in the ABox, we consider $\mathsf{MSC}^k(a)$ and $\mathsf{MSC}^k(b)$ (supposed in normal form). Now, in order to assess the dissimilarity between the individuals, the d measure can be applied to these concept descriptions, as follows:

$$d(a, b) := d(\mathsf{MSC}^k(a), \mathsf{MSC}^k(b))$$

Analogously, the dissimilarity value between an individual a and a concept description C can be computed by determining the (approximation of the) MSC of the individual and then applying the dissimilarity measure:

$$\forall a: \ d(a, C) := d(\mathsf{MSC}^k(a), C)$$

These cases may turn out to be particularly handy in several tasks, namely both in inductive reasoning (construction, repairing of knowledge bases) and in information retrieval.

4 Final Remarks

Similarity and distance measures turn out to be useful in several tasks such as classification, case-based reasoning, clustering, etc. A novel (dis)similarity measure has been introduced, based on the information on concepts and roles as it can be approximated on the grounds of the underlying semantics of the ABox.

We have also shown how to apply this function to measuring the (dis)similarity between individuals and also between individual-concept (useful in knowledge discovery tasks). In particular, defining a measure, that is applicable for comparing both concepts and individuals, is suitable for agglomerative and divisional clustering. A further investigation will concern the derivation of a distance measure, which amounts to finding a measure that fulfils the triangular property.

The presented measure can be refined introducing a weighting factor, useful for decreasing the impact of the dissimilarity between nested sub-concepts in the descriptions on the determination of the overall value.

Another natural extension may concern the definition of dissimilarity measures in more expressive languages. For example, a normal form for $\mathcal{ALCN}$ can be obtained based on those for $\mathcal{ALN}$ and $\mathcal{ALC}$. Then, by exploiting the notion of existential mappings [15], already used for computing the LCS in $\mathcal{ALCN}$, the measure presented in this paper may be extended to the richer DL.

Kernels are another means to express the similarity in some unknown feature space. We are working at the definition of kernel functions on DLs representations, thus allowing the exploitation of the efficiency of kernel methods (e.g. support vector machines) in a relational setting.

References

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[2] F. Bacchus. Lp, a logic for representing and reasoning with statistical knowledge. *Computational Intelligence*, 6:209–231, 1990.

[3] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[4] H.H. Bock. *Analysis of Symbolic Data: Exploratory Methods for Extracting Statistical Information from Complex Data*. Springer-Verlag, 1999.

[5] A. Borgida, T.J. Walsh, and H. Hirsh. Towards measuring similarity in description logics. In *Working Notes of the International Description Logics Workshop*, CEUR Workshop Proceedings, Edinburgh, UK, 2005.

[6] M. W. Bright, A. R. Hurson, and Simin H. Pakzad. Automated resolution of semantic heterogeneity in multidatabases. *ACM Transaction on Database Systems*, 19(2):212–253, 1994.

[7] W.W. Cohen and H. Hirsh. Learning the CLASSIC description logic. In P. Torasso, J. Doyle, and E. Sandewall, editors, *Proceedings of the 4th International Conference on the Principles of Knowledge Representation and Reasoning*, pages 121–133. Morgan Kaufmann, 1994.

[8] C. d'Amato, N. Fanizzi, and F. Esposito. A dissimilarity measure for concept descriptions in expressive ontology languages. In H. Alani, C. Brewster, N. Noy, and D. Sleeman, editors, *Proceedings of the KCAP2005 Ontology Management Workshop*, Banff, Canada, 2005.

[9] C. d'Amato, N. Fanizzi, and F. Esposito. A semantic similarity measure for expressive description logics. In A. Pettorossi, editor, *Proceedings of Convegno Italiano di Logica Computazionale (CILC05)*, Rome, Italy, 2005. `http://www.disp.uniroma2.it/CILC2005/downloads/papers/15.dAmato_CILC05.pdf`.

[10] C. d'Amato, N. Fanizzi, and F. Esposito. A dissimilarity measure for $\mathcal{ALC}$ concept descriptions. In *Proceedings of the 21st Annual ACM Symposium of Applied Computing, SAC2006*, Dijon, France, 2006.

[11] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. *Information and Computation*, 134(1):1–58, 1997.

[12] F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Deduction in concept languages: From subsumption to instance checking. *Journal of Logic and Computation*, 4(4):423–452, 1994.

[13] J.-U. Kietz. Learnability of description logic programs. In S. Matwin and C. Sammut, editors, *Proceedings of the 12th International Conference on Inductive Logic Programming*, volume 2583 of *LNAI*, pages 117–132, Sydney, 2002. Springer.

[14] R. Küsters and R. Molitor. Approximating most specific concepts in description logics with existential restrictions. In F. Baader, G. Brewka, and T. Eiter, editors, *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence, KI/ÖGAI01*, volume 2174 of *LNCS*, pages 33–47. Springer, 2001.

[15] R. Küsters and R. Molitor. Computing least common subsumers in $\mathcal{ALEN}$. In B. Nebel, editor, *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI2001*, pages 219–224, 2001.

[16] T. Mantay. Commonality-based ABox retrieval. Technical Report FBI-HH-M-291/2000, Department of Computer Science, University of Hamburg, Germany, 2000.

[17] R. Molitor. Structural subsumption for $\mathcal{ALN}$. Technical Report LTCS-98-03, LuFg Theoretical Computer Science, RWTH Aachen, Germany, 1998.

[18] OWL. Web Ontology Language Reference Version 1.0, 2003. `http://www.w3.org/TR/owl-ref`.

[19] P. Resnik. Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language. *Journal of Artificial Intelligence Research*, 11:95–130, 1999.

[20] A. Rodriguez. *Assessing semantic similarity between spatial entity classes*. PhD thesis, University of Maine, 1997.

[21] M. A. Rodríguez and M. J. Egenhofer. Determining semantic similarity among entity classes from different ontologies. *IEEE Transaction on Knowledge and Data Engineering*, 15(2):442–456, 2003.

[22] C. Rouveirol and V. Ventos. Towards learning in CARIN-$\mathcal{ALN}$. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *LNAI*, pages 191–208. Springer, 2000.

[23] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics*, IV:146–171, 2005.

[24] A. Tversky. Features od similarity. *Psycological Review*, 84(4):327–352, 1997.

[25] P. Weinstein and P. Bimingham. Comparing concepts in differentiated ontologies. In *Proceedings of 12th Workshop on Knowledge Acquisition, Modelling, and Management*, 1999.

Prof. Francesco M. Donini

UNIVERSITA' DELLA TUSCIA – VITERBO
Dipartimento di Scienze della Comunicazione
Facoltà di Scienze Politiche

Knowledge Representation Tools for Electronic Commerce

The graft of Semantic Annotation into Electronic Commerce brings new opportunities for the application of Knowledge Representation techniques originally devised for Knowledge Bases.
In this talk, I tackle two phases of an Electronic Commerce transaction, namely matchmaking and negotiation, and argue that – in addition to standard Deductive Reasoning, and Decision Theory – some non-standard forms of reasoning are present when accomplishing matchmaking and negotiation: Abduction, Belief Revision, Preference-based reasoning. However, the use of these forms of reasoning is not the same as their use in Knowledge Bases. I point out the changes that are needed, and sketch open problems and possible lines of research.

Semantic-based matchmaking and query refinement for B2C e-marketplaces

Simona Colucci[1], Tommaso Di Noia[1], Eugenio Di Sciascio[1], Francesco M. Donini[2], Azzurra Ragone[1], Raffaele Rizzi[1],

[1] Politecnico di Bari, Via Re David, 200, I-70125, Bari, Italy
{s.colucci,t.dinoia,disciascioa.ragone}@poliba.it,
raffaele@raffaelerizzi.com
[2] Università della Tuscia, via San Carlo, 32, I-01100, Viterbo, Italy
donini@unitus.it

Abstract. We present an application in the framework of semantic-enabled e-marketplaces aimed at fully exploiting semantics of supply/demand descriptions in B2C and C2C e-marketplaces. Distinguishing aspects of the framework include logic-based explanation of query results, semantic ranking of matchmaking results, logic-based request refinement.

1 INTRODUCTION

In this paper we present a web application for semantic discovery and selection of products in a B2C e-marketplace. The main objective we tackle is providing users with benefits of semantic annotation, including richness of descriptions, semantic matchmaking and logic-based ranking and explanation services, while hiding from them all technicalities and letting users experience interaction with the system in an immediate and user friendly way. The query formulation process is very important for the success of a retrieval system, especially an ontology-based one. The query language has to be very simple for the end user but, at the same time, its expressiveness must be able to capture the real user needs and retrieve only what the user is really looking for. Users are often unable to use logic formulas needed to use a formal ontology [2], they need visual representation to manipulate the domain of interest similarly to what Visual Query Systems do [4].

It is well-known that a challenge for B2C e-marketplaces is to match resources in the e-marketplace to potential buyer's interests, but also to present available goods in an appealing manner, facilitating exploration and selection of product characteristics. As pointed out in [19], selecting a product to buy in e-marketplaces is usually quite a frustrating experience: finding products best fitting users needs and/or financial capabilities often requires too much effort and time, spent browsing web sites or taxonomies in the web sites. Especially when the searched product is not a perfectly defined item, users may have a vague idea of what they are actually looking for, being unaware of all the characteristics of the product. Searching for a product or service often requires domain knowledge that users do not have, so that many potential buyers tend to prefer traditional sales channels, such as physical stores where shop assistants can help the customer to make the right choice and answer to users requests or doubts.

A central issue in e-commerce is hence to support the user in the searching process of the products or services: converting site visitors to buyers in e-commerce environments is a recognized challenging subject [18].

The promise of the Semantic Web is to make information available on the web machine-understandable. By means of formal ontologies, modeled using OWL[16], the knowledge on specific domain can be modeled and exploited in order to make explicit the implicit knowledge, and reason on it thanks to the formal semantics expressed in OWL. Since its launch, the semantic Web initiative has attracted several researchers, has raised big money in research projects, has provided many useful insights in knowledge representation, but up to now has provided few working applications. Obviously, semantic web technologies open extremely interesting new scenarios, including: formalization of annotated descriptions that are machine understandable and interoperable, without being biased by usual drawbacks of natural language expressions; the possibility to reason on descriptions and infer new knowledge; the validity of the Open World Assumption, overcoming limits of structured-data models. There are several reasons for this strange situation: the annotation effort is considerable, though promising results are being obtained on automated extraction and ontology mapping and merging [20]; computational complexity is often demanding also for simple reasoning tasks; interaction with semantic-based systems is often cumbersome and requires skills that most end users do not have –and are not willing to learn.

Furthermore we believe that the effort of annotation should be rewarded with inferences smarter than purely deductive services such as classification and satisfiability, which, although extremely useful show their limits in approximate searches. The question ”show us something useful you could not do without semantic web technologies” starts to come up often now.

In this paper we face some of the above issues in the framework of semantic-enabled e-commerce and present an approach and a system, which allow using a fully graphical user interface semantic-based matchmaking and retrieval of offers –and query refinement– in an intuitive way.

Main features of our approach are: full exploitation of non-standard inferences for explanation services in the query-retrieval-refinement loop; semantic-based ranking in the request answering; fully graphical and usable interface, which requires no prior knowledge of any logic principles, though fully exploiting it in the back-office. Modeling the marketplace domain using an OWL ontology, the user is able to browse the domain knowledge starting form ”her vague idea” on the good she wants to buy. Once the request of the user is formalized with respect to the domain ontology, its formal relations are exploited in order to find goods within the marketplace able to satisfy her needs. Based on the formal semantics of both the request and the returned goods descriptions, an explanation of the matchmaking process results is then provided to the user, who can simply use such new knowledge to refine or change, in a visual environment, her request

The remaining of the paper is structured as follows: next section outlines common sense user needs to be satisfied by an e-marketplace. In Section 3 motivations for a semantic based approach for matchmaking in e-marketplaces are presented together with a brief summary of semantic-based matchmaking in Description Logics. The descrip-

tion of an ontology-based web application satisfying common sense user needs is then presented. Related work and conclusion close the paper.

2 Identifying common sense user needs

We present, with the aid of real–life example, some of the issues an E-marketplace system should successfully face to help satisfying users needs. Here and in the rest of the paper we focus on an automotive domain and on a car E-marketplace, though the approach is obviously general.

"*Gioseppe has been hired by a new company. He loves the new job, the salary is good. But there is a drawback: the company is in an isolated place, 20 Km far from his city and there is no bus connection. So he needs a car immediately. Then Gioseppe goes to a used car seller, sets his budget, and asks for a car endowed of good safety features – he has to travel up and down for 40 Kms a day – but absolutely the color must not be yellow. He likes Italian style so he preferably would like a car of an Italian manufacturer. Based on these requirements, the shop dealer proposes to Gioseppe some cars he has in stock. While examining proposals, Gioseppe discovers new characteristics he had not asked for, he now considers interesting. Then he reformulates his request and asks for a car of an Italian make endowed of ABS system, airbags, as before excluding yellow color, but now he also would like leather seats and an alarm system. Again the dealer proposes a new set of cars but they do not satisfy Gioseppe so he decides to look also for non-Italian cars, even if he continues to desire an Italian one. He continues refining his requirements until he finally finds, among all the available ones, the car satisfying his needs.*"

The above description illustrates what is the level of interaction humans expect when they interact with other humans. In which ways an automated B2C e-markeplace can provide comparable levels of interaction?

Support to the user in the searching process. Facilitate browsing and selection of product characteristics. The selection of the product really fulfilling user requirements should be the goal of any system for e-commerce. In a real store, users can rely on shop assistants who can help in choosing among various brands or various models. Often a user may not know exactly what she is looking for as she enters a shop, because of lack of specific domain knowledge *e.g.*, she may want to buy a car, but she does not know all the features or optionals a car has.

In the same way, starting from your initial and vague idea on what you wish to buy, you may go around supermarket aisles and then find the product that best fits your wishes: a user in an e-marketplace could "discover" some new unknown product characteristics searching for the ones she already knew. A system should support users in the searching process starting from incomplete information they have on the product to be bought/sold. It should manage incomplete information in the user requirements and also in the products descriptions. The (potential) buyer, especially in the initial stage, might be not aware of all the possible characteristics she can specify for a particular product. As said before, this may happen both because the user could be not a domain expert and because she has not an "exact" idea on what

she wants to buy. On the other hand, the individual who describes the items in the marketplace decides the characteristics she wants to emphasize in the description. The system should support users in the elicitation of their needs, in order to refine the initial query and reflect their true needs or wishes. One of the main problems with *preference elicitation* is that preferences expressed by user on the initial stage of the search process can be uncertain and erroneous so preference elicitation process has to be part of the searching process, as preferences can depend on partial search results [2]. Moreover the process of eliciting preferences should not require neither much effort on the user's side or force him to set a large number of weights on items. It is well-assessed that assigning a precise and explicit value (weight) to each item, especially when the number of items increase, is extremely frustrating for users.

Efficiency and trust . A key issue for a system supporting users in e-marketplace is not simply finding *a* product, but finding the *right* product[19], or a set of products, best matching the user needs. Furthermore users should be *confident* that the system has made the best choice for them. A simple presentation of a list of items after a query may be not sufficient to convince the user they are the best choice. In the same way a user may not be convinced they are proposed because of their match degree with the request or because some other "unclear" factor. Explanations on match degree may help in both cases.

Ranking criteria . Often products in e-marketplaces, or in general in Internet shopping malls, are compared only through their *price*. The most common tools for product selection over internet sites present offers as an ordered list, sorted according to their price, or more generally in increasing order of a quantitative attribute. The limit is here the possibility to choose each time only one criterion to display the results [18]. Nevertheless, price is not the only characteristic to be considered in the request unless the item has been perfectly identified. Users preferences are expressed over a set of attributes describing the good to be bought. Considering only one criterion is not realistic. For instance, an E-marketplace supporting the sale of cars has to model different features, as look, comfort, optionals, model and not only the price, quantity or delivery time. So a system should be able to provide overall rankings according to users requests.

Friendliness . an e-marketplace system should not need any specific skill or learning effort to be immediately usable also by non expert-users. Experience in information retrieval shows that users may encounter problems even with simple text-based Boolean expressions, by far preferring graphical query interfaces [2]. [3]

3 Why ontologies?

Currently, the approaches adopted by web-based tools in order to retrieve resources within a repository – as an e-marketplace can be seen as a particular repository – are mostly either database oriented or text retrieval based, while a minority of them uses taxonomies. Why are they not enough? What do they miss with respect to the user

[3] In this paper we are not specifically interested in a usability analysis and do not perform any usability tests, which is part of future work.

needs introduced in the previous section? Database systems are extremely efficient in handling huge amounts of data (items). The state of the art in database systems allows the management of very big marketplaces whose items expose many characteristics. A drawback of such an approach is in its "closed world" assumption. What is not specified is considered as a constraint of absence, as a negation by default approach is used. It is not hence possible to manage incomplete information. Using a database is also not possible to have explanations on the results presented. Using text retrieval based techniques, well known problems of noise and bad recall have to be taken into account [3]. It is difficult to find the best match and if some heuristics are used to refine the results, the system does not present any explanation on them to clarify system behavior. Taxonomies are very useful to browse classes of items. Each node in a taxonomy can represent a set of items sharing a common characteristic. But, once this initial set of items has been found, it is not possible to use the taxonomy to refine the query.

Besides the above mentioned limitations, all these approaches lack of the possibility to deal with the semantics of the descriptions – both the user request and resources descriptions; a very useful feature in the search process. In taxonomy-based approaches a very basic semantic search (IS-A relation between category in the tree) is presented, but it results very weak. We believe that especially in e-marketplaces, the "meaning" of the terms rather than the terms themselves is very important. Turning back to Gioseppe, if he was looking for a `safe car`, then a car endowed with `ABS system` and `airbags` would be a good choice. In order to catch these logical correlation, ontologies [13] would help **Gioseppe** in the search process. An ontology allows to relate terms with each other and give a formal model to the knowledge of the marketplace domain, and consequently express that a `safe car` is a `car` `endowed_with` an `ABS_system` and `endowed_with` `airbags`. Exploiting the formal semantics of the language used to build the ontology, logic based inference processes can be performed, successfully dealing also with incomplete information (Open World Assumption – OWA). Based on such inference services an efficient retrieval process can be carried out.

Nevertheless, using standard deductive inference services only exact matches can be identified. Neither logical ranking nor explanation services on resources discarded during the search process are available, as the reasoning engine behaves as a boolean oracle. As in database systems, a list of results is presented to the user apparently without any justification.

To overcome such limitations, in [8] an ontology based approach has been proposed exploiting abductive inference services and belief revision techniques [7] in a Description Logics based framework. Using these services both explanation on the results can be provided to the user and new knowledge elicitation can be performed in order to guide the user in her query refinement.

3.1 Semantic Based Matchmaking

A close relation exists between OWL and Description Logics. In fact, the formal semantics of OWL DL sub-language is grounded in the Description Logics theoretical studies.

For the sake of completeness, we briefly recall standard inference services in Description Logics (DLs) and Concept Abduction and Concept Contraction services, showing how they can be used in a matchmaking process for match explanations and knowledge elicitation. We assume the reader be familiar with the basics of Description Logics [1].

Match Classes. Given an ontology $\mathcal{T}$ and two DLs formulas D (for demand) and S (for supply), two standard inference services are provided by a DL reasoner:

Subsumption : Check if S is more specific than D with respect to the ontology $\mathcal{T}$. In a formal way: $\mathcal{T} \models \mathsf{S} \sqsubseteq \mathsf{D}$.

Satisfiablity : Check if S (conversely D) is satisfiable with respect to the ontology $\mathcal{T}$. In a formal way: $\mathcal{T} \not\models \mathsf{S} \sqsubseteq \bot$.

Based on these standard inferences, given a request D (for demand) and a resource S (for supply) the following match classes can be identified with respect to an ontology $\mathcal{T}$[10, 14, 17].

exact $\mathcal{T} \models \mathsf{D} \equiv \mathsf{S}$. S *is semantically equivalent to* D. All the characteristics expressed in D are presented in S and S does not expose any additional characteristic with respect to D.

full $\mathcal{T} \models \mathsf{S} \sqsubseteq \mathsf{D}$. S *is more specific than* D. All the characteristics expressed in D are provided by S and S exposes also other characteristics both not required by D and not in conflict with the ones in D.

plug-in $\mathcal{T} \models \mathsf{D} \sqsubseteq \mathsf{S}$. D *is more specific than* S. All the characteristics expressed in S are provided by D and D requires also other characteristics both not exposed by S and not in conflict with the ones in S.

potential $\mathcal{T} \not\models \mathsf{S} \sqcap \mathsf{D} \sqsubseteq \bot$. D *is compatible with* S. Nothing in D is logically in conflict with anything in S.

partial $\mathcal{T} \models \mathsf{S} \sqcap \mathsf{D} \sqsubseteq \bot$. D *is not compatible with* S. Something in D is logically in conflict with some characteristic in S.

Actually, it is questionable whether a **plug-in** match type should be considered better than **potential** one. We note that some researchers also consider **plug-in** match more favorable than **full** match (*e.g.*, see [14, 17]), motivating this choice with the idea that if D *is more specific than* S one may expect that the advertiser offering resource S will probably have also more specific resources; in an e-commerce setting if the advertiser offers a *sedan* car it will also probably offer specific types of *sedans*. Nevertheless we argue that this idea prevents a fully automated matchmaking, which is possible when S *is more specific than* D, and furthermore it favors underspecified resource description, *i.e.*, an advertisement offering a sedan will always plug-in match any request for a specific sedan, but will leave on the requester the burden to determine the right one – if any is actually available – for her needs. Even though exact match is surely the best match, full match might be considered –not always anyway– equivalent from the requester point of view, because it states that at least all the features specified in D are also expressed in S. We can give a rank to the match classes:

$$\textbf{partial} \rightarrow \textbf{potential} \rightarrow \textbf{full} \rightarrow \textbf{exact}$$

Largest part of logic-based approaches only allow, as pointed out before, a categorization within match types.

Non Standard Inference Services for Logical Matchmaking. Notice that even if **exact** and **full** matches are obviously the best possible matches, in resource retrieval scenarios the most frequent cases are **potential** and **partial** matches. We can evaluate a score for **potential** and **partial** matches considering their distance from a **full** match and explain the match degree proposing: (a) in case of **partial** match, which characteristics have to be retracted from D in order to reach a **potential** match with S; (b) in case of **potential** match, what is not specified[4] in S in order to be more specific than $\mathsf{D}$and then have a **full** match. The knowledge elicitation for query refinement can be performed evaluating what is exposed in S and is not required by D. In order to perform these evaluation two non-standard inference services for DLs can be exploited. We briefly recall them.

Given two DLs formulas D and S both satisfiable with respect to an ontology $\mathcal{T}$

Concept Contraction : If D and S are not compatible with each other $-\mathcal{T} \models \mathsf{D} \sqcap \mathsf{S} \sqsubseteq \bot-$ find two DLs formulas G (for give up) and K (for keep), such that both $\mathcal{T} \models \mathsf{D} \equiv \mathsf{G} \sqcap \mathsf{K}$ and $\mathcal{T} \not\models \mathsf{K} \sqcap \mathsf{S} \sqsubseteq \bot$.

Concept Abduction : If S is not more specific than D $-\mathcal{T} \not\models \mathsf{S} \sqsubseteq \mathsf{D}-$ find a formula H (for hypotheses) satisfiable with respect to $\mathcal{T}$and such that $\mathcal{T} \models \mathsf{S} \sqcap \mathsf{H} \sqsubseteq \mathsf{D}$.

(partial → potential) Hence, if D and S are in a **partial** match, solving a Concept Contraction it is possible to compute G representing why D is in conflict with S and K representing the new contracted request. After the contraction we have K in potential match with S.

(potential → full) If D and S are in **potential** match, solving a Concept Abduction problem, hypotheses H on why there is not a **full** match between D and S are computed. Hence the conjunction $\mathsf{S} \sqcap \mathsf{H}$ is a **full** match for D.

Using Concept Contraction and Concept Abductions is then possible to compute and explain how far is a **partial** match or a **potential** match from a **full** match.

If D and S are in potential match, the characteristics B (for bonus)[9] specified in S but not requested in D represent the knowledge that can be elicited and proposed to the requester in order to still refine the initial query. At this point it should be easy to see how B can be computed solving a Concept Abduction problem.

In [8] we proposed a formalization on how to deal with **strict** and **negotiable** characteristics, in a DL framework. Roughly speaking, the demander sets a characteristic as strict, if she never wants to give up that strict feature. Then all the resources that are in a partial match because of the strict constraints have to be discarded (see [8] for further details). If all the request characteristics are set strict, then only potential matches are allowed. This case models the situation where the user is not willing to contract any part of her request in order to reach a potential match within the marketplace. Typically in a refinement iterative process this is the case when the user formulate her initial query.

[4] We write "not specified" instead of "missing" in order to emphasize the underlying Open World Assumption.

In fact, if she is not satisfied by the system results to her first query, then she starts to negotiate on some constraints in order to find appealing offers.

4 I would like to buy a car, but...

Based on the theoretical framework presented in the previous sections we developed an application fully exploiting semantic-based matchmaking procedures and able to satisfy user needs specified in Section 2.[5] The information presented within the system interface is ontology independent, *i.e.*, it is built on the fly once the reference ontology –hence the chosen marketplace domain– has been selected (Figure 1).

Fig. 1. Marketplace/Ontology Selection

Then, if the marketplace changes, the new ontology can be loaded to dynamically present the new available domain knowledge to the user. What the user see within the GUI is a visual representation of the knowledge modeled by the ontology. Actually, the effectiveness of how the information is visualized is strongly dependent on the quality of the ontology. As pointed out also in [11], it is very difficult to manage the visualization of a poorly structured ontology.
Here, we are not interested in the marketplace population. Once the ontology is selected, the corresponding marketplace and all its semantically annotated supplies become available for the discovery process.

4.1 The Tool

When the application starts, the shown GUI is divided in two main sections: the left-side one (Figure 2(a)(b)(d)), from now on navigation panel, is devoted to ontology brows-

[5] http://sisinflab.poliba.it/marketplace/

ing – intensional navigation [5], the right-side one (Figure 2(c)(e)), from now on query panel, to graphically visualize the user query. The **navigation panel** is divided in two

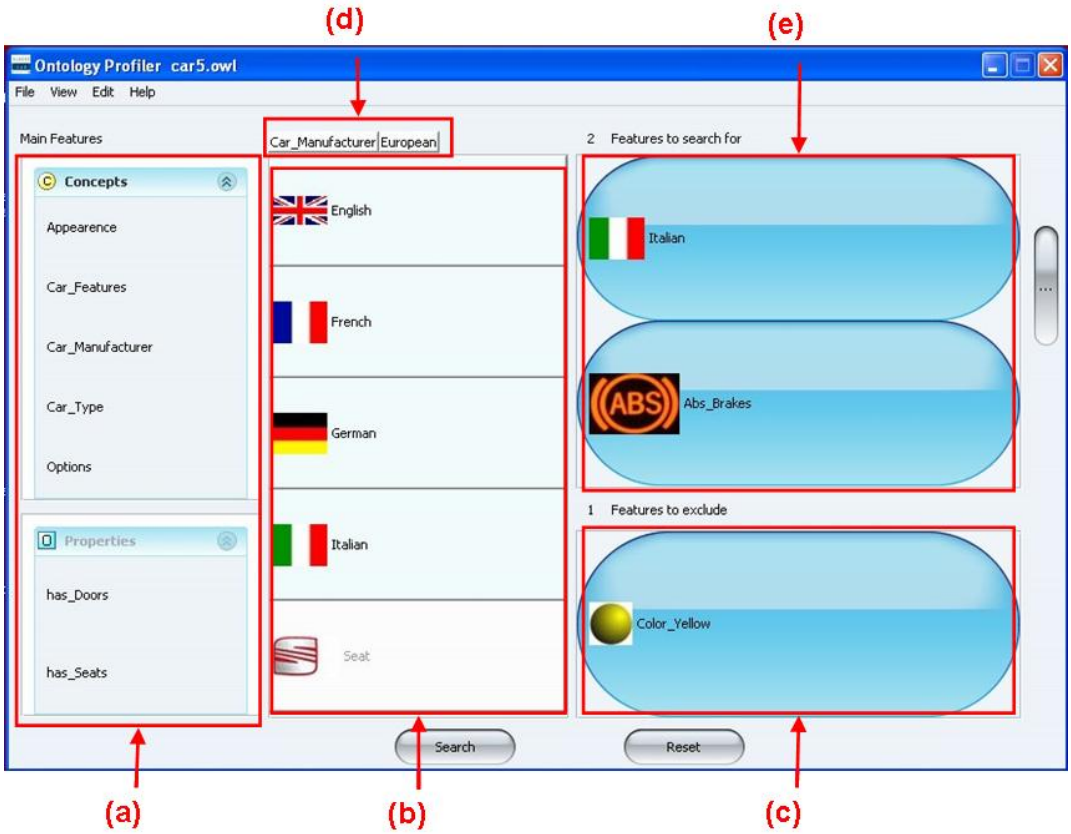


Fig. 2. The Graphical User Interface for intensional navigation and query formulation

main panels. In most-left one (Figure 2(a)) the entry-points for the intensional navigation are represented in the top side. The navigational approach is *top-down*. Initial characteristics the user is able to select in order to represent her query are only the most generic ones within the ontology – that is, all those classes which are direct children of the `<owl:Thing/>` class. By selecting one of these classes, both all its sub-classes and the roles having the selected class as domain are visualized within the right side of the visualization panel (Figure 2(b)). The user starts from a general aspect of the domain and then sees a deep aspect of it with recursive zooms on the ontological model. Clicking on one of these just visualized characteristics, the intensional navigation continues recursively exploiting the sub-class or the domain/range relations and the new information is visualized within the same panel. What the user sees in the navigation panel are local views of the ontology. Doing so the user is able to see only the current

focus of her search in the right side of the navigation panel and is able to change the entry point if she decides to look for something different. To help the user in the navigation and to come back to an upper level – zoom out – a history bar is visualized on the navigation panel (Figure 2(d)). The information of the visualized characteristic within the GUI, is not just the name of either the class or the role within the ontology. Exploiting `<rdfs:comment/>` meta-information within an OWL file we associate an image to each class and role name and show it as an icon of the class/role. Moreover `<rdfs:label/>` is used to manage multilingual information.

If the user finds a characteristic she is looking for, she drags it in the **query panel** and add it to her final query. If the dragged elements is a `<owl:SameClassAs/>`, then its definition is added to the query panel. The query panel is divided in two sections, so that the user can express preferences both positive and negative. In the top side (Figure 2(c))the user drags characteristics she would like the retrieved supplies own, in the bottom side (Figure 2(e)) the user drags the characteristics she absolutely does not want in the retrieved supplies. Again we stress that under an OWA the negative information must be clearly stated. The user only sees atomic characteristics to be added or removed from the query panel. All the relations between these characteristics are coded within the ontology and completely hidden to her. As well as the user is able to add elements to the query panel, she can remove them just right-clicking on the corresponding item. It is noteworthy that in the initial query all the characteristics are set strict.

Once the user formulates the query, the matchmaking process is performed with all the supplies semantic-enabled descriptions within the marketplace helped by a reasoner exploiting the formal semantics of both the query and supplies descriptions.
The reasoner is not embedded within the tool. This one communicates with the inference engine via a DIG 1.1 interface over HTTP. Since the tool exploits both standard and non-standard inference services as presented in Section 3.1 we use **MaMaS**[6] reasoner system, which exposes a standard DIG 1.1 interface enhanced with additional tags to support the above mentioned services.
The matchmaking results are shown in a **results window**. The information within this window is both related to a *ranked list* of appealing supplies – with respect to the query – within the marketplace, *explanation* on the match results and *suggestions* on how to refine the query adding new characteristics found in the retrieved offers but not specified in the user request. The results window is shown in Figure 3: in the left side – **list panel**, the ranked list endowed with match explanation for each retrieved supply is represented (Figure 3(a)(b)); in the right side – **query refinement panel**, a visualization of the query is presented in the top side (Figure 3(c)) and suggestions on characteristics to be added to the query in the bottom side (Figure 3(d)).
In the **list panel**, for each retrieved item, the system provides the information detailed in the following:

description An image representing the retrieved item together with a natural language description of the item itself. The system provides also a transliteration of the semantic-based supply description. The verbalization of the OWL description is provided automatically by the system [5].

[6] http://sisinflab.poliba.it/MAMAS-tng/

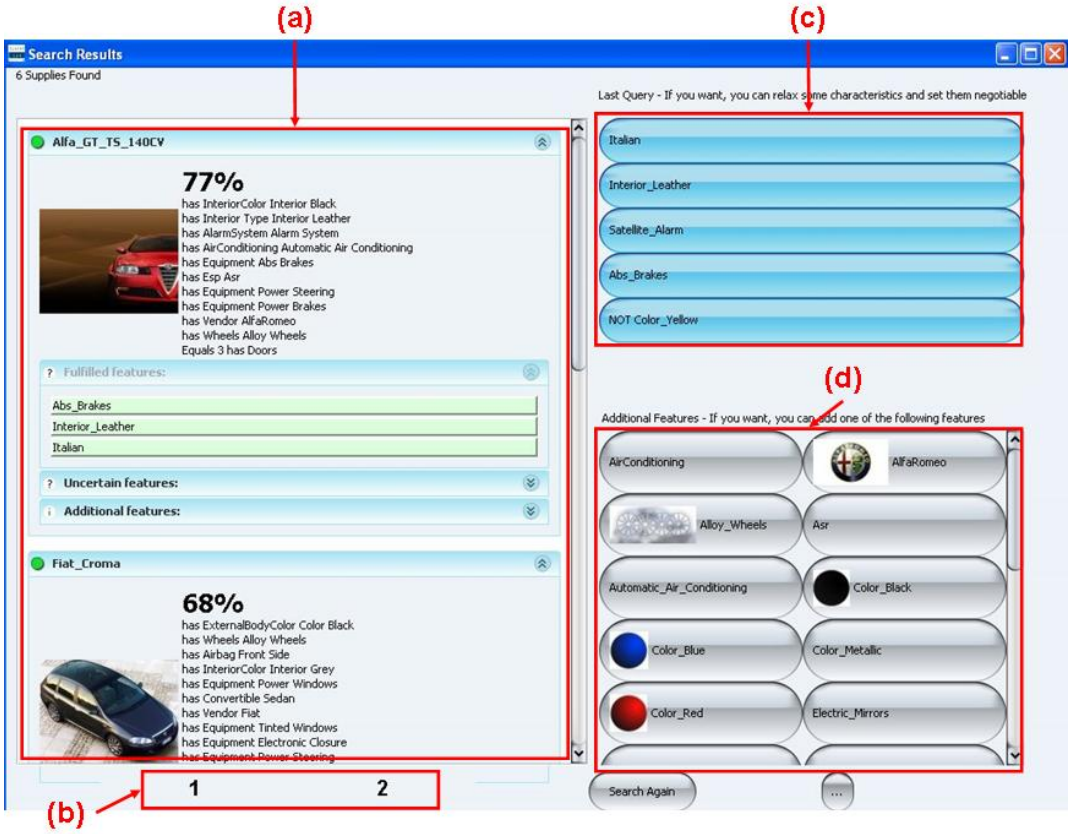


Fig. 3. The results window

match value Based on the semantics of both the query and the offers descriptions, a semantic similarity value is computed [8]. This value is used to rank the results list.

match explanation A semantic based explanation for the match result is displayed. Fulfilled, unspecified, conflict and additional characteristics, with respect to the request, are displayed for each item. Obviously, if all the the query characteristics are set to strict, the conflicting elements set is empty and is not displayed. Additional features represent what is not specified in the query but is specified in the offer.

The list panel has a multi-page visualization (Figure 3(b)). For each page only five items are displayed.

The **query refinement panel** is divided in two sections: in the top side panel the query is visualized, in the bottom side all the additional information – bonus – related to the offers visualized in the current page of the list panel, are displayed (Figure 3(d)). The query refinement panel allows the user to refine the query in two different ways: relaxing some characteristics setting them to negotiable or adding new characteristics from the additional ones of the currently displayed supplies. If the user set the characteris-

tic to negotiable (gray colored features in the top side panel – Figure 4), then also the supplies exposing a characteristic in conflict with the negotiable ones are taken into account during the matchmaking process. Notice that setting characteristics to negotiable is not equivalent to removing them from the query. It is not a *don't care* specification. A negotiable characteristic has to be interpreted as a *wish* specification. That is why, also supplies in conflict with the negotiable features are considered during the matchmaking process and ranked in the final result list. The bonus characteristics in the query refinement panel represent information the user might not be aware of or she initially considers not relevant for the search. Nevertheless, it is related to what the user is looking for. If the user asks for a `sedan` and the retrieved supplies expose among additional features the `air conditioning`, then the user could be interested in this bonus exposed by some sedan cars.
Once the user adds new features to the query selecting them from the additional ones or she sets some characteristics to negotiable, a new search can start based on the new refined query. The process can be iterated until the user finds a supply best fitting her desires.

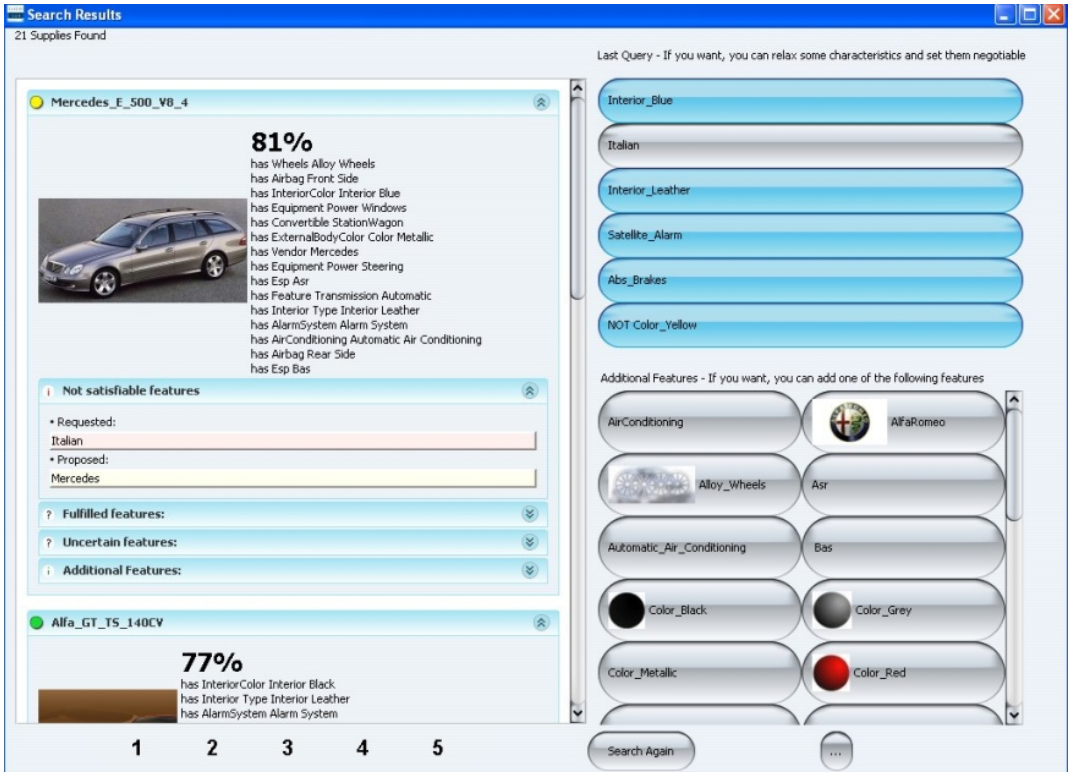


Fig. 4. The results window after a query refinement

4.2 The matchmaking process

In this section we describe the matchmaking process, *i.e.*, what happens behind the scenes during the search process after the query (re-)formulation.

