Università di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

The CIFF Proof Procedure for Abductive Logic Programming with Constraints: Definition, Implementation and a Web Application

Giacomo Terreni

Supervisor
Paolo Mancarella

Supervisor
Francesca Toni

May 5, 2008

Abstract

Abduction has found broad application as a powerful tool for hypothetical reasoning with incomplete knowledge, which can be handled by labeling some pieces of information as abducibles, i.e. as possible hypotheses that can be assumed to hold, provided that they are consistent with the given knowledge base.
Attempts to make the abductive reasoning an effective computational tool gave rise to *Abductive Logic Programming* (ALP) which combines abduction with standard logic programming. A number of so-called *proof procedures* for ALP have been proposed in the literature, e.g. the IFF procedure, the Kakas and Mancarella procedure and the SLDNFA procedure, which rely upon extensions of different semantics for logic programming. ALP has also been integrated with *Constraint Logic Programming* (CLP), in order to combine abductive reasoning with an arithmetic tool for *constraint solving*.
In recent years, many proof procedures for abductive logic programming with constraints have been proposed, including ACLP and the A-System which have been applied to many fields, e.g. multi-agent systems, scheduling, integration of information.

This dissertation describes the development of a new abductive proof procedure with constraints, namely the CIFF proof procedure. The description is both at the theoretical level, giving a formal definition and a soundness result with respect to the three-valued completion semantics, and at the implementative level with the implemented CIFF System 4.0 as a Prolog meta-interpreter.
The main contributions of the CIFF proof procedure are the advances in the expressiveness of the framework with respect to other frameworks for abductive logic programming with constraints, and the overall computational performances of the implemented system.

The second part of the dissertation presents a novel application of the CIFF proof procedure as the computational engine of a tool, the CIFFWEB system, for checking and (possibly) repairing faulty web sites.
Indeed, the exponential growth of the WWW raises the question of maintaining and automatically repairing web sites, in particular when the designers of these sites require them to exhibit certain properties at both structural and data level. The capability of maintaining and repairing web sites is also important to ensure the success of the Semantic Web vision. As the Semantic Web relies upon the definition and the maintenance of consistent data schemas (XML/XMLSchema, RDF/RDFSchema, OWL and so on), tools for reasoning over such schemas (and possibly extending the reasoning to multiple web pages) show great promise.
The CIFFWEB system is such a tool which allows to verify and to repair XML web sites instances, against sets of requirements which have to be fulfilled, through abductive reasoning.
We define an expressive characterization of rules for checking and repairing web sites' errors and we do a formal mapping of a fragment of a well-known XML query language, namely Xcerpt, to abductive logic programs suitable to fed as input to the CIFF proof procedure.
Finally, the CIFF proof procedure detects the errors and possibly suggests modifications to the XML instances to repair them. The soundness of this process is directly inherited from the soundness of CIFF.

Contents

1 Introduction **5**
- 1.1 Abductive Logic Programming and CIFF . . . 6
- 1.2 CIFF for repairing XML Web sites instances . . . 9
- 1.3 Overview of the thesis . . . 11

2 Preliminaries **13**
- 2.1 First Order Logic . . . 13
  - 2.1.1 Interpretations and Models . . . 15
  - 2.1.2 Substitutions and unification . . . 17
- 2.2 Definite Logic Programming . . . 19
  - 2.2.1 Definite clauses, programs and goals . . . 19
  - 2.2.2 Semantics of Definite Logic Programming . . . 20
  - 2.2.3 SLD-resolution . . . 24
- 2.3 Negation in Logic Programming . . . 27
  - 2.3.1 The Negation As Failure (NAF) rule . . . 27
  - 2.3.2 Completion of a logic program . . . 28
  - 2.3.3 SLDNF for Definite Logic Programs . . . 29
  - 2.3.4 Normal Logic Programming . . . 31
  - 2.3.5 SLDNF resolution for Normal Logic Programs . . . 34
- 2.4 Alternative Semantics of Normal Logic Programming . . . 35
  - 2.4.1 Three-valued completion semantics . . . 36
  - 2.4.2 Stable models semantics . . . 38
  - 2.4.3 Well-founded semantics . . . 39
- 2.5 Constraint Logic Programming (CLP) . . . 41
- 2.6 Answer Sets Programming (ASP) . . . 43

3 Abductive Logic Programming with Constraints (ALPC) **47**
- 3.1 Abductive reasoning . . . 47
- 3.2 Abduction in logic . . . 48
- 3.3 Abductive Reasoning in Applications . . . 50
- 3.4 Abductive Logic Programming (ALP) . . . 51
- 3.5 Abductive proof procedures . . . 53
  - 3.5.1 The Kakas and Mancarella procedure . . . 53
  - 3.5.2 The SLDNFA procedure . . . 57
  - 3.5.3 The IFF proof procedure . . . 59
  - 3.5.4 Other computational approaches to abduction . . . 62
- 3.6 ALP + CLP = ALPC . . . 63
- 3.7 Abductive proof procedures with constraints . . . 65
  - 3.7.1 The ACLP proof procedure . . . 65
  - 3.7.2 The $\mathcal{A}$-System . . . 65

4 The CIFF Proof Procedure and the CIFF$^\neg$ extension **67**
4.1 The CIFF proof procedure . . . 67
4.1.1 CIFF proof rules . . . 72
4.1.2 CIFF derivation and answer extraction . . . 78
4.2 Soundness of the CIFF Proof Procedure . . . 82
4.3 The CIFF$^\neg$ proof procedure . . . 92
4.4 Soundness of the CIFF$^\neg$ proof procedure . . . 98

5 The CIFF System **105**
5.1 The CIFF System: an Overview . . . 106
5.1.1 Input Programs, Preprocessing and Abductive Answers . . . 109
5.1.2 Variable Handling . . . 113
5.1.3 Constraints Handling . . . 114
5.1.4 Loop Checking and CIFF Proof Rules Ordering . . . 116
5.