1. Thanks to the intensional navigation the user formalizes her request with respect to the ontology. In this initial query all the requested characteristics are set strict.
2. All the supplies within the marketplace compatible with the request, *i.e.*, in potential match with it, are retrieved. For each of them, solving concept abduction problems via **`MaMaS`**, fulfilled, uncertain and additional feature are computed based on the knowledge modeled within the ontology. They represent respectively: which part of the request is also present in the supply description; what is requested by the user but is not specified in the supply description; the bonus characteristics. Based on fulfilled and uncertain characteristics, a semantic-based match value is computed.
3. All the retrieved supplies are ranked with respect to their semantic-based match value and then grouped in sets of five elements each. The first group of supplies is displayed in the list panel. All the additional features related to these supplies are put together and displayed in the bottom side of the query refinement panel. If the user selects another group/page (Figure 3(b)) then the bottom side of the query refinement panel is updated with the bonuses related to the new displayed supply.
4. If the user does not find any supply satisfying her needs, she can refine the query.
5. After the query refinement, a new retrieved process is performed. All the supplies in conflict with the new strict characteristics are discarded. For the remaining supplies, if they are in partial match with the query then, for each of them, contraction problems are solved to compute both a contracted request (see Section 3.1) which is in potential match with the supply and conflicting features. Concept abduction problems are then solved in order to compute fulfilled, uncertain an bonus characteristics with respect to the contracted request. Based on conflicting, fulfilled and uncertain characteristics the semantic-based match value is computed.
6. The process restarts form point 3.

4.3 User needs satisfaction

Turning back to the user requirements and needs outlined in Section 2, we now explain how the proposed tool satisfies them.

Support to the user in the searching process. Using intensional navigation, the user is guided through the exploration of the marketplace knowledge domain. Even if she is not a expert, she can start from very general concepts of the domain and discover what she is really looking for. Furthermore, since the user sees only local views of the ontology, she can focus only on it in each step of the query formulation.

The preference elicitation is managed by the tool using bonus characteristics. In fact since only the additional information related to the current page in the list panel is shown, then if the user is interested in one the supply presented in that page, probably she is also interested in their bonus features.

Giving the opportunity to set negotiable some characteristics in the query, the user is helped in expressing also their wishes – "*preferably, I would like*".

Efficiency and trust. All the retrieved supplies are selected considering user's strict specifications and desires. Exploiting the semantics of the request all the supplies in conflict with the user strict requirements are discarded. For the retrieved supplies, the rank is computed based on the meaning of their description and their degree of request satisfaction.
For each retrieved supply, an explanation on the match degree is shown to the user. Then she can verify why the system chose a supply rather than another one.

Ranking criteria. The ranking is established based on the semantic similarity of the demand with each supply. Of course, the semantic match degree value can be combined with other extra information, *e.g.*, price, quantity or delivery time, in order to refine the ranking function.

5 Related Work

Recently, there has been a growing interest towards systems supporting semantics exploitation, in different domains. In [12] an application is presented, improving traditional web searching using semantic web technologies: two Semantic Search applications are presented, running on an application framework called TAP, which provides a set of simple mechanisms for sites to publish data onto the Semantic Web and for applications to consume these data via a query interface called GetData. The results provided by the system are then compared with traditional text search results of Google. Story Fountain [15] is an ontology-based tool, which provides a guided exploration of digital stories using a reasoning engine for the selection and organization of resources. Story Fountain provides support for six different exploration facilities to aid users engaged in exploration process. The system is being used by the tour guides at Bletchley Park. The approach has been further investigated in [6]. An intelligent query interface exploiting an ontology-based search engine is presented in [5]; the system enables access to data sources through an integrated ontology and supports a user in formulating a query even in the case of ignorance of the vocabulary of the underlying information system.

6 Conclusion

In this contribution we have presented a system that, in our opinion, clearly shows the benefits of semantic markup of descriptions in an e-marketplace. Exploiting ontologies and non-standard inference services for OWL DL ontologies, it allows to satisfy common sense user needs during the interaction within an e-marketplace.
The user is guided in the query formulation through the intensional navigation of a specific marketplace domain knowledge without any underlying knowledge of the so called semantic web technologies. The semantics of the ontology-based query and supplies descriptions is used to perform a semantic-based matchmaking process and to provide explanations on match results.
We are carrying out preliminary tests on the system, with the aid of human volunteers. The domain we selected was one of used cars. Experiments are related to evaluate both the theoretical approach and the usability of the tool. The evaluation of different match degree functions, combining extra-ontological information, is also under investigation.

Acknowledgment

We acknowledge partial support of projects PITAGORA, CNOSSO, and ACAB-C2.

References

1. F. Baader, D. Calvanese, D. Mc Guinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2002.
2. A. Bernstein, E. Kaufmann, A. Göhring, and C. Kiefer. Querying ontologies: A controlled english interface for end-users. In *4th International Semantic Web Conference (ISWC 2005)*, pages 112–126, November 2005.
3. D. C. Blair and M. E. Maron. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Comm. of the ACM*, 28(3):289–299, 1985.
4. T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *J. Vis. Lang. Comput.*
5. T. Catarci, P. Dongilli, T. D. Mascio, E. Franconi, G. Santucci, and S. Tessaris. An ontology based visual tool for query formulation support. In *ECAI*, pages 308–312, 2004.
6. T. Collins, P. Mulholland, and Z. Zdráhal. Semantic browsing of digital collections. In *International Semantic Web Conference*, pages 127–141, 2005.
7. S. Colucci, T. Di Noia, E. Di Sciascio, F. M. Donini, and M. Mongiello. A uniform tableaux-based method for concept abduction and contraction in description logics. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI '04)*, pages 975–976. IOS Press, 2004.
8. S. Colucci, T. Di Noia, E. Di Sciascio, F. M. Donini, and M. Mongiello. Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an e-marketplace. *Electronic Commerce Research and Applications*, 4(4):345–361, 2005.
9. S. Colucci, T. Di Noia, E. Di Sciascio, F. M. Donini, and A. Ragone. Knowledge elicitation for query refinement in a semantic-enabled e-marketplace. In *7th International Conference on Electronic Commerce ICEC 05 ACM Press*, pages 685–691. ACM, 2005.
10. T. Di Noia, E. Di Sciascio, F. Donini, and M. Mongiello. A system for principled Match-making in an electronic marketplace. In *Proc. of WWW '03*, pages 321–330, 2003.
11. E. Franconi, S. Tessaris, T. Catarci, T. Di Mascio, G. Santucci, and G. Vetere. Specification of the ontology design tool. Technical report, D6.2, SEWASIE Consortium, 2003.
12. R. V. Guha, R. McCool, and E. Miller. Semantic search. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003*, pages 700–709, 2003.
13. I. Horrocks. Owl: A description logic based ontology language. In *ICLP*, pages 1–4, 2005.
14. L. Li and I. Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. In *Proc. of WWW '03*, 2003.
15. P. Mulholland, T. Collins, and Z. Zdráhal. Story fountain: intelligent support for story research and exploration. In *Intelligent User Interfaces*, pages 62–69, 2004.
16. OWL. Web Ontology Language. www.w3.org/TR/owl-features/, 2004.
17. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *The Semantic Web - ISWC 2002*, number 2342 in LNCS. 2002.
18. P. H. Z. Pu and P. Kumar. Evaluating example-based search tools. In *ACM Conference on Electronic Commerce*, pages 208–217, 2004.
19. G. M. Sacco. The intelligent e-store: Easy interactive product selection and comparison. In *CEC*, pages 240–248, 2005.
20. P. Shvaiko and J. Euzenat. A Survey of Schema-based Matching Approaches. *Journal on Data Semantics*, 4, 2005.

Memory-driven dynamic behavior checking in Logical Agents

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 deal with detecting behavioral anomalies in agents. In particular we consider agents defined via logic languages, and we take as a case-study the DALI language previously defined by the authors. We start proposing a formalization of aspects of agent memory and experience and we emphasize how the construction of memories can be a concrete aid for verification of future activities. The anomalies detection process is based on a set of constraints that use the agent past experience to identify behavioral traces different from those expected. Anomalies verification can be performed without stopping agent life and new constraints can be dynamically added thus allowing to check what the agent learns during its activity.

1 Introduction

Like human communities, agents can count only upon their capabilities for facing complex tasks and maintaining a high performance in dynamic environments where they are put at work. In order to cope with the unknown stimuli of the world, an agent needs to observe its past behavior and to deduce the best actions to do, trying to avoid the errors performed in previous situations. This motivates the importance of recording the most relevant facts which happened in the past and of recovering error and behavioral anomalies by means of appropriate strategies.

By providing agents with the capability of maintaining a track of past behavior and with a mechanism to elicit performed errors and a wrong behavior recovery strategies, it is possible to make these entities more powerful and reliable. The definition of frameworks for checking agent behavior during its life based on experience has not been really treated up to now. In fact, correspondingly to the agent framework complexity, there has been an increasing need for agent platforms whose entities would be capable of exhibiting a correct and rigorous behavior with respect to the expectations, but typically developers have applied model-checking techniques to system abstract models thus neglecting the verification of behavioral anomalies during the agent life.

The Model-checking paradigm allows to model a system S in terms of automata by building an implementation P_s of the system by means of a model-checker friendly language and then verifying some formal specifications. These are commonly expressed either as formulae of the branching time temporal logic CTL [4,23] or as formulae of

Linear Temporal Logic [14,30]. Model-checking techniques have been adopted in order to check systems implemented in AgentSpeak(L) [3]. In order to apply model-checking techniques, the authors have defined a variation of the language aimed at allowing its algorithmic verification. This variation is then submitted to model checkers. Penczek and Lomuscio have defined bounded semantics of CTLK [22], a combined logic of knowledge and time. The approach is to translate the system model and a formula ϕ, indicating the property to be verified, to sets of propositional formulae then submitted to a SAT-solver.

Methods for checking behavior are generally applied at the initial phase of the agent life. But, as it is well known, agents live in open environment where they can learn new knowledge or rules: then there are implicit difficulties for checking from time to time the behavior correctness by means of either model-checking or traditional approaches, that are generally static. Moreover, as mentioned before, model-checking techniques are applied by rewriting the interpreter in another language and this operation cannot be re-executed whenever the agent learns a new fact or rule.

In contrast to model checking, the deductive approach to verification uses a logical formula to describe all possible executions of the agent system and then attempts to prove the required property from this logical formula. The required properties are often captured by using modal and temporal logics. Deductive approaches have been adopted by Shapiro, Lesperance and Levesque that defined CASLve [28], a verification environment for the Cognitive Agent Specification Language. A limit of the theorem proving approach is the problems complexity, and thus a human interaction is often required.

Another possible approach to agent validation requires to observe the agent's behavior as it performs its task in a series of test scenarios before putting it at work. But this approach, as observed by Wallace in [31], by its very nature is incomplete, since all possible scenarios cannot be examined. Nor the future agent knowledge is knowable in advance. So, it is necessary to individuate a new mechanism capable of verifying the agent behavior correctness without stopping its life.

In this paper, we propose a method for checking the agent behavior correctness during the agent activity, based on maintaining information on its past behavior. This information is useful in that it records what has happened in the past to the agent (events perceived, conclusions reached and actions performed) and thus encodes relevant aspects of an agent's *experience*. If augmented by time-stamps, these records (that we call *past events*) constitute in a way the *history* of the agent activity. The set of past events evolves in time, and can be managed for instance by distinguishing the most recent versions of each past event, that contribute to the agent present perception of the state of the world. Past events can moreover be exploited for the purpose of self-checking agent activities: we propose in fact the introduction of specific constraints, aimed at checking if the entity has performed actions that it should not have done or has not performed actions that it should have done in a certain time and/or under some conditions. Alberti et al. in [1] have adopted a similar approach based on social constraints in order to model the interactions among (possibly heterogeneous) agents that form an open society. The main advantages of their approach are in the design of societies of agents, and in the possibility to detect undesirable behavior.

Our checking behavior proposal is not aimed at verifying the agents behaviors correctness in the context of communication acts exchange but to evaluate the reliability of a single agent by means of constraints. The constraints that we introduce are purposedly of a quite simple form so as to be easily and efficiently checked. As in this context we assume that our agent are not malicious, whenever the self-check should reveal an anomaly, the agent might for instance try a self-correction or report the anomaly to a supervisor agent. However, details of possible error-recovery strategies are outside the scope of this paper.

Another interesting class of techniques for agent behavior verification is based on variations of Kowalski and Sergots Event Calculus, used in conjunction with abduction. We intend in the future to perform a comparison between our behavioral constraints and these techniques. We mention here the interesting approaches in [27] and [2] that analyze safety properties and formalize Policy Specification. In [2], the abduction process is applied to a specification that models both the systems behavior and the policy specification, allowing to detect conflicts when the applicability of the policies is constrained on the runtime state of the system.

This paper is organized as follows. In Section 2 we discuss some kind of anomalies that agent behavior can reveal. In Section 3 we introduce concepts related to agent memory and experience, that we define more formally in Section 4. In Sections 5 and 6 we propose a set of constraints useful to detect behavioral anomalies. In Section 7 briefly discuss the semantics of our approach and, in Section 8, as a case-study, we show in detail the application of constraints in the DALI language [6] [7]. Finally, we conclude in Section 9.

2 Behavioral Anomalies Detection

Any mechanism for checking and possibly recovering agent behavioral anomalies relies on the ability to identify a model describing the expected agent behavior. This description can be provided at various levels of detail and under different points of view. However, any description should in general state what the agent has to do and what it must not do. In this context, we do not assume either a deep knowledge of the agent inner structure or the possibility of making all its state explicit. Rather, we assume to have a meta-description including a component specification of the expected observable agent behavior.

In order to define a framework for verifying the agent behavior correctness, we concentrate on six possible behavioral anomalies partially inspired by the work of Wallace in [31]:

- **Incorrect time execution:** An action or goal is accomplished at the incorrect time. This anomaly happens when we expect an agent to do an action or pursue a goal at a certain time but the action takes place before or after the established threshold.
- **Incorrect duration:** It is possible that an an action or goal lasts beyond a reasonable time or a specific related condition. In this case its duration is incorrect and determines an anomaly.

– **Omission:** The agent fails to perform the required action or to pursue its goal. This anomaly happens when an action/goal is not executed within a certain amount of time and cannot be executed later.
– **Intrusion:** The agent performs a goal or action that is not proper. This anomaly takes place when the expected behavior contains some actions/goals that the agent must not perform.
– **Duplication:** An action or a goal is repeated inappropriately. This happens when an agent performs more than once the same action/goal.
– **Incoherency:** An action or goal is executed a number of times greater than an expected threshold. The agent program can require that an action or a goal is executed a certain number of times but the agent performs that action beyond the prefixed threshold.

These behavioral anomalies can be very dangerous in critical contexts. Then, their capture not only improves the entities performance but prevents irreversible damages to the system. In order to identify similar anomalies, Wallace's approach [31] proposes a Behavior Comparison System based on traces. It can be viewed as a machine that takes two specifications of behavior as input and produces a summary of how these behaviors differ. We start from Wallace theoretical work, and develop a constraints-based system capable to detect the anomalies described above while the agent is active. This system, integrated in an agent framework, works for detection without stopping the agent life.

In the approach proposed in this paper, an agent is able to detect these anomalies via a self-check based on special constraint (clearly, in this context we assume that our agent are not malicious). What can be done if an anomaly is actually detected? Although details of possible error-recovery strategies are outside the scope of this paper, we can suggest that the agent might for instance try a self-correction or report the anomaly to a supervisor agent.

3 About agent experience

A rule-based agent consists of a knowledge base and of rules aimed at providing the entity with rational, reactive, pro-active and communication capabilities. The knowledge base constitutes the agent "memory" while rules define the agent behavior. We suppose that agents repeatedly execute a *observe-think-act cycle* as suggested by Kowalski in [16] in order to sense the environment, to decide the better action to perform and, finally, to modify the world by means of the action execution.

Imagine an agent that is capable of remembering the received external stimuli, the reasoning process adopted and the performed actions. Through "memory", the agent is potentially able to learn from experiences and ground what it knows through these experiences [18]. The interaction between the agent and the environment can play an important role in constructing its "memory" and may affect its future behavior. Most methods to design agent memorization mechanisms have been inspired by models of human memory as ([24],[21]).

In 1968, Atkinson and Shiffrin proposed a model of human memory which posited two distinct memory stores: short-term memory and long-term memory. This model

has been suggested by Gero and Liew for constructive memory whose implementation has been presented in [19]. Memory construction [in this model] occurs whenever a design agent uses past experiences in the current environment in a situated manner. In a constructive memory system, any information about the current design environment, the internal state of the agent and the interactions between the agent and the environment is used as cues in its memory construction process.

Gero and Liew introduce the notion of working memory, a workspace for reflective and reactive processes where explicit design-based reasoning occurs. Items of information within the working memory are combined with the stored knowledge and experiences, manipulated, interpreted and recombined to develop new knowledge, assist learning, form goals, and support interaction with the external environment. At this point it is clear that memory, experience and knowledge are strongly related. Correlation between these elements can be obtained via neural networks as in [19], via mathematical models as in [20] or via logical deduction.

The authors of this paper have proposed in [5], [6],[7] a method of correlating agent experience and knowledge by using a particular construct, the internal events, that has been introduced in the DALI language (though it can be in principle adopted in any computational logic setting). We have defined the "static" agent memory in a very simple way as composed of the original knowledge based augmented with *past events* that record the external stimuli perceived, the internal conclusions reached and the actions performed.

Past events can play a role in reaching internal conclusions. These conclusions, that are proactively pursued, take the role of "dynamic" memory that supports decision-making and actions: in fact, the agent can inspect its own state and its view of the present state of the world, so as to identify the better behavioral strategy in that moment. The agent re-elaboration of its experiences creates a particular view of the external world. By "particular" we mean that each agent, on the basis of its knowledge and experience, can interpret what has changed in the world in its peculiar manner. In our view therefore, an agent must record not only perceived external stimuli but also the internal conclusions reached by the entity and the actions performed. This allows in principle all aspects of agent behavior to be related, thus potentially improving its performance.

More specifically, *Past events* in our view, have at least two relevant roles:

- To describe the agent experience: for example, if an agent, after putting an apple on the table, takes it from the table and places it in the fruit-dish, all these actions must be recorded in the set of past events with the annotation of when they were performed, so that their sequence ca be reconstructed.
- To keep track of the state of the world and of its changes, possibly due to the agent intervention. In the above example, the agent should have a record (as past event) of the apple being on the table. Then, due to its own action (if successful), there will be a new record of the apple being in the fruit dish. The former item of information will be kept, but it will be no more "actual", while the latter one will now be "actual". Then, the most recent past events may be seen as representing the current "state of affairs".

With time, on the one hand past events can be overridden by more recent ones of the same kind (take for example temperature measurement: the last one is the "current" one) and on the other hand can also be overridden also by more recent ones of different kinds, which are somehow related.

In our approach, we introduce a set P of current "valid" past events that describe the state of the world as perceived by the agent. We also introduce a set PNV where we store all previous ones. The content of set PNV can be seen as the agent "memory" in a broader sense (we do not cope here with practical efficiency reasons that might force the agent to "purge" PNV in some way so as to regain store).

Then, we need a mechanism for keeping P up-to-date. The mechanism that we propose consists in defining in the agent program a set of constraints that express what and when to remove a past event from P.

4 Defining agent experience

We abstractly formalize an agent as the tuple $< Ag, P_{rg}, E, I, A >$ where Ag is the agent name, P_{rg} describes the agent behavioral rules, E is the external events set (events that the agent is capable to perceive and recognize), I is the internal events set (distinguished internal conclusions) and, finally, A is the actions set (actions that the agent can possibly perform). We emphasize that, while external events allow agents to react to the environmental stimuli, internal events generate the agent proactive capabilities. Finally, by using actions the agent influences the external world.

We suppose that each action performed and each external or internal stimulus received will be recorded by the agent in order to keep track of its past behavior. The events that happen are kept in a set $H = E \cup I \cup A$ where however each event in H is annotated (time-stamped) with the time when the event has happened. In practice, the set H is dynamically augmented with new events that happen, and is totally ordered w.r.t. the event time-stamps. Each event in H is then actually in the form $X : T_i$ where X is the event and T_i is its time-stamp; by abuse of notation for the sake of coinciseness we will occasionally indicate $X : T_i$ as X_i or simply X. We introduce a function $\Pi_Y^T(X)$ that takes in input an $X \in H$ and transforms it in the corresponding past form X_P^T where: T is the time-stamp of X; P is a postfix that syntactically indicates past events and Y is a label indicating what is X, i.e., if it belongs to E, I or A. I.e., $\Pi_Y^T(X) = X_P^T$

We can indicate the generation process of past events as the agent *history* h. The history is related to the order of set H, that reflects which events (non deterministically) happen. In fact, we assume to build the history according to this order. Several histories are thus in principle possible depending on both the interaction of the agent with the external environment and its internal choices.

$$h : init \xrightarrow{X_0 \in H} X_{P_0}^{T_0} \xrightarrow{X_1 \in H} X_P^{T_1} .... \xrightarrow{X_n \in H} X_P^{T_n}$$

while the set of past events P_h related to this specific history is:

$$P_h = \{X_{0P}^{T_0}, X_{1P}^{T_1}, ..., X_{nP}^{T_n}\}.$$

In the rest of this paper, we say P instead of P_h referring to the set of past events resulting from a generic agent history. Moreover, a generic element of P will be indicated by X_P^T or even X_P if the time-stamp is irrelevant. P synthesizes the whole agent life maintaining track of its actions, reactions and internal thoughts and its relevance is based on the possibility, for the agent, of knowing what it did and of deciding new strategies according to the old ones.

P is not static: it grows while the agent accomplishes its activities but it also loses those items that don't contribute any longer to the agent experience description. In the rest of this section, we introduce a set of constraints useful to determine which past events concur in the current agent life description and which don't. As mentioned before, past events no longer valid are eliminated from P and put in another set *PNV*, from which the agent can deduce some conclusions useful for performing its tasks.

Past Constraints define which past events must be eliminated and under which conditions. They are verified from time to time to maintain the agent memory consistent with the external world. More formally, we define a *Past Constraint* as follows:

Definition 1 (Past Constraint). *A Past Constraint has the syntax:*

$$X_{kP} : T_k, ..., X_{mP} : T_m \doteq X_{sP} : T_s, ..., X_{zP} : T_z, \{C_1, ..., C_n\}.$$

where $X_{kP} : T_k, ..., X_{mP} : T_m$ are the past events to be eliminated whenever past events $X_{sP} : T_s, ..., X_{zP} : T_z$ are in P and conditions $C_1, ..., C_n$ are true.

Each condition C_i can express either time or knowledge constraints. Consider the apple example introduced above. A corresponding constraint could be: $put\_apple\_in\_the\_table_P : T_1 \doteq put\_apple\_in\_the\_fruit\_dish_P : T_2, \{T_2 > T_1\}$.

The directive that it expresses is: if in P there is the past event $put\_apple\_in\_the\_fruit\_dish_P$ with a time greater than that of the past event $put\_apple\_in\_the\_table_P$, then delete from P the latter one (occurring in the head of the rule) because it is no longer valid and put it in the PNV set.

Formally, we can define a particular function, $\Phi(X)$ that determines the transmigration of the event X from P to PNV. This transmigration is based on the satisfaction of the related constraints. Let PC the set of Past Constraints. A generic past event X_P^T will be transformed into X_{PNV}^T iff there exists a Past Constraint in PC related to X_P^T whose conditions expressed in the body are satisfied.

$$\Phi(X_P^T) = \begin{cases} X_{PNV}^T & \exists C \in PC \mid C(X_P^T)\ is\ true \\ \\ X_P^T & else \end{cases}$$

More precisely, the PNV set will be composed of a certain number of X_{PNV}^T items:

$$PNV = \{X^{T_0}_{0PNV}, X^{T_1}_{1PNV}, ..., X^{T_n}_{nPNV}\}.$$

Past events in PNV may still have a relevant role for the entity decisional process. In fact, an agent could be interested for instance in knowing how often an action has been performed or a particular stimuli has been received by the environment. For this reason, we introduced some operators capable of deducing from PNV data useful to draw internal conclusions:

– **How often an action, a reaction or an internal conclusion has been performed.** This information can be obtained by the agent via the operator $\Delta^{PNV}_n(X)$ that returns the number n of occurrences of X in PNV. In fact, the parameter n has been adopted to indicate the number of occurrences that Δ has to return, and can take the special values *first* and *last* to indicate in fact that one is searching for either the first or the last occurrence, given the temporal order specified by time-stamps. For example, consider an agent that bought a car through instalments. If it wants to know when it paid all the money, it could check the number of instalments that has been paid:

  $$N = \#\Delta^{PNV}_n(pay\_instalment_A(Value, Date))$$

– **The first or last past event occurrence.** An agent would want to know when it performed an action, reached a conclusion or reacted to a certain event for either the first or the last time. This information can be obtained searching for event X in $P \cup PNV$:

  $$E_{first} = \Delta^{P \cup PNV}_{first}(X) \text{ or } E_{last} = \Delta^{P \cup PNV}_{last}(X)$$

  For example, the agent that bought the car could be interested in verifying when it paid the first or last instalment:

  $$First\_payment = \Delta^{P \cup PNV}_{first}(pay\_instalment(Value, Date))$$
  $$Last\_payment = \Delta^{P \cup PNV}_{last}(pay\_instalment(Value, Date))$$

– **Past event occurrences in a time interval.** Sometimes an agent is interested to know how many past event occurrences are present in a certain time interval T_1,T_2. In this case it will search the response in PNV:

  $$Occurrences\_list = \Delta^{PNV}_{T_1,T_2}(X)$$

  Suppose for instance that the bank that lent money to our agent, at a certain time contests the payment of the instalments from March to June. The agent is able to verify the situation via the following operator:

  $$Instalments = \Delta^{PNV}_{March,June}(pay\_instalment(Value, Date))$$

Finally, in order to formalize the consideration for which experience and knowledge concur to generate agent "memory", we formally define the *agent memory* M as:

$$M = \{X \in P \cup PNV \cup KB_{facts}\}$$

where KB_{facts} is the set of facts belonging to the agent knowledge base. Then, memory (in terms of facts) includes what the agent knows from start and what it records later. This "static" memory is then to be elaborated so as to reach internal conclusions about the state of either the world or the agent itself, that constitute the "dynamic" memory, or, in an alternative interpretation, the "self-consciousness" of the entity.

The agent *experience* synthesized by the sets P and PNV constitutes the starting point for our behavior checking approach.

5 Constraints for behavioral anomalies checking

Past events have a main role in defining our detection anomalies system. In fact, the possibility of discovering wrong behavioral sequences is strictly linked to the study of performed actions, reactions and internal conclusions. Given that past events resume previous agent behavior, as skillful archaeologists, we propose a method based on several *behavioral constraints* useful to discover specific traces. What kind of traces? In this section we cope with those suggested by the anomalies classification proposed in section 2. We introduce the following kinds of *behavioral constraints*.

– **Existential Constraint:** *Existential Constraints* contain a reference to the past, one to the present and a set of conditions useful to describe the class of histories that we are considering. They check whether the agent performed an action, a reaction or a conclusion that the expected behavior considers anomalous. More formally:

Definition 2 (Existential Constraint). *Let P be the set of past events, let $X_i \in P$, let T_i be its time-stamp and let $C_1, ..., C_n$ be a set of additional conditions (a similar reasoning can be performed with PNV events), for us an Existential Constraint has the following structure:*
$X_{kP} : T_k, ..., X_{mP} : T_m \exists \triangleleft X_{sP} : T_s, ..., X_{zP} : T_z, \{C_1, ..., C_n\}$.

The meaning is: if in P there are past events $X_{sP} : T_s, ..., X_{zP} : T_z$ and conditions $C_1, ..., C_n$ are true, then $X_{kP}^{\not\in} T_k, ..., X_{mP} : T_m$ past events must not be in P. Otherwise, we are in presence of an anomaly. The constraint is existential in that if just one of $X_{kP}^{\not\in} T_k, ..., X_{mP} : T_m$ is in P, this constitutes an anomaly.

– **Inquiring Constraint:** This kind of behavioral constraint checks past events that, considered some conditions, must be in P at a certain time. I.e., if an agent has performed some actions, reacted to some external stimuli or reached some internal conclusions and if some conditions are true, then necessarily, after a certain amount of time, the agent should have performed certain actions, reactions or reasoning as specified in the expected behavior. Or else, the entity is assuming an incomprehensible behavior. Formally, an *inquiring constraint* has the form:

Definition 3 (Inquiring Constraint). *An Inquiring Constraint has the structure:*

$X_{kP} : T_k, ..., X_{k+mP} : T_{k+m} \neg\exists \triangleleft X_{sP} : T_s, ..., X_{zP} : T_z, \{C_1, ..., C_n\}$.
where P is the set of past events, each $X_{iP} \in P$ with time-stamp T_i and $C_1, ..., C_n$ are a set of conditions.

The meaning is: if in P there are the past events $X_{sP} : T_s, ..., X_{zP} : T_z$ conditions and $C_1, ..., C_n$ are true, then we expect that at most at the time T_k the past event $X_{kP} : T_k$ be in P, and ... at most at time T_{k+m} the past event X_{k+mP} be in P. If any of them is lacking at the limit time, we are in presence of an anomaly.

In the next section we show how the behavioral constraints we have introduced allow us to detect the anomalies suggested by Wallace.

6 Detecting anomalies

In this section we propose, for each anomaly, how *behavioral constraints* are able to detect the specific traces. We will adopt some examples in order to explain the potentiality of our approach.

Incorrect time execution: *An action or goal is performed at the incorrect time*. Suppose for instance that our agent bought a goldfish and an aquarium. To keep the fish safe it is necessary that it first fills the aquarium up with water and then puts the animal inside. We will indicate an action with the postfix A for the sake of readability. So, the expected action sequence will be: $fill\_the\_aquarium_A$, $put\_inside\_fish_A$.

In order to check the actions execution correctness, we must verify that agent does not perform the second action before the first one. To this aim, we can adopt the *existential constraint*:

$fill\_the\_aquarium_P : T_1 \exists \triangleleft put\_inside\_fish_P : T_2, \{T_2 < T_1\}$.

indicating that if the agent has accomplished the action $put\_inside\_fish_A$ (it became past event) at the time T_2 and in P there is the past event $fill\_the\_aquarium_P^{T_1}$, which occurred later ($T_2 < T_1$) then the action sequence is not correct and we are in the presence of *Incorrect time execution* anomaly.

Incorrect duration: *An action or goal last beyond a reasonable time or a specifiied condition.* Consider the following simple program where we indicate external events (perceptions) with postfix E and we mean that opening the umbrella is a reaction to the perception of rain. We assume that an external event is recorded as a past events only after the related reaction has successfully taken place and that an action is recorded as a past events as soon as it has been successfully performed.

$rain_E : -open\_the\_umbrella_A$.

Then, in P we cannot have the past events $rain_P : T_1$ and $open\_the\_umbrella_P : T_2$ with $T_1 > T_2$. This would mean in fact that reaction has been considered to have

been successfully accomplished while the action had not yet been completed. It is an anomalous behavior. In order to detect this anomaly we can employ an *inquiring constraint*:

$open\_the\_umbrella_P : T_1, \neg\exists \triangleleft rain_P : T_2, \{T_1 < T_2\}$.

Omission: *The agent fails to perform the required action or pursue the required goal.* An action omission can be detected if we expect that the agent performs an actions sequence but one of the corresponding past events is lacking after a certain time limit. Suppose that our agent is on an island and must go fishing for surviving. For reaching its purpose, the agent must take the fishing-rod, then must bait the hook and finally must drop the fish-hook in the sea. So, the corresponding actions sequence will be: $take\_fishing\_rod_A$, $bait\_hook_A$, $drop\_fish\_hook_A$. Now, we can check an *omission anomaly* by adopting the following *inquiring constraint*:

$take\_fishing\_rod_P : T_k, \neg\exists \triangleleft$
$bait\_hook_P : T_2, drop\_fish\_hook_P : T_3, \{T_k < T_2 + Th_2, T_k < T_3 + Th_3\}$.

where Th_i are predefined time thresholds. The meaning is: if in P there are the past events $bait\_hook_P : T_2$ and $drop\_fish\_hook_P : T_3$ but at the time T_k defined by the conditions the past event $take\_fishing\_rod_P : T_k$ is absent, we can deduce that the corresponding action has been omitted.

Intrusion: *The agent performs a goal or action that is not allowed.* Suppose that our agent is situated in a critical environment where we need be sure that it never performs a dangerous action. We can verify if this happens by using an *existential constraint*. In particular, consider an agent that, if enters into the red room, must not push the green button. The corresponding constraint will be:

$push\_green\_button_P : T_1, \exists \triangleleft enter\_in\_red\_room_P : T_2, \{T_1 > T_2\}$.

Duplication: *An action or a goal is repeated inappropriately.* This anomaly can be detected by searching for two past events corresponding to the same action, reaction or conclusion having the same time T_i. The existential constraint capable of detecting this incorrect behavior is:

$push\_green\_button_P : T_1, \exists \triangleleft push\_green\_button_P : T_2, \{T_1 = T_2\}$.

Incoherency: *An action or goal is executed a number of times greater than an expected threshold.* This anomaly is strictly correlated with PNV events. In fact, the agent experience maintains the information on how often an action/goal has been accomplished. Consider again the agent that bought the car. It must pay twelve instalments in the current year. If it pays a further instalment, the expected behavior is violated:

$pay\_instalment(Value, Date))_P : T\ \exists \triangleleft$
$\{N = \Delta_n^{PNV}(pay\_instalment(Value, Date)), N > 20, T > 0\}$.

Behavioral constraints are to be checked from time to time by the system in order to point out the anomalies. When an incorrect behavior takes place, a particular past event can be generated. This event may contain information about the kind of violation for allowing the agent to activate possible recovery strategies. Moreover, by inspecting the sets P and PNV, one can discover the motivation of the entity anomalous behavior

examining all past events generated by the system. Consequently, appropriate countermeasures can be taken either by the agent itself, or by an external controller. In the next sections, after giving the semantics of the approach, we will put our idea at work in a real multi-agent system called DALI, a logic language capable of generating autonomous, reactive, pro-active and communicative agents.

7 Semantics of Past and Behavioral Constraints

The semantics of Computational Logic agent languages may in principle be expressed as outlined in [6] for the DALI language. In particular, given program P_{Ag}, the semantics is based on the following.

1. An *initialization step* where P_{Ag} is transformed into a corresponding program P_0 by means of some sort of knowledge compilation (which can be understood as a rewriting of the program in an intermediate language).
2. A sequence of evolution steps, where the reception of an event or a certain expired time threshold is understood as a transformation of P_i into P_{i+1}, where the transformation specifies how the event affects the agent program (e.g., it is recorded). The time threshold allows one to verify the behavioral constraints if no event is happened.

Then, one has a Program Evolution Sequence $PE = [P_0, ..., P_n]$ and a corresponding Semantic Evolution Sequence $[M_0, ..., M_n]$ where M_i is the semantic account of P_i (in [6] M_i is the model of P_i).

This semantic account can be adapted by transforming the initialization step into a more general knowledge compilation step, to be performed:

(i) At the initialization stage, as before.
(ii) When a new behavioral constraint is added.
(iii) In consequence to a violation evidence.

8 Anomalies detection in DALI

DALI [6] [7] [29] [9] [5] is an Active Logic Programming language designed in the line of [17] for executable specification of logical agents. The Horn-clause language is a subset of DALI, that in fact procedurally employs an Extended Resolution Procedure that interleaves different activities (based on the declarative semantics outlined above, and on an operational semantics based on Dialogue Games Theory [8]).

The reactive and proactive behavior of a DALI agent is triggered by several kinds of events: external, internal, present and past ones. All the events and actions are time-stamped, so as to record when they occurred. 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.

Definition 4 (Past Event). *A past event is syntactically indicated by the postfix P:*
$PastEvent ::= << Atom_P >>$

Original DALI program definition has been modified for introducing the set of behavioral constraints:

Definition 5 (DALI logic program). *A DALI logic program Pr_i is the tuple:*

$\langle Action_i, Reactive_i, Active_i, Hclause_i, Past\_constraints, Beh\_constraints_i \rangle$

where $Action_i$ is the set of the Action rules, $Reactive_i$ is the set of the Reactive rules, $Active_i$ is the set of the Active rules, $Hclause_i$ is the set of Horn clause rules, $Past\_constraints_i$ is the set of Past constraints and $Beh\_constraints_i$ is the set of Behavioral constraints. Reactive rules have an external or internal event in the head, Action rules specify the action-preconditions, Active rules have actions in their body, Horn clause rules are general prolog-like rules while Past and Behavioral constraints have been defined above.

Constraints on DALI agent behavior maintain a very similar syntax to that presented in Definitions 2 and 3.

Definition 6 (DALI Past constraints). *Let $E_{1P}, ..., E_{nP}$ be DALI past events, let $C_1, ..., C_s$ be conditions and $T_1, ..., T_n$ be time variables, we define a DALI past constraint as:*
$E_{1P} : T_1, ..., E_{kP} : T_k \ \sim / \ E_{k+1P} : T_{k+1}, ..., E_{nP} : T_n, \{C_1, ..., C_s\}.$

Definition 7 (DALI Behavioral constraints). *Let $E_{1P}, ..., E_{nP}$ be DALI past events, let $C_1, ..., C_s$ be conditions, let $T_1, ..., T_n$ be time variables and let Time indicate a precise moment, we define DALI Behavioral Constraints as follows:*

- **Existential constraint**
  $E_{1P} : T_1, ..., E_{kP} : T_k \ < / \ E_{k+1P} : T_{k+1}, ..., E_{nP} : T_n, \{C_1, ..., C_s\}.$
- **Inquiring constraint**
  $E_{1P} : at(Time) \ ?/ \ E_{k+1P} : T_{k+1}, ..., E_{nP} : T_n, \{C_1, ..., C_s\}.$

The meaning of these constraints corresponds to the Existential and Inquiring constraints introduced in previous sections. DALI constraints can be added to the logic agent program either at the initialization phase or when the agent is active. In fact, DALI agents are capable of learning rules and constraints through a particular mechanism, as discussed in [10]. This improvement allows one to expand the detection anomalies system potentialities by adapting the constraints to new knowledge learned by the entity during its life. Consider a simple DALI example on an agent being at home when it receives the perception that something dangerous is happened. In this example, we use the postfix E to indicate an external perception and A to identify the actions.

$danger_E :> once(ask\_for\_help)$.
$ask\_for\_help : -call\_police_A$.
$call\_police :< have\_a\_phone_P$.
$ask\_for\_help : -scream_A$.
$danger :< at\_home_P$.

The new tokens introduced have the following meaning: $:>$ denotes reaction, i.e. the body of the rule is involved whenever the head occurs as an external event; $:<$ denotes (for sake of clarity) that the rule expresses preconditions of an action. The meaning of this example is: if the agent receives from the environment the stimulus $danger_E$, it can react either by calling the police, if it has a phone, or by screaming. The reaction is allowed only if the agent is at home when the danger happens. The contextual information on the phone and the agent position is synthesized by *past events*. Two actions, $call\_police_A$ and $scream_A$ constitute agent alternative options and describe as past events a different evolution of the world. If the agent called the police, then it cannot have in its memory the past event of the opposite action $scream_P$. For keeping the world coherency we introduce in the agent program the following *past constraints*:

$call\_police_P : T \ \sim / \ scream_P : T_1, \{T < T_1\}$.
$scream_P : T \ \sim / \ call\_police_P : T_1, \{T < T_1\}$.

The directive expressed in the first constraint is: if in the agent memory there is the past event $call\_police_P$ happened at the time T and at the time T_1 greater than T the entity performs the action $scream_A$, the system must eliminate the previous action because it is no longer valid for describing the current world. The second constraint copes with the opposite situation.

At this point we complicate the agent behavior by introducing new rules:

$remain\_at\_home : -danger_P, call\_police_P, robber\_in\_kitchen_P$.
$remain\_at\_home_I :> go\_to\_bathroom_A, close\_the\_door_A$.
$go\_out : -danger_P, scream_P, robber\_in\_garage_P$.
$go\_out_I :> go\_to\_police\_station_A$.

We suppose that the agent, on the basis of performed actions $scream_A$ or $call\_police_A$, could draw some internal conclusions. In fact, when the agent decides to call the police it knows that the robber is in the kitchen. Then, not being able to escape, goes to the bathroom and locks the door. Instead, if the entity screamed and the robber is in the garage, then it goes to the police station. At this point, in order to guarantee the agent behavior correctness, we will introduce an *existential constraint* indicating that an agent cannot be in different places within a restricted time interval:

$go\_to\_police\_station_P : T \ < / \ remain\_at\_home_P : T_1, \{T_1 - 20 \leq T \leq T_1 + 20\}$.

By this constraint we define as anomalous behavior the situation in which the agent is within an interval of 20 seconds both in the police station and at home. In this case we have an incorrect execution.

Inquiring Constraints can help the agents system user to individuate actions that the agent has not performed but that it should have done within a certain time T. Suppose in our example that the police received the call and the policemen are sent to the agent's home. When they arrive, the agent receives the external stimulus $arrived\_police_E$ and it is happy because robber escapes:

$arrived\_police_E :> robber\_escapes_A.$
$i\_am\_happy : -robber\_escapes_P.$
$i\_am\_happy_I :> i\_embrace\_the\_policeman_A.$

However, we suppose that our agent will also be happy if it reaches the police station. Then we update the previous internal event adding a new condition:

$i\_am\_happy : -go\_to\_police_P.$

At this point, we have an agent that receives the external event $danger_E$ and, after a certain time it becomes happy because it meets the police. We can verify the agent behavior correctness by means of the following *Inquiring Constraint*:

$i\_am\_happy_{1P} : at([2006, 01, 16, 23, 44, 55])\ ?/$
$danger_P : T_1, at\_home_P : T_2, \{T_2 < T_1\}.$

The meaning is: after the dangerous situation described by the example, we will expect that, within a certain time interval fixed by the date, the entity will be happy. If this does not happen, the agent behavior is anomalous (Omission). We conclude this section emphasizing that Past and Behavioral Constraints are attempted by the DALI interpreter from time to time in order to detect as quickly as possible the anomalies.

9 Concluding Remarks

In this paper we have presented an approach to update agent memory and to detect behavioral anomalies by using logic constraints. The approach is based on introducing particular events, past events, that records what has happened. Past events are used to identify contexts in which agents adopt a behavior different from the one expected. Relating past behavior to future one is also proper of Temporal Logic. This is a special branch of modal logic that investigates the notion of time and order. With Temporal Logic one can specify and verify how components, protocols, and objects behave as time progresses ([11],[26]). Temporal Logic is also the core of an important language for multi-agent systems, Concurrent METATEM [12]. Despite Concurrent METATEM demonstrated that Temporal Logic is a fruitful land to generate software entities, some perplexities on its use in time-critical applications remain, due to limited efficiency.

Our approach aims in principle at introducing mechanisms similar to that of Temporal Logic but based on a simple and efficient constraint language. We are conscious that detecting behavioral anomalies is not sufficient in multi-agent systems. Actually, one should provide not only a mechanism to point out anomalies during the agent life but also error recovery strategies usable by the running agent without asking for human intervention. This is in fact a topic of our future research.

References

1. M. Alberti, M. Gavanelli, E. Lamma, P, Mello and P, Torroni, *An Abductive Interpretation for Open Agent Societies*. AI*IA 2003: 287-299
2. A. K. Bandara, E. C. Lupu and A. Russo. Using Event Calculus to Formalise Policy Specification and Analysis. In Proceedings of the 4th IEEE international Workshop on Policies For Distributed Systems and Networks (June 04 - 06, 2003). POLICY. IEEE Computer Society, Washington, DC, 26.

3. R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge, *Model checking agentspeak*. In Proceedings of the Second international Joint Conference on Autonomous Agents and Multiagent Systems (Melbourne, Australia, July 14 - 18, 2003),2003. AAMAS '03. ACM Press, New York, NY, 409-416. DOI= http://doi.acm.org/10.1145/860575.860641
4. E. M. Clarke, O. Grumberg and D. A. Peled, *Model Checking*, The MIT Press: Cambridge,MA,2000.
5. S. Costantini, *Towards active logic programming*, In A. Brogi and P. Hill, editors, *Proc. of 2nd International Workshop on component-based Software Development in Computational Logic (COCL'99)*, PLI'99, (held in Paris, France, September 1999), Available on-line, URL http://www.di.unipi.it/
brogi/ResearchActivity/COCL99/proceedings/index.html.
6. 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.
7. S. Costantini and A. Tocchio, *The DALI Logic Programming Agent-Oriented Language*. In J. J. Alferese and J. Leite(eds.), Logics in Artificial Intelligence, Proceedings of the 9th European Conference, Jelia 2004, Lisbon, September 2004. LNAI 3229, Springer-Verlag, Germany, 2004.
8. S. Costantini, A. Tocchio and A. Verticchio, *A Game-Theoretic Operational Semantics for the DALI Communication Architecture*, proc. of WOA04, Turin, Italy, December 2004, ISBN 88-371-1533-4.
9. S. Costantini and A. Tocchio, *About declarative semantics of logic-based agent languages*. In: "Declarative Agent Languages and Technologies", (revised selected papers presented at DALT 2005), Lecture Notes in Artificial Intelligence, Springer-Verlag, Germany, to appear.
10. , S. Costantini and A. Tocchio, *Learning by Knowledge Exchange in Logical Agents*. In: Simulazione ed Analisi Formale di Sistemi Complessi, Proceedings of WOA05, Universit di Camerino, Novembre 2005.
11. D. Drusinsky, *Monitoring Temporal Rules Combined with Time Series*. In Proc. of CAV'03: Computer Aided Verification, volume 2725 of Lecture Notes in Computer Science, pages 114–118, Boulder, Colorado, USA, 2003. SpringerVerlag.
12. M. Fisher, *Concurrent METATEM - A Language for Modeling Reactive Systems*. In Parallel Architectures and Languages, Europe (PARLE), Munich, Germany, June 1993. SpringerVerlag.
13. M. Fisher and M. Wooldridge, *Executable Temporal Logic for Distributed A.I.*. In Proc. of the 12th International Workshop on Distributed Artificial Intelligence, Hidden Valley, PA, May 1993. http://citeseer.ist.psu.edu/article/fisher93executable.html
14. G. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall International: Hemel Hempstead, england, 1991.
15. M. Kacprzak, A. Lomuscio and W. Penczek, *Verification of Multiagent Systems via Unbounded Model Checking*. In Proc. of the Third international Joint Conf. on Autonomous Agents and Multiagent Systems - Volume 2 (New York, New York, July 19 - 23, 2004), pp. 638-645. DOI= http://dx.doi.org/10.1109/AAMAS.2004.296
16. R. Kowalski and F. Sadri, *Logic Programming and the Real World* . Logic Programming Newsletter, 2001.
17. 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.
18. P-K, Liew and JS, Gero, *A memory system for a situated design agent based on constructive memory*. In Eshaq, A., Khong, C., Neo, K., Neo, M., and Ahmad, S. (eds.), Proc. of CAADRIA2002, Prentice Hall, New York, pp. 199-206.

19. P-K, Liew and JS, Gero, *An Implementation model of constructive memory for a design agent*. In Agents in Design 2002, J. S. Gero and F. Brazier (eds.), 257-276. University of Sydney, Australia: Key Centre of Design Computing and Cognition.
20. K. Lerman, and A. Galstyan, *Agent memory and adaptation in multi-agent systems*. In Proc. of the Second International Joint Conf. on Autonomous Agents and Multi-agent Systems (Melbourne, Australia, July 14 - 18, 2003) AAMAS '03. ACM Press, New York, NY, 797-803. DOI= http://doi.acm.org/10.1145/860575.860703
21. R. H. Logie, *Visuo-Spatial Working Memory*. Lawrence Erlbaum: Hove, 1995.
22. A. Lomuscio, T. Lasica, and W. Penczek. *Bounded model checking for interpreted systems: preliminary experimental results*. In M. Hinchey (ed.), Proc. of FAABS II, LNCS 2699. Springer Verlag, 2003.
23. K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers: Boston, MA, 1993.
24. D.G. Pearson and R. H. Logie, *Working memory and mental synthesis*. In S. O'Nuallan (Ed.), Spatial Cognition: Foundations and applications. John Benjamins Publishing Company.
25. D.G. Pearson, A. Alexander, and R. Webster, *Working memory and expertise differences in design*. In J. Gero, B. Tversky, and T. Purcell (eds.), Visual and Spatial Reasoning in Design II. Sydney: Key Centre of Design Computing and Cognition.
26. A. Pnueli, *The Temporal Logic of Programs*. In: Proc. 18th IEEE Symp. on Foundations of Computer Science, pp. 46-57, 1977.
27. A.Russo, R.Miller, B. Nuseibeh, and , J. Kramer. An Abductive Approach for Analysing Event-Based Requirements Specifications. In Proc. of the 18th international Conf. on Logic Programming (July 29 - August 01, 2002). P. J. Stuckey, Ed. Lecture Notes In Computer Science, vol. 2401. Springer-Verlag, London, 22-37.
28. S. Shapiro, Y. Lesprance, and H. J. Levesque, *The cognitive agents specification language and verification environment for multiagent systems*. In Proceedings of the First international Joint Conference on Autonomous Agents and Multiagent Systems: Part 1 (Bologna, Italy, July 15 - 19, 2002). AAMAS '02. ACM Press, New York, NY, 19-26.
29. A. Tocchio. *Multi-Agent sistems in computational logic*. Ph.D. Thesis, Dipartimento di Informatica, Universitá degli Studi di L'Aquila, 2005.
30. M. Y. Vardi, *Branching vs. linear time: Final showdown*. In T. Margaria and W. Yi (eds.), Proc. of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001 (LNCS Volume 2031), pages 1-22. Springer-Verlag: Berlin, Germany, april 2001.
31. S. A. Wallace, *Identifying Incorrect Behavior: The Impact of Behavior Models on Detectable Error Manifestations*. Fourteenth Conf. on Behavior Representation in Modeling and Simulation (BRIMS-05) 2005.

Learning for Biomedical Information Extraction with ILP

Margherita Berardi[1], Vincenzo Giuliano[2], and Donato Malerba[1]

[12]Dipartimento di Informatica
Università degli Studi di Bari, via Orabona, 4 - 70126 Bari - Italy
[1]{berardi,malerba}@di.uniba.it
[2]enzoju@hotmail.com

Abstract. Information in text form remains a greatly unexploited source of biological information. Information Extraction (IE) techniques are necessary to map this information into structured representations that allow facts relating domain-relevant entities to be automatically recognized. In biomedical IE tasks, extracting patterns that model implicit relations among entities is particularly important since biological systems intrinsically involve interactions among several entities. In this paper, we resort to an Inductive Logic Programming (ILP) approach for the discovery of mutual recursive theories from text. Mutual recursion allows dependencies among entities to be explored in data and extraction models to be applied in a context-sensitive mode. In particular, IE models are discovered in form of classification rules encoding the conditions to fill a pre-defined information template. An application to a real-world dataset composed by publications selected to support biologists in the task of automatic annotation of a genomic database is reported.

1 Introduction

The last decade has witnessed an unexampled expansion of biomedical data and related literature. Advances of genome sequencing techniques have mainly risen an overwhelming increase in the literature discussing discovered genes, proteins and their role in biological processes. The ability to survey this literature and extract relevant portions of information is crucial for researchers in biomedicine. However, finding explicit entities (e.g., a protein or a kinase) and facts (e.g., phosporylation and interaction relationships) in unstructured text is a time consuming and boring task because of the size of available resources, data sparseness and continuous updating of information. Information Extraction (IE) is the process of mapping unstructured text into structured form, such as knowledge bases or databases, by filling predefined templates of information describing objects of interest and facts about them. This motivates the interest of IE and text mining practitioners towards the biomedical field [12, 19, 13].

In a machine learning perspective, IE can be tackled as a classification task, where classification models composed by rules or patterns encoding the conditions to fill a given slot of a template of interest are learned from a set of

annotated texts (i.e., examples of filled templates) [17]. Although natural language research has widely made use of statistical techniques (e.g., hidden Markov models and probabilistic context-free grammars) because of their robustness and wide coverage peculiarities, logic-based approaches are able to overcome some intrinsic defects of statistical approaches due to their inability to cope with the level of semantic interpretation and the linguistically impoverished nature of discovered models that are difficult to interpret and extend [16]. Indeed, logical approaches, such as those that are developed in the Inductive Logic Programming (ILP) framework, allow to naturally encode natural language statements representing both data and background knowledge and to learn or revise these logical representations [8]. Moreover, IE tasks can be naturally framed in the ILP relational setting where data have a relational structure and examples can be related each other. In the literature, there are several works that take advantages of ILP principles to learn rule-based models from logical representations of texts [1, 9, 14, 10, 4], whereas few attempts have been conducted to solve IE problems from biomedical texts [5, 11], despite the fact that it promises to become a major application area where ILP may converge [7]. Difficulties are due to the complexity of the biomedical language which is characterized by inconsistent naming conventions, that is, ambiguities occurring when the same term is used to denote more than one semantic class (e.g., p53 is used to specify both a gene and a protein) or when many terms lead to the same semantic class (abbreviations, acronym variations), continuous creation of new biological terms or evolutions of the same biological object (e.g., genes are renamed once their function is known), non standard grammatical structures as well as domain-specific jargon combined with English. This makes the data processing phase in the learning process really difficult. On the other hand, it is available a large amount of controlled vocabularies, lexicons and ontologies that can be exploited both in the data processing and reasoning steps. This further motivates the use of an ILP approach since it allows to naturally handle explicitly expressed background knowledge.

In biomedical IE tasks, extracting patterns that model implicit relations among entities is particularly important since biological systems intrinsically involve interactions among several entities, e.g., genes and proteins interacting in regulation networks. Implicit relations can be captured in form of mutual recursive patterns, that is, patterns relating more than one entity of interest. Mechanisms supporting mutual recursion exploration in the space of candidate patterns allow hidden dependencies among entities to be discovered in data and extraction models to be applied in a context-sensitive mode.

In this paper, IE models are discovered in form of classification rules including mutual recursive definitions of classes of interest, where each class plays the role of a template slot while recursive patterns are searched among slots of the same template. In an ILP perspective, classification is tackled as a concept learning problem in a multiple predicate learning framework. In the following section, we describe how the annotation of a genomic databases can be supported by properly defining a biomedical information extraction problem. In Section 3, representation and algorithmic issues faced in the ATRE system to discover con-

cept dependencies are briefly described. Data preparation techniques adopted to process data and the representation model used to describe data and background knowledge are reported in Section 4. Results obtained on a real-world dataset composed by publications concerning studies on mitochondrial pathologies are reported in Section 5. Finally, some conclusions are drawn in Section 6.

2 The Information Extraction problem

The application we are addressing concerns the annotation in the HmtDB resource [1] of variability and clinical data associated to mitochondrial pathological phenotypes [2]. Currently, HmtDB stores data from healthy subjects while variability and clinical data are manually extracted from published literature. A peculiarity of this fragment of the scientific literature is that biomedical documents are organized according to a regular section structure (composed by Abstract, Introduction, Methods, Results and Discussion) and that often biologists already know which part of the documents may contain a certain kind of information. This suggests to conduct the IE process in a local way to pre-categorized sections of interest [3]. Indeed, selecting relevant portions of text is a prerequisite step to IE as the lack of robustness and data sparseness make IE methods inapplicable to large corpora and irrelevant documents. In this application, selected publications concern mitochondrial mutations and biologists are interested in automating the identification of occurrences of specific biological objects (i.e., mitochondrial mutations) and their features (i.e., type, position, involved nucleotides, expressing locus, related pathology) as well as the particular method and experimental setting adopted in the scientific study (i.e., dimension, age, sex, nationality of the sample) discussed in the publication. Each of these information to be annotated can be considered as a single entity of interest, but more effective and expressive "extractors" might be mined when instances of relations among objects are modelled. Implicit relations among relating entities can be mined in data when observations include some relational knowledge, e.g., by simply describing word neighbourhood in text or when a domain-specific taxonomy is available, distances among textual objects can be described on the basis of the path in the hierarchy linking categories to which pairs of textual objects belong. More complex relations can be modelled by using ontologies of is_a or part_of relations that allow to recognize sentences containing specializations of concepts expressed by previous sentences.

Considering the following example of a text fragment of the collection:

```
Cytoplasts from two unrelated patients with MELAS (mitochondrial
myopathy, encephalopathy, lactic acidosis, and strokelike episodes)
harboring an A-*G transition at nucleotide position 3243 in the
tRNALeU(UUR) gene of the mitochondrial genome were fused with human
cells lacking endogenous mitochondrial DNA (mtDNA)
```

[1] http://www.hmdb.uniba.it/

MELAS is an instance of the *pathology* associated to the mutation under study, $A-*G$ is an instance of the *substitution* that causes the mutation, *transition* is the *type* of the mutation, 3243 is the *position* in the DNA where the mutation occurs, and $tRNALeU(UUR)$ gene is the instance of the *gene* correlated to the mutation.

By modelling the sentence structure, annotation rules that take into account the dependence between these entities can be discovered, as the following logical clauses show:

```
substitution(X) ← follows(Y,X), type(Y).

type(X) ← distance(X,Y,3), position(Y),
   word_between(X,Y,‘‘nucleotide position’’).
```

In the above example, it is noteworthy that *type*, *substitution* and *position* are classes of entities of interest, and that learning their classification models independently could not lead to the expected result. Most of the studies on inductive learning make the implicit assumption that concepts are independent (independence assumption). In many real applications, like in this one, this is not always true since capturing dependence among entities may lead to more accurate extraction models.

3 Learning Concept Dependencies

Rules for automated entity extraction are automatically learned in form of mutually recursive patterns by means of ATRE [2] [15]. ATRE is an inductive learning system that supports multiple concept learning. In multiple concept learning, the aim is to learn for each concept a set of interacting predicate definitions or properties that hold among various predicates. In this application, each concept plays the role of an annotation class (i.e., template slot) and the learning problem is formulated as a single-class problem, namely a textual object can not be multi-annotated. ATRE solves the following learning problem:

Given

- a set of concepts $C_1, C_2, \ldots, C_r$ to be learned,
- a set of observations O described in a language $\mathcal{L}_O$,
- a background knowledge BK described in a language $\mathcal{L}_{BK}$,
- a language of hypotheses $\mathcal{L}_H$ that defines the space of hypotheses S_H
- a user's preference criterion PC,

Find a (possibly recursive) logical theory $T \in S_H$, defining the concepts $C_1, C_2, \ldots, C_r$, such that T is complete and consistent with respect to the set of observations and satisfies the preference criterion PC.

[2] http://www.di.uniba.it/ malerba/software/atre

The language of hypotheses $\mathcal{L}_H$ and the language of background knowledge $\mathcal{L}_{BK}$ is that of linked, range-restricted definite clauses. The language of observations is *object-centered*, that is, observations are represented as ground multiple-head clauses, called *objects*, which have a conjunction of simple literals in the head that are examples of the concepts $C_1, C_2, \ldots, C_r$. They can be considered either positive or negative according to the problem definition. In this application domain, the set of concepts to be learned is defined by means of a set of predicates `annotation(X)=annotation_class`. We are interested in finding rules which predict the class label of a textual object, namely a predicate definition for each class. No rule is generated for the case `annotation(X)=no_tag` that corresponds to negative examples. The main assumption made in ATRE is that each object contains examples explained by some base clauses of the underlying recursive theory. Therefore, by choosing as seeds *all* examples of different concepts represented in the head of one training object, it is possible to induce some of the correct base clauses. A parallel exploration of all candidate seeds is feasible and mutually recursive concept definitions will be generated only after some base clauses have been added to the theory.

Therefore, the search space is a forest of as many search-trees as the number of chosen seeds. Each search-tree is rooted with a unit clause and ordered by generalized implication. The forest can be processed in parallel by as many concurrent tasks as the number of search-trees (parallel-conquer search). Each task traverses the specialization hierarchy top-down through a separate-and-conquer strategy, but synchronizes traversal with the other tasks at each level. Search proceeds towards deeper and deeper levels of the specialization hierarchies until at least a user-defined number of consistent clauses is found. Task synchronization is performed after that all "relevant" clauses at the same depth have been examined. A supervisor task decides whether the search should carry on or not on the basis of the results returned by the concurrent tasks. When the search is stopped, the supervisor selects the "best" consistent clause according to the user's preference criterion. In this work, short rules, which explain a high number of positive examples and a low number of negative examples, are preferred.

This separate-and-parallel-conquer search strategy provides us with a solution to the problem of *interleaving* the induction of distinct concept definitions. Mined models reflecting dependencies among annotations may enable a context-sensitive recognition of them when the annotation is automatically performed.

4 Data preparation

The dataset is composed by a set of manually annotated pre-categorized texts. In the observation language adopted by ATRE, the body of a clause contains descriptions of texts reporting information obtained by a preprocessing phase, while related positive and negative examples are reported in the head of the clause on the basis of expert users' annotations. Counterexamples for all the classes of annotations to be learned are all the unlabelled tokens. Therefore, each text generates as many training examples as the number of described tokens.

Annotated texts are preprocessed by means of natural language facilities provided in the GATE (General Architecture for Text Engineering) infrastructure [6]. In particular, we exploit the ANNIE (A Nearly-New IE system) component to perform tokenisation, sentence splitting, part-of-speech tagging, named-entity recognition (e.g., persons, locations, organizations), mapping into dictionaries. We use both predefined dictionaries available with ANNIE (e.g., organization names, job title, geographical locations, dates, etc.) and domain-specific dictionaries that categorize biological entities such as diseases, enzymes, genes, etc. General domain dictionaries are used to disambiguate some terms (e.g. places and geographical locations are useful to recognize terms about the ethnic origin of the sample). Biology dictionaries are flat dictionaries of entities that are peculiar of the mitochondrial genome, they include lists of names about diseases, genes, methods of analysis, nucleic acids, enzymes, etc. Domain-specific dictionaries are also useful to perform a rough resolution of acronyms that in this domain is one of the sources of redundancy and ambiguity in data. This text engineering framework allows also to define user-specific components to integrate in a pipeline of text processing. For instance, it includes a finite-state transduction engine to recognize regular expressions over processed texts. We define regular expressions to identify appositions occurring in texts and some particular numeric and alphanumeric strings that are really frequent in this domain. Stopwords are removed, such as articles, adverbs, prepositions etc. (taken from Glimpse-glimpse.cs.arizona.edu), stemming is performed by means of Porter's algorithm for English texts [18].

4.1 Data Representation

The relational representation used to describe data considers a text fragment as the composition of smaller passages described in terms of properties of occurring tokens and relations among them. Properties express statistical (e.g., token frequency), lexical (e.g., alphanumeric, capitalized token), structural (e.g., structure of complex tokens such as alphanumeric strings, abbreviations, acronyms, hyphenated tokens), syntactical (e.g., singular/plural proper/not proper nouns, base/conjugated verbs) and domain-specific knowledge (e.g., an entity belonging to a dictionary). Relations describe structural properties, such as the composition of sentences in passages of text and tokens in chunks or directly in sentences. Properties or attributes of a token are represented by unary descriptors, while relations between two tokens are represented by binary descriptors. Each token and sentence is given a unique identifier based on its ordering within the given text. The predicate `word_to_string` maps an identifier to the corresponding stemmed token, `word_frequency` expresses the relative frequency of a token in the given text, `type_of` refers to morphological features as `allcaps`, `mixedcaps`, `upperinitial`, `numeric`, `percentage`, `alphanumeric`, `real number`. Part-of-speech are encoded by the predicate `type_pos`. Semantics is added by the `word_category` predicate. Structural knowledge is expressed by means of the binary predicate `follows` that states the relation of successor among tokens, while complex tokens are described by using `s_part_of` relations

on component tokens, `first` and `last` predicates that state strings corresponding to the first and second part of an hyphenated token, `lenght` predicate defining the lenght of component tokens, some predicates (e.g., `first_is_numeric`, `middle_is_char`) stating the lexical nature of tokens composing an alphanumeric string. Relational knowledge is also asserted with respect to domain dictionaries by expressing the distance among categorized tokens in the context of a sentence (`distance_word_category`). Numeric knowledge is handled by ATRE by means of on-line discretization algorithm.

In this application, we have basically defined two templates, one for the entity *mutation* composed by the following slots: *position* (i.e., position in the DNA where the mutation occurs), *type* (i.e., type of the mutation: insertion, deletion, translation, substitution, etc.), *type_position* (i.e., pieces of the DNA involved in the mutation and at which position in the DNA), *locus* (gene involved in the mutation), *substitution* (i.e., type of substitution: which nucleotide is substituted by which other). The other template concerns the *subjects* under study that is described by the following slots: *method* (method of analysis of mutations), *nationality* (geographic origin of the sample population that is under study), *category* (category of the sample population, e.g., families, single individuals, patients, etc.), *number* (dimension of the sample population), *pathology* (pathologies affecting the population).

Concerning the unit of observation for the learning step, namely the dimension of the context of example annotations, we restrict observations only to sentences containing at least a positive example (target sentences). Moreover, no relation among sentences is currently used. Hence, the extraction is local to sentences. This is to prevent the generation of unbalanced data sets from which IE systems typically suffer since only a very small number of phrases contains examples.

Following ATRE's language of observations, the body of an object describes the tokens composing a sentence while the head describes annotations associated to tokens. All literals in the head of the clause are examples (either positive or negative) of the concept `annotation(_)`. The complete list of concepts of interest for this domain is obtained by varying the label associated to the `annotation(_)` predicate in the set of annotation class values corresponding to template slots. It is noteworthy that models for automatic entity extraction can be learned by looking for dependencies among slots intra-template as well as slots inter-template. An instance of training observation is reported in the following:

```
annotation(22)=no_tag, ..., annotation(27)=pathology,...,
annotation(35)=no_tag, section(1)=abstract ←
sentence(21)=true, part_of(21,22)=true, ..., part_of(21,35)=true,
word_to_string(22)=preval, word_to_string(23)=trans-membran, ...,
word_to_string(34)=t14634c, word_to_string(35)=famili,
type_of(25)=upperinitial, ..., type_of(34)=alphanumeric,
s_part_of(23,36)=true, s_part_of(23,37)=true, first(23)=trans,
last(23)=membran, lenght(36)=5, lenght(37)=7,
```

```
first(24)=t, last(24)=c, lenght(24)=7, first_is_char(24)=true,
middle_is_numeric(24)=true, last_is_char(24)=true, ...
type_POS(22)=nn, ..., type_POS(35)=nns, word_frequency(22)=1,
..., word_frequency(35)=3, word_category(27)=ethnic_group, ...,
word_category(33)=disease, distance_word_category(27,29)=2, ...,
distance_word_category(32,33)=1, follows(22,23)=true,
follows(23,24)=true, ..., follows(34,35)=true
```

The constant 1 denotes the whole text, which belongs to an abstract of the collection, the constant 21 denotes the sentence described in this clause, while the constants 22, 23, ..., 35 denote identifiers of tokens that are described in the body of the clause.

4.2 Background Knowledge

The specification of the following domain specific knowledge:

```
follows(X,Z)=true ← follows(X,Y)=true, follows(Y,Z)=true
```

permits to define in a transitive way the relation of "successor" among tokens, while some domain knowledge can be specified to reduce redundancy in token values by expressing a sort of synonymy relation among biological terms, such as by means of the following clauses:

```
word_to_string(X)=transition ← word_to_string(X)=transversion
word_to_string(X)=substitution ← word_to_string(X)=replacement
```

that allow ATRE to generalize over tokens with the same meaning.

Moreover, in ATRE, it is possible to prevent the use of some predicates that are involved in the body of training objects and, rather, to use some other predicates that are intensionally defined in the background knowledge. This allows to support a form of abstraction in the inference strategy, that is, to perform a shift of representational language in order to eliminate superfluous details from the representation language. For instance, the following statements allow to compact some patterns into a unique predicate that unburdens the comprehensibility of the mined models.

```
char_number_char(X)=true ← first_is_char(X)=true,
      middle_is_numeric(X)=true, last_is_char(X)=true

number_char_char(X)=true ← first_is_numeric(X)=true
      middle_is_char(X)=true, last_is_char(X)=true

char_char_number(X)=true ← first_is_char(X)=true,
      middle_is_char(X)=true, last_is_numeric(X)=true
```

5 Experiments

We considered a data set composed by seventy-one papers concerning mithocondrial mutations selected for the annotation of HmtDb. We considered the abstract of each paper and we present results obtained for annotation classes related to the *mutation* template. The total number of annotated tokens is 355, that is, 1.68 tokens per target sentence and 8.65 per abstract. They correspond to about 10.3% of the total number of tokens that are described in the data set. The remaining tokens are considered as *no tagged* tokens (negative examples).

Table 1. Distribution of examples per folds.

Fold	**# abstract**	**# sentences**	**# locus**	**# position**	**# substitution**	**# type**	**# type_position**	**# no_tag**	**# literals in body**
F1	9	36	16	12	4	8	12	424	2424
F2	9	40	27	13	13	5	4	474	2552
F3	13	40	22	14	6	17	0	550	3098
F4	13	34	16	5	5	17	24	553	3260
F5	15	39	24	15	8	8	18	524	3083
F6	12	27	14	6	2	6	31	531	3199
Total	41	211	119	38	55	61	89	3085	17793

Performances are evaluated by means of a 6-fold cross-validation, that is, the set of seventy-one papers is firstly divided into six blocks (or folds) (Table 1), and then, for every block, ATRE is trained on the remaining blocks and tested on the hold-out block. Results have been evaluated on the basis of different perspectives on accuracy of a classifier. In particular, we computed for each concept, the number of omission and commission errors and the value of precision and recall. *Omission* errors occur when annotations of tokens are missed, while *commission* errors occur when wrong annotations are "recommended" by a rule. The omission measure is reported as the ratio of the number of omission errors and the number of positive examples, while the commission measure as the ratio of the number of commissions and the total of examples. The *recall* measure is computed as the ratio of positive examples correctly annotated (i.e., true positives) and the sum of true positives and the omissions (i.e., false negatives). The *precision* measure is computed as the ratio of true positives and the sum of true positives and the commissions (i.e., false positives). Experimental results are reported in Table 2 for each trial and average values on errors are also given.

We can observe that there is a high variability among trials. This is mainly due to an heterogeneous distribution of examples that leads to different degrees of data sparseness. However, the percentage of commission errors is very low with respect to the percentage of omission errors (the system misses annotations rather than suggesting wrong annotations) independently on the trail. This means that learned rules are quite specific. Some explanations can be drawn

Table 2. Experimental results: percentage values are reported.

Fold	**locus**		**position**		**substitution**		**type**		**type_position**	
	omiss.	**comm.**	**omiss.**	**comm.**	**omiss.**	**comm.**	**omiss.**	**comm.**	**omiss.**	**comm.**
F1	75	0.18	1	0	25	0	1	0.37	0	0
F2	40.7	0.31	46.15	1.54	0	3.08	80	0	0	3.08
F3	68.2	0.65	1	0	33.3	0	1	0	–	0.33
F4	68.75	0.32	1	0	40	0.32	88.23	0	0	0
F5	62.5	0.33	53.3	0.16	0	0	50	0.16	0	0%
F6	64.3	0.34	1	0	0	0.34	50	0.169	19.35	0.34
Avg	*63.24*	*0.35*	*83.25*	*0.28*	*16.39*	*0.16*	*78.04*	*0.117*	*3.87*	*0.16*
St.D.	*11.84*	*0.157*	*26.05*	*0.619*	*18.57*	*0.177*	*22.99*	*0.148*	*8.65*	*0.178*
Prec.	**76.2**		–		**82.7**		–		**76.5**	
Rec.	**36.75**		**16.75**		**83.61**		**21.9**		–	

by considering the complexity of learned theories described in Table 3 that reports coverage rates of discovered clauses as the ratio of the number of clauses and the number of training examples per trial. Indeed, we find that coverage rate is low for annotation classes affected by a more evident difference among the number of commissions and omissions (*locus*, *type*, *position*). Low coverage rate of learned theories explains also some low recall values. This can be due to the preprocessing module that is not completely apt to manage the variety of morpho-syntactic variations on the same term that affect this type of domain. By scanning the learned theories, we discover that for some annotation classes, many rules take into account the information on the specific occurrence of a token (predicate *word_to_string*) rather than involving information on the context of the example. This might also explain the nature of some commission errors such as in the case of *locus*. Specificities of learned theories is also due to the low percentage of positive examples with respect to negatives. Best performances are obtained on the *substitution* class for which the system learns more general and accurate theory. Indeed, examples of this class are the most homogeneous and the preprocessing module is able to produce discriminative descriptions.

Table 3. Experimental results. Complexity of the learned theory.

Fold	**locus**	**position**	**substitution**	**type**	**type_position**
	# clauses/ # pos. ex.	**# clauses/ # pos. ex.**	**# clauses/ # pos. ex.**	**# clauses/ # pos. ex.**	**# clauses/ # pos. ex.**
F1	37/103	31/53	3/34	25/53	6/77
F2	40/82	20/52	3/19	21/56	6/108
F3	34/97	18/51	2/32	21/44	5/89
F4	41/103	20/60	3/92	19/34	6/55
F5	38/95	17/50	3/30	24/53	6/49
F6	47/105	22/59	3/36	23/55	2/58
Avg	40.72	39.47	8.74	45.8	7.59

For the sake of completeness, some clauses learned by ATRE have been analyzed. Some of them follow:

```
annotation(X1)=type_position ←
   char_number_char(X1)=true

annotation(X1)=type_position ←
   type_of(X1)=alphanumeric, length(X1) ∈ [5..6]

annotation(X1)=position ← follows(X2,X1)=true,
   type_of(X1)=numeric, follows(X1,X3)=true,
   word_category(X3)=gene, word_to_string(X2)=position
```

The first rule states that $X1$ is annotated as *type_position* if it is an alphanumeric token composed by a char, a number and another char. This is one of the first rule that ATRE adds to the theory, that generally are the most general rules that cover many examples. Actually, information on *type_position* of a mutation are tokens such as $A1262G$, that means that A is substituted by G at position 1262 of the DNA. The second rule concerns the same concept and it states that $X1$ is annotated as *type_position* if it is an alphanumeric token which is about 5 long. The third rule states that $X1$ is annotated as *position* if it is a numeric token that is preceded by the token "position" and followed by a token of the "gene" category. This rule captures patterns like "an A-to-G mutation at *position 3426 (tRNALeu)*".

It is noteworthy that ATRE is able to discover meaningful dependencies as the following rules show:

```
annotation(X1)=position ← follows(X2,X1)=true
   annotation(X2)=substitution, follows(X3,X1)=true,
   follows(X1,X4)=true, word_frequency(X4) ∈ [6..6],
   annotation(X3)=type, follows(X1,X5)=true,
   annotation(X5)=locus, word_frequency(X1) ∈ [1..2]
```

This rule states that $X1$ is annotated as *position* if it is preceded by two tokens that have been annotated as *type* and *substitution*, respectively. Moreover it is followed by a token occurring about 6 times in the abstract and that is followed by a *locus* annotation. Finally, $X1$ is quite infrequent in the abstract.

```
annotation(X1)=position ← follows(X2,X1)=true
   annotation(X2)=substitution, follows(X1,X3)=true,
   annotation(X3)=locus, follows(X4,X1)=true, word_to_string(X4)=np
```

This rule states that $X1$ is annotated as *position* if it is preceded by two tokens, the first is the string "np" and the second has been annotated as *substitution*. Moreover, it is followed by a *locus* annotation.

Through this first-order logic formalism, the automatic entity extraction task can be reformulated as a matching test between a logic formula that describes a model and another logic formula that represents the properties of the text to be annotated. The antecedent of a rule describes properties that should hold between some tokens of the sentence, while the consequent specifies the annotation class of some token involved in the antecedent part.

6 Conclusions

Biomedical information extraction is an appealing task for ILP thanks to its ability to reason in presence of logically described examples and abundant domain knowledge. In this paper, we have proposed an application of recursive logical theories learning to a real-world IE task on the biomedical literature. Recursive rules are discovered by inducing mutually dependent definitions of predicates by means of the ILP system ATRE. This system allows us to discover meaningful dependencies among biomedical entities of interest that reflect implicit relations among entities. These can also subsume relations at the semantic level, such as the association of a mutation type with the responsible/involved gene. "Implicit" refers to the fact that the kind of relation is not found out. Moreover, results show that ATRE performs well when descriptions of examples are safe from inconsistencies and training observations are sufficiently homogeneous; anyway, it leads to low recall values when the inaccurate data preprocessing causes specificity of learned theories.

As future work, we plan to investigate the use of additional domain dictionaries such as taxonomical dictionaries to reduce redundancy in data as well as domain-specific tools for acronym and abbreviations resolution. Evaluation of the system on some recently made available biomedical datasets for ILP [7] will also be conducted to test performances on noisy free data. Combination of logical theories induction with probabilistic measures to handle noisy data is also worth to be investigated. Finally, we plan to explore the application of association rule mining from text to discover implicit relations among entities in form of strong co-occurrences of entities as alternative way to mine data.

References

1. J. S. Aitken. Learning information extraction rules: An inductive logic programming approach. In *ECAI*, pages 355–359, 2002.
2. M. Attimonelli, M. Accetturo, M. Santamaria, D. Lascaro, G. Scioscia, G. Pappada, M. Tommaseo-Ponzetta, and A. Torroni. Hmtdb, a human mitochondrial genomic resource based on variability studies supporting population genetics and biomedical research. *BMC Bioinformatics*, 1(6), 2005.
3. M. Berardi, M. Ceci, and D. Malerba. A hybrid strategy for knowledge extraction from biomedical documents. In *ICDAR workshop on "Neural Networks and Learning in Document Analysis and Recognition"*, Seoul, Korea, 2005.

4. M. E. Califf and R. J. Mooney. Relational learning of pattern-match rules for information extraction. In *AAAI '99/IAAI '99*, pages 328–334. American Association for Artificial Intelligence, 1999.
5. M. Craven and J. Kumlien. Constructing biological knowledge bases by extracting information from text sources. In *ISMB*, pages 77–86, 1999.
6. H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. Gate: A framework and graphical development environment for robust nlp tools and application. In *Proc. of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*, Philadelphia, USA, 2002.
7. J. Cussens and C. Nedellec, editors. *Proceedings of the 4th ICML Workshop on Learning Language in Logic (LLL05)*, Bonn, Germany, 2005.
8. S. Dzeroski, J. Cussens, and S. Manandhar. An introduction to inductive logic programming and learning language in logic. In J. Cussens and S. Dzeroski, editors, *Learning Language in Logic*, volume 1925, pages 3–35. 2000.
9. S. Ferilli, N. Fanizzi, and G. Semeraro. Learning logic models for automated text categorization. In *AI*IA 01: Proceedings of the 7th Congress of the Italian Association for Artificial Intelligence on Advances in Artificial Intelligence*, pages 81–86, London, UK, 2001. Springer-Verlag.
10. D. Freitag. Toward general-purpose learning for information extraction. In *Proceedings. of the 17th int. conf. on Computational linguistics*, pages 404–408, Morristown, NJ, USA, 1998. Association for Computational Linguistics.
11. M. Goadrich, L. Oliphant, and J. W. Shavlik. Learning ensembles of first-order clauses for recall-precision curves: A case study in biomedical information extraction. In *ILP*, pages 98–115, 2004.
12. L. Hirschman, J. C. Park, J. Tsujii, and L. Wong. Accomplishments and challenges in literature data mining for biology. *Bioinformatics*, 18:1553–61, 2002.
13. L. Hirschman, A. Yeh, C. Blaschke, and A. Valencia. Overview of biocreative: critical assessment of information extraction for biology. *Bioinformatics*, 6, 2005.
14. M. Junker, M. Sintek, and M. Rink. Learning for text categorization and information extraction with ilp. In *Learning Language in Logic*, pages 247–258, 1999.
15. D. Malerba. Learning recursive theories in the normal ilp setting. *Fundamenta Informaticae*, 57(1):39–77, 2003.
16. R. Mooney. Learning for semantic interpretation: Scaling up without dumbing down. In J. Cussens, editor, *Proc. of the 1st Workshop on Learning Language in Logic*, pages 7–15, Bled, Slovenia, 1999.
17. C. Nedellec, editor. *Machine Learning for Information Extraction in Genomics - State of the art and perspectives*, volume 138 of *Studies in Fuzziness and Soft Computing*. Springer Verlag, 2004.
18. M. F. Porter. *Readings in information retrieval*, chapter An algorithm for suffix stripping, pages 313–316. 1997.
19. H. Shatkay and R. Feldman. Mining the biomedical literature in the genomic era: an overview. *Journal of Computational Biology*, 10:821–855, 2003.

A-Priori Verification of Web Services with Abduction

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

[1] ENDIF, Università di Ferrara - Via Saragat, 1 - 44100 Ferrara (Italy).
Email: {marco.alberti|marco.gavanelli|evelina.lamma}@unife.it
[2] DEIS, Università di Bologna - Viale Risorgimento 2 - 40126 Bologna (Italy).
Email: {fchesani|pmello|mmontali}@deis.unibo.it

Abstract. Although stemming from very different research areas, Multi-Agent Systems (MAS) and Service Oriented Computing (SOC) share common topics, problems and settings. A common problem is the need to formally verify the conformance of individuals (Agents or Web Services) to common rules and specifications (resp. Protocols/Choreographies), in order to provide a coherent behaviour and to reach the user's goals.
In previous publications, we developed a framework, $\mathcal{S}$CIFF, for the automatic verification of compliance of agents to protocols. The framework includes a language based on abductive logic programming and on constraint logic programming for formally defining the social rules. Suitable proof-procedures to check on-the-fly and a-priori the compliance of agents to protocols have been defined.
Building on our experience in the MAS area, in this paper we make a first step towards the formal verification of web services conformance to choreographies in Abductive Logic Programming. We adapt the $\mathcal{S}$CIFF framework for the new settings, and propose a heir of $\mathcal{S}$CIFF, the framework A^{l}LoWS (Abductive Logic Web-service Specification). A^{l}LoWS comes with a language for defining formally a choreography and a web service specification. As its ancestor, A^{l}LoWS has a declarative and an operational semantics. We show examples of how A^{l}LoWS deals correctly with interaction patterns previously identified. Moreover, thanks to its constraint-based semantics, A^{l}LoWS deals seamlessly with other cases involving constraints and deadlines.

Note. An extended version of this paper will appear in the Proceedings of the Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06).

1 Introduction

The mating between high availability of network resources and the growing of software applications is breeding in recent years new technologies and software tools. Network ubiquity, joined together with the software engineering requirement to combine existing tools and divide the complexity of applications, is spawning new programming paradigms.

One of the most successful is Service Oriented Computing, one of the children of Object Oriented Computing. Web service technology is an important instance of Service Oriented Computing aiming at facilitating the integration of new applications, avoiding difficulties due to different platforms, languages, etc. In this context the way to build complex services from simpler ones is called *composition* and it is a very interesting and promising research area. Service composition promises to considerably reduce development time and costs by taking components off-the-shelf and joining them in a working application. Although very appealing, it leaves open questions, such as correctness of the composition, or ensuring interoperability of the web services. Choreographies propose an answer to such questions. The behaviour of the various web services is defined through a language (e.g., WS-CDL [5]), explaining the information flows amongst the components. However, the formal proofs of conformance of a web service to a choreography are not yet fully given.

Another technology descending from networks and artificial intelligence is the Multi Agent Systems (MAS). In the MAS area there exists a wide literature about checking the conformance of an agent to social rules (or protocols), both at run-time and at design-time. Baldoni et al. [4], amongst others, pointed out the similarities of requirements in the two areas of web service composition and multi agent systems. Both are devoted to define a collaboration between a collection of peers that share common goals. Both choreographies and societies should capture interactions and dependencies between interactions (time constraints, deadlines, control-flow dependencies, etc.). Both describe the external behaviour of members avoiding the internal details of the implementation of the peers.

Stemming from previous experience in multi-agent systems, we follow the path of Baldoni et al. and propose to apply agent techniques to web service composition. Within the SOCS european project [14], we proposed a formal language to define multi agent protocols [2]. We gave an abductive semantics to the devised language, and developed an abductive proof procedure, called $\mathcal{S}$CIFF [3], to check on-line the compliance of agents to protocols. More recently, we applied a variant of $\mathcal{S}$CIFF, called g-$\mathcal{S}$CIFF [1], to the problem of proving properties of a protocol itself (such as security properties). In this work, we apply these technologies to web service composition, and propose a framework, called A^{l}LoWS (Abductive Logic Web-service Specification), that exploits the variants of $\mathcal{S}$CIFF for checking the interoperability of web services.

2 The A^{l}LoWS framework

We propose an abductive based framework, A^{l}LoWS (Abductive Logic Web-service Specification), to verify conformance of web services to choreographies.

An Abductive Logic Program (ALP) [11] is a triple $\mathcal{P} \equiv \langle KB, \mathcal{E}, \mathcal{IC} \rangle$ where KB is a logic program, $\mathcal{E}$ is a set of predicates, called *abducibles*, that have no definition in KB, and $\mathcal{IC}$ is a set of *Integrity Constraints*, that must always hold true. The aim is to find a set $\Delta \subseteq \mathcal{E}$ that explains a goal G and such that the integrity constraints are satisfied

$$KB \cup \Delta \models G, \qquad KB \cup \Delta \models \mathcal{IC}.$$

A^{l}LoWS builds upon the tools and technologies developed in the SOCS project in the Multi Agent Systems area. In A^{l}LoWS, the behaviour of the web services is represented by means of *events*. Since we focus on the interactions between web services, events represent exchanged messages. We adopt the syntax used in [4]: a message is a term $m_x(Sender, Receiver, Content)$, where m_x is the type of message, and the arguments retain their intuitive meaning. We may omit some of the parameters when the meaning is clear from the context.

In A^{l}LoWS we have two types of events. Happened events are represented as $\mathbf{H}(Message, Time)$, where $Message$ has the syntax previously defined, and $Time$ is an integer, representing the time point in which the event happened. As we will see in the following, the **H** predicate can be abduced, when making hypotheses on the possible interactions. In other phases, they are considered as given a priori, thus considered as a defined predicate.

The second type of events are *expectations*. From both web services and choreography's viewpoint, given a past history of happened events, more events are expected to happen, in a conformant evolution. We represent expectations with the predicate $\mathbf{E}_X(Message, Time)$ expressing the fact that the corresponding event is expected to happen, in order to fulfil the coherent evolution, from the viewpoint of X (where X might be either the choreography or a web service). An expectation is called *fulfilled* if there exists a matching **H** event. For example, the expectation $\mathbf{E}(p(X), T)$ can be fulfilled by the event $\mathbf{H}(p(a), 1)$.

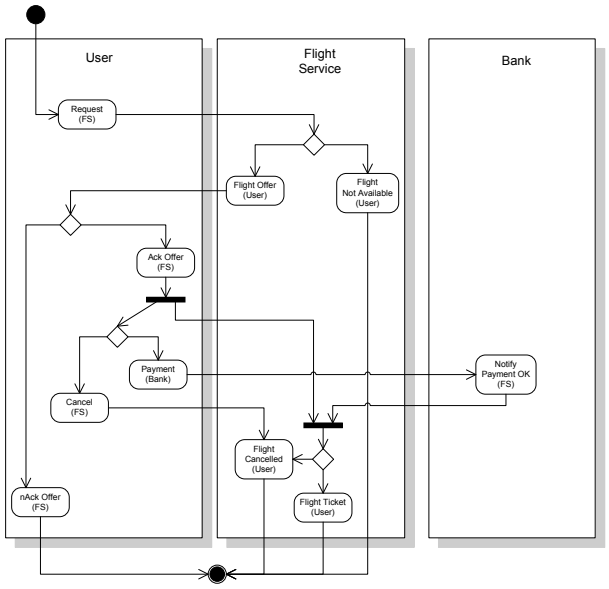


Fig. 1. Graphical representation of a simple choreography

Specification 2.1 The specification of the choreography shown in Fig. 1.

$$\mathbf{H}(request(User, FS, Flight), T_r) \rightarrow \mathbf{E}_{chor}(offer(FS, User, Flight, Price), T_o) \quad (1)$$
$$\vee\ \mathbf{E}_{chor}(notAvailable(FS, User, Flight), T_{na})$$
$$\mathbf{H}(offer(FS, User, Flight, Price), T_o) \rightarrow \mathbf{E}_{chor}(ackOffer(User, FS, Flight, Price), T_a) \quad (2)$$
$$\vee\ \mathbf{E}_{chor}(nAckOffer(User, FS, Flight, Price), T_a)$$
$$\mathbf{H}(ackOffer(User, FS, Flight, Price), T_a) \rightarrow \mathbf{E}_{chor}(payment(User, Bank, Price, FS), T_f) \quad (3)$$
$$\vee\ \mathbf{E}_{chor}(cancel(User, FS, Flight), T_f)$$
$$\mathbf{H}(ackOffer(User, FS, Flight, Price), T_a) \wedge \quad (4)$$
$$\mathbf{H}(notifyPayment(Bank, FS, Price), T_p) \rightarrow \mathbf{E}_{chor}(flightTicket(FS, User, Flight), T_f)$$
$$\vee\ \mathbf{E}_{chor}(flightCancelled(FS, User, Flight), T_f)$$
$$\mathbf{H}(cancel(User, FS, Flight), T_a) \rightarrow \mathbf{E}_{chor}(flightCancelled(FS, User, Flight), T_f) \quad (5)$$
$$\mathbf{H}(payment(User, Bank, P, Cred), T_p) \rightarrow \mathbf{E}_{chor}(notifyPayment(Bank, Cred, P), T_n) \quad (6)$$

2.1 Specification of a Choreography

A choreography describes, from a global viewpoint, the patterns of communication allowed in a system adopting such choreography [5]. The choreography specification defines the allowed messages: all messages that are not explicitly specified are forbidden. The choreography also enlists the participants, the roles the participants can play, and other knowledge about the web service interaction.

We specify a choreography by means of an ALP. A choreography specification $\mathcal{P}_{chor}$ is defined by $\mathcal{P}_{chor} \equiv \langle KB_{chor}, \mathcal{E}_{chor}, \mathcal{IC}_{chor} \rangle$. The abducibles $\mathcal{E}_{chor}$ include the happened events ($\mathbf{H}$) and the choreography's expectations ($\mathbf{E}_{chor}$).

The *Knowledge Base* (KB_{chor}) specifies declaratively pieces of knowledge of the choreography, such as roles descriptions, list of participants, etc. KB_{chor} is a set of clauses (a logic program); the clauses may contain in their body expectations about the behaviour of participants, defined literals, and constraints.

Choreography Integrity Constraints $\mathcal{IC}_{chor}$ are forward rules, of the form *Body* $\rightarrow$ *Head*, whose *Body* can contain literals and (happened, $\mathbf{H}$, and expected, $\mathbf{E}_{chor}$) events, and whose *Head* can contain (disjunctions of) conjunctions of expectations. The syntax of $\mathcal{IC}_{chor}$ is a simplified version of that defined for the $\mathcal{S}$CIFF Integrity Constraints [3]. In particular in A^{l}LoWS we do not need negative expectations and explicit negation.

In Fig. 1 a multi-party interaction is shown, expressed by the $\mathcal{IC}_{chor}$ in Spec. 2.1: the depicted scenario is about a User that wants to buy a flight ticket from a Flight Service, and pay by sending a payment order to a Bank.

CLP constraints [10] can be used to impose relations on any of the variables that occur in an expectation, like conditions on the role of the participants, or on the time instants the events are expected to happen. For example, time conditions might define orderings between the messages, or enforce deadlines.

A choreography can be *goal directed*, i.e. a specific goal $\mathcal{G}_{chor}$ can be specified: for example, a choreography used in an electronic auction system could have the goal of selling all the goods in the store. The syntax of the goal is the same as the body of a clause. If no particular goal is required, $\mathcal{G}_{chor} = true$.

Specification 2.2 The interface behaviour specification of the web service shown in Fig. 2.

$$\mathbf{H}(request(User, fs, Flight), T_r) \rightarrow \mathbf{E}_{fs}(offer(fs, User, Flight, Price), T_o) \quad (7)$$
$$\vee\ \mathbf{E}_{fs}(notAvailable(fs, User, Flight), T_{na})$$
$$\mathbf{H}(offer(fs, User, F, P), T_o) \rightarrow \mathbf{E}_{fs}(User, fs, ackOffer(User, fs, F, P), T_a) \quad (8)$$
$$\vee\ \mathbf{E}_{fs}(User, fs, nAckOffer(User, fs, F, P), T_a)$$
$$\mathbf{H}(notifyPayment(Bank, fs, Price), T_p) \rightarrow \mathbf{E}_{fs}(flightCancelled(fs, User, Flight), T_c) \quad (9)$$
$$\wedge T_p > T_a + \delta \wedge T_c > T_p$$
$$\vee\ \mathbf{E}_{fs}(flightTicket(fs, User, Flight), T_t)$$
$$\wedge T_t > T_p$$
$$\mathbf{H}(ackOffer(User, fs, Flight, Price), T_a) \rightarrow \mathbf{E}_{fs}(notifyPayment(Bank, fs, Price), T_p) \quad (10)$$
$$\vee\ \mathbf{E}_{fs}(User, fs, cancel(User, fs, Flight), T_c)$$
$$\mathbf{H}(cancel(User, fs, Flight), T_a) \rightarrow \mathbf{E}_{fs}(flightCancelled(fs, User, Flight), T_f) \quad (11)$$

2.2 Representing Web services

Similarly to the specification of a choreography, we describe the interface behaviour of a web service by means of an ALP. We restrict our analysis to the communicative aspects of the interface behaviour of a web service. A Web Service Interface Behaviour Specification $\mathcal{P}_{ws}$ is the ALP $\mathcal{P}_{ws} \equiv \langle KB_{ws}, \mathcal{E}_{ws}, \mathcal{IC}_{ws} \rangle$.

KB_{ws} and $\mathcal{IC}_{ws}$ are analogous to their counterparts in the choreography specification, except that they are an individual, rather than global, perspective: they represent the declarative knowledge and the policies of the web service.

$\mathcal{E}_{ws}$ is the set of abducible predicates: as for the choreographies, it contains both expectations and happened events. The expectations in $\mathcal{E}_{ws}$ can be divided into two significant subsets:

- expectations about messages whose sender is ws, $\mathbf{E}_{ws}(m_x(ws, A, Content))$, are interpreted as actions that ws intends to do;
- expectations about messages uttered by other participants to ws (of the form $\mathbf{E}_{ws}(m_x(A, ws, Content))$, with $A \neq ws$), can be intended as the messages that ws is able to understand.

In Fig. 2 the communicative part of a web service's interface behaviour is represented. The corresponding translation in terms of $\mathcal{IC}_{ws}$ is in Spec. 2.2.

3 Conformance: declarative semantics

Intuitively, conformance is the characteristics of a web service to comply to a choreography, provided that the other peers will behave according to the choreography. From the declarative semantics viewpoint, the test of conformance requires to assume further hypotheses about events ws expects to utter, and events that the choreography expects other peers to utter. We consider the predicate $\mathbf{H}$ as abducible, and use the web service's interface behaviour $\mathcal{P}_{ws}$ to foresee the messages the web service will send in every possible situation, provided that the other peers behave as specified by the choreography. Formally, all the messages the web service ws expects to send will be delivered, i.e.:

$$\mathbf{E}_{ws}(m_x(ws, R, C), T) \rightarrow \mathbf{H}(m_x(ws, R, C), T) \quad (12)$$

Symmetrically, the messages sent by other peers are those prescribed by the choreography specification $\mathcal{P}_{chor}$:

$$\mathbf{E}_{chor}(m_x(S,R,C),T), S \neq ws \rightarrow \mathbf{H}(m_x(S,R,C),T) \tag{13}$$

The possible interactions amongst the web service ws and the other peers will be the sets $\mathbf{HAP}^*$ satisfying equations 12 and 13.

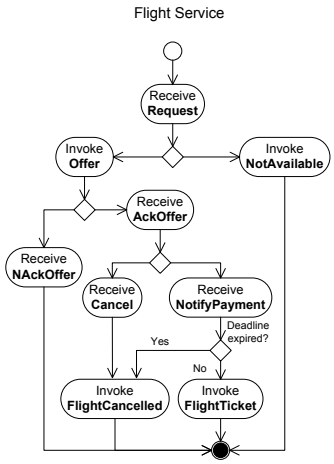


Fig. 2. Example of a behavioural interface

Definition 1. *Given the ALP* $\langle KB_U, \mathcal{E}_U, \mathcal{IC}_U \rangle$ *(where* $KB_U \triangleq KB_{chor} \cup KB_{ws}$, $\mathcal{E}_U \triangleq \mathcal{E}_{chor} \cup \mathcal{E}_{ws}$, $\mathcal{IC}_U \triangleq \mathcal{IC}_{chor} \cup \mathcal{IC}_{ws}$*), a* possible interaction *of a web service* ws *in a choreography* $chor$ *is a pair* $(\mathbf{HAP}^*, \mathbf{EXP}^*)$ *such that:*

$$KB_U \cup \mathbf{HAP}^* \cup \mathbf{EXP}^* \models G_U \tag{14}$$

$$KB_U \cup \mathbf{HAP}^* \cup \mathbf{EXP}^* \models \mathcal{IC}_U \tag{15}$$

$$KB_U \cup \mathbf{HAP}^* \cup \mathbf{EXP}^* \models (12) \cup (13) \tag{16}$$

(where by Eq. 16 we mean that equations 12 and 13 must hold). The set $\mathbf{HAP}^*$ *is also called* possible history.

When the goal G_U is *true*, the empty set is typically one of the possible histories. The empty history is often of little (or no) interest for proving conformance. When the interesting histories are only those containing at least one event, the expectation of such event can be inserted as the goal G_U. Typically, we use as goal the expectation (both from the web service's viewpoint, $\mathbf{E}_{ws}$, and from the choreography's viewpoint, $\mathbf{E}_{chor}$) of the first event of an interaction. This poses no serious restriction on the types of protocols that can be tested.

Example 1. Suppose a choreography prescribes the following protocol:

$$\mathbf{H}(ask(ws, R, X)) \rightarrow \mathbf{E}_{chor}(answer(R, ws, X)) \tag{17}$$
$$\mathbf{H}(answer(R, ws, X)) \rightarrow \mathbf{E}_{chor}(ack(ws, R, X)) \tag{18}$$

while the web service's integrity constraints contain only the first rule

$$\mathbf{H}(ask(ws, R, X)) \rightarrow \mathbf{E}_{ws}(answer(R, ws, X)).$$

Let $G_U = \mathbf{E}_{ws/chor}(ask(ws, peer, X))$.[3] Given the goal G_U, ws has the intention to send an *ask* message to the *peer*, so all the possible histories for G_U will contain the event $\mathbf{H}(ask(ws, peer, X))$. The *peer*'s behaviour is simulated through the rules in the choreography specification. Since the choreography has an expectation (generated by rule 17) $\mathbf{E}_{chor}(answer(peer, ws, X))$, this will become a happened event: $\mathbf{H}(answer(peer, ws, X))$. Now, rule 18 provides a choreography's expectation about the third message: ws is supposed to send an *ack*. But, as we can see from ws's specification, it does not have an expectation to send such message, so the simulation will not suppose it will comply to the choreography's expectation. So, the (only) possible history for the goal G_U is

$$\mathbf{HAP}^* = \{\mathbf{H}(ask(ws, peer, X)), \mathbf{H}(answer(peer, ws, X))\}. \tag{19}$$

In a possible history, messages sent by the other peers comply by definition to the choreography. However, the messages uttered by the web service ws under test might be non conformant. ws is conformant if all the possible histories are conformant. Also, ws should be able to understand the messages it receives, otherwise there might be requests which it is unable to serve. We require all possible histories to satisfy both the choreography and the web service expectations.

Definition 2. *A possible history* $\mathbf{HAP}^*$ *is* Feeble Conformant *if there exists a set* $\mathbf{EXP}$ *such that*[4]

$$KB_U \cup \mathbf{HAP}^* \cup \mathbf{EXP} \models G \tag{20}$$
$$KB_U \cup \mathbf{HAP}^* \cup \mathbf{EXP} \models \mathcal{IC}_U \tag{21}$$
$$\mathbf{HAP}^* \cup \mathbf{EXP} \models \mathbf{E}_{ws}(X) \rightarrow \mathbf{H}(X) \tag{22}$$
$$\mathbf{HAP}^* \cup \mathbf{EXP} \models \mathbf{E}_{chor}(X) \rightarrow \mathbf{H}(X) \tag{23}$$

A web service is feeble conformant if all the possible histories are feeble conformant. A pair $(\mathbf{HAP}^*, \mathbf{EXP})$ *is a* Feeble Conformant Interaction *if* $\mathbf{HAP}^*$ *is a feeble conformant history and* $\mathbf{EXP}$ *is a set of expectations satisfying equations (20-23) which is minimal with respect to set inclusion.*

Example 2. Cont. from Example 1. In the history of Eq. 19, the choreography's expectation for the message *ask* is unfulfilled, so ws is not (feeble) conformant.

[3] $\mathbf{E}_{ws/chor}$ represents an event expected both by the choreography and the web service.

[4] Note the difference between Eq. 22-23 and Eq. 12-13: Eq. 22 is used as a test, and requires *all* the expectations of the web service to be fulfilled, while Eq. 12 is used to *generate* the behaviour of the web service and imposes only the fulfilment of the expectations the web service has about *itself*. Analogously for Eq. 23 and 13.

Feeble conformance ensures that ws will utter all the messages requested by the choreography, but it does not require ws to avoid the forbidden messages. We extend feeble conformance to a stronger version.

A possible history is strong conformant if (it is feeble conformant and) all the happened events were expected both by the choreography and the web service. We include in this concept only the communications that involve the web service under observation (the other messages, exchanged between other peers in a multi-party interaction, are considered conformant).

Definition 3. *A feeble conformant interaction* (**HAP***, **EXP**) *is also a* Strong Conformant Interaction *if the following conditions hold:*

$$\mathbf{H}(m_x(ws, R, C)) \leftrightarrow \mathbf{E}_{chor}(m_x(ws, R, C)) \tag{24}$$
$$\mathbf{H}(m_x(S, ws, C)) \leftrightarrow \mathbf{E}_{ws}(m_x(S, ws, C)). \tag{25}$$

A Strongly Conformant History *is a history for which there exists a strongly conformant interaction. A web service is* Strongly Conformant *if all the possible histories are strongly conformant.*

Example 3. Let us change in the previous example the specifications of the choreography and of the web service, i.e., the web service specification is

$$\mathbf{H}(ask(ws, R, X)) \rightarrow \mathbf{E}_{ws}(answer(R, ws, X))$$
$$\mathbf{H}(answer(R, ws, X)) \rightarrow \mathbf{E}_{ws}(ack(ws, R, X))$$

and the choreography is

$$\mathbf{H}(ask(ws, R, X)) \rightarrow \mathbf{E}_{chor}(answer(R, ws, X)).$$

ws intends to send ack, so it will indeed send it in all the possible histories. The choreography does not prescribe this message. The possible history is

$$\mathbf{HAP}^*{}_2 = \{\mathbf{H}(ask(ws, peer, X)), \mathbf{H}(answer(peer, ws, X)), \mathbf{H}(ack(ws, peer, X))\}.$$

All expectations of the choreography are fulfilled by some message of ws, so ws is feeble conformant. However, it will also send an unrequested message, that might confound the other $peer$, undermining the interoperability. The ack message is unexpected by the choreography, therefore ws is not strong conformant.

3.1 Operational semantics

A^{l}LoWS operational semantics is based on the two versions of the $\mathcal{S}$CIFF proof procedure developed in the SOCS project. $\mathcal{S}$CIFF is sound and complete; it terminates for acyclic programs. The $\mathcal{S}$CIFF proof procedure considers the **H** events as a predicate defined by a set of incoming atoms; it is devoted to generate expectations corresponding to such history and to check that expectations indeed match with happened events. $\mathcal{S}$CIFF was developed to check the compliance of agents to protocols [3]. The $\mathcal{S}$CIFF proof procedure is based on a rewriting system transforming one node to another (or to others) as specified by rewriting steps called *transitions*. A node is defined by the

tuple $T \equiv \langle R, CS, PSIC, \mathbf{PEND}, \mathbf{HAP}, \mathbf{FULF}, \mathbf{VIOL} \rangle$, where R is the resolvent, CS is the constraint store à la CLP [10], $PSIC$ is a set of implications, derived from the $\mathcal{IC}$s, **HAP** is the history of happened events, and the set of expectations is partitiond into **PEND** (pending), **FULF** (fulfilled) and **VIOL** (violated). We cannot report here all the transitions, due to lack of space. As an example, the *fulfilment* transition is devoted to prove that an expectation $\mathbf{E}(X, T_x)$ has been fulfilled by an event $\mathbf{H}(Y, T_y)$. Two nodes are generated: in the first, X and Y are unified, and the expectation is fulfilled (i.e., it is moved to the set **FULF**); in the second the new constraint $X \neq Y$ is added to the constraint store CS. At the end of the computation, a *closure* transition is applied, and all the expectations remaining in the set **PEND** are considered as violated.

The g-$\mathcal{S}$CIFF proof procedure, instead, considers **H** an abducible predicate and provides both the set of expectations and the history that fulfils the goal. g-$\mathcal{S}$CIFF was used to prove properties of protocols, such as security [1]. It has the same transitions in $\mathcal{S}$CIFF; in the version adopted in this paper, it also contains as integrity constraints the rules 12 and 13. g-$\mathcal{S}$CIFF generates events, so it does not contain the *closure* transition, that enforces a closed world assumption on the set of happened events. A derivation without *closure* is called *open*.

In order to prove conformance, we apply the two proof procedures to the two phases implicitly defined in the previous section. We decompose the proof of feeble conformance into a *generative* phase and a *test* phase. In the generative phase, we generate, by means of g-$\mathcal{S}$CIFF, all the possible histories. Of course, those histories need not be generated as ground histories (the set of ground histories can be infinite), but intensionally: the **H** events can contain variables, possibly with constraints à la Constraint Logic Programming (CLP) [10].

In the test phase, we check with $\mathcal{S}$CIFF the compliance of the generated histories both with respect to the web service and the choreography specifications. If all the histories are conformant, the web service is feeble conformant to the choreography. Otherwise, if there exists at least one history that is not conformant, the web service is not (feeble) conformant.

Finally, we can prove strong conformance by checking that all the happened events were expected both by the choreography and by the web service. This can be performed during the second phase ($\mathcal{S}$CIFF) by adopting the same technique used in the fulfilment transition: if a **H** event matches both with an $\mathbf{E}_{ws}$ and a $\mathbf{E}_{chor}$ expectation, it is labelled *expected*; after the application of the *closure* transition, all events that were not expected are considered *unexpected*, showing that the web service was not strongly conformant.

3.2 Examples

Baldoni et al. [4] show various examples of conformance and non-conformance of a web service to a choreography, and propose a framework based on Finite State Automata, to prove conformance. We show how their examples are addressed in A^{l}LoWS, based on Computational Logics.

Web service with more capabilities In the first example of [4], the choreography specification defines only one allowed interaction: ws sends a message m_1 and the other peer will reply with m_2 (Eq. 26). The specification of ws is wider:

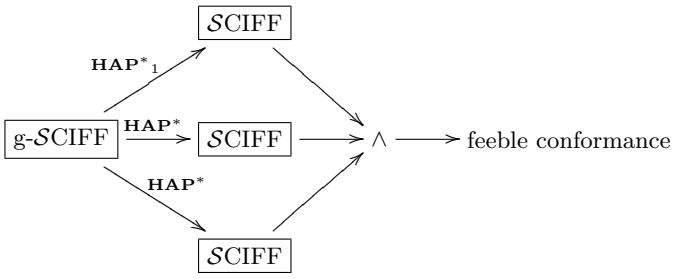


Fig. 3. A^{l}LoWS architecture

after m_1, ws accepts either m_2 or m_3 (Eq. 27).

$$\mathbf{H}(m_1(ws, X)) \rightarrow \mathbf{E}_{chor}(m_2(X, ws)) \tag{26}$$

$$\mathbf{H}(m_1(ws, X)) \rightarrow \mathbf{E}_{ws}(m_2(X, ws)) \vee \mathbf{E}_{ws}(m_3(X, ws)) \tag{27}$$

Baldoni et al. state that ws is conformant. In fact, in a legal conversation message m_3 will never be received by ws, so the interoperability is ensured.

The g-$\mathcal{S}$CIFF proof procedure is started with the goal containing the expectation, both from the web service's and from the choreography's viewpoint, of the first event: $G_U = \mathbf{E}_{ws/chor}(m_1(ws, X))$. g-$\mathcal{S}$CIFF abduces (Fig. 4) one possible history: $\mathbf{HAP}^* = \{\mathbf{H}(m_1), \mathbf{H}(m_2)\}$. There are two alternative sets of expectations from the viewpoint of ws, $\{\mathbf{E}_{ws}(m_1), \mathbf{E}_{ws}(m_2)\}$ and $\{\mathbf{E}_{ws}(m_1), \mathbf{E}_{ws}(m_3)\}$, but in the first phase the correspondence between expectations and happened events is not required (open derivation). In the second phase, the (only) generated history is checked; since there exists one set of expectations fulfilled by $\mathbf{HAP}^*$, ws is feeble conformant. Since in $\mathbf{HAP}^*$ there are no unexpected events, ws is also strong conformant.

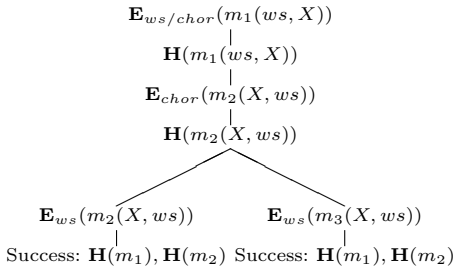


Fig. 4. g-$\mathcal{S}$CIFF derivation.

Missing capability The web service accepts as reply only m_2 (Eq 28), while the choreography defines as valid two interactions (Eq 29):

$$\mathbf{H}(m_1(ws, X)) \rightarrow \mathbf{E}_{ws}(m_2(X, ws)) \tag{28}$$

$$\mathbf{H}(m_1(ws, X)) \rightarrow \mathbf{E}_{chor}(m_2(X, ws)) \vee \mathbf{E}_{chor}(m_4(X, ws)). \tag{29}$$

In this case, g-$\mathcal{S}$CIFF provides two possible histories: $\mathbf{HAP^*}_1 = \{\mathbf{H}(m_1), \mathbf{H}(m_2)\}$ and $\mathbf{HAP^*}_2 = \{\mathbf{H}(m_1), \mathbf{H}(m_4)\}$. In phase 2, $\mathcal{S}$CIFF detects non conformance of $\mathbf{HAP^*}_2$, because the expectation $\mathbf{E}_{ws}(m_2(S, ws, C))$ remains unfulfilled in all possible derivation paths. This means that ws is blocked waiting for m_2, and will not process other messages, so it is not (feeble) conformant.

Wrong reply ws assumes to have the freedom to reply either m_2 or m_3 to a question m_1 (Eq. 30), while the choreography does not grant such a freedom: only m_2 is legal (Eq. 31):

$$\mathbf{H}(m_1(X, ws)) \rightarrow \mathbf{E}_{ws}(m_2(ws, X)) \vee \mathbf{E}_{ws}(m_3(ws, X)) \tag{30}$$

$$\mathbf{H}(m_1(X, ws)) \rightarrow \mathbf{E}_{chor}(m_2(ws, X)). \tag{31}$$

This case is non conformant according to [4], as in some cases ws might utter the forbidden message m_3. g-$\mathcal{S}$CIFF abduces two possible histories ($\{\mathbf{H}(m_1), \mathbf{H}(m_2)\}$ and $\{\mathbf{H}(m_1), \mathbf{H}(m_3)\}$). The first is compliant, according to $\mathcal{S}$CIFF, while in the second the choreography's expectation $\mathbf{E}_{chor}(m_2)$ remains pendent.

Predefined answer The dual of the previous example is when the choreography lets the web service choose to reply m_2 or m_3 to a question m_1 (Eq 32), while the web service sticks to the reply m_2 (Eq 33). Again, A^{l}LoWS provides a correct proof: g-$\mathcal{S}$CIFF gives one possible history $\{\mathbf{H}(m_1), \mathbf{H}(m_2\}$, which is reported (feeble and strong) conformant by $\mathcal{S}$CIFF in the second phase.

$$\mathbf{H}(m_1(X, ws)) \rightarrow \mathbf{E}_{chor}(m_2(ws, X)) \vee \mathbf{E}_{chor}(m_3(ws, X)) \tag{32}$$

$$\mathbf{H}(m_1(X, ws)) \rightarrow \mathbf{E}_{ws}(m_2(ws, X)). \tag{33}$$

Thus, in all the examples by Baldoni et al., A^{l}LoWS provides the same answer proposed in [4]. We now propose other examples that highlight the enhanced expressive power provided by computational logics, in particular the use of constraints, that are embedded in the $\mathcal{S}$CIFF and g-$\mathcal{S}$CIFF proof procedures.

Mutual exclusion Many protocols include mutual exclusion between choices: a choreography might prescribe that if a given condition on a message m_1 holds, a message m_2 should be exchanged, otherwise another message m_3 should be sent. In A^{l}LoWS, conditions can be expressed by means of constraints (either the ones predefined in the underlying solver, i.e., CLP(FD), or user-defined) or by means of defined predicates. As a simple example, suppose the choreography prescribes to reply either m_2 or m_3, depending on the content of the previous message m_1 (Eq. 34) while the web service always replies m_2 (Eq. 35):

$$\begin{aligned}\mathbf{H}(m_1(X, ws, C)) \rightarrow\ & \mathbf{E}_{chor}(m_2(ws, X, C_2)), C > 0 \\ \vee\ & \mathbf{E}_{chor}(m_3(ws, X, C_3)), C \leq 0\end{aligned} \tag{34}$$

$$\mathbf{H}(m_1(X, ws, C)) \rightarrow \mathbf{E}_{ws}(m_2(ws, X, C_2)) \tag{35}$$

g-$\mathcal{S}$CIFF generates two possible histories, with variables and constraints upon the variables (Fig 5). In both the messages m_1 and m_2 are generated, but while in the first the proof procedure assumes that C takes a value greater than 0, in the second C is non positive. In the second phase, $\mathcal{S}$CIFF takes as input both the happened events and the constraint store, and accepts as conformant the first history, while discarding as non-conformant the second.

Notice that constraints scope is not restricted only to variables in the content, but might involve all the variables in the message, including *time*.

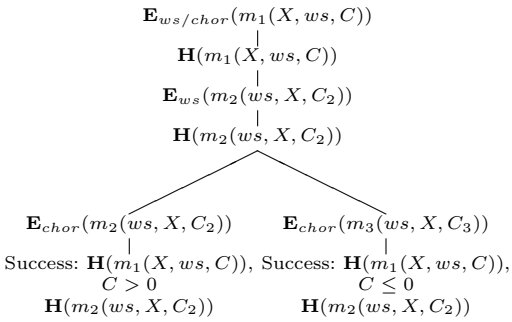


Fig. 5. g-$\mathcal{S}$CIFF derivation for Example of Mutual Exclusion.

Deadlines Suppose that the choreography specifies a deadline for the receipt of a given message m_2 (Eq. 36), and that the web service ws replies within a deadline that might be different (Eq. 37):

$$\mathbf{H}(m_1(X, ws, C_1), T_1) \rightarrow \mathbf{E}_{chor}(m_2(ws, X, C_2), T_2) \wedge T_2 < T_1 + \delta_{chor} \quad (36)$$

$$\mathbf{H}(m_1(X, ws, C_1), T_1) \rightarrow \mathbf{E}_{ws}(m_2(ws, X, C_2), T_2) \wedge T_2 < T_1 + \delta_{ws}. \quad (37)$$

In this case, the only possible history is

$$\mathbf{HAP}^* = \{\, \mathbf{H}(m_1(X, ws, C_1), T_1), \mathbf{H}(m_2(ws, X, C_2), T_2), T_2 < T_1 + \delta_{ws}\}$$

Applying $\mathcal{S}$CIFF to the history $\mathbf{HAP}^*$, generates the choreography's expectation $\mathbf{E}_{chor}(m_2(ws, X, C_2), T_2) \wedge T_2 < T_1 + \delta_{chor}$; this expectation matches with the second item of $\mathbf{HAP}^*$ if a further condition holds: $T_2 < T_1 + \delta_{chor}$. Coherently with the philosophy of CLP, $\mathcal{S}$CIFF provides this constraint in output, as a conditional answer: the web service is conformant provided that the answer arrives before the deadline in the choreography specification.

4 A test conformance example

Consider the choreography specification in Fig. 1. The interaction is initiated by a $User$ that asks the Flight Service FS to book a flight. If there are seats available on the plane, FS will reply with *flightOffer*, specifying the $Price$. Otherwise, FS replies with *notAvailable*. The *offer* can be accepted (*ackOffer*) or refused (*nAckOffer*) by the $User$. If the *offer* is accepted, FS will book the

seat. After booking, the $User$ can still $Cancel$ the reservation. Otherwise, it will issue a payment order ($payment$) to the $Bank$, that will send the notification ($notifyPayment$) to the creditor, FS. When FS has received both the booking order ($ackOffer$) and the payment ($notifyPayment$), it will normally issue the $flightTicket$ to the $User$; however, FS retains the right to refuse the ticket and send a $flightCancelled$ message in case of problems (e.g., overbooking).

Fig. 2 shows the behavioural interface of a Flight Server web service; the specification in terms of ICs is in Spec. 2.2. The FS establishes that the late payment is an error condition, and will cancel the booking if the payment notification does not arrive within δ time units after the booking.

In next section, we show how the conformance of *fs* is proven in A^{l}LoWS.

4.1 Conformance of the Flight Service

The test of conformance of the Flight Service fs is performed by generating, through g-$\mathcal{S}$CIFF, the set of the possible histories. The g-$\mathcal{S}$CIFF derivation provides five possible histories:

$$\begin{aligned}
\mathbf{HAP^*}_1 = \{&\mathbf{H}(request(U, fs, F), T_r), \mathbf{H}(offer(fs, C, F, P), T_o), \mathbf{H}(ackOffer(C, fs, F, P), T_a),\\
&\mathbf{H}(payment(C, B, P, fs), T_p), \mathbf{H}(notifyPayment(B, fs, P)), T_n) \wedge T_n > T_a + \delta\\
&\mathbf{H}(flightCancelled(fs, C, F), T_c)\},\\
\mathbf{HAP^*}_2 = \{&\mathbf{H}(request(U, fs, F), T_r), \mathbf{H}(offer(fs, C, F, P), T_o), \mathbf{H}(ackOffer(C, fs, F, P), T_a),\\
&\mathbf{H}(payment(C, B, P, fs), T_p), \mathbf{H}(notifyPayment(B, fs, P)), T_n) \wedge T_n \leq T_a + \delta\\
&\mathbf{H}(flightTicket(fs, C, F), T_t)\},\\
\mathbf{HAP^*}_3 = \{&\mathbf{H}(request(U, fs, F), T_r), \mathbf{H}(offer(fs, C, F, P), T_o), \mathbf{H}(ackOffer(C, fs, F, P), T_a),\\
&\mathbf{H}(cancel(C, fs, F), T_c), \mathbf{H}(flightCancelled(fs, C, F))\},\\
\mathbf{HAP^*}_4 = \{&\mathbf{H}(request(U, fs, F), T_r), \mathbf{H}(offer(fs, C, F, P), T_o), \mathbf{H}(nAckOffer(C, fs, F, P), T_a)\},\\
\mathbf{HAP^*}_5 = \{&\mathbf{H}(request(U, fs, F), T_r), \mathbf{H}(notAvailable(fs, C, F, P), T_n)\}.
\end{aligned}$$

Two of the histories include time constraints. All the possible histories are trivially conformant: they satisfy both the expectations of the choreography, and those of the web service fs. Thus, fs is feeble conformant. Moreover, all the generated events are expected, and this shows that fs is also strong conformant.

5 Related Work

A number of languages for specifying service choreographies and testing "a priori" and/or "run-time" conformance have been proposed in the literature. Two examples of these languages are state machines [6] and Petri nets [8].

In [6], the authors focus on two-party choreographies involving a requester and a provider (named service conversations) and formulate some requirements for a modelling language suitable for them. The requirements include genericity, automated support, and relevance. The authors argue that state machines satisfy these requirements and sketch an architecture of a service conversation controller capable of monitoring messages exchanged between a requester and provider in order to determine whether they conform to a conversation.

An example of use of Petri nets for the formalization of choreographies is in [8]. Four different viewpoints (interface behaviour, provider behaviour, choreography, and orchestration) and relations between viewpoints are identified and

formalised. These relations are used to perform (global) consistency checking of multi-viewpoint service designs thereby providing a formal foundation for incremental and collaborative approaches to service-oriented design. Our proposal is limited to a deep analysis of the relation between choreographies and behaviour interfaces but deal with both "a priori" and "run-time" conformance.

Our work is highly inspired by Baldoni et al. [4]. We adopt, like them, a MAS viewpoint, in defining a priori conformance in order to guarantee interoperability. As in [4], we interpret a-priori conformance as a property that relates two formal specifications: the global one determining the conversations allowed by the choreography and the local one related to the single web service. But, while they represent choreographies as finite state automata, we claim that the formalisms and technologies developed in the area of Computational Logic in providing a declarative representation of the social knowledge, could provide a higher expressive power. This paper can be considered as a first step in this direction. We also manage concurrency, which they do not consider at the moment.

Endriss et al. [9] apply a formalism based on computational logic to the a-priori conformance in the MAS field. They restrict their analysis to the so-called *shallow protocols*. They address only 2-party interactions, without the possibility of expressing conditions over the content of the exchanged messages, and without considering concurrency. While [9] and our work agree on the notion of strong/exhaustive conformance, we have dual notions of feeble/weak conformance: in [9] weak conformant is an agent that does not perform forbidden actions, but might fail to perform required actions. Dually, in this work, we call feeble conformant an agent that executes all the required actions, but might also perform forbidden actions.

Abduction has been applied to verification in other work; in particular, Russo et al. [13] use an abductive proof procedure for analysing event-based requirements specifications. They use abduction for analysing the correctness of specifications, while A^{l}LoWS is focussed on conformance checking of web services.

In [12], the authors tackle the problem of verifying (general and specific) properties of a service obtained from the composition of many web services. Each web service specification (written in BPEL4WS) is translated into a *labelled state transition system* (labelled STS); then, by applying a composition operator, they get the STS representing the composed service. Finally, model checking techniques are applied to this latter model, to the end of verifying the properties. We share the same intuition that *"the situation where some messages can be emitted without being ever consumed should not occur in valid composition"*. Our approach consists in representing both the web service specification and the choreography with the same logic-based formalism, and then interoperability is verified, before actual composition.

6 Conclusions and Future Work

This paper represents a first step in checking the conformance of web services to choreographies in computational logics. We proposed a framework, called A^{l}LoWS, for defining web services and choreographies specifications, and checking their conformance. We showed various examples of the expressivity of com-

putational logics, including reasoning on constraints, deadlines, and other structures that are not currently easily addressed by classical tools.

We are aware that many issues should be addressed in the future. For example, A^ILoWS generates the possible histories, though in intensional version, so it cannot currently handle histories of unbounded length, such as those possible in choreographies containing *cycles*. In such a case, the resulting program could be non acyclic, so the proof of termination might not hold. Those choreographies might be tested using an iterative deepening search strategy.

We are currently developing a graphical language to define choreographies, and a compiler to generate integrity constraints from the graphical specification.

Acknowledgments
This work has been partially supported by the MIUR PRIN 2005 projects *Specification and verification of agent interaction protocols*, and *Vincoli e preferenze come formalismo unificante per l'analisi di sistemi informatici e la soluzione di problemi reali*.

References

1. M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Security protocols verification in Abductive Logic Programming: a case study. In A. Pettorossi, M. Proietti, and V. Senni, editors, *CILC 2005*, June 21-22 2005.
2. 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, *CEEMAS 2003*, volume 2691 of *LNAI*, pages 204–213.
3. M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. The $\mathcal{S}$CIFF abductive proof-procedure. In *AI*IA 2005*, volume 3673 of *LNAI*, pages 135–147.
4. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In Bravetti et al. [7].
5. A. Barros, M. Dumas, and P. Oaks. A critical overview of the web services choreography description language (WS-CDL). *BPTrends*, 2005.
6. B. Benattallah, F. Casati, F. Toumani, and R. Hamadi. Conceptual modeling of web service conversations. 2681:449–467, 2003.
7. M. Bravetti, L. Kloul, and G. Zavattaro, editors. volume 3670 of *Lecture Notes in Computer Science*. Springer, 2005.
8. R. Dijkman and M. Dumas. Service-oriented design: A multi-viewpoint approach. *International Journal of Cooperative Information Systems*, 13(4):337–378, 2004.
9. U. Endriss, N. Maudet, F. Sadri, and F. Toni. Logic-based agent communication protocols. In F. Dignum, editor, *Advances in Agent Communication*, volume 2922 of *LNAI*, pages 91–107. Springer-Verlag, 2004.
10. J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
11. A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.
12. R. Kazhamiakin and M. Pistore. A parametric communication model for the verification of BPEL4WS compositions. In Bravetti et al. [7], pages 318–332.
13. A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for analysing event-based requirements specifications. In P. Stuckey, editor, *ICLP 2002*, volume 2401 of *LNCS*, pages 22–37. Springer-Verlag, 2002.
14. Societies Of ComputeeS (SOCS). http://lia.deis.unibo.it/Research/SOCS/.

Improving scalability in ILP incremental systems

Marenglen Biba, Teresa Maria Altomare Basile, Stefano Ferilli, Floriana Esposito

Dipartimento di Informatica – Università degli Studi di Bari
via Orabona 4 - 70126 Bari
{biba, basile, ferilli, esposito}@di.uniba.it

Abstract. This paper presents an approach for extending an existing ILP system to deal with the scalability issue of inductive learning systems. Efficient data access in ILP has been tackled by previous approaches through the use of SQL, subsampling and local coverage test. The work presented here provides another method for efficient access of terms that allows also to generate and preserve precious information about the learning process that could be used during the learning task to guide the search for hypotheses. Although the learning algorithms in the ILP system do not change the way they work, coverage test of examples and theory refinement are employed using a hash based access to terms which are no longer loaded in main memory. Experiments conducted with the improved system indicate that it performs better for large datasets.

1. Introduction

Scalability is gaining increasing interest in ILP research in order to make it the main stream for multirelational data mining [1] [2]. In these domains, huge datasets must be explored in order to discover relationships among data, leading therefore to critical efficiency matters to be dealt with.

One of the approaches to the scalability problem could be using commercial database management systems (DBMS) that translate a goal query to the corresponding SQL query. These methods might not lead to scalable ILP systems because the internal efficiency of DBMSs should be taken into account. Recent attempts aim at ILP specific solutions. For instance, sub-sampling methods that focus on smaller sets of examples have produced good results [3]. Another ILP-tailored solution is local coverage test within a learning from interpretations setting where independence assumptions among examples are reasonable and coverage of each example can be tested without considering the entire background knowledge [4]. Furthermore, ad-hoc theta-subsumption techniques to speed-up the coverage test have been investigated in [5], [6].

In this paper, we present a powerful method for fast access to large datasets without changing the learning algorithms, applied to the ILP system INTHELEX (INcremental THEory Learner from EXamples) [7]. Such an approach stores the examples and the theory in an external database, that is then accessed for coverage tests of examples and for clause generation without loading any example into main memory.

All the terms in the database are indexed, this way enhancing the scalability and efficiency of the learning system.

To empirically demonstrate the efficiency gain, we extended the ILP system INTHELEX in order to fetch terms no longer from main memory but by calling predicates that interact with the external storage of terms. Such storage is a hash-based database based on the Berkeley-DB [8] data engine that provides fast access to data. While the two versions of INTHELEX produce almost the same set of rules, runtimes are quite different showing that the new version is more efficient on large datasets. For a solid comparison of the results, different datasets were used for the experiments.

The paper is organized as follows. The next section illustrates the learning system INTHELEX which loads data in main memory. Section 3 presents the scalable version of INTHELEX. Then, Section 4 reports the experimental results and Section 5 contains conclusions.

2 INTHELEX: an internal memory ILP system

Almost all ILP systems are based on an internal memory usage to process examples and clauses. INTHELEX is an ILP system for the induction of hierarchical theories from positive and negative examples [9], whose architecture is depicted in Figure 1. It is fully and inherently incremental: this means that, in addition to the possibility of taking as input a previously generated version of the theory to be refined, learning can also start from an empty theory and from the first available example; moreover, at any moment the theory is guaranteed to be correct with respect to all of the examples encountered so far. This is a fundamental issue, since in many cases deep knowledge about the world is not available. It adopts a *close loop* learning process, where feedback on performance is used to activate the theory revision phase. INTHELEX can learn simultaneously various concepts, possibly related to each other. The correctness of the learned theory is checked on any new example and, in case of failure, a revision process is activated on it, in order to restore completeness and consistency. It adopts a full memory storage strategy − i.e., it retains all the available examples, thus the learned theories are guaranteed to be valid on the whole set of known examples. It incorporates two inductive operators, one for generalizing definitions that reject positive examples, and the other for specializing definitions that explain negative examples. Both these operators, when applied, change the answer set of the theory, i.e. the set of examples it accounts for.

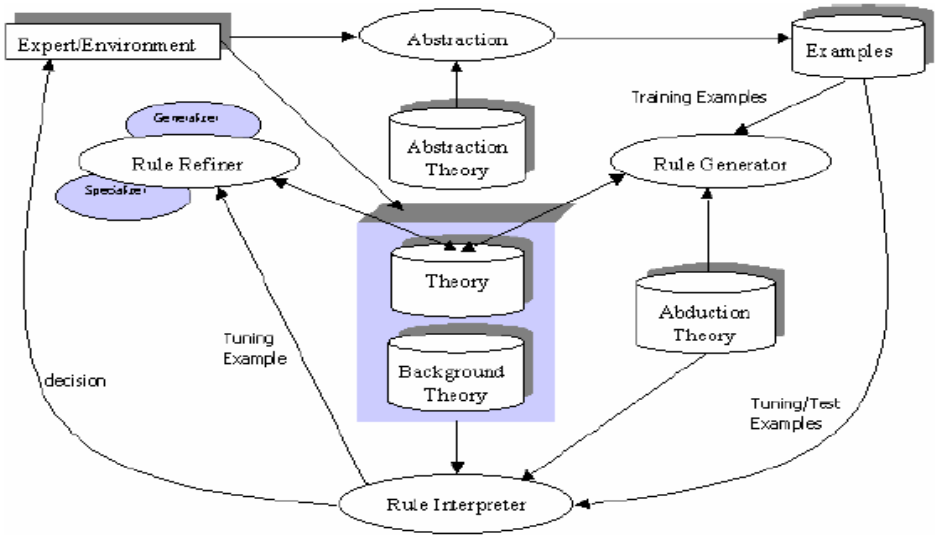


Figure 1. Architecture of INTHELEX

Algorithm 1

```
Procedure Tuning (E: example; T: theory; M: historical memory);
       If E is a positive example AND ¬ covers(T,E) then
              Generalize(T,E,M⁻)
       else
              if E is a negative example AND covers(T,E) then
                     Specialize(T,E,M⁺)
              end if
       end if
M := M U E.
```

Algorithm 1 shows the procedure implementing the Rule Refiner. It concerns the tuning phase of the system, where $M = M^+ \cup M^-$ represents the set of all negative (M^-) and positive (M^+) processed examples, E is the example currently examined, T represents the theory generated so far according to M plus the background knowledge BK. When a positive example is not covered by the theory, the procedure Generalize is called in order to restore the completeness of the theory.

Algorithm 2.

```
Procedure Generalize (E: positive example, T: theory, M⁻: negative
examples);
L := (non BK)clauses in the definition of the concept which
      E refers to
generalized := false; gen_lgg := 0
while ((¬ generalized) AND (L ≠ ∅)) do
       C := choose_clause(L) ; L := L\{C}
       while ((∃ another l ∈ lgg_OI(C,E)) AND
              (¬generalized) AND (gen_lgg ≤ max_gen)) do
              gen_lgg + 1
```

```
            if (consistent({T\{C}} U {l},M⁻)) then
                  generalized := true ; T := {T\{C}} U {l}
            end if
      end while
end while
if (¬generalized) then
      G := turn_constants_into_variables(E)
      if consistent(T U {G},M⁻) then
            T := T U {G}
      else
            T := T U {E} {positive exception}
      end if
end if
```

Algorithm 2 shows the procedure *Generalize*. When searching for a *least general generalization* (lgg), the algorithm must check whether the generated *lgg*s are consistent with the negative examples in the main memory. To do this, it must scan the entire set of previous examples, that have been loaded in main memory, in order to ensure that no negative example is covered by the generated *lgg*. This makes the generalization process heavily dependent on the number of examples. Since the system loads examples and theory into main memory, picking up examples and coverage test are simply obtained by calling the corresponding goals where goal proving is done automatically within a Prolog runtime system. Therefore it is not possible to access only the examples related to a certain clause, and every time a clause is being generalized, all the examples in the main memory must be considered instead of checking only those referred to the given clause. For instance, when a clause that covers some positive examples is generalized, it will obviously still cover those examples after generalization, so completeness of the theory is trivial and it becomes useless checking all positive examples in the main memory. However, the generalized clause could cover also other previous examples of the same concept, in addition to the new one, and remembering such examples could help to ensure that, after future refinements, each positive example has at least one clause covering it. As for negative examples checking, only those belonging to the same concept as the generalized clause should be tested and this could be obtained through a relational schema that links together clauses and examples that have in common the target concept.

The same problems arise when the system tries to specialize a clause that covers a negative example (Algorithm 3 shows the procedure). The consistency check scans all the available positive examples in main memory in order to check whether the specialized clause still covers the examples that were covered before specialization. In order to speed up such a verification, the coverage test could take advantage from a data organization allowing to link clauses to the examples they cover, so that once a clause is specialized, only the examples that were covered by the clause before being specialized need to be checked for consistency.

Algorithm 3

```
Procedure Specialize(E: negative example; T: theory; M⁺:positive
examples);
specialized := false ; gen_spec := 0
while ∃ a (new) derivation D of E from T do
      L := Input program clauses in D sorted by decreasing
           depth in derivation
```

```
        while ((¬ specialized) AND (∃C ∈ L) and
               (gen_spec ≤ maxgen)) do
               while ((∃ another S ∈ rho_OI_pos(C,E)) AND
                      (¬ specialized)) do
                      gen_spec + 1
                      if (complete({T\{C}} U {S},M+)) then
                             T := {T\{C}} U {S}
                      end if
               end while
        end while
        if (¬ specialized) then
               C := first clause in the derivation of E
               while ((∃ another S ∈ rho_OI_neg(C,E)) AND
                     (¬ specialized)) do
                      if (complete({T\{C}} U {S},M+)) then
                             T := {T\{C}} U {S}
                      end if
               end while
        end if
        if (¬ specialized) then
               T := T U {E} {negative exception}
        end if
end while
```

Inspired by these considerations, a possible integration of an external terms storage was taken into account in order to increase the system performance. Additionally, developing a more structured framework for handling data, it becomes possible to maintain and exploit during the learning process precious information about the examples and the clauses.

3. Scalable version of INTHELEX

This section presents the extension of INTHELEX with an external storage of terms. Instead of choosing a general DBMS for the external handling of terms, a tailored embedded solution was developed using the Sicstus Prolog interface [10] to the Berkeley DB toolset for supporting persistent storage of Prolog terms. The idea is to obtain a behavior similar to the built-in Prolog predicates such as assert/1, retract/1 and clause/2, but having the terms stored on files instead of main memory.

Some differences with respect to the Prolog database are:

- The functors and the indexing specifications of the terms to be stored have to be given when the database is created.
- The indexing is specified when the database is created. It is possible to index on other parts of the term than just the functor and first argument.
- Changes affect the database immediately.
- The database will store variables with blocked goals as ordinary variables.

Some commercial databases can't store non-ground terms or more than one instance of a term. This interface can however store terms of either kind.

3.1 Interface to Berkeley DB

The interface we exploit in this work uses the Concurrent Access Methods product of Berkeley DB [10]. This means that multiple processes can open the same database, but transactions and disaster recovery are not supported. The environment and the database files are ordinary Berkeley DB entities that use a custom hash function.

The db-specification (*db-spec*) defines which functors are allowed and which parts of a term are used for indexing in a database. The *db-spec* is a list of atoms and compound terms where the arguments are either + or -. A term can be inserted in the database if there is a specification (*spec*) in the *db-spec* with the same functor. Multilevel indexing is not supported, terms have to be "flattened". Every *spec* with the functor of the indexed term specifies an indexing. Every argument where there is a + in the *spec* is indexed on.

A *db-spec* has the form of a specification-list (*speclist*):

speclist = [spec1, . . . , specM]
spec = functor(argspec1, . . . , argspecN)
argspec = + | -

where functor is a Prolog atom. The case N = 0 is allowed. A *spec* F(argspec1, . . . , argspecN) is applicable to any ground term with principal functor F/N.

When storing a term T, it is generated a hash code for every applicable *spec* in the *db-spec*,and a reference to T is stored with each of them. (More precisely with each element of the set of generated hash codes). If T contains ground elements on each + position in the *spec*, then the hash code depends on each of these elements. If T contains some variables on + position, then the hash code depends only on the functor of T. When fetching a term Q we look for an applicable *spec* for which there are no variables in Q on positions marked +. If no applicable *spec* can be found a domain error is raised. If no *spec* can be found where on each + position a ground term occurs in Q an instantiation error is raised. Otherwise, we choose the *spec* with the most + positions in it breaking ties by choosing the leftmost one. The terms that contain ground terms on every + position will be looked up using indexing based on the principal functor of the term and the principal functor of terms on + positions. The other (more general) terms will be looked up using an indexing based on the principal functor of the term only.

3.2 Designing storage of terms

In order to relate clauses to examples, a simple relational schema was designed. It makes possible to link clauses to examples that are covered by these clauses. Whenever one of these clauses is refined, the relational schema permits a fast access only to those examples that are related to the clause. We chose to include examples in a database and clauses in another. The following schema in Figure 2 shows how the logical model of the tables was designed. We maintained two separate tables for examples and descriptions in order to have the possibility to preserve useful information about descriptions that can be used either afterwards to assess and study the learning process or during the learning task for accelerating the access to data. We also designed a table with only two fields that serves as a hash table to fetch all the examples covered by a clause.

In fact, the two fields represent respectively the key of the clause and the key of the example. Another table was introduced in order to maintain all the versions of the theory during the refinement steps. This could be very useful for the learning task and the previous versions of the theory could be exploited for improving the searching on the space of the hypothesis. Regarding the information about the examples, we keep useful statistics about the examples and the observations. For example, the attribute *type* permits to distinguish between the examples that have modified the theory and those that have not. This could be useful in a partial memory context, in which examples could be considered based on their history of modifications of the theory, thus during the learning task only the examples that have modified the theory could be considered. For this purpose, we also planned to group the examples in order to distinguish between those that have been given in input to the system at a time and those have been given at another moment. This might be useful to study the learning task from an *example grouping* point of view, trying to analyze and distinguish those examples that have a greater impact on the learning task. As regards the clauses, we keep other information such as the generating example of the clause. This represents another precious fact that could be useful in understanding better the learning process.

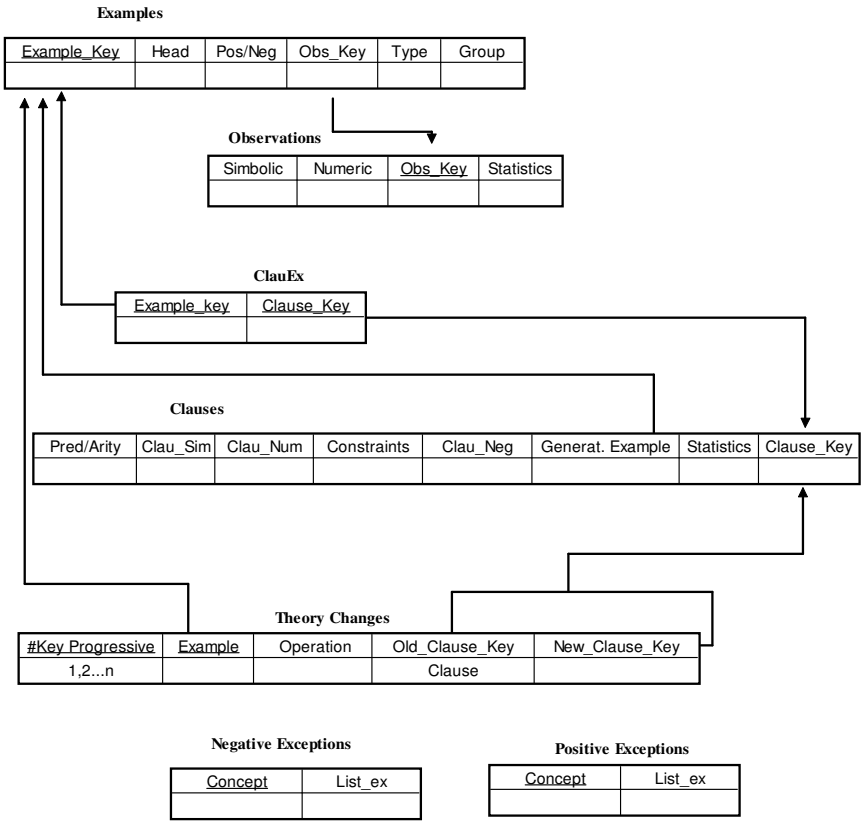


Figure 2. Logical model of the database of terms

Another similar information kept in the table *theory_changes* is the key of the example that have caused the refinement of a certain clause. This information could be exploited by another improved version of INTHELEX, named INTHELEX_back, [11], [12], able to handle the problem of ordering effects [13] in incremental learning.

The *db-spec* that resulted form the logical model was very simple. The following specifications show how the tables were implemented through the *db-spec* of Berkeley DB. As it can be seen, the attributes which are marked with "+, represent the keys of the clauses and examples, thus the database is indexed on these attributes.

[examples(+,-,-,-,-,-), obs(+,-,-,-), max(+,-,-,-)]

[clauses(+,-,-,-,-,-,-,-,-), theory_changes(+,-,-,-,-), clauEx(+,-), pos_excep(+,-,-), neg_excep(+,-,-), max(+,-,-,-,-,-)]

We have used the indexed attributes for our purpose of implementing a relational schema between clauses and examples but however the database engine of Berkeley DB uses these attributes for efficient fetching of the terms. This is due to the schema adopted by this database engine whose fetching mechanism is based on the indexed terms.

4. Experimenting scalability

The framework proposed was tested on the domain of document processing. The exploited datasets belong to two learning tasks: document understanding and document classification. Both these tasks are quite hard to deal with, in terms of computational effort and we chose them because of the very high times of elaboration of the first version of INTHELEX in order to process the datasets involved. The datasets contain descriptions of documents formatted according to different conference and formatting styles, specifically `ISMIS` (the International Symposium on Methodologies for Intelligent Systems), `TPAMI` (Transactions on Pattern Analysis and Machine Intelligence) and `ICML` (the International Conference on Machine Learning) [7]. The task for the learning system is to learn theories able to classify the new incoming documents and to recognize the significant logical components for each class of document. In particular, the interesting logical components identified by the expert in the classes were: *title*, *abstract*, *author* and *page number* for the documents that belong to ICML; *title*, *abstract*, *author* and *affiliation* for ISMIS documents and *title*, *abstract*, *affiliation*, *index terms*, *page number* and *running head* for TPAMI. All the experiments were performed on a Pentium 4, 2,4 GHz, using Sicstus Prolog 3.11.1. The first experiments were conducted between the two versions of INTHELEX with database. In the first version nothing is changed but the simple fetching of terms. Whereas in the second version we consider during the refinement steps only examples covered by a clause during the clause refinement. Table 1 reports the results of these experiments. As it can be seen, the second version of INTHELEX that does not simply use the database as a fast-fetching mechanism but makes use also of the relational schema between clauses and examples has better results than the first one. We have reported here the mean time in seconds that resulted from the experiments on 33 folds.

Finally, experiments were performed between the initial version of INTHELEX that uses main memory for loading examples and clauses and is based on text files and the second version of INTHELEX with database. As it is possible to note in Table 2 there is a far better performance for the scalable version of INTHELEX. This is due to not only the relational schema that makes possible considering only the examples covered by a clause but can be attributed also to the fast access that the database engine guarantees comparing it with the main memory approach of Prolog. As regards the dataset dimension, the results showed that for large datasets the gain in elaboration time is considerable. For small datasets the scalable version has a slightly better performance. This enforces the idea that the scalable version is oriented to large datasets where the elaboration times are very high. As for the theories' accuracy in both experiments the accuracy of the theories was similar.

Table 1. Comparison of the two versions of INTHELEX with database

Class of documents	Initial version Inthelex-Berkeley Runtime (secs)	Second version Inthelex-Berkeley Runtime (secs)
Classification		
ICML	3.61	3.15
ISMIS	25.23	19.45
TPAMI	37.26	27.55
Understanding		
ICML logical components		
Abstract	167.86	111.13
Author	31.35	29.07
Page Number	77.65	76.22
Title	55.73	51.66
ISMIS logical components		
Abstract	173.99	133.70
Affiliation	65.80	50.94
Author	64.83	47.88
Title	54.11	33.46
TPAMI logical components		
Absctract	1276.09	1089.88
Affiliation	476.12	387.76
Page Number	1652.31	1265.47
Running Head	1161.31	1148.17
Title	1272.71	1234.43

5. Conclusions and future work

This paper described an approach for implementing scalable ILP systems. It is based on the idea that loading datasets in main memory is an operation which causes the ILP systems to have low performance on large datasets. For this reason it was

implemented a scalable version of the ILP system INTHELEX in order to work without loading data in main memory. There has been some work in this direction such as that in [3] which proposed sub-sampling as a possible approach for implementing scalability. The work presented here is based on the use of high performance database engine for storing terms. The results were very promising since for large datasets it was evident that the scalable version has better results in terms of elaboration time. Our aim was that of showing how the scalability extension of ILP systems is possible through the integration of valid engines for terms retrieval.

As future work we intend to use the information we stored during the learning task in order to guide the search for the hypotheses. This will render the system not only scalable for dealing with large datasets but will also provide a valid mechanism for improving the learning process through the use of information such as the different versions of theory and the modifying examples for each couple of clauses, the old and the refined one.

Table 2. Comparison between the original version of INTHELEX and the scalable one

Class of documents	INTHELEX Text Files Mean time (secs.) on 33 folds	INTHELEX BERKELEY DB Mean time (secs.) on 33 folds	Coefficient of time gain
Classification			
ICML	15.67 sec	3.15 sec	4,96
ISMIS	67.53 sec	19.45 sec	3,47
TPAMI	64.20 sec	27.55 sec	2.32
Understanding			
ICML logical components			
Abstract	131.54 sec	111.13 sec	1,18
Author	130.41 sec	29.07 sec	4,48
Page Number	270.23 sec	76.22 sec	3,54
Title	249.04 sec	51.66 sec	4,82
ISMIS logical components			
Abstract	178.75 sec	133.70 sec	1,33
Affiliation	129.50 sec	50.94 sec	2,54
Author	159.35 sec	47.88 sec	3,32
Title	65.52 sec	33.46 sec	1,95
TPAMI logical components			
Abstract	1687.72 sec	1089.88 sec	1,54
Affiliation	1155.10 sec	387.76 sec	2,97
Page Number	2669.03 sec	1265.47 sec	2,10
Running Head	1798.31 sec	1148.17 sec	1,56
Title	2027.57 sec	1234.43 sec	1,64

References

[1] A. Knobbe, H. Blockeel, A. Siebes and D. Van der Wallen, Multi-Relational Data Mining, *Technical Report INS-R9908*, Maastricht University, 9, 1999.

[2] H. Blockeel and M. Sebag, Scalability and efficiency in multi-relational data mining. *SIGKDD Explor. Newsl*, Vol 5, pp. 17-30, 2003.

[3] A. Srinivasan, A study of Two Sampling Methods for Analyzing Large Datasets with ILP, *Data Mining and Knowledge Discovery,* Vol. 3, pp. 95-123, 1999.

[4] H. Blockeel, L. De Raedt, N. Jacobs and B. Demoen. Scaling up Inductive Logic Programming by Learning from Interpretations. *Data Mining and Knowledge Discovery,* Vol. 3, No. 1, pp. 59-64, 1999.

[5] N. Di Mauro, T.M.A. Basile, S. Ferilli, F. Esposito and N. Fanizzi. An Exhaustive Matching Procedure for the Improvement of Learning Efficiency. In T. Horváth and A. Yamamoto, Inductive Logic Programming: 13th International Conference (ILP03), 2003, Vol. 2835, LNCS, pp. 112-129. Springer Verlag.

[6] J. Maloberti and M. Sebag. Fast Theta-Subsumption with Constraint Satisfaction Algorithms. Machine Learning, Vol 55, N. 2, pp. 137-174, Kluwer Accademic Publishers, 2004.

[7] F. Esposito, S. Ferilli, N. Fanizzi, T. M.A. Basile and N. Di Mauro, Incremental Multistrategy Learning for Document Processing, *Applied Artificial Intelligence: An International Journal*, 2003, Vol. 17, n. 8/9, pp. 859-883, Taylor Francis.

[8] Berkeley DB reference: http://www.sleepycat.com

[9] F. Esposito, S. Ferilli, N. Fanizzi, T.M.A Basile and N. Di Mauro. Incremental Learning and Concept Drift in INTHELEX. *Intelligent Data Analysis Journal, ISSN 1088-467X, Special Issue on Incremental Learning Systems Capable of Dealing with Concept Drift,* 8(3):213-237, IOS Press, Amsterdam, 2004.

[10] Sisctus Prolog reference www.sics.se/sicstus

[11] N. Di Mauro, F. Esposito, S. Ferilli and T.M.A. Basile. A Backtracking Strategy for Order-Independent Incremental Learning. In R. Lopez de Mantaras and L. Saitta (Eds), *Proceedings 16th European Conference on Artificial Intelligence* (ECAI 2004), 2004, pp. 460-464, IOS Press.

[12] N. Di Mauro, F. Esposito, S. Ferilli and T. M.A. Basile. Avoiding Order Effects in Incremental Learning. In S. Bandini and S. Manzoni (Eds.), *Advances in Artificial Intelligence* (AI*IA05), 2005 LNCS, Vol. 3673, pp. 110-121, Springer-Verlag

[13] P. Langley. Order effects in incremental learning. In: P. Reimann, H. Spada (Eds.), *Learning in Humans and Machines: Towards an Interdisciplinary Learning Science*, Elsevier, Amsterdam, 1995.

Proving Properties of Constraint Logic Programs by Eliminating Existential Variables

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@disp.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 proving first order properties of constraint logic programs which manipulate finite lists of real numbers. Constraints are linear equations and inequations over reals. Our method consists in converting any given first order formula into a stratified constraint logic program and then applying a suitable unfold/fold transformation strategy that preserves the perfect model. Our strategy is based on the elimination of existential variables, that is, variables which occur in the body of a clause and not in its head. Since, in general, the first order properties of the class of programs we consider are undecidable, our strategy is necessarily incomplete. However, experiments show that it is powerful enough to prove several non-trivial program properties.

1 Introduction

It has been long recognized that program transformation can be used as a means of proving program properties. In particular, it has been shown that unfold/fold transformations introduced in [4,20] can be used to prove several kinds of program properties, such as equivalences of functions defined by recursive equation programs [5,9], equivalences of predicates defined by logic programs [14], first order properties of predicates defined by stratified logic programs [15], and temporal properties of concurrent systems [7,19]. In this paper we consider stratified logic programs *with constraints* and we propose a method based on unfold/fold transformations to prove first order properties of these programs.

The main reason that motivates our method is that transformation techniques may serve as a way of eliminating *existential variables* (that is, variables which occur in the body of a clause and not in its head) and the consequent quantifier elimination can be exploited to prove first order formulas. Quantifier elimination is a well established technique for theorem proving in first order logic [18] and one of its applications is Tarski's decision procedure for the theory of the field of reals. However, no quantifier elimination method has been developed so far to prove formulas within theories defined by constraint logic programs, where the constraints are themselves formulas of the theory of reals.

Consider, for instance, the following constraint logic program which defines the membership relation for finite lists of reals:

$Member$: $member(X, [Y|L]) \leftarrow X = Y$
$member(X, [Y|L]) \leftarrow member(X, L)$

Suppose we want to show that every finite list of reals has an upper bound, i.e.,

φ: $\forall L\, \exists U\, \forall X\, (member(X, L) \rightarrow X \leq U)$

Tarski's quantifier elimination method cannot help in this case, because the membership relation is not defined in the language of the theory of reals. The transformational technique we propose in this paper, proves the formula φ in two steps. In the first step we transform φ into clause form by applying a variant of the Lloyd-Topor transformation [11], thereby deriving the following clauses:

$Prop_1$: 1. $prop \leftarrow \neg p$
2. $p \leftarrow list(L) \wedge \neg q(L)$
3. $q(L) \leftarrow list(L) \wedge \neg r(L, U)$
4. $r(L, U) \leftarrow X > U \wedge list(L) \wedge member(X, L)$

where $list(L)$ holds iff L is a finite list of reals. The predicate $prop$ is equivalent to φ in the sense that $M(Member) \models \varphi$ iff $M(Member \cup Prop_1) \models prop$, where $M(P)$ denotes the perfect model of a stratified constraint logic program P. In the second step, we eliminate the existential variables by extending to constraint logic programs the techniques presented in [16] in the case of definite logic programs. For instance, the existential variable X occurring in the body of the above clause 4, is eliminated by applying the unfolding and folding rules and transforming that clause into the following two clauses: $r([X|L], U) \leftarrow X > U \wedge list(L)$ and $r([X|L], U) \leftarrow r(L, U)$. By iterating the transformation process, we eliminate all existential variables and we derive the following program which defines the predicate $prop$:

$Prop_2$: 1. $prop \leftarrow \neg p$
$2'$. $p \leftarrow p_1$
$3'$. $p_1 \leftarrow p_1$

Now, $Prop_2$ is a propositional program and has a *finite* perfect model, which is $\{prop\}$. Since all transformations we have made can be shown to preserve the perfect model, we have that $M(Member) \models \varphi$ iff $M(Prop_2) \models prop$ and, thus, we have completed the proof of φ.

The main contribution of this paper is the proposal of a proof method for showing that a closed first order formula φ holds in the perfect model of a stratified constraint logic program P, that is, $M(P) \models \varphi$. Our proof method is based on program transformations which eliminate existential variables.

The paper is organized as follows. In Section 2 we consider a class of constraint logic programs, called *lr-programs* (*lr* stands for lists of reals), which is Turing complete and for which our proof method is fully automatic. Those programs manipulate finite lists of reals with constraints which are linear equations and inequations over reals. In Section 3 we present the transformation strategy which defines our proof method and we prove its soundness. Due to the undecidability of the first order properties of *lr*-programs, our proof method is

necessarily incomplete. Some experimental results obtained by using a prototype implementation are presented in Section 5. Finally, in Section 6 we discuss related work in the field of program transformation and theorem proving.

2 Constraint Logic Programs over Lists of Reals

We assume that the reals are defined by the usual structure $\mathcal{R} = \langle R, 0, 1, +, \cdot, \leq \rangle$. In order to specify programs and formulas, we use a *typed* first order language [11] with two types: (i) *real*, denoting the set of reals, and (ii) *list of reals* (or *list*, for short), denoting the set of finite lists of reals.

We assume that every element of R is a constant of type *real*. A term p of type *real* is defined as follows:

$p ::= a \mid X \mid p_1 + p_2 \mid a \cdot X$

where a is a real number and X is a variable of type *real*. We also write aX, instead of $a \cdot X$. A term of type *real* will also be called a *linear polynomial*. An *atomic constraint* is a formula of the form: $p_1 = p_2$, or $p_1 < p_2$, or $p_1 \leq p_2$, where p_1 and p_2 are linear polynomials. We also write $p_1 > p_2$ and $p_1 \geq p_2$, instead of $p_2 < p_1$ and $p_2 \leq p_1$, respectively. A *constraint* is a finite conjunction of atomic constraints. A *first order formula over reals* is a first order formula constructed out of atomic constraints by using the usual connectives and quantifiers (i.e., $\neg, \wedge, \vee, \rightarrow, \exists, \forall$). By $F_{\mathcal{R}}$ we will denote the set of first order formulas over reals. A term l of type *list* is defined as follows:

$l ::= L \mid [\,] \mid [p \,|\, l\,]$

where L is a variable of type *list* and p is a linear polynomial. A term of type *list* will also be called a *list*. An *atom* is a formula of the form $r(t_1, \ldots, t_n)$ where r is an n-ary predicate symbol (with $n \geq 0$ and $r \notin \{=, <, \leq\}$) and, for $i = 1, \ldots, n$, t_i is either a linear polynomial or a list. An atom is *linear* if each variable occurs in it at most once. A *literal* is either an atom (i.e., a positive literal) or a negated atom (i.e., a negative literal). A *clause* C is a formula of the form: $A \leftarrow c \wedge L_1 \wedge \ldots \wedge L_m$, where: (i) A is an atom, (ii) c is a constraint, and (iii) $L_1, \ldots, L_m$ are literals. A is called the *head* of the clause, denoted $hd(C)$, and $c \wedge L_1 \wedge \ldots \wedge L_m$ is called the *body* of the clause, denoted $bd(C)$.

A *constraint logic program over lists of reals*, or simply a *program*, is a set of clauses. A program is *stratified* if no predicate depends negatively on itself [2]. Given a term or a formula f, $vars(f)$ denotes the set of variables occurring in f. Given a clause C, a variable V is said to be an *existential variable* of C if $V \in vars(bd(C)) - vars(hd(C))$.

The *definition* of a predicate p in a program P, denoted by $Def(p, P)$, is the set of the clauses of P whose head predicate is p. The *extended definition* of p in P, denoted by $Def^*(p, P)$, is the union of the definition of p and the definitions of all predicates in P on which p depends (positively or negatively). A program is *propositional* if every predicate occurring in the program is 0-ary. Obviously, if P is a propositional program then, for every predicate p, $M(P) \models p$ is decidable.

Definition 1 (*lr*-program). Let X denote a variable of type *real*, L a variable of type *list*, p a linear polynomial, r_1 and r_2 two predicate symbols, and c a constraint. An *lr-clause* is a clause defined as follows:

$$\begin{array}{lll} \textit{head term}: & h & ::= \; X \mid [\,] \mid [X|L] \\ \textit{body term}: & b & ::= \; p \mid L \\ \textit{lr-clause}: & C & ::= \; r_1(h_1,\ldots,h_k) \leftarrow c \\ & & \quad \mid \; r_1(h_1,\ldots,h_k) \leftarrow c \wedge r_2(b_1,\ldots,b_m) \\ & & \quad \mid \; r_1(h_1,\ldots,h_k) \leftarrow c \wedge \neg r_2(b_1,\ldots,b_m) \end{array}$$

where: (i) $vars(p) \neq \emptyset$, (ii) $r_1(h_1,\ldots,h_k)$ is a linear atom, and (iii) clause C has no existential variables. An *lr-program* is a finite set of *lr*-clauses. □

We assume that the following *lr*-clauses belong to every *lr*-program (but we will omit them when writing *lr*-programs):

$$\textit{list}([\,]) \leftarrow$$
$$\textit{list}([X|L]) \leftarrow \textit{list}(L)$$

The specific syntactic form of *lr*-programs is required for the automation of the transformation strategy we will introduce in Section 3. Here is an *lr*-program:

P_1: $\textit{sumlist}([\,], Y) \leftarrow Y = 0$
$\textit{sumlist}([X|L], Y) \leftarrow \textit{sumlist}(L,\, Y - X)$
$\textit{haspositive}([X|L]) \leftarrow X > 0$
$\textit{haspositive}([X|L]) \leftarrow \textit{haspositive}(L)$

The following definition introduces the class of programs and formulas which can be given in input to our proof method.

Definition 2 (Admissible Pair). Let P be an *lr*-program and φ a closed first order formula with no other connectives and quantifiers besides $\neg, \wedge$, and $\exists$. We say that $\langle P, \varphi\rangle$ is an *admissible pair* if: (i) every predicate symbol occurring in φ and different from $\leq, <, =$, also occurs in P, (ii) every predicate of arity n (>0) occurring in P and different from $\leq, <, =$, has at least one argument of type *list*, and (iii) for every proper subformula σ of φ, if σ is of the form $\neg\psi$, then either σ is a formula in $F_{\mathcal{R}}$ or σ has a free occurrence of a variable of type *list*. □

Conditions (ii) and (iii) of Definition 2 are needed to guarantee the soundness of our proof method (see Theorem 3).

Example 1. Let us consider the above program P_1 defining the predicates *sumlist* and *haspositive*, and the formula

$$\pi: \quad \forall L\, \forall Y\, ((\textit{sumlist}(L, Y) \wedge Y > 0) \rightarrow \textit{haspositive}(L))$$

which expresses the fact that if the sum of the elements of a list is positive then the list has at least one positive member. This formula can be rewritten as:

$$\pi_1: \quad \neg\exists L\, \exists Y\, (\textit{sumlist}(L, Y) \wedge Y > 0 \wedge \neg \textit{haspositive}(L))$$

The pair $\langle P_1, \pi_1\rangle$ is admissible. Indeed, the only proper subformula of π_1 of the form $\neg\psi$ is $\neg\textit{haspositive}(L)$ and the free variable L is of type *list*. □

In order to define the semantics of our logic programs we consider $\mathcal{LR}$-interpretations where: (i) the type *real* is mapped to the set of reals, (ii) the type *list* is mapped to the set of lists of reals, and (iii) the symbols $+$, $\cdot$, $=$, $<$, $\leq$, $[\,]$, and $[\_|\_]$ are mapped to the usual corresponding operations and relations on reals and lists of reals. The semantics of a stratified logic program P is assumed to be its *perfect $\mathcal{LR}$-model* $M(P)$, which is defined similarly to the perfect model of a stratified logic program [2,12,17] by considering $\mathcal{LR}$-interpretations, instead of Herbrand interpretations. Note that for every formula $\varphi \in F_{\mathcal{R}}$, we have that $\mathcal{R} \models \varphi$ iff for any $\mathcal{LR}$-interpretation $\mathcal{I}$, $\mathcal{I} \models \varphi$.

Now we present a transformation, called *Clause Form Transformation*, that allows us to derive stratified logic programs starting from formulas, called *statements*, of the form: $A \leftarrow \beta$, where A is an atom and β is a typed first order formula. Our transformation is a variant of the transformation proposed by Lloyd and Topor in [11]. When applying the Clause Form Transformation, we will use the following well known property which guarantees that existential quantification and negation can always be eliminated from first order formulas on reals.

Lemma 1 (Variable Elimination). *For any formula $\varphi \in F_{\mathcal{R}}$ there exist n (≥ 0) constraints $c_1, \ldots, c_n$ such that:* (i) $\mathcal{R} \models \forall(\varphi \leftrightarrow (c_1 \vee \ldots \vee c_n))$, *and* (ii) *every variable in vars$(c_1 \vee \ldots \vee c_n)$ occurs free in φ.*

In what follows we write $C[\gamma]$ to denote a formula where the subformula γ occurs as an *outermost conjunct*, that is, $C[\gamma] = \gamma_1 \wedge \gamma \wedge \gamma_2$ for some (possibly empty) conjunctions γ_1 and γ_2.

Clause Form Transformation.
Input: A statement S whose body has no other connectives and quantifiers besides $\neg, \wedge$, and $\exists$. *Output*: A set of clauses denoted $CFT(S)$.

(*Step* A) Starting from S, repeatedly apply the following rules A.1–A.5 until a set of clauses is generated.

(A.1) If $\gamma \in F_{\mathcal{R}}$ and γ is *not* a constraint, then replace $A \leftarrow C[\gamma]$ by the n statements $A \leftarrow C[c_1], \ldots, A \leftarrow C[c_n]$, where $c_1 \vee \ldots \vee c_n$, with $n \geq 0$, is a disjunction of constraints which is equivalent to γ. (The existence of such a disjunction is guaranteed by Lemma 1 above.)

(A.2) If $\gamma \notin F_{\mathcal{R}}$ then replace $A \leftarrow C[\neg\neg\gamma]$ by $A \leftarrow C[\gamma]$.

(A.3) If $\gamma \wedge \delta \notin F_{\mathcal{R}}$ then replace the statement $A \leftarrow C[\neg(\gamma \wedge \delta)]$ by the two statements $A \leftarrow C[\neg newp(V_1, \ldots, V_k)]$ and $newp(V_1, \ldots, V_k) \leftarrow \gamma \wedge \delta$, where *newp* is a new predicate and $V_1, \ldots, V_k$ are the variables which occur free in $\gamma \wedge \delta$.

(A.4) If $\gamma \notin F_{\mathcal{R}}$ then replace the statement $A \leftarrow C[\neg\exists V\, \gamma]$ by the two statements $A \leftarrow C[\neg newp(V_1, \ldots, V_k)]$ and $newp(V_1, \ldots, V_k) \leftarrow \gamma$, where *newp* is a new predicate and $V_1, \ldots, V_k$ are the variables which occur free in $\exists V\, \gamma$.

(A.5) If $\gamma \notin F_{\mathcal{R}}$ then replace $A \leftarrow C[\exists V\, \gamma]$ by $A \leftarrow C[\gamma\{V/V_1\}]$, where V_1 is a new variable.

(*Step* B) For every clause $A \leftarrow c \wedge G$ such that $L_1, \ldots, L_k$ are the variables of type *list* occurring in G, replace $A \leftarrow c \wedge G$ by $A \leftarrow c \wedge list(L_1) \wedge \ldots \wedge list(L_k) \wedge G$.

Example 2. The set $CFT(prop_1 \leftarrow \pi_1)$, where π_1 is the formula given in Example 1, consists of the following two clauses:

D_2 : $prop_1 \leftarrow \neg new_1$
D_1 : $new_1 \leftarrow Y > 0 \wedge list(L) \wedge sumlist(L, Y) \wedge \neg haspositive(L)$

(The subscripts of the names of these clauses follow the bottom-up order in which they will be processed by the UF_{lr} strategy we will introduce below.) □

By construction, we have that if $\langle P, \varphi \rangle$ is an admissible pair and *prop* is a new predicate symbol, then $P \cup CFT(prop \leftarrow \varphi)$ is a stratified program. The Clause Form Transformation is correct with respect to the perfect $\mathcal{LR}$-model semantics, as stated by the following theorem.

Theorem 1 (Correctness of CFT). *Let $\langle P, \varphi \rangle$ be an admissible pair. Then, $M(P) \models \varphi$ iff $M(P \cup CFT(prop \leftarrow \varphi)) \models prop$.*

In general, a clause in $CFT(prop \leftarrow \varphi)$ is *not* an *lr*-clause because, indeed, existential variables may occur in its body. The clauses of $CFT(prop \leftarrow \varphi)$ are called *typed-definitions*. They are defined as follows.

Definition 3 (Typed-Definition, Hierarchy). A *typed-definition* is a clause of the form: $r(V_1, \ldots, V_n) \leftarrow c \wedge list(L_1) \wedge \ldots \wedge list(L_k) \wedge G$ where: (i) $V_1, \ldots, V_n$ are distinct variables of type *real* or *list*, and (ii) $L_1, \ldots, L_k$ are the variables of type *list* that occur in G. A sequence $\langle D_1, \ldots, D_n \rangle$ of typed-definitions is said to be a *hierarchy* if for $i = 1, \ldots, n$, the predicate of $hd(D_i)$ does not occur in $\{bd(D_1), \ldots, bd(D_i)\}$. □

One can show that given a closed formula φ, the set $CFT(prop \leftarrow \varphi)$ of clauses can be ordered as a hierarchy $\langle D_1, \ldots, D_n \rangle$ of typed-definitions such that $Def(prop, \{D_1, \ldots, D_n\}) = \{D_k, D_{k+1}, \ldots, D_n\}$, for some k with $1 \leq k \leq n$.

3 The Unfold/Fold Proof Method

In this section we present the transformation strategy, called UF_{lr} (Unfold/Fold strategy for *lr*-programs), which defines our proof method for proving properties of *lr*-programs. Our strategy applies in an automatic way the transformation rules for stratified constraint logic programs presented in [8]. In particular, the UF_{lr} strategy makes use of the definition introduction, (positive and negative) unfolding, (positive) folding, and constraint replacement rules. (These rules extend the ones proposed in [6,12] where the unfolding of a clause with respect to a negative literal is not permitted.)

Given an admissible pair $\langle P, \varphi \rangle$, let us consider the stratified program $P \cup CFT(prop \leftarrow \varphi)$. The goal of our UF_{lr} strategy is to derive a program *TransfP*

such that $Def^*(prop, TransfP)$ is propositional and, thus, $M(TransfP) \models prop$ is decidable. We observe that, in order to achieve this goal, it is enough that the derived program *TransfP* is an *lr*-program, as stated by the following lemma, which follows directly from Definition 1.

Lemma 2. *Let P be an lr-program and p be a predicate occurring in P. If p is 0-ary then $Def^*(p, P)$ is a propositional program.*

As already said, the clauses in $CFT(prop \leftarrow \varphi)$ form a hierarchy $\langle D_1, \ldots, D_n \rangle$ of typed-definitions. The UF_{lr} strategy consists in transforming, for $i = 1, \ldots, n$, clause D_i into a set of *lr*-clauses. The transformation of D_i is performed by applying the following three substrategies, in this order: (i) *unfold*, which unfolds D_i with respect to the positive and negative literals occurring in its body, thereby deriving a set *Cs* of clauses, (ii) *replace-constraints*, which replaces the constraints appearing in the clauses of *Cs* by equivalent ones, thereby deriving a new set *Es* of clauses, and (iii) *define-fold*, which introduces a set *NewDefs* of new typed-definitions (which are not necessarily *lr*-clauses) and folds all clauses in *Es*, thereby deriving a set *Fs* of *lr*-clauses. Then each new definition in *NewDefs* is transformed by applying the above three substrategies, and the whole UF_{lr} strategy terminates when no new definitions are introduced. The substrategies *unfold*, *replace-constraints*, and *define-fold* will be described in detail below.

The UF_{lr} Transformation Strategy.
Input: An *lr*-program P and a hierarchy $\langle D_1, \ldots, D_n \rangle$ of typed-definitions.
Output: A set *Defs* of typed-definitions including $D_1, \ldots, D_n$, and an *lr*-program *TransfP* such that $M(P \cup Defs) = M(TransfP)$.

TransfP := *P*; *Defs* := $\{D_1, \ldots, D_n\}$;
for $i = 1, \ldots, n$ **do** *InDefs* := $\{D_i\}$;
  while *InDefs* $\neq \emptyset$ **do**
  unfold(*InDefs*, *TransfP*, *Cs*);
  replace-constraints(*Cs*, *Es*);
  define-fold(*Es*, *Defs*, *NewDefs*, *Fs*);
  TransfP := *TransfP* $\cup$ *Fs*; *Defs* := *Defs* $\cup$ *NewDefs*; *InDefs* := *NewDefs*;
  end-while;
  eval-props: for each predicate p such that $Def^*(p, TransfP)$ is propositional,
  if $M(TransfP) \models p$ **then** *TransfP* := (*TransfP* − *Def*(*p*, *TransfP*)) $\cup$ $\{p \leftarrow\}$
    else *TransfP* := (*TransfP* − *Def*(*p*, *TransfP*))
end-for

Our assumption that $\langle D_1, \ldots, D_n \rangle$ is a hierarchy ensures that, when transforming clause D_i, for $i = 1, \ldots, n$, we only need the clauses obtained after the transformation of $D_1, \ldots, D_{i-1}$. These clauses are those of the current value of *TransfP*.

The following *unfold* substrategy transforms a set *InDefs* of typed-definitions by first applying the unfolding rule with respect to each positive literal in the

body of a clause and then applying the unfolding rule with respect to each negative literal in the body of a clause. In the sequel, we will assume that the conjunction operator $\wedge$ is associative, commutative, idempotent, and with neutral element *true*. In particular, the order of the conjuncts will *not* be significant.

The *unfold* Substrategy.
Input: An *lr*-program *Prog* and a set *InDefs* of typed-definitions.
Output: A set *Cs* of clauses.

Initially, no literal in the body of a clause of *InDefs* is marked as 'unfolded'.

Positive Unfolding: **while** there exists a clause C in *InDefs* of the form $H \leftarrow c \wedge G_L \wedge A \wedge G_R$, where A is an atom which is not marked as 'unfolded' **do**
Let C_1: $K_1 \leftarrow c_1 \wedge B_1$, ..., C_m: $K_m \leftarrow c_m \wedge B_m$ be all clauses of program *Prog* (where we assume $vars(Prog) \cap vars(C) = \emptyset$) such that, for $i = 1, \ldots, m$, (i) there exists a most general unifier ϑ_i of A and K_i, and (ii) the constraint $(c \wedge c_i)\vartheta_i$ is satisfiable. Let U be the following set of clauses:

$$U = \{(H \leftarrow c \wedge c_1 \wedge G_L \wedge B_1 \wedge G_R)\vartheta_1, \ldots, (H \leftarrow c \wedge c_m \wedge G_L \wedge B_m \wedge G_R)\vartheta_m\}$$

Let W be the set of clauses derived from U by removing all clauses of the form

$$H \leftarrow c \wedge G_L \wedge A \wedge \neg A \wedge G_R$$

Inherit the markings of the literals in the body of the clauses of W from those of C, and mark as 'unfolded' the literals $B_1\vartheta_1, \ldots, B_m\vartheta_m$;
$InDefs := (InDefs - \{C\}) \cup W$;
end-while;

Negative Unfolding: **while** there exists a clause C in *InDefs* of the form $H \leftarrow c \wedge G_L \wedge \neg A \wedge G_R$, where $\neg A$ is a literal which is not marked as 'unfolded' **do**
Let C_1: $K_1 \leftarrow c_1 \wedge B_1, \ldots, C_m$: $K_m \leftarrow c_m \wedge B_m$ be all clauses of program *Prog* (where we assume that $vars(Prog) \cap vars(C) = \emptyset$) such that, for $i = 1, \ldots, m$, there exists a most general unifier ϑ_i of A and K_i. By our assumptions on *Prog* and on the initial value of *InDefs*, and as a result of the previous *Positive Unfolding* phase, we have that, for $i = 1, \ldots, m$, B_i is either the empty conjunction *true* or a literal and $A = K_i\vartheta_i$. Let U be the following set of statements:

$$U = \{H \leftarrow c \wedge d_1\vartheta_1 \wedge \ldots \wedge d_m\vartheta_m \wedge G_L \wedge L_1\vartheta_1 \wedge \ldots \wedge L_m\vartheta_m \wedge G_R \mid$$
(i) for $i = 1, \ldots, m$, either ($d_i = c_i$ and $L_i = \neg B_i$) or ($d_i = \neg c_i$ and $L_i = true$), and (ii) $c \wedge d_1\vartheta_1 \wedge \ldots \wedge d_m\vartheta_m$ is satisfiable$\}$

Let W be the set of clauses derived from U by applying as long as possible the following rules:

- remove $H \leftarrow c \wedge G_L \wedge \neg true \wedge G_R$ and $H \leftarrow c \wedge G_L \wedge A \wedge \neg A \wedge G_R$
- replace $\neg\neg A$ by A, $\neg(p_1 \leq p_2)$ by $p_2 < p_1$, and $\neg(p_1 < p_2)$ by $p_2 \leq p_1$
- replace $H \leftarrow c_1 \wedge \neg(p_1 = p_2) \wedge c_2 \wedge G$ by $H \leftarrow c_1 \wedge p_1 < p_2 \wedge c_2 \wedge G$
  $H \leftarrow c_1 \wedge p_2 < p_1 \wedge c_2 \wedge G$

Inherit the markings of the literals in the body of the clauses of W from those of C, and mark as 'unfolded' the literals $L_1\vartheta_1, \ldots, L_m\vartheta_m$;
$InDefs := (InDefs - \{C\}) \cup W$;

end-while;
$Cs := InDefs$.

Negative Unfolding is best explained through an example. Let us consider a program consisting of the clauses C: $H \leftarrow c \wedge \neg A$, $A \leftarrow c_1 \wedge B_1$, and $A \leftarrow c_2 \wedge B_2$. The negative unfolding of C w.r.t. $\neg A$ gives us the following four clauses:

$H \leftarrow c \wedge \neg c_1 \wedge \neg c_2$
$H \leftarrow c \wedge c_1 \wedge \neg c_2 \wedge \neg B_1$
$H \leftarrow c \wedge \neg c_1 \wedge c_2 \wedge \neg B_2$
$H \leftarrow c \wedge c_1 \wedge c_2 \wedge \neg B_1 \wedge \neg B_2$

whose conjunction is equivalent to $H \leftarrow c \wedge \neg((c_1 \wedge B_1) \vee (c_2 \wedge B_2))$.

Example 3. Let us consider the program-property pair $\langle P_1, \pi_1 \rangle$ of Example 1. In order to prove that $M(P_1) \models \pi_1$, we apply the UF_{lr} strategy starting from the hierarchy $\langle D_1, D_2 \rangle$ of typed-definitions of Example 2. During the first execution of the body of the for-loop of that strategy, the *unfold* substrategy is applied, as we now indicate, by using as input the program P_1 and the set $\{D_1\}$ of clauses. We have the following positive and negative unfolding steps.

Positive Unfolding. By unfolding clause D_1 w.r.t. $list(L)$ and then unfolding the resulting clauses w.r.t. $sumlist(L, Y)$, we get:

C_1: $new_1 \leftarrow Y > 0 \wedge list(L) \wedge sumlist(L, Y - X) \wedge \neg haspositive([X|L])$

Negative Unfolding. By unfolding clause C_1 w.r.t. $\neg haspositive([X|L])$, we get:

C_2: $new_1 \leftarrow Y > 0 \wedge X \leq 0 \wedge list(L) \wedge sumlist(L, Y - X) \wedge \neg haspositive(L)$ $\square$

The correctness of the *unfold* substrategy follows from the fact that the positive and negative unfoldings are performed according to the rules presented in [8]. The termination of that substrategy is due to the fact that the number of literals which are not marked as 'unfolded' and which occur in the body of a clause, decreases when that clause is unfolded. Thus, we have the following result.

Lemma 3. *Let Prog be an lr-program and let InDefs be a set of typed-definitions such that the head predicates of the clauses of InDefs do not occur in Prog. Then, given the inputs Prog and InDefs, the unfold substrategy terminates and returns a set Cs of clauses such that* $M(Prog \cup InDefs) = M(Prog \cup Cs)$.

The *replace-constraints* substrategy derives from a set Cs of clauses a new set Es of clauses by applying equivalences between existentially quantified disjunctions of constraints. We use the following two rules: *project* and *clause split*.

Given a clause $H \leftarrow c \wedge G$, the *project* rule eliminates all variables that occur in c and do not occur elsewhere in the clause. Thus, *project* returns a new clause $H \leftarrow d \wedge G$ such that $\mathcal{R} \models \forall((\exists X_1 \ldots \exists X_k\, c) \leftrightarrow d)$, where: (i) $\{X_1, \ldots, X_k\} = vars(c) - vars(\{H, G\})$, and (ii) $vars(d) \subseteq vars(c) - \{X_1, \ldots, X_k\}$. In our prototype theorem prover (see Section 5), the *project* rule is implemented by using a variant of the Fourier-Motzkin Elimination algorithm [1].

The *clause split* rule replaces a clause C by two clauses C_1 and C_2 such that, for $i = 1, 2$, the number of occurrences of existential variables in C_i is less

than the number of occurrences of existential variables in C. The clause split rule applies the following property, which expresses the fact that $\langle \mathcal{R}, \leq \rangle$ is a linear order: $\mathcal{R} \models \forall X \forall Y\, (X < Y \vee Y \leq X)$. For instance, a clause of the form $H \leftarrow Z \leq X \wedge Z \leq Y \wedge G$, where Z is an existential variable occurring in the conjunction G of literals and X and Y are not existential variables, is replaced by the two clauses $H \leftarrow Z \leq X \wedge X < Y \wedge G$ and $H \leftarrow Z \leq Y \wedge Y \leq X \wedge G$. The decrease of the number of occurrences of existential variables guarantees that we can apply the clause split rule a finite number of times only.

The *replace-constraints* **Substrategy.**
Input: A set Cs of clauses. *Output*: A set Es of clauses.

• *Introduce Equations.* (A) From Cs we derive a new set R_1 of clauses by applying as long as possible the following two rules, where p denotes a linear polynomial which is not a variable, and Z denotes a fresh new variable:

(R.1) $H \leftarrow c \wedge G_L \wedge r(\ldots, p, \ldots) \wedge G_R$ is replaced by
$H \leftarrow c \wedge Z = p \wedge G_L \wedge r(\ldots, Z, \ldots) \wedge G_R$

(R.2) $H \leftarrow c \wedge G_L \wedge \neg r(\ldots, p, \ldots) \wedge G_R$ is replaced by
$H \leftarrow c \wedge Z = p \wedge G_L \wedge \neg r(\ldots, Z, \ldots) \wedge G_R$

(B) From R_1 we derive a new set R_2 of clauses by applying to every clause C in R_1 the following rule. Let C be of the form $H \leftarrow c \wedge G$. Suppose that $\mathcal{R} \models \forall\, (c \leftrightarrow (X_1 = p_1 \wedge X_n = p_n \wedge d))$, where: (i) $X_1, \ldots, X_n$ are existential variables of C, (ii) $vars(X_1 = p_1 \wedge \ldots \wedge X_n = p_n \wedge d) \subseteq vars(c)$, (iii) $\{X_1, \ldots, X_n\} \cap vars(\{p_1, \ldots, p_n, d\}) = \emptyset$. Then we replace C by $H \leftarrow X_1 = p_1 \wedge \ldots \wedge X_n = p_n \wedge d \wedge G$.

• *Project.* We derive a new set R_3 of clauses by applying to every clause in R_2 the *project* rule.

• *Clause Split.* From R_3 we derive a new set R_4 of clauses by applying as long as possible the following rule. Let C be a clause of the form $H \leftarrow c_1 \wedge c_2 \wedge c \wedge G$ (modulo commutativity of $\wedge$). Let E be the set of existential variables of C. Let $X \in E$ and let d_1 and d_2 be two inequations such that $\mathcal{R} \models \forall\, ((c_1 \wedge c_2) \leftrightarrow (d_1 \wedge d_2))$. Suppose that: (i) $d_1 \wedge d_2$ is of one of the following six forms:

$X \leq p_1 \wedge X \leq p_2$	$X \leq p_1 \wedge X < p_2$	$X < p_1 \wedge X < p_2$
$p_1 \leq X \wedge p_2 \leq X$	$p_1 \leq X \wedge p_2 < X$	$p_1 < X \wedge p_2 < X$

and (ii) $(vars(p_1) \cup vars(p_2)) \cap E = \emptyset$.
Then C is replaced by the following two clauses: C_1: $H \leftarrow d_1 \wedge p_1 < p_2 \wedge c \wedge G$ and C_2: $H \leftarrow d_2 \wedge p_2 \leq p_1 \wedge c \wedge G$, and then each clause in $\{C_1, C_2\}$ with an unsatisfiable constraint in its body is removed.

• *Eliminate Equations.* From R_4 we derive the new set Es of clauses by applying to every clause C in R_4 the following rule. If C is of the form $H \leftarrow X_1 = p_1 \wedge \ldots \wedge X_n = p_n \wedge d \wedge G$ where $\{X_1, \ldots, X_n\} \cap vars(\{p_1, \ldots p_n, d\}) = \emptyset$, then C is replaced by $(H \leftarrow d \wedge G)\{X_1/p_1, \ldots, X_n/p_n\}$.

The transformation described at Point (A) of *Introduce Equations* allows us to treat all polynomials occurring in the body of a clause in a uniform way as arguments of constraints. The transformation described at Point (B) of *Introduce Equations* identifies those existential variables which can be eliminated during the final *Eliminate Equations* transformation. That elimination is performed by substituting, for $i=1,\ldots,n$, the variable X_i by the polynomial p_i.

Example 4. By applying the *replace-constraints* substrategy, clause C_2 of Example 3 is transformed as follows. By introducing equations we get:

C_3: $new_1 \leftarrow Y>0 \wedge X\leq 0 \wedge Z=Y-X \wedge list(L) \wedge sumlist(L,Z) \wedge \neg haspositive(L)$

Then, by applying the *project* transformation, we get:

C_4: $new_1 \leftarrow Z>0 \wedge list(L) \wedge sumlist(L,Z) \wedge \neg haspositive(L)$ □

The correctness of the *replace-constraints* substrategy is a straightforward consequence of the fact that the *Introduce Equations*, *Project*, *Clause Split*, and *Eliminate Equations* transformations are performed by using the rule of *replacement based on laws* presented in [8]. The termination of *Introduce Equations* and *Eliminate Equations* is obvious. The termination of *Project* is based on the termination of the specific algorithm used for variable elimination (e.g., Fourier-Motzkin algorithm). As already mentioned, the termination of *Clause Split* is due to the fact that at each application of this transformation the number of occurrences of existential variables decreases. Thus, we get the following lemma.

Lemma 4. *For any program Prog and set Cs ⊆ Prog of clauses, the replace-constraints substrategy with input Cs terminates and returns a set Es of clauses such that* $M(Prog) = M((Prog - Cs) \cup Es)$.

The *define-fold* substrategy eliminates all existential variables in the clauses of the set *Es* obtained after the *unfold* and *replace-constraints* substrategies. This elimination is done by folding all clauses in *Es* that contain existential variables. In order to make these folding steps we use the typed-definitions in *Defs* and, if necessary, we introduce new typed-definitions which we add to the set *NewDefs*.

The *define-fold* **Substrategy.**
Input: A set *Es* of clauses and a set *Defs* of typed-definitions.
Output: A set *NewDefs* of typed-definitions and a set *Fs* of *lr*-clauses.

Initially, both *NewDefs* and *Fs* are empty.
for each clause C: $H \leftarrow c \wedge G$ in *Es* **do**
if C is an *lr*-clause **then** $Fs := Fs \cup \{C\}$ **else**

- *Define.* Let E be the set of existential variables of C. We consider a clause *NewD* of the form $newp(V_1,\ldots,V_m) \leftarrow d \wedge B$ constructed as follows:
(1) let c be of the form $c_1 \wedge c_2$, where $vars(c_1) \cap E = \emptyset$ and for every atomic constraint a occurring in c_2, $vars(a) \cap E \neq \emptyset$; let $d \wedge B$ be the most general (modulo variants) conjunction of constraints and literals such that there exists a substitution ϑ with the following properties: (i) $(d \wedge B)\vartheta = c_2 \wedge G$, and (ii) for

each binding V/p in ϑ, V is a variable not occurring in C, $vars(p) \neq \emptyset$, and $vars(p) \cap E = \emptyset$;
(2) *newp* is a new predicate symbol;
(3) $\{V_1, \ldots, V_m\} = vars(d \wedge B) - E$.
NewD is added to *NewDefs*, unless in *Defs* there exists a typed-definition D which is equal to *NewD*, modulo the name of the head predicate, the names of variables, equivalence of constraints, and the order and multiplicity of literals in the body. If such a clause D belongs to *Defs* and no other clause in *Defs* has the same head predicate as D, then we assume that $NewD = D$.

- *Fold.* Clause C is folded using clause *NewD* as follows:
$Fs := Fs \cup \{H \leftarrow c_1 \wedge newp(V_1, \ldots, V_m)\vartheta\}$.
end-for

Example 5. Let us consider the clause C_4 derived at the end of Example 4. The *Define* phase produces a typed-definition which is a variant of the typed-definition D_1 introduced at the beginning of the application of the strategy (see Example 2). Thus, C_4 is folded using clause D_1, and we get the clause:

C_5: $new_1 \leftarrow new_1$

Let us now describe how the proof of $M(P_1) \models \pi_1$ proceeds. The program *TransfP* derived so far consists of clause C_5 together with the clauses defining the predicates *list*, *sumlist*, and *haspositive*. Thus, $Def^*(new_1, TransfP)$ consists of clause C_5 only, which is propositional and, by *eval-props*, we remove C_5 from *TransfP* because $M(TransfP) \not\models new_1$. The strategy continues by considering the typed definition D_2 (see Example 2). By unfolding D_2 with respect to $\neg new_1$ we get the final program *TransfP*, which consists of the clause $prop_1 \leftarrow$ together with the clauses for *list*, *sumlist*, and *haspositive*. Thus, $M(TransfP) \models prop_1$ and, therefore, $M(P_1) \models \pi_1$. □

The proof of correctness for the *define-fold* substrategy is more complex than the proofs for the other substrategies. The correctness results for the unfold/fold transformations presented in [8] guarantee the correctness of a folding transformation if each typed-definition used for folding is unfolded w.r.t. a positive literal during the application of the UF_{lr} transformation strategy. The fulfillment of this condition is ensured by the following two facts: (1) by the definition of an admissible pair and by the definition of the Clause Form Transformation, each typed-definition has at least one positive literal in its body (indeed, by Condition (iii) of Definition 2 each negative literal in the body of a typed-definition has at least one variable of type *list* and, therefore, the body of the typed-definition has at least one *list* atom), and (2) in the *Positive Unfolding* phase of the *unfold* substrategy, each typed-definition is unfolded w.r.t. all positive literals.

Note that the set *Fs* of clauses derived by the *define-fold* substrategy is a set of *lr*-clauses. Indeed, by the *unfold* and *replace-constraints* substrategies, we derive a set *Es* of clauses of the form $r(h_1, \ldots, h_k) \leftarrow c \wedge G$, where $h_1, \ldots, h_k$ are head terms (see Definition 1). By folding we derive clauses of the form

$r(h_1, \ldots, h_k) \leftarrow c_1 \wedge newp(V_1, \ldots, V_m)\vartheta$

where $vars(c_1 \wedge newp(V_1, \ldots, V_m)\vartheta) \subseteq vars(r(h_1, \ldots, h_k))$, and for $i = 1, \ldots, m$, $vars(V_i\vartheta) \neq \emptyset$ (by the conditions at Points (1)–(3) of the *Define* phase). Hence, all clauses in *Fs* are *lr*-clauses.

The termination of the *define-fold* substrategy is obvious, as each clause is folded at most once. Thus, we have the following result.

Lemma 5. *During the UF_{lr} strategy, if the define-fold substrategy takes as inputs the set Es of clauses and the set Defs of typed-definitions, then this substrategy terminates and returns a set NewDefs of typed-definitions and a set Fs of lr-clauses such that $M(TransfP \cup Es \cup NewDefs) = M(TransfP \cup Fs \cup NewDefs)$.*

By using Lemmata 3, 4, and 5 we get the following correctness result for the UF_{lr} strategy.

Theorem 2. *Let P be an lr-program and $\langle D_1, \ldots, D_n \rangle$ a hierarchy of typed-definitions. Suppose that the UF_{lr} strategy with inputs P and $\langle D_1, \ldots, D_n \rangle$ terminates and returns a set Defs of typed-definitions and a program TransfP. Then:* (i) *TransfP is an lr-program and* (ii) $M(P \cup Defs) = M(TransfP)$.

Now, we are able to prove the soundness of the unfold/fold proof method.

Theorem 3 (Soundness of the Unfold/Fold Proof Method). *Let $\langle P, \varphi \rangle$ be an admissible pair and let $\langle D_1, \ldots, D_n \rangle$ be the hierarchy of typed-definitions obtained from $prop \leftarrow \varphi$ by the Clause Form Transformation. If the UF_{lr} strategy with inputs P and $\langle D_1, \ldots, D_n \rangle$ terminates and returns a program TransfP, then:*

$$M(P) \models \varphi \quad \textit{iff} \quad (prop \leftarrow) \ \in TransfP$$

Proof. By Theorem 1 and Point (ii) of Theorem 2, we have that $M(P) \models \varphi$ iff $M(TransfP) \models prop$. By Point (i) of Theorem 2 and Lemma 2 we have that $Def^*(prop, TransfP)$ is propositional. Since the last step of the UF_{lr} strategy is an application of the *eval-props* transformation, we have that $Def^*(prop, TransfP)$ is either the singleton $\{prop \leftarrow\}$, if $M(TransfP) \models prop$, or the empty set, if $M(TransfP) \not\models prop$. □

4 A Complete Example

As an example of application of our transformation strategy for proving properties of constraint logic programs we consider the *lr*-program *Member* and the property φ given in the Introduction. The formula φ is rewritten as follows:

$\varphi_1 : \ \neg\exists L\, \neg\exists U\, \neg\exists X\, (X > U \wedge member(X, L))$

The pair $\langle Member, \varphi_1 \rangle$ is admissible. By applying the Clause Form Transformation starting from the statement $prop \leftarrow \varphi_1$, we get the following clauses:

D_4: $prop \leftarrow \neg p$
D_3: $p \leftarrow list(L) \wedge \neg q(L)$
D_2: $q(L) \leftarrow list(L) \wedge \neg r(L, U)$
D_1: $r(L, U) \leftarrow X > U \wedge list(L) \wedge member(X, L)$

where $\langle D_1, D_2, D_3, D_4\rangle$ is a hierarchy of typed-definitions. Note that the three nested negations in φ_1 generate the three atoms p, $q(L)$, and $r(L,U)$ with their typed-definitions D_3, D_2, and D_1, respectively. The arguments of p, q, and r are the free variables of the corresponding subformulas of φ_1. For instance, $r(L,U)$ corresponds to the subformula $\exists X\,(X > U \wedge \mathit{member}(X,L))$ which has L and U as free variables. Now we apply the UF_{lr} strategy starting from the program *Member* and the hierarchy $\langle D_1, D_2, D_3, D_4\rangle$.

• *Execution of the for-loop with* $i = 1$. We have: $\mathit{InDefs} = \{D_1\}$. By unfolding clause D_1 w.r.t. the atoms $\mathit{list}(L)$ and $\mathit{member}(X,L)$ we get:

1.1 $r([X|T],U) \leftarrow X > U \wedge \mathit{list}(T)$
1.2 $r([X|T],U) \leftarrow Y > U \wedge \mathit{list}(T) \wedge \mathit{member}(Y,T)$

No replacement of constraints is performed. Then, by folding clause 1.2 using D_1, we get:

1.3 $r([X|T],U) \leftarrow r(T,U)$

After the *define-fold* substrategy the set *Fs* of clauses is $\{1.1, 1.3\}$, and at this point the program *TransfP* is *Member* $\cup\, \{1.1, 1.3\}$. No new definitions are introduced and, thus, $\mathit{InDefs} = \emptyset$ and the while-loop terminates. *eval-props* is not performed because the predicate r is not propositional.

• *Execution of the for-loop with* $i = 2$. We have: $\mathit{InDefs} = \{D_2\}$. We unfold clause D_2 w.r.t. $\mathit{list}(L)$ and $\neg r(L,U)$, we get:

2.1 $q([\,]) \leftarrow$
2.2 $q([X|T]) \leftarrow X \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

No replacement of constraints is performed. Then we introduce the new definition:

2.3 $q_1(X,T) \leftarrow X \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

and we fold clause 2.2 using clause 2.3. We get:

2.4 $q([X|T]) \leftarrow q_1(X,T)$

Since $\mathit{NewDefs} = \mathit{InDefs} = \{2.3\}$ we execute again the body of the while-loop. By unfolding clause 2.3 w.r.t. $\mathit{list}(T)$ and $\neg r(T,U)$, we get:

2.5 $q_1(X,[\,]) \leftarrow$
2.6 $q_1(X,[Y|T]) \leftarrow X \leq U \wedge Y \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

By applying *replace-constraints*, clause 2.6 generates the following two clauses:

2.6.1 $q_1(X,[Y|T]) \leftarrow X > Y \wedge X \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$
2.6.2 $q_1(X,[Y|T]) \leftarrow X \leq Y \wedge Y \leq U \wedge \mathit{list}(T) \wedge \neg r(T,U)$

By folding clauses 2.6.1 and 2.6.2 using clause 2.3, we get:

2.7 $q_1(X,[Y|T]) \leftarrow X > Y \wedge q_1(X,T)$
2.8 $q_1(X,[Y|T]) \leftarrow X \leq Y \wedge q_1(Y,T)$

At this point the program *TransfP* is *Member* $\cup \{1.1, 1.3, 2.1, 2.4, 2.5, 2.7, 2.8\}$. No new definitions are introduced and, thus, the while-loop terminates. *eval-props* is not performed because the predicates q and q_1 are not propositional.

• *Execution of the for-loop with* $i = 3$. We have: $\mathit{InDefs} = \{D_3\}$. By unfolding clause D_3 w.r.t. $\mathit{list}(L)$ and $\neg q(L)$, we get:

3.1 $p \leftarrow \mathit{list}(T) \wedge \neg q_1(X,T)$

No replacement of constraints is performed. The following new definition:

3.2 $p_1 \leftarrow \mathit{list}(T) \wedge \neg q_1(X,T)$

is introduced. Then by folding clause 3.1 using clause 3.2, we get:

3.3 $p \leftarrow p_1$

Since $\mathit{NewDefs} = \mathit{InDefs} = \{3.2\}$ we execute again the body of the while-loop. By unfolding clause 3.2 w.r.t. $\mathit{list}(T)$ and $\neg q_1(X,T)$, we get:

3.4 $p_1 \leftarrow X > Y \wedge \mathit{list}(T) \wedge \neg q_1(X,T)$

3.5 $p_1 \leftarrow X \leq Y \wedge \mathit{list}(T) \wedge \neg q_1(Y,T)$

Since the variable Y occurring in the constraints $X > Y$ and $X \leq Y$ is existential, we apply the *project* rule to clauses 3.4 and 3.5 and we get the following clause:

3.6 $p_1 \leftarrow \mathit{list}(T) \wedge \neg q_1(X,T)$

This clause can be folded using clause 3.2, thereby deriving the following clause:

3.7 $p_1 \leftarrow p_1$

Clauses 3.3 and 3.7 are added to *TransfP*. Since the predicates p and p_1 are both propositional, we execute *eval-props*. We have that: (i) $M(\mathit{TransfP}) \not\models p_1$ and (ii) $M(\mathit{TransfP}) \not\models p$. Thus, clauses 3.3 and 3.7 are removed from *TransfP*. Hence, $\mathit{TransfP} = \mathit{Member} \cup \{1.1, 1.3, 2.1, 2.4, 2.5, 2.7, 2.8\}$.

• *Execution of the for-loop with* $i = 4$. We have: $\mathit{InDefs} = \{D_4\}$. By unfolding clause D_4 w.r.t. $\neg p$, we get the clause:

4. $\mathit{prop} \leftarrow$

This clause shows that, as expected, property φ holds for any finite list of reals.

5 Experimental Results

We have implemented our proof method by using the MAP transformation system [13] running under SICStus Prolog on a 900MHz Power PC. Constraint satisfaction and entailment were performed using the clp(r) module of SICStus. Our prototype has automatically proved the properties listed in the following table, where the predicates *member*, *sumlist*, and *haspositive* are defined as shown in Sections 1 and 2, and the other predicates have the following meanings: (i) $\mathit{ord}(L)$ holds iff L is a list of the form $[a_1, \ldots, a_n]$ and for $i = 1, \ldots, n-1$, $a_i \leq a_{i+1}$, (ii) $\mathit{sumzip}(L, M, N)$ holds iff L, M, and N are lists of the form $[a_1, \ldots, a_n]$, $[b_1, \ldots, b_n]$, and $[a_1 + b_1, \ldots, a_n + b_n]$, respectively, and (iii) $\mathit{leqlist}(L, M)$ holds iff L and M are lists of the form $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_n]$, respectively, and for $i = 1, \ldots, n$, $a_i \leq b_i$. We do not write here the *lr*-programs which define the predicates $\mathit{ord}(L)$, $\mathit{sumzip}(L, M, N)$, and $\mathit{leqlist}(L, M)$.

Property	Time
$\forall L\, \exists M\, \forall Y\, (\mathit{member}(Y, L) \rightarrow Y \leq M)$	140 ms
$\forall L\, \forall Y\, ((\mathit{sumlist}(L, Y) \wedge Y > 0) \rightarrow \mathit{haspositive}(L))$	170 ms
$\forall L\, \forall Y\, ((\mathit{sumlist}(L, Y) \wedge Y > 0) \rightarrow \exists X(\mathit{member}(X, L) \wedge X > 0))$	160 ms
$\forall L\, \forall M\, \forall N\, ((\mathit{ord}(L) \wedge \mathit{ord}(M) \wedge \mathit{sumzip}(L, M, N)) \rightarrow \mathit{ord}(N))$	160 ms
$\forall L\, \forall M\, ((\mathit{leqlist}(L, M) \wedge \mathit{sumlist}(L, X) \wedge \mathit{sumlist}(M, Y)) \rightarrow X \leq Y)$	50 ms

6 Related Work and Conclusions

We have presented a method for proving first order properties of constraint logic programs based on unfold/fold program transformations, and we have shown that the ability of unfold/fold transformations to eliminate existential variables [16] can be turned into a useful theorem proving method. We have provided a fully automatic strategy for the class of *lr*-programs, which are programs acting on reals and finite lists of reals, with constraints as linear equations and inequations over reals. The choice of lists is actually a simplifying assumption we have made and we believe that the extension of our method to any finitely generated data structure is quite straightforward. However, the use of constraints over the reals is an essential feature of our method, because quantifier elimination from constraints is a crucial subprocedure of our transformation strategy.

The first order properties of *lr*-programs are undecidable (and not even semi-decidable), because one can encode every partial recursive function as an *lr*-program without list arguments. As a consequence our proof method is necessarily incomplete. We have implemented the proof method based of program transformation and we have proved some simple, yet non-trivial, properties. As the experiments show, the performance of our method is encouraging.

Our method is an extension of the method presented in [15] which considers logic programs without constraints. The addition of constraints is a very relevant feature, because it provides more expressive power and, as already mentioned, we may use special purpose theorem provers for checking constraint satisfaction and for quantifier elimination. Our method can also be viewed as an extension of other techniques based on unfold/fold transformations for proving equivalences of predicates [14,19], and indeed, our method can deal with a class of first order formulas which properly includes equivalences.

Some papers have proposed transformational techniques to prove propositional temporal properties of finite and/or infinite state systems (see, for instance, [7,10,19]). Since propositional temporal logic can be encoded in first order logic, in principle these techniques can be viewed as instances of the unfold/fold proof method presented here.

However, it should be noted that the techniques described in [7,10,19] have their own peculiarities because they are tailored to the specific problem of verifying concurrent systems.

Finally, we think that a direct comparison of the power of our proof method with that of traditional theorem provers is somewhat inappropriate. The techniques used in those provers are very effective and are the result of a well established line of research (see, for instance, [3] for a survey on the automation of mathematical induction). However, our approach has its novelty and is based on principles which have not been explored in the field of theorem proving. In particular, the idea of making inductive proofs by unfold/fold transformations for eliminating quantifiers, has not yet been investigated within the theorem proving community.

7 Acknowledgments

We would like to thank the anonymous referees for their helpful comments and suggestions.

References

1. K. R. Apt. *Principles of Constraint Programming*. Cambridge Univ. Press, 2003.
2. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
3. A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, volume I, pages 845–911. North Holland, 2001.
4. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
5. B. Courcelle. Equivalences and transformations of regular systems – applications to recursive program schemes and grammars. *Theor. Comp. Sci.*, 42:1–122, 1986.
6. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings VCL'01, Florence, Italy*, pages 85–96. University of Southampton, UK, 2001.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In *Program Development in Computational Logic*, LNCS 3049, pages 292–340. Springer, 2004.
9. L. Kott. The McCarthy's induction principle: 'oldy' but 'goody'. *Calcolo*, 19(1):59–69, 1982.
10. 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*, LNCS 1817, pages 63–82. Springer, 1999.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1987. 2nd Edition.
12. M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.
13. The MAP System. `http://www.iasi.cnr.it/~proietti/system.html`.
14. A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2&3):197–230, 1999.
15. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In *Proc. CL 2000, London, UK*, LNAI 1861, pp. 613–628. Springer, 2000.
16. M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theor. Comp. Sci.*, 142(1):89–124, 1995.
17. T. C. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1987.
18. M. O. Rabin. Decidable theories. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, 1977.
19. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Proc. TACAS 2000*, LNCS 1785, pp. 172–187. Springer, 2000.
20. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, ed., *Proceedings of ICLP '84*, pages 127–138, Uppsala, Sweden, 1984.

A Master-Slave Architecture to Integrate Sets and Finite Domains in Java

F. Bergenti[1], E. Panegai[2] and G. Rossi[2]

[1] Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
bergenti@ce.unipr.it
[2] Dipartimento di Matematica
Università degli Studi di Parma
panegai@cs.unipr.it, gianfranco.rossi@unipr.it

Abstract. This paper summarizes the lessons learned from the integration of two Java constraint solvers: a set solver (namely JSetL) and a finite domains solver (namely JFD). The most relevant outcome of this experience is the definition of a generic master-slave architecture that can be used to support the cooperation of different solvers. Each slave is responsible for managing constraints of a particular sort and the master, which is also a solver, is in charge of distributing tasks according to a static, a-priori policy. This paper first presents this generic architecture in an abstract form; then, its concrete instantiation to the selected case study, i.e., the integration of JSetL and JFD, is also described. This case study was selected because it fully demonstrates the possibilities of this architecture as: *(i)* the poor performances of JSetL on non-set variables are overwhelmed by the cooperation with JFD; and *(ii)* the expressive power of JSetL is fully preserved and the integration with JFD demands no restrictions.

1 Introduction

Rarely a single constraint solver outperforms all others in all situations. As a matter of fact solvers are normally implemented on the basis of more or less explicit tradeoffs between:

1. Capabilities: the kinds of constraints they manage and under which assumptions; and
2. Performances: the adopted strategies, heuristics and optimizations.

If we accept the assumption that no single solver will be sufficient to accommodate the requirements of future applications, we should better take into account the possibility of synergically orchestrating the work of different solvers in order to efficiently overcome the limitations of each solver [9].

Along these lines, the main objective of this work is to implement and validate a case study of synergic cooperation between two solvers in the attempt to find out a generic approach that would help us in similar integration tasks. The realization of this combined solver guided us in the definition of a generic

master-slave architecture that can be easily and fruitfully applied in many interesting situations. The reason why we chose this bottom-up approach is because we believe that it can give us the instruments for deeply understanding and criticizing our results and, most notably, that it can ensure a fine-grained analysis of the impacts of important aspects, e.g., nondeterminism, in the cooperation of different solvers.

Many frameworks and architectures that explore the cooperative integration of constraint solvers are available in the literature (see, e.g., [6, 9]). They all share a common architectural outline made of the following three major components:

1. A top level of meta-resolution (namely a *meta-solver*);
2. A set of constraint solvers all grouped at an object level; and
3. An interface between the meta-solver and the object-level solvers.

Broadly speaking, our architecture can be thought as a contraction of this general architecture. The master-slave approach excludes the need of a layer of meta-resolution: one of the solvers is selected and promoted to the role of master. Such a solver does not work at the meta-level, rather it works at the object level and it is enriched with the capability of dispatching tasks to, and gathering results from, all other solvers. This may require some modifications to the selected master solver, but it saves us the trouble of implementing a solver from scratch when no meta-level functionality, e.g., expressing the dispatching policy in term of constraints, is requested.

The case study that motivated our work deals with the integration of two Java libraries JSetL [11] and JFD, both developed at the University of Parma[3], that implement respectively a constraint solver over sets and a constraint solver over finite integer domains. Both work well on their reference domains, but they are rather bad (or null) in all other situations: JSetL shows very poor performances on non-set variables; on the contrary, JFD shows good efficiency but it does not provide any notion of set or aggregate data type. Therefore, we decided to integrate the two solvers into an added-value Java constraint solver capable of exploiting:

1. The full expressive power of JSetL, with its inherent flexibility and generality; and
2. The efficiency of JFD in treating finite integer domains variables.

Integrating a constraint solver over general sets, namely CLP($\mathcal{SET}$) [5], and an efficient solver over finite domains, namely CLP($\mathcal{FD}$) [2], has been already explored in [4], where feasibility and usefulness of the approach are clearly pointed out. Since the constraint solving algorithms of JSetL and JFD are basically the same exploited in CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$), [4] also provides the theoretical basis of our current work. The work in [4], however, is deeply rooted in a logic programming framework, assuming a logic programming language as the common implementation language for the integrated solver. Moving to a more common programming context, such as that of Java, causes some implementation

[3] JSetL is available open source at `http://www.math.unipr.it/~gianfr/JSetL`. JFD is available on request from the authors of this paper.

decisions to become more evident—e.g., how to handle nondeterminism—and it requires explicit solutions for them to be provided. Moreover, differently from [4], we use the integration of the two specific solver as an opportunity to generalize the proposed solutions to a wider setting of cooperative constraint solving based on a master-slave architecture, possibly involving more than one slave solver.

The paper starts from the description of the generic master-slave architecture (next section) and its operational behaviour (Section 3). The instantiation of this architecture to our case study is postponed to Section 4. Then, our results are discussed together with other comparable results in Section 5. Section 6 draws some conclusion and presents some future work.

2 Master-Slave Constraint Solving

In this section we describe a generic master-slave architecture that we designed to support the integration of solvers with different capabilities. Each solver is responsible for a predefined set of constraints and the master, which is also a solver, is in charge of distributing tasks to, and gathering results from, the slaves. We work under the assumption that the distribution policy is static and based only on a fixed, a-priori mapping between kinds of constraints and solvers.

We start with a bunch of solvers to be integrated and we treat all of them, but the one selected to be the master, as black boxes. All solvers share a common high-level description as follows. We assume that each solver is equipped with a constraint store that holds its constraints. We also assume that all solvers regard each constraint as a collection of atomic constraints, possibly containing just a single atomic constraint, and interpreted logically as a conjunction of atomic constraints. We do not require all solvers to share the same set of atomic constraints. Typically, the sets of atomic constraints of the slaves are overlapping and some atomic constraint, e.g., the equality constraint, is in all sets.

As long as the interaction with the outer world is concerned, each solver is characterized in terms of:

1. The kinds of constraints that it can manage;
2. The way we can add constraints to its constraint store; and
3. The way the constraint store is made inspectable from the outside world in order to collect the results of computations.

The main task of each solver is to try to reduce any conjunction of atomic constraints to a simplified form that it cannot further simplify, i.e., that it considers *irreducible*. The detection of a failure (logically, the reduction to *false*) implies the unsatisfiability of the original constraint. Conversely, the ability to obtain the irreducible form may eventually imply the satisfiability of the original collection of constraints. In *complete* constraint solvers, the success of the reduction process allows to conclude the satisfiability of the original constraint and each of the obtained irreducible constraints represents a *solution* for the original constraint. On the contrary, *incomplete* constraint solvers usually return one irreducible constraint which has the same set of solutions as the original constraint, but

that is not guaranteed to be satisfiable (actually, determining its satisfiability can be potentially hard).

Our general master-slave architecture applies to both complete and incomplete solvers. However, the specific instantiation that we consider in our case study aims at developing a complete constraint solver in order to save the results of [4].

2.1 Selection of the Master

Given a set of solvers described in terms of the aforementioned characteristics, the first issue we have to tackle in order to enable cooperation among them regards assigning the role of master to a particular solver. The master represents the front end of the integrated solver and has the following duties:

1. It enables the programmer to express constraints;
2. It distributes tasks to slaves according to an allocation policy;
3. It gathers results from slaves; and
4. It provides a common view of results.

Differently from most of the approaches in the literature, our starting point is the selection of a solver to let it play the role of master. We are not concerned with realizing a new, special-purpose solver for the sole purpose of enabling cooperation. This would require the realization of a new solver from scratch and it would require to rewrite many algorithms and data structures that are already present in all solvers we are integrating. We would need a language for this new solver, a constraint store, and some rewriting procedures. All these features are already implemented and functioning in all solvers we are integrating.

Unfortunately, we cannot devise a single criterion for selecting the optimal solver to elect to the role of master. This choice is driven by the experience and by estimations of the required implementation effort. We can only enumerate some issues to consider for a good selection of the master:

1. The expressive power of the constraint languages involved, i.e., the master should provide a constraint language that subsumes the union of the languages of all slaves;
2. The performaces, i.e., slaves should perform better than the master in their reference domains;
3. The support for constraint programming abstractions, e.g., logical variables and nondeterminism;
4. The possibility of extending the selected solver to integrate master-specific procedures, e.g., a constraint dispatching procedure and its result processing counterpart; and
5. The public functionalities provided to programmers.

Reasonably, if we can access a solver with a good expressive power and such that it can cover most of the constraints of all other solvers, we have a good candidate for the role of master. Taking into account our case study, the constraint language of JSetL is sufficiently expressive and therefore we selected JSetL to play the role of master, after some minor modifications described in Section 4.

2.2 The Master's Solving Process

Once the role of master is assigned, the solving process works as follows. First, all constraints are loaded into the master's constraint store, then each constraint in the store is analyzed separately. For each constraint, the master performs one of the following mutually exclusive actions:

1. It solves the constraint with no help from slaves;
2. It delegates the resolution of the constraint to a bunch of slaves; or
3. It forwards the constraint to a bunch of slaves to let them exploit it for current or future use, and then it solves it.

The choice of which action to perform and when is a core part of the design of the master and it may heavily influence the overall speed-up that the integrated solver gains over single solvers.

Previous actions progress the master's constraint store towards a solved form and they are repeated until the solving process terminates successfully or until it fails. The process terminates successfully when no further constraint can be reduced or allocated to slaves and when all results are gathered from the constraint stores of slaves. On the contrary, it fails when the master, or any slave, detects an inconsistency and no further nondeterministic choices are left open.

2.3 Communications between the Master and the Slaves

In our architecture, the allocation of constraints to slaves is performed by means of a static, a-priori policy. This policy is chosen because it is very easy to implement and because it demands no considerable amount of computation compared to reasonable constraint solving processes.

Given an n-ary atomic constraint $C = op(t_1, \ldots, t_n)$, we first use the operator op and its arity n to allocate C to a group of solvers. If this is not sufficient to identify the needed solvers, we consider the data types of the t_i and we use this information to finally allocate C.

In general, given an atomic constraint C, and given S, the pre-computed set of slaves that can handle C, the allocation of C to slaves is performed by means of one of the four mutually exclusive cases:

1. If C is a constraint of the master and $S = \emptyset$, then the master solves it;
2. If C is not a constraint of the master and $S \neq \emptyset$, then C is posted to all slaves in S;
3. If C is a constraint of the master and $S \neq \emptyset$, then the master posts C to all slaves in S and then solves it; or
4. If C is not a constraint of the master and $S = \emptyset$, the allocation fails, i.e., the constraint is unknown.

The first option (that we call *MASTER*) is selected when the master can manage the constraint at hand and no slave needs it in for future computations. In our case study, this is the case of all set constraints, such as $op_{\subset}(x, y)$ which enforces $x \subset y$: JSetL reduces them with no help from JFD. The second option (called *SLAVE*) is the opposite, i.e., the master has no means to exploit the constraint and it needs a group of slaves to handle it. In our case study, this is the case of

arithmetic constraints like $op_+(z, x, y)$, which enforces $z = x + y$: JSetL cannot efficiently handle such a constraint and therefore it delegates it to JFD. The third option ($BOTH$) is selected when the master can handle the constraint on its own, but it knows that slaves may eventually need it. In our case study, this is the case of equality constraints like $op_=(x, y)$, which imposes $x = y$: if x and y are either free variables or integer constants, both JSetL and JFD need this constraint to make sure that no information contained in the original constraint store is lost during the solving process. The last of these four options ($UNKNOWN$) indicates an error in the integration of solvers and it should never occur.

The allocation of constraints to slaves is only the first part of the process the master performs to solve its constraint store. The second part consists of the master gathering the results from slaves and reconciling them into its constraint store. In our architecture, we do not take into account the possibility for a slave to generate constraints that are directly passed to other slaves.

The master is interested only in the final outcome of the computation of slaves, i.e., the irreducible constraints, involving variables known from the master, that the slaves leave in their constraint store after a complete reduction of any allocated constraint. Hence, the computation of slaves is immaterial from the point of view of the master.

Generally speaking, a slave should provide back to the master any constraint left in its constraint store that is available also in the language of the master. This would guarantee that no information is lost during the resolution and that each solver is free to choose the best approach for solving its constraints.

3 General Constraint Solving Procedures

The procedure the master performs for solving its constraint store is described in Algorithm 1, where *Store* is the conjunction of all constraints in the initial constraint store, conventionally ended with a *true*:

$$Store = C_1 \wedge C_2 \wedge \ldots \wedge true$$

This procedure moves from one atomic constraint C_i to the next C_{i+1} and solves each of them until *Store* is irreducible. The procedure *reset* sets the current constraint to point to the first atomic constraint in *Store*. Then, procedure *step* eventually allocates the current constraint to slaves or tries to solve it. Finally, function *is_final_form* tests if the reduction process is complete.

The master constraint solving procedure assumes that each slave provides a few methods to modify or inspect its constraint store, as follows.

3.1 Slave Interface

The following methods represent the interface that slave solvers provide to the outer world:

1. *add(C)*, where C is a slave constraint to be added to the constraint store of the slave;

Algorithm 1 The master's procedure for constraint solving

```
procedure master_solve(Constraint Store)
    repeat
        reset(Store);
        step(Store)
    until is_final_form(Store)
end procedure;
```

2. *get_constraints()* that returns the conjunction of constraints that are present in the store of the slave;
3. *solve(B)*, where B is a value from an enumerative type used to specify which solving process is requested.

The type of the argument of *solve* contains, at least, the value *REDUCE* and, possibly, the value *LABEL_ALL*. The first is used to ask the slave to process its constraint store until an irreducible form is found. The second asks the slave to label nondeterministically all variables in its constraint store. This process causes equality constraints $op_{=}(x, v)$ to be added to the constraint store of the slave to force $x = v$, where v is a value in the domain of variable x.

The previous methods are sufficient to let the master delegate constraints to slaves. However we need some glue functions to make a perfect match between the master's and each slave's view of the constraint store. Such glue functions have the following responsibilities:

1. They *translate* master's constraints to the corresponding constraints for each slave and vice versa; and
2. They *filter* constraints to determine which of them should be passed from the master to the slaves and which should not, and vice-versa.

In details, such glue functions are:

1. *master_to_slave(Slave,C)* that translates a master constraint C to the corresponding slave constraint, or to *true* if C is not a constraint to be sent to slave *Slave*;
2. *slave_to_master(Slave,D)* acts as the opposite of *master_to_slave* and it translates a slave constraint D to the corresponding master constraint;
3. *which_type(C)* that allows C to be classified as belonging to one of the four types of constraints mentioned above (*MASTER*, *SLAVE*, *BOTH* and *UNKNOWN*); and
4. *which_slave(C)* that maps C to a possibly empty set of slaves that can handle it.

Procedures *master_to_slave* and *slave_to_master* take also care of mapping master's variables to slaves' counterparts, and vice versa.

Few other procedures are provided by the slave interface to handle nondeterminism and will be discussed in subsection 3.3.

3.2 The Master's Constraint Solving Procedure

The procedure *step*—see Algorithm 2—is the core of the solving process of the master. It embodies all actions necessary to solve each atomic constraint in

Algorithm 2 The core of the solving procedure

```
procedure step(Constraint Store)
    Type T;
    Constraint C, D;
    Set Slaves, Selected_Slaves;

    C ← extract(Store);
    Selected_Slaves ← ∅;

    while C ≠ true do
        T ← which_type(C);
        Slaves ← which_slave(C);
        Selected_Slaves ← Selected_Slaves ∪ Slaves;

        if T = SLAVE or T = BOTH then
            for all Slave ∈ Slaves do
                Slave.add(master_to_slave(Slave, C))
            end for
        end if;

        if T = BOTH or T = MASTER then
            handle_constraint(Store, C)
        end if;

        C ← extract(Store);
    end while;

    for all Slave ∈ Selected_Slaves do
        Slave.solve(REDUCE);

        slave_try_next(Slave, Store)
    end for;
end procedure;
```

Store, either by using its own procedures or by interacting with the slave solvers. Eventually *step* modifies the constraint store *Store* as a side effect.

The invocation *extract(Store)* removes the current atomic constraint C from *Store* and returns it, moving the current atomic constraint to point to the next atomic constraint in *Store*. Then *step* has the task of deciding whether the selected constraint C must be allocated to some slave solver or not. More precisely, if the master decides—invoking procedures *which_type* and *which_slave*—to solve an atomic constraint on its own, it exploits the procedure *handle_constraint*; after its complete execution, either the atomic constraint is in an irreducible form, or it will be further processed at next step. On the contrary, if the master decides to delegate an atomic constraint to a group of slaves it posts the (translated version of the) selected constraint C to each slave in the group. A specific instance of procedure *handle_constraint* will be shown in next section, when presenting our case study.

After the whole constraint store of the master solver has been examined, the procedure *step* requests to the slaves to solve the constraints possibly posted to them. This is obtained by invoking the procedure *solve* for each slave solver in the set *Selected_Slaves*. The subsequent invocation of the procedure *slave_try_next*—see Algorithm 3—allows the master to get the results from the slave solver and to add them to its constraint store. This procedure also takes care of the possible nondeterminism occurring in the slave constraint solving.

Algorithm 3 The procedure used to explore nondeterministic choices left open by slaves

```
procedure slave_try_next(Solver Slave, Constraint Store)
    Constraint D;

    either
        D ← slave_to_master(Slave, Slave.get_constraints());

        if D = false then
            fail
        else
            insert(Store, D);
        end if;
    orelse                                          ▷ try next nondeterministic alternative
        Slave.next_solution();

        slave_try_next(Slave, Store)
    end either
end procedure;
```

3.3 Handling Nondeterminism

In our architecture we assume that both the master and the slave solvers can be sources of nondeterminism and we need to take care of both possibilities in a coherent way.

The fact that slave solvers can be nondeterministic requires that the slave interface modules further provide the following methods:

1. *next_solution()* to explore the next nondeterministic alternative that a previous call to *solve* might have left open.
2. *save()* that returns a snapshot of the current state of computation of a slave; and
3. *restore()* that restores a previously saved state.

save() and *restore()* will allow saving and restoring of the constraint store of the slave solvers whenever a choice point is detected in the rewriting process of a constraint of the master.

To be able to explore nondeterministic choices possibly left open by the slave solvers the master uses the procedure *slave_try_next*. This procedure uses the construct *either-orelse* to manage the nondeterminism that a previous call to procedure *solve* might have created. The idea of this construct (see [1]) is that the *either* branch is explored on first and just before executing it, the complete state of computation is pushed onto a backtracking stack. Subsequent branches of nondeterminism are triggered by a call to *fail* and they will execute the first unexplored *orelse* branch until no choices are left open. Before executing any *orelse* branch, the backtracking stack is used to restore the computation state, so that all branches start from the same state.

Furthermore, the procedures that handle the backtracking stack in the master solver need to be slightly modified. As a matter of fact, our use of the construct *either-orelse* is based on the assumption that the *either* branch can save the complete computation state of all slaves, and that the *orelse* branch can restore it. In details, whenever the master has to add a choice point, saving its computation state (in particular, its constraint store) onto its backtracking stack, it

also save the state of all the slave solver, as obtained by invoking the relevant *save* methods. Whenever the master solver backtracks to an unexplored choice point, it pops the computation state from the stack, including the saved states of the slaves, and it automatically forces the latter to turn back to their saved state, by invoking the relevant *restore* methods.

It is worth noting that methods *save* and *restore* are not normally explicitly considered when dealing with cooperative constraint solvers and the work is under the (implicit) assumption that solvers are implemented in a programming language that supports nondeterminism. Under this assumption, any change in the computation state of any slave occurred since last choice point is automatically undone when the master rolls back (this is the case of the Prolog implementation of CLP($\mathcal{SET} + \mathcal{FD}$) [4]). This is obviously false in a programming language like Java that do not support nondeterminism and we need to explicitly take care of it.

4 A Case Study: JSetL+JFD

In this section we describe the integration of two constraint solvers, namely JSetL and JFD, as an instance of the generic master-slave architecture described above. In such instance we assume that one single slave is used and that the implementation language, i.e., Java, does not support nondeterminism. The presented technique can be easily generalized to the case of many slaves.

JSetL is a Java library that endows Java with a number of facilities to support general purpose declarative programming like those usually found in *Constraint Logic Programming* (*CLP*) languages. In particular, JSetL provides logical variables, unification, list and set data structures, constraint solving and nondeterminism, like those supported by CLP($\mathcal{SET}$) [5].

As noted in [11], computation efficiency is not a primary design requirement of JSetL. JSetL is mainly conceived as a tool for rapid prototyping, where easiness of program development and program understanding prevail over efficiency. Moreover, JSetL is meant to provide researchers with a study tool for all issues related to set constraint solving. This is the reason why JSetL provides rich and efficient techniques for managing set variables and constraints, while it relies on basic *generate & test* constraint solving for the case of scalar variables.

The Java Finite Domains (JFD) is a library that provides the programmer with an object-oriented view of well-known constraint solving techniques for variables with finite domains. In details, it implements the CLP($\mathcal{FD}$) language for integers [2] and it provides some facilities for inspecting the computation of the solver and for integrating it with third party Java objects.

The language that JSetL provides for the definition of constraints is a superset of JFD's one and therefore JSetL can play the role of master, while JFD becomes the one and only slave. This choice requires some minor modifications to JSetL, but it saves us the trouble of realizing a new meta-solver from scratch. The result of this integration, namely JSetL+JFD, has the best of both worlds:

Algorithm 4 The *handle_constraint* procedure of JSetL+JFD

```
procedure handle_constraint(Constraint Store, Constraint C)
    if C = op∈(o1,o2) then
        member(Store, C)
    else if C = op∉(o1,o2) then
        equals(Store, C)
    else if ... then
    else if C = next then
        slave_try_next(Store, C)                    ▷ Unroll nondeterministic choice
    end if
end procedure
```

it retains the expressive power and flexibility of JSetL, while allowing an efficient finite-domains processing when needed.

The synergic cooperation of JSetL and JFD is completely transparent to the programmer because he/she deals only with the API, i.e., the language, that JSetL provides. The presence of JFD as a back-end slave is perceived only at runtime because of the improvement of performances that it brings when dealing with finite domains variables: whenever JSetL detects constraints that JFD can handle efficiently, e.g., membership constraints over finite sets of integers, it delegates their resolution to JFD.

Algorithm 4 shows (part of) the instance of procedure *handle_constraint* in the case of JSetL+JFD. *handle_constraint* implements the constraint handling for master's constraints, using a dedicated procedure for each different type of constraint. One of these procedures, namely *member*, dedicated to the management of $op_{\in}(x, y)$ constraints, is shown in Algorithm 5.

Following [4], we want our integrated solver JSetL+JFD to preserve the completeness property that characterizes CLP($\mathcal{SET}$) and JSetL. To this end, we force the slave solver JFD to label variables before returning its results to the master. To obtain this, the call *Slave.solve(REDUCE)* of Algorithm 2 is replaced by the call:

jfd.solve(LABEL_ALL)

Forcing labeling, allows the slave solver JFD to communicate back to the master solver JSetL only equality constraints, provided JFD has received enough information to associate a domain to each variable it has to deal with.

The main differences with respect to the general scheme of Section 3 is the way nondeterminism is handled within the master solver. The problem is how the abstract construct *either-orelse* can be rendered concretely using JSetL facilities for nondeterminism handling.

JSetL allows to express nondeterminism by explicitly creating choice points through the use of the method *add_choice_point*, and to backtrack to one of the open choice points using the control information stored in the constraint itself and obtainable through the method *get_alternative* of the class *Constraint*. Nondeterminism in JSetL, however, is confined to constraint solving, i.e., the aforementioned methods can be only used within constraint solving procedures (either predefined or user defined).

Algorithm 5 The *member* procedure of JSetL+JFD.

```
procedure member(Constraint Store, Constraint C)
    if C = op∈(o1, ∅) then
        fail
    else if C = op∈(o1, X) then
        insert(Store, op=(X, {o1 | N}))
    else if C = op∈(o1, {o2 | s}) then
        if C.get_alternative() = 0 then
            add_choice_point(C);                    ▷ Save nondeterministic choice

            insert(Store, op=(o1, o2))
        else
            insert(Store, op∈(o1, s))
        end if
    else if ... then
    end if
end procedure
```

As an example, consider the implementation of the constraint rewriting procedure *member* shown in Algorithm 5. Such a procedure is a source of nondeterminism because it addresses two possibilities when the constraint it handles is $C = op_{\in}(o_1, \{o_2 \mid s\})$:

$$o_1 = o_2 \vee o_1 \in s$$

The nondeterministic alternatives are implemented as the different alternatives of nested *if-else* statements, which are selected using the control information obtained through the method *get_alternative*. Each *if-else* alternative, but the last one, creates a choice point and adds it to the backtracking stack by invoking the method *add_choice_point*. Then the remaining code of the *if-else* alternative implements the proper constraint rewriting.

Algorithm 6 shows the refined versions of the procedure *slave_try_next* for JSetL+JFD. To implement the nondeterministic computation of Algorithm 3 we introduce a new constraint, called *next*, and we turn the procedure *slave_try_next* into a constraint rewriting procedure called to handle the new constraint *next*. The way this constraint is processed is exactly the same shown in procedure *slave_try_next* of Algorithm 3 except that, after calling *next_solution* of the slave, it inserts again the constraint *next* in its current constraint store *Store*. Some minor modifications of the procedure *handle_constraint* are also required to account for the new constraint *next*—see Algorithm 4. Similarly, the invocation *slave_try_next(Slave,Store)* in the procedure *step* is replaced by the invocation *insert(Store,next)* that adds the constraint *next* to the constraint store of the master.

Furthermore we assume that the procedures that handle the backtracking stack in JSetL are slightly modified to allow the JFD state to be saved/restored to/from the JSetL backtracking stack, according to the technique described in Section 3.3. This requires that JFD is equipped with the *save* and *restore* procedures described in the general case. Using this technique, the implementation of nondeterministic constraint rewriting procedures, such as member, remains completely unchanged with respect to the case of a stand-alone master.

Algorithm 6 The *slave_try_next* procedure of JSetL+JFD

```
procedure slave_try_next(Constraint Store, Constraint C)
    Constraint D;

    if C.get_alternative() = 0 then
        add_choice_point(next);                          ▷ save computation state

        D ← slave_to_master(jfd, jfd.get_constraints());

        if D = false then
            fail
        else
            insert(Store, D)
        end if
    else
        jfd.next_solution();                             ▷ try next nondeterministic alternative
        insert(Store, next)
    end if
end procedure;
```

5 Related Work

The possibility of synergically orchestrating the work of different solvers in order to efficiently overcome the limitations of each single solver is normally explored in the area of *cooperative constraint solving*.

The literature on cooperative constraint solving tries to capture this idea by designing a *composite constraint solver* that provides a unified view of the languages and the features that atomic constraint solvers offer. The composite solver has the responsibilities of all cooperation-related tasks, e.g., providing a unified constraint store, managing the communication and task breakdown between atomic solvers and translating constraints back and forth atomic solvers. Then, the language of this composite solver is likely to enrich the combination of the languages of atomic solvers with meta-level features that allow expressing strategies and heuristics for cooperation.

The literature on this subject is rich and a number of architectures have been proposed [3, 6, 9, 13]. For example, Hofstedt [9] proposes a very general, yet quite complex, architecture capable of accommodating different solvers on-the-fly in order to solve complex problems that no single solver could solve efficiently. Moreover, a substantial number of theoretical works are devoted to the study of necessary conditions for supporting efficient cooperation between different solvers [9, 10].

Many of the approaches that the literature of cooperative constraint solving proposes could be possibly applied to the case study that we consider in this work, i.e., the integration of a set solver with a finite domains solver. Nevertheless, such approaches seem to overkill the problem with very generic, yet very redundant, architectures that are not easily implementable. Moreover, many details regarding the internal mechanisms used for cooperation are not specified and it is not clear whether they may put little problems in practice, or not. Finally, most of the proposals found in the literature assume that composite solvers and atomic solvers are all implemented in a (single) logic language. This hidden

hypothesis has a strong impact when implementing solvers in an object-oriented language like Java because most of the issues related to nondeterminism must be treated explicitly.

A related research area uses the synergic cooperation of constraint-solvers to deal with *Distributed Constraint Satisfaction Problems* (see, e.g., [12]), i.e., constraint satisfaction problems that can be solved by means of an efficient distribution of tasks to solver agents. Normally, all agents have the same characteristics and a single agent would be able to efficiently solve the whole problem. The distribution is meant to boost performances and it does not deal with different capabilities agents may have. This is significantly different from the main objective of our work that targets the integration of solvers with different capabilities. We need the cooperation of solvers because no single solver is capable of efficiently addressing the whole complexity of the problem at hand; not because a single solver is not quick enough for our purposes.

The basic motivation of our work lies in the working assumption that we are not probably going to deliver *the* perfect constraint solver capable of meeting all requirements of future applications. This lead us to the study of cooperative solvers as a means to overcome the limitations of each solver. For the sake of completeness, we shall mention that the same working hypothesis led other researcher towards the realization of tools for the ad-hoc construction of application-specific solvers. For example, *Constraint Handling Rules* (*CHR*s) [7] is a language for the realization (from scratch) of the right solver for the problem at hand.

6 Conclusions and Future Work

This paper gathers ideas and results obtained from a very practical experience: the integration of JSetL and JFD into a value-added Java constraint solver that can fully exploit the expressive power of JSetL, e.g., set variables and constraints, and the efficiency of JFD in treating finite integer domains variables. The outcome of this experience is twofold:

1. A generic master-slave architecture for the integration of constraint solvers with different capabilities has been identified and studied; and
2. The JSetL+JFD Java library for constraint solving is implemented.

Future work is in the direction of improving the generic architecture that we defined by relaxing some design choices that may prohibit the use of it in many interesting situations. For example, we are in the process of changing the policy for allocation of constraints to slaves in order to make it more flexible and dynamic.

From the point of view of the integration of set and finite domains constraints, we envisage the possibility of allowing the manipulation of finite domains by means of JSetL high-level set operations. The importance of this feature has been emphasized by various authors (see, e.g., [8]). In practice this would be achieved by implementing the constraint rewriting procedures defined in [4].

Other interesting directions of future development are inherent to the choice of forcing slaves, when possible, to label their variables before returning their results to the master, in order to gain completeness. This labeling, however, would probably cause inefficiencies. A first cut to such inefficiencies could be achieved by carefully deciding when to ask slaves to label variables and which variables they should label. Obviously, the more we delay the labeling request, the better. Then, a possibly larger improvement of efficiency could be achieved by dropping the completeness requirement to let slaves deciding the best way to process their constraint stores.

Acknowledgments The work is partially supported by MIUR project "Constraints and preferences as a unifying formalism for system analysis and solution of real-life problems".

References

1. K. R. Apt, J. Brunekreef, V. Partington and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM TOPLAS*, 20(5), 1014–1066, 1998.
2. P. Codognet and D. Diaz. Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3), 185–226, 1996.
3. M. Correia, P. Barahona and F. Azevedo. CaSPER: A programming environment for development and integration of constraint solvers. *Procs.* 1^{st} *International Workshop on Constraint Programming Beyond Finite Integer Domains* (*BeyondFD'05*), 59–73, 2005.
4. A. Dal Palù, A. Dovier, E. Pontelli and G. Rossi. Integrating finite domain constraints and CLP with sets. *Procs.* 5^{th} *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (*PPDP'03*), ACM Press, 219–229, 2003.
5. A. Dovier, C. Piazza, E. Pontelli and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.
6. S. Frank, P. Hofstedt and P. R. Mai. A flexible meta-solver framework for constraint solver collaboration. *Procs.* 26^{th} *German Conference on Artificial Intelligence*, *KI'2003*, LNCS 2821, Springer-Verlag, 2003.
7. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3), 1998.
8. M. Gavanelli, E. Lamma, P. Mello and M. Milano. Dealing with incomplete knowledge on CLP(FD) variable domains, In *ACM TOPLAS*, 27(2):236–263, 2005.
9. P. Hofstedt. Cooperating constraint solvers. *Procs. International Conference on Principle and Practice of Constraint Programming*, LNCS 1894, Springer-Verlag, 520–524, 2000.
10. E. Monfroy and C. Castro. Basic components for constraint solver cooperations. *Procs. ACM SAC 2003*, ACM Press, 367–374, 2003.
11. G. Rossi, E. Panegai and E. Poleo. JSetL: A Java library for supporting declarative programming in Java. *Software-Practice & Experience*, in print, 2006.
12. A. Petcu, B. Faltings and D. Parkes. MDPOP: Faithful distributed implementation of efficient social choice problems. *Procs. Autonomous Agents and Multiagent Systems* (*AAMAS'06*), 2006.
13. P. Zoeteweij and F. Arbab. A component-based parallel constraint solver. *Procs. of COORDINATION 2004*, LNCS 2949, 307–322, Springer-Verlag, 2004.

Checking UML Model Consistency

A. Baruzzo[1] and M. Comini[1]

Dipartimento di Matematica e Informatica (DIMI), University of Udine,
Via delle Scienze 206, 33100 Udine, Italy.

Abstract UML is nowadays a de-facto standard for design and development of (object-oriented) software. With version 2.0 UML has achieved a more precise formal semantics. The same happened to OCL, a specification language which is an integral part of UML that allows to embed software contracts in the model.
In this work we propose an approach for a static verification of consistency of UML models which relies on OCL constraints.
Many approaches for model validation and verification rely on generation of suitable code which dynamically (i.e., at run-time) checks the validity of OCL constraints. This approach has several drawbacks. For example, it cannot generally guarantee that a constraint will never be violated, unless an infinite number of tests is performed. Also the generation of just a *significative* finite subset is not so feasible because, on one hand, a considerable manual effort is needed even to produce a single test scenario and, on the other hand, test-case generation is well-known to be a hard problem.
On the other hand, static approaches based on model checking suffer of the state explosion problem and thus cannot scale to real system sizes.
In this paper we propose a static verification technique that, by using OCL constraints together with class diagrams, certifies that the dynamic part of the model is satisfied. This encompasses the weaknesses of the other mentioned methods as it does not require users to build test scenarios and also performs the verification of *all* system properties at the same time.

1 Introduction

Finding program bugs is a long-standing problem in software construction. There has been considerable theoretical research activities and published results starting from the mid-nineties about using formal specifications to help the debugging phase. All these theoretical efforts have produced also relevant practical results. Indeed, the software engineering community has nowadays accepted the fact that the specification of various kinds of pieces of software is not only a topic of theoretical interest but also one of practical importance. A steadily increasing number of papers dealing with the concepts and the practical use of assertions in general have been published during the last few years (amongst all [21]). The basic foundations have been laid by Bertrand Meyer with his concept of *Design by Contract* (DbC) as realized in the Eiffel language (see [16,15]). This approach

has then rapidly spread to other languages, for instance there emerged lot of support for assertions for Java (e.g. Jass, *etc.*) and C++ (e.g. the iContract tool, *etc.*). Mostly important the Unified Modeling Language (UML) has now, as one of its integral parts, the Object Constraint Language (OCL), which has its root in the Syntropy method. With OCL we can naturally implement the contract mechanism of Eiffel.

Many approaches for system debugging and for model validation/verification rely on generation of suitable code which dynamically (i.e., at run-time) checks the validity of OCL constraints (i.e., the compliance of the system status w.r.t. the constraint). This approach has several drawbacks. For example, it undoubtedly slows down performance and can potentially alter the behavior (if the inserted code has by mistake side effects). But most of all it does not ensure to reveal a bug unless the specific run of the system effectively enters a state which is not compliant w.r.t. the specification. One can argue that not all runs are actually needed to manifest an error, since most symptoms (wrong traces) are caused by the same error. However also the generation of just a *significative* finite subset of the possible runs is not so feasible because, on one hand, a considerable manual effort is needed even to produce a single test scenario and, on the other hand, test-case generation is well-known to be a hard problem.

On the contrary *static* (semantics-based) tools could guarantee that *any* run will be compliant w.r.t. the specification, without even adding extra overhead. The problem with this approach is that it is in general (well-known to be) undecidable and, in any case, much difficult to tackle.

Many researchers are proposing static approaches based on Model Checking, but this suffers of the state explosion problem and thus (while suitable for protocols and small hardware systems) cannot scale to real software system sizes. Moreover there is also an inherent limit to verification of a single specific property of the system at a time.

This paper is motivated by the fact that we believe we can attack the undecidability of the static approach by using Abstract Interpretation techniques [9,10,11,12,8,2]. Abstract Interpretation is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems. Abstract interpretation can be applied to the systematic construction of methods and effective algorithms to approximate undecidable or very complex problems in computer science such as the semantics, the proof, the static analysis, the verification, the safety and the security of software or hardware computer systems. In particular, abstract interpretation-based static analysis, which automatically infers dynamic properties of computer systems, has been very successful these last years to automatically verify complex properties of real-time, safety critical, embedded systems.

We already had plenty of experience in Debugging and Verification of Declarative Languages where, by using Abstract Interpretation techniques, we could develop effective semantic-based tools [7,3,6,5,4,1]. The nice of this approach is that it can discover bugs even in absence of symptoms. Moreover it does not

need of a complete system to work, since we can (must) use, in place of missing components, their specification to diagnose existing parts.

This could be the case also for real systems providing UML diagrams with OCL specifications. However, given the level of complexity of such systems, it can easily be the case that the UML diagram *in itself* is not consistent. This would render the use of (complex), either static or dynamic, code diagnosis tools completely pointless. Hence it is important to have a tool to statically check the consistency of an UML model to achieve a good design *even before* the implementation starts. It can help further debugging stages and it is important in itself for Model Validation.

This is why now we propose a conceptual framework for static verification of UML model consistency.

The paper is structured as follows: in Section 2 we introduce some concepts about assertions in UML with OCL. In Section 3 we present our Conceptual Framework with an example to show how it works. Then in Section 4 we discuss about its applicability for development of software verification tools.

2 Assertions in the Software Engineering Practice

2.1 Design by Contract

In this section we will look how some of the concepts introduced above can be transferred to software systems in practice. Design by Contract (DbC) [17] is inspired by formal approaches embodied in specification languages such as Z, VDM, etc. Bertrand Meyer has coined the concept of DbC to denote a software development style which (1) emphasizes the importance of formal specifications, and (2) interleaves them with actual code. DbC is a systematic method of assertion usage and interpretation introduced as a standard feature of the Eiffel language [16]. Without it, no trial would have ever been made to provide a similar mechanism in other languages and, by no means, would we have discussion papers like this and the ones mentioned in the references.

Software contracts have been invented to capture mutual obligations and benefits among classes, as they are e.g. needed in design patterns, where each of the involved classes is expected to exhibit a "proper" behavior [13,14]. A software contract is the specification of the behavior of a class and its associated methods. The contract outlines the responsibilities of both the caller and the method being called. Failure to meet any of the responsibilities stated in the contract results in a break of the contract, and indicates the existence of a bug somewhere in the design, in the implementation, in both of them, or - one must not forget this possibility in earlier project phases - in the assertions themselves. Software contracts can be completely specified through the use of preconditions, postconditions, and class invariants in object-oriented software. DbC views software construction as based on contracts between clients (callers) and suppliers (routines). Each party expects some benefits from the contract, and accepts some obligations in return. As in human affairs, the contract document

spells out these mutual benefits and obligations and protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope. The DbC paradigm is as follow:

> The client's obligation is to call a method only in a program state where both the class invariant and the method's precondition hold. The method, in return, guarantees that the work specified in the postcondition has been done, and the class invariant is still respected.

A precondition violation is a manifestation of an error in the client, while a postcondition failure is a manifestation of a bug in the (implementation of the) supplier, which does not fulfill its promise (Note: The phrase "An assertion fails" in real life means just the opposite: the assertion did its job well, because it has found a bug). For this reason, in order to call a method, the client should verify only its preconditions. If the preconditions are satisfied, it should take for grant the postcondition after the termination of the method execution. The supplier, vice versa, should check the postconditions in order to guarantee its part of the contract, but under no circumstances shall the body of the method ever test for its preconditions. Under the *Non Redundancy Principle* [17], hence, the DbC encourage the developer to "check less and get more". DbC is, in this respect, the opposite of defensive programming, which recommends to protect every software module by as many checks as possible. This may result in redundancy and makes it also difficult to precisely assign responsibilities among modules.

2.2 UML and OCL

In the last years, many efforts has been spent to make the UML language more precise. Since its beginning, UML was conceived as a standard graphical language suitable to support the development of object-oriented systems. A clear intent in the UML design was the unification of the previous modeling languages, which all provided different notations for the same concepts. The standardization process was made by the Object Management Group (OMG), involving both the industry and the academia worlds. The results of this process was a relatively stable language, with an informal semantics. This level of definition was sufficient for sketching analysis and design models. However, when the model needed to be elaborated by automated tools for validation and verification purposes, the lack of a more formal foundation was immediately recognized. Because UML focused primarily on the diagrammatic elements and gave meaning to those elements through English text, a constraint language was added to the specification, in order to provide a more precise definition of UML meta-model. That language was the Object Constraint Language (OCL) [22], initially developed in 1995 at IBM. OCL allows the integration of both well-formedness rules and assertions (i.e., preconditions, postconditions, invariants) in UML models. The former are useful to validate especially the syntax of a UML model, whereas the latter can be exploited to verify the conceptual constraints.

Preconditions and postconditions provide a mechanism to specify the properties required before and after the execution of an operation, respectively. They do not specify how that operation internally works. The recent development of version 2 for both OCL [18] and UML [19] is a breakthrough in order to completely define the semantics of a method in an object-oriented system. In these last versions, it is possible to define a behavior specification in OCL for any query operation (an operation without side-effects).

Following [20], now we summarize the relevant concepts about UML diagrams, the OCL specification language and the action semantics. For the sake of simplicity, here we present just a summary of the most important results.

In this work we use OCL as specification language to define software contracts such as method preconditions and postconditions, class invariants, and assertions in general. Hence we now define an object model $\mathcal{M}$ that contains the UML elements relevant for this task. Because preconditions, postconditions and invariants are defined typically for class diagram elements (i.e., class attributes and methods), we consider for the moment only the static structure of a UML model. A (static) object model $\mathcal{M}$ can be represented by the following tuple:

$$\mathcal{M} = \langle CLASS, ATT, OP, ASSOC, \preceq, associates, roles, multiplicities \rangle$$

where *CLASS* is a set of UML classes, *ATT* is a set of attributes, OP is a set of operations, ASSOC is a set of associations, $\preceq$ is a generalization hierarchy over classes, and *associations*, *roles*, and *multiplicities* are functions that give for each $as \in ASSOC$ its dedicated classes, classes' role names, and multiplicities, respectively.

For an object model $\mathcal{M}$ providing a set of types $T_{\mathcal{M}}$, a relation $\leq$ on types reflecting the type hierarchy, and a set of operations $\Omega_{\mathcal{M}}$, the definition of OCL expressions is based upon the signature:

$$\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$$

By using this signature, we can define the OCL expressions syntax in the following way. Let $\texttt{Var} = \{\texttt{Var}_t\}_{t \in T_M}$ be a family of variable sets where each variable set is indexed by a type t. An expression over the signature $\texttt{Expr}_{\mathcal{M}}$ is given by a set $\texttt{Expr} = \{\texttt{Expr}_t\}_{t \in T_M}$ and a function $\texttt{free} : \texttt{Expr} \rightarrow \mathcal{F}(\texttt{Var})$ defined as follow.

- If $v \in \texttt{Var}_t$ then $v \in \texttt{Expr}_t$ and $\texttt{free}(v) := \{v\}$.
- If $v \in \texttt{Var}_{t_1}, e_1 \in \texttt{Expr}_{t_1}, e_2 \in \texttt{Expr}_{t_2}$ then $\texttt{let } \mathbf{v} = \mathbf{e_1} \textbf{ in } \mathbf{e_2} \in \texttt{Expr}_{t_2}$ and $\texttt{free}(\texttt{let } v = e_1 \textit{ in } e_2) := \texttt{free}(v) - \{v\}$.
- If $\omega : t_1 \text{ x } ... \text{ x } t_n \rightarrow t \in \Omega_{\mathcal{M}}$ and $e_i \in \texttt{Expr}_{t_i}$ for all $i = 1, ..., n$ then $\omega(\mathbf{e_1}, ..., \mathbf{e_n}) \in \texttt{Expr}_t$ and $\texttt{free}(\omega(e_1, ..., e_n)) := \texttt{free}(e_1) \cup ... \cup \texttt{free}(e_n)$.
- If $e_1 \in \texttt{Expr}_{Boolean}$ and $e_2, e_3 \in \texttt{Expr}_t$ then $\texttt{if } \mathbf{e_1} \texttt{ then } \mathbf{e_2} \texttt{ else } \mathbf{e_3} \texttt{ endif} \in \texttt{Expr}_t$ and $\texttt{free}(\text{ if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif}) := \texttt{free}(e_1) \cup \texttt{free}(e_2) \cup \texttt{free}(e_3)$.
- If $e \in \texttt{Expr}_t$ and $t' \leq t$ or $t \leq t'$ then $(\mathbf{e} \texttt{ asType } \mathbf{t'}) \in \texttt{Expr}_{t'}$,$(\mathbf{e} \texttt{ isType } \mathbf{t'}) \in \texttt{Expr}_{Boolean}$, $(\mathbf{e} \texttt{ isKindOf } \mathbf{t'}) \in \texttt{Expr}_{Boolean}$ and $\texttt{free}((e \texttt{asType} t')) := \texttt{free}(e)$, $\texttt{free}((e \texttt{ isTypeOf } t')) := \texttt{free}(e), \texttt{free}((e \texttt{ isKindOf } t')) := \texttt{free}(e)$.

– If $\mathbf{e_1} \rightarrow \mathbf{iterate}(\mathbf{v_1}; \mathbf{v_2} = \mathbf{e_2} | \mathbf{e_3}) \in \mathtt{Expr}_{t_2}$ and $\mathtt{free}(e_1 \rightarrow iterate(v_1; v_2 = e_2 | e_3)) := (\mathtt{free}(e_1) \cup \mathtt{free}(e_2) \cup \mathtt{free}(e_3)) - \{v_1, v_2\}$.

In order to properly address the subtyping relation, an expression of type t' is also an expression of a more general type t. Hence, for all $t' \leq t$, if $e \in \mathtt{Expr}_{t'}$ then $e \in \mathtt{Expr}_t$.

Using the syntax defined above, we can start to write assertions in OCL, embedding them in a UML model, as we will show in Example 2.

3 A Conceptual Framework for Static Verification of Dynamic Diagrams Consistency

The expressive power of object-oriented paradigm makes it better suited for development of large software systems than the traditional imperative paradigm. However, the statically checks enforced by e.g. C++ or Java compilers test for such syntactic and typing restrictions only that guarantee the lack of runtime type errors. This is the contracting and specification level that has been used for too many years in the past by most software developers. Obviously, this is not enough to prevent surprising and often disastrous behavior of programs. In other words, the checks done by compilers are only part of what is needed to reason about the behavior (i.e., the semantics) of software.

Software contracts are a necessary prerequisite for being able to introduce a notion of correctness: if you do not state what your program should do, you are lacking the norm to which to compare what your program does in reality. In defining class correctness we follow [17], p. 370:

Definition 1 ([17]). *A class C is* correct *with respect to its specification if*

– *For any set of valid arguments $e_1, \ldots, e_n$ to a creation procedure p:*

$$\{Default_C \wedge Pre_{\mathtt{p}}[\boldsymbol{x}/\boldsymbol{e}]\}\ p\ \{Post_{\mathtt{p}}[\boldsymbol{x}/\boldsymbol{e}] \wedge Inv_{\mathtt{C}}\}$$

– *For every public method m and any set of valid arguments $e_1, \ldots, e_n$:*

$$\{Pre_{\mathtt{m}}[\boldsymbol{x}/\boldsymbol{e}] \wedge Inv_{\mathtt{C}}\}\ m\ \{Post_{\mathtt{m}}[\boldsymbol{x}/\boldsymbol{e}] \wedge Inv_{\mathtt{C}}\}$$

where $Default_C$ denotes the assertion expressing that the attributes of C have the default values of their type.

What happens, for example, when a diagram specifies a call to a method when $Pre_{\mathtt{m}}[\boldsymbol{x}/\boldsymbol{e}] \wedge Inv_{\mathtt{C}}$ does not hold?

As already said, failure to meet any of the responsibilities stated in the contract results in a break of the contract, and indicates the existence of a bug somewhere in the design or implementation of the software or in the assertions themselves. Due to the size of real systems the latter chance is not so unlikely. In this paper we want to focus on this choice and propose a conceptual framework to reason about consistency of a UML diagram. In particular we want

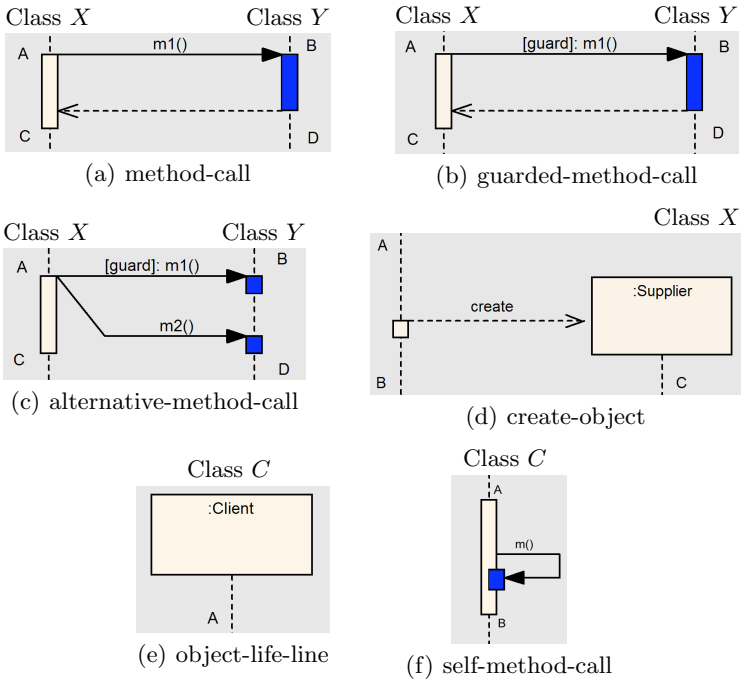


Figure 1. Sequence Diagrams Building Blocks

to check dynamic diagrams against static diagrams and OCL specifications. In other words, the idea is to consider class diagrams and OCL specifications as a kind of meta-specification and all the dynamic diagrams as meta-code which has to conform the specification. For the moment we restrict our attention to sequence diagrams.

Thus we aim to guarantee that, by following the control flow on the diagram, the state is strong enough to satisfy the entry precondition of methods calls.

We will define our verification method by structural induction on the (graphical) syntax of sequence diagrams. Since for some diagram elements there is not a precise semantics, we consider now only the ones which (as far as we are aware of) are the most relevant in practice and have a precise meaning. We believe these are enough to maintain a good level of generality and to be useful for practical cases.

In order to proceed we need to specify in a formal way the graphical syntax of sequence diagrams; mainly how we can compose (connect) basic diagrams to obtain bigger ones. All graphical blocks (we consider) refer to the lifetime of at most 2 objects at the same time. Hence they fall in one of the taxonomy shown in Figure 1. They have entry and exit points which are graphically connected to exits and entries of other blocks. We introduce a function *link* that, given an

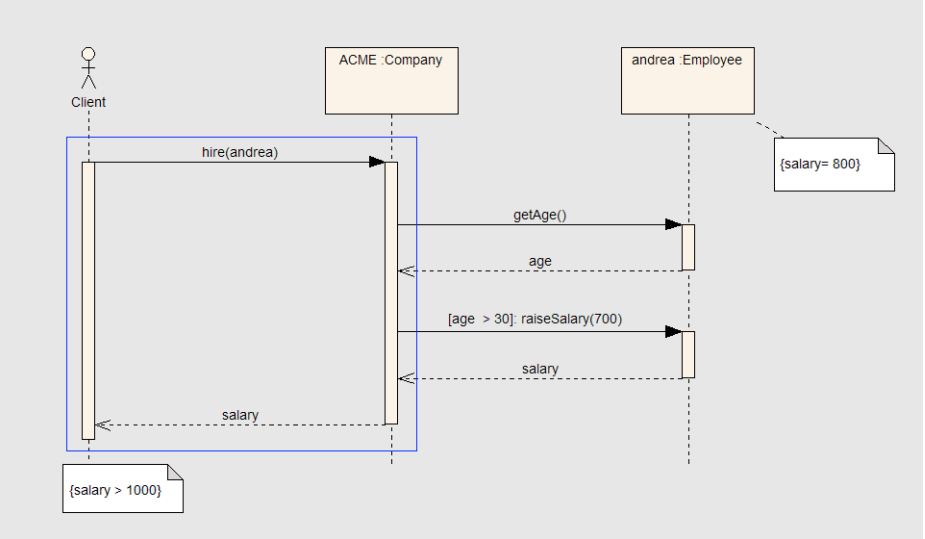


Figure 2. An Example of Sequence Diagram

entry point of a block, returns the exit point of the block to which the former is connected.

Most important than this, blocks can be nested. Inside the colored parts of blocks of type 1(a), 1(b), 1(c) and 1(f) we can plug blocks of type 1(f) or the left side of blocks of type 1(a), 1(b), 1(c) and 1(d). We can also plug any arbitrary sequential composition of the latter. We can trivially extend function *link* to take this kind of connections into account.

Example 1 (Decomposition of Sequence Diagrams in Blocks).

The sequence diagram of Figure 2 is decomposed in blocks according to our schema as in Figure 3. The whole diagram is composed of 2 blocks β_1, β_2 of type 1(e) connected to a outer block β_3 of type 1(a) which inside contains the sequential composition of two other blocks β_4, β_5, both of them of type 1(a). Thus function *link* in this case is defined as

$$\begin{array}{lll} link(B) = L & link(N) = B & link(C) = M \\ link(D) = O & link(E) = P & link(G) = Q \end{array}$$

while the blue boxes in the diagram indicate the block division.

We can now define our verification method by structural induction on the graphical syntax of sequence diagrams. The idea we follow here is to introduce formula variables for all points of the blocks, then we collect equalities between formula variables of the linked points and then we add all the implications that must hold within the formula variables inside the various blocks according to their semantics. The implications that do not hold show manifestly an inconsis-

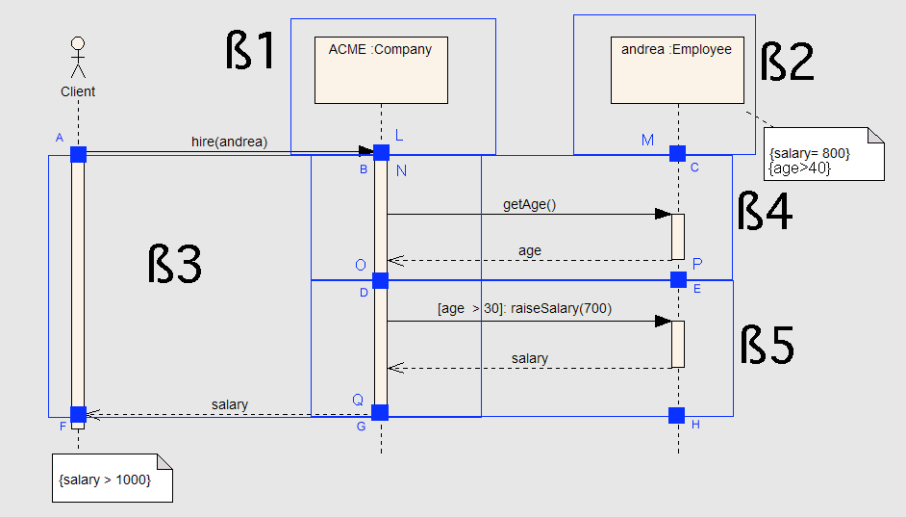


Figure 3. Decomposition of Sequence Diagrams in Building Blocks

tency of the sequence diagram, which can exploit a problem in the diagram or in the OCL specification of the method or of the class invariants involved.

Let now present the various possibilities. We assume now that methods are called with actual arguments $e_1, \ldots, e_n$ (denoted by $\boldsymbol{e}$) and that its formal parameters are $x_1, \ldots, x_n$ (denoted by $\boldsymbol{x}$).

Guarded Method Call (Figure 1(b)) We need to impose that

$$\Phi_C = result(\Phi_A) \wedge Post_{\texttt{m1}}\,[\boldsymbol{x}/\boldsymbol{e}] \qquad \Phi_A = \Phi_{link(A)}$$
$$\Phi_D = \Phi_B \wedge Post_{\texttt{m1}}\,[\boldsymbol{x}/\boldsymbol{e}] \qquad \Phi_B = \Phi_{link(B)}$$

and check that

$$\Phi_A \wedge guard \wedge \Phi_B \Longrightarrow Pre_{\texttt{m1}}\,[\boldsymbol{x}/\boldsymbol{e}] \tag{3.1}$$
$$\Phi_D \Longrightarrow Inv_{\texttt{Y}} \tag{3.2}$$
$$\Phi_C \Longrightarrow Inv_{\texttt{X}} \tag{3.3}$$

where $result(\Phi_A)$ denotes the formula Φ_A modified (if it is the case) by inserting the result of method `m1` in the container specified by the call in class X.

Equation (3.1) prescribes that in order to call method `m1` the states of caller and callee, under the guard condition, have to be strong enough to guarantee that the precondition of the method holds. Equations (3.2) and (3.3) prescribe that the states reached by the caller and that by the callee do not invalidate the corresponding class invariant.

(Unguarded) Method Call (Figure 1(a)) This is analogous to the previous with $guard := True$.

Conditional Method Call (Figure 1(c)) We need to impose that

$$\Phi_C = \mathit{result}(\Phi_A) \wedge ((\mathit{guard} \wedge \mathit{Post}_{\mathtt{m1}}[\boldsymbol{x}/\boldsymbol{e}]) \vee (\neg \mathit{guard} \wedge \mathit{Post}_{\mathtt{m2}}[\boldsymbol{x}/\boldsymbol{e}]))$$
$$\Phi_A = \Phi_{\mathit{link}(A)}$$
$$\Phi_D = (\mathit{guard} \wedge \Phi_B \wedge \mathit{Post}_{\mathtt{m1}}[\boldsymbol{x}/\boldsymbol{e}]) \vee (\neg \mathit{guard} \wedge \Phi_B \wedge \mathit{Post}_{\mathtt{m2}}[\boldsymbol{x}/\boldsymbol{e}])$$
$$\Phi_B = \Phi_{\mathit{link}(B)}$$

and check that

$$\Phi_A \wedge \mathit{guard} \wedge \Phi_B \Longrightarrow \mathit{Pre}_{\mathtt{m1}}[\boldsymbol{x}/\boldsymbol{e}]$$
$$\Phi_A \wedge \neg \mathit{guard} \wedge \Phi_B \Longrightarrow \mathit{Pre}_{\mathtt{m2}}[\boldsymbol{x}/\boldsymbol{e}]$$
$$\Phi_D \Longrightarrow \mathit{Inv}_{\mathtt{Y}}$$
$$\Phi_C \Longrightarrow \mathit{Inv}_{\mathtt{X}}$$

Self Method Call (Figure 1(f)) We need to impose that

$$\Phi_B = \mathit{result}(\Phi_A) \wedge \mathit{Post}_{\mathtt{m}}[\boldsymbol{x}/\boldsymbol{e}] \qquad \Phi_A = \Phi_{\mathit{link}(A)}$$

and check that

$$\Phi_A \wedge \mathit{guard} \Longrightarrow \mathit{Pre}_{\mathtt{m}}[\boldsymbol{x}/\boldsymbol{e}]$$
$$\Phi_B \Longrightarrow \mathit{Inv}_{\mathtt{C}}$$

Object Life Line (Figure 1(e)) We need to impose that

$$\Phi_A = \mathit{Inv}_{\mathtt{C}}$$

Create Object (Figure 1(d)) We need to impose that

$$\Phi_B = \Phi_A \qquad \Phi_A = \Phi_{\mathit{link}(A)} \qquad \Phi_C = \mathit{Default}_X$$

and check that

$$\Phi_C \Longrightarrow \mathit{Inv}_{\mathtt{X}}$$

Example 2 (Method at work). Now we provide a complete example in order to show how our method can be applied in practice. We start to describe the static description of a software system, building the class diagram shown in Figure 4. In this diagram, the Company and Employee classes are defined. In particular, Employee has the following attributes: age of type Integer, name of type String, and salary of type Double. Similarly, the attributes of class Company are location and name (both of type String). Employee has two methods: getAge, which takes no arguments and returns an Integer value (the age), and raiseSalary which takes a Double and return a Double (the raised salary). Company has two methods: fire and hire, both of which takes an object of type Employee as argument. The method hire returns a Double (the salary of the hired employee). If the employee to be hired has an age greater than 30 years, the hire method call raiseSalary

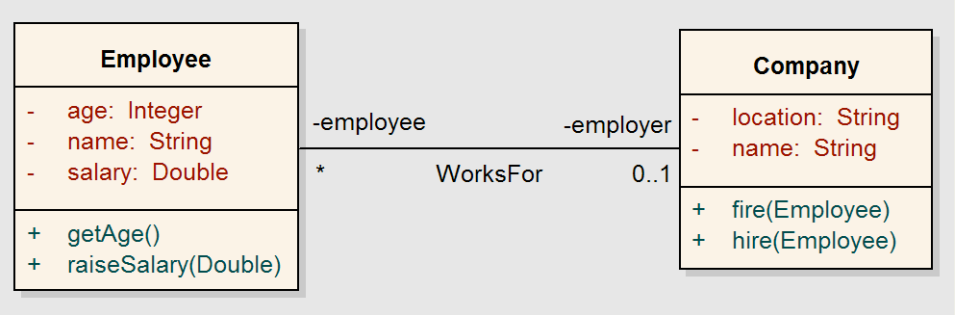


Figure 4. Example Class Diagram

in order to increase the current employee's salary of 700 units. This scenario is depicted in the sequence diagram shown in Figure 3. Then we now embed in the class diagram the software contracts specifications for both Employee and Company classes. In particular, we should define the classes invariants and the precondition - postconditions for each public method defined. We provide here part of this specification for the class Employee:

```
context Employee
  inv: (self.age >= 18)

context Employee::getAge() : Integer
  pre:  true
  post: (result =  self.age)

context Employee::raiseSalary(amount : Double) : Double
  pre:  true
  post: (self.salary = (self.salary@pre + amount))
  post: (result = self.salary)
```

The specification of the class Company is as follows:

```
context Company
  inv: self.employee->size() == self.employee->asSet()->size()

context Company::hire(p : Employee)
  pre  hirePre1: p.isDefined
  pre  hirePre2: self.employee->excludes(p)
  post hirePost: self.employee->includes(p)

context Company::fire(p : Employee)
  pre  firePre: self.employee->includes(p)
  post firePost: self.employee->excludes(p)
```

Now we show what we obtain with our method about the sequence diagram of Figure 3. The equalities on formula variables are:

$$\Phi_B = \Phi_N = \Phi_L = Inv_{Company} = \\ (Company.employee \dashrightarrow size() \equiv Company.employee \dashrightarrow asSet() \dashrightarrow size())$$
$$\Phi_C = \Phi_M = (Andrea.age \geq 40 \wedge salary \equiv 800)$$
$$\Phi_D = \Phi_O = (Inv_{Company} \wedge Result \equiv Andrea.age)$$
$$\Phi_E = \Phi_P = (Andrea.age \geq 40 \wedge salary \equiv 800 \wedge Result \equiv Andrea.age)$$
$$\Phi_Q = (Inv_{Company} \wedge Result \equiv 1500)$$
$$\Phi_H = (Andrea.age \geq 40 \wedge salary \equiv 1500 \wedge Andrea.salary \equiv 1500 \wedge Result \equiv 1500)$$
$$\Phi_A = (Company.isDefined \wedge Andrea.isDefined)$$
$$\Phi_F = (salary \equiv 1500 \wedge Result \equiv 1500)$$
$$\Phi_G = (Inv_{Company} \wedge Company.employee \dashrightarrow includes(Andrea))$$

While the implications that we have to check are

$$Inv_{Company} \wedge Andrea.age \geq 40 \wedge salary \equiv 800 \Longrightarrow True$$
$$Andrea.age \geq 40 \wedge salary \equiv 800 \wedge Result \equiv Andrea.age \Longrightarrow Andrea.age \geq 18$$
$$Inv_{Company} \wedge Result \equiv Andrea.age \Longrightarrow Inv_{Company}$$
$$Inv_{Company} \wedge Andrea.age \geq 30 \wedge Andrea.age \geq 40 \wedge salary \equiv 800 \Longrightarrow True$$
$$Inv_{Company} \wedge Result \equiv 1500 \Longrightarrow Inv_{Company}$$
$$Andrea.age \geq 40 \wedge Andrea.salary \equiv 1500 \Longrightarrow Andrea.age \geq 18$$
$$Company.isDefined \wedge Andrea.isDefined \wedge Inv_{Company} \Longrightarrow \\ Andrea.isDefined \wedge Company.employee \dashrightarrow exclude(Andrea) \quad \text{(i)}$$
$$Inv_{Company} \wedge Company.employee \dashrightarrow includes(Andrea) \Longrightarrow Inv_{Company}$$

All implications can be verified except of (i). Actually looking at (i) we discover that there is nothing in the diagram which specifies that *Andrea* is not already an hired employee. If we add in the diagram an initial constraint specifying that $Company.employee \dashrightarrow excludes(Andrea)$ then we can prove the new (i).

This example suggests that in practical situations the assertions to be added in order to reach consistency can be quite easy by just inspecting the failing formula.

4 Applicability of the conceptual method

As already stated, the conceptual method that we have just presented is just a first step in a much more ambitious direction. Clearly in its generality it cannot be implemented because automatic proof of the verification formulas is undecidable. We think that even in this case the dimension of the state space generated is so large that it cannot be explored explicitly by model-checking nor reasonably covered by testing.

However there are two possible nowadays well explored directions which we can follow from now on. One is that of using some proof assistant, like Coq [**?**]. The Coq tool is a formal proof management system: a proof done with Coq is mechanically checked by the machine. This direction of research is quite fertile in the literature. Several tools are being built on top of Coq, for object-oriented software verification purposes. For example Krakatoa [**?**] is a Java code certification tool that uses Coq to verify the soundness of implementations with regards to the specifications and Caduceus [**?**] is a verification tool for C programs.

However even computer-aided formal proofs tend to be humanely demanding and economically costly. An alternative is to use Abstract Interpretation Techniques where an abstraction of the semantics of the programs is automatically computed. This leaves out all information about reachable states which is not strictly necessary for the proof. Of course if the abstraction is too precise, the computation cost are too high (resource exhaustion) and if it is too rough, nothing can be proved (false alarm). Although the best abstraction does exist, it is not computable, and so, must be found experimentally.

There has been a lot of research on these topics and recently we find tools based on Abstract Interpretation like Astree [2] that can be used with great success for verification purposes of large C software systems.

5 Conclusions

Many approaches for model validation and verification rely on generation of suitable code which dynamically checks the validity of OCL constraints. This approach has several drawbacks. For example, it cannot generally guarantee that a constraint will never be violated, unless an infinite number of tests is performed. Also the generation of just a *significative* finite subset is not so feasible because, on one hand, a considerable manual effort is needed even to produce a single test scenario and, on the other hand, test-case generation is well-known to be a hard problem.

On the other hand, static approaches based on model checking suffer of the state explosion problem and thus cannot scale to real system sizes. Moreover they are also inherently limited to verification of a single specific property of the system at a time.

In this paper we presented an approach for a static verification of consistency of UML models. By using OCL constraints together with class diagrams, it certifies that the dynamic part of the model is satisfied. We hope to have convinced the reader that, to some extent, this encompasses the weaknesses of the other mentioned methods as it does not require users to build test scenarios and also performs the verification of *all* system properties at the same time.

Since the method is not necessarily effective, we suggested two possible directions of research to render it effective (without loosing its nice properties).

References

1. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected Papers*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, 2003. Springer-Verlag.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), June 7–14, San Diego, California, USA*, pages 196–207, New York, NY, USA, 2003. ACM Press.
3. M. Comini. VeriPolyTypes: a tool for Verification of Logic Programs with respect to Type Specifications. In M. Falaschi, editor, *Proceedings of 11th International Workshop on Functional and (constraint) Logic Programming*, number UDMI/18/2002/RR in Research Reports, pages 233–236, Udine, Italy, 2002. Dipartimento di Matematica e Informatica, Università di Udine.
4. M. Comini, R. Gori, and G. Levi. Assertion based Inductive Verification Methods for Logic Programs. In A. K. Seda, editor, *Proceedings of MFCSIT'2000*, volume 40 of *Electronic Notes in Theoretical Computer Science*, pages 1–18, North Holland, 2001. Elsevier Science Publishers. Available at URL: `http://www.elsevier.nl/locate/entcs/volume40.html`.
5. M. Comini, R. Gori, and G. Levi. How to Transform an Analyzer into a Verifier. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming and Automated Reasoning. Proceedings of the 8th International Conference (LPAR'01)*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 595–609, Berlin, 2001. Springer-Verlag.
6. M. Comini, R. Gori, and G. Levi. Logic programs as specifications in the inductive verification of logic programs. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Declarative Programming – Selected Papers from AGP 2000*, volume 48 of *Electronic Notes in Theoretical Computer Science*, pages 1–16, North Holland, 2001. Elsevier Science Publishers. Available at URL: `http://www.elsevier.nl/locate/entcs/volume48.html`.
7. M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract Interpretation based Verification of Logic Programs. *Science of Computer Programming*, 49(1–3):89–123, 2003.
8. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 1-2:47–103, 2002.
9. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Los Angeles, California, January 17–19*, pages 238–252, New York, NY, USA, 1977. ACM Press.
10. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, January 29–31*, pages 269–282, New York, NY, USA, 1979. ACM Press.
11. P. Cousot and R. Cousot. 'A la Floyd' induction principles for proving inevitability properties of programs. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 277–312. Cambridge University Press, Cambridge, UK, 1985.

12. P. Cousot and R. Cousot. A language independent proof of the soundness and completeness of generalized Hoare logic. *Information and Computation*, 80(2):165–191, 1989.
13. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
14. J.-M. Jézéquel, M. Train, and C. Mingins. *Design Pattern and Contracts*. Addison-Wesley, Reading, MA, 2000.
15. B. Meyer. Applying "Design by Contract". *Computer: Innovative Technology for Computer Professionals*, 25(10):40–51, 1992.
16. B. Meyer. *Eiffel – the Language*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
17. B. Meyer. *Object-Oriented Software Construction, 2/E*. Prentice-Hall, Englewood Cliffs, NJ, 1997.
18. Object Management Group. *UML 2.0 OCL Specification*. Document – ptc/05-06-06 (OCL FTF report – convenience document).
19. Object Management Group. *UML 2.0 Superstructure Specification, v2.0*. Document – formal/05-07-04 (UML Superstructure Specification, v2.0).
20. M. Richters and M. Gogolla. OCL: Syntax, Semantics, and Tools. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer-Verlag, Berlin, 2002.
21. H. Toth. On theory and practice of Assertion Based Software Development. *Journal of Object Technology*, 4(2):109–130, 2005.
22. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, Reading, MA, 2003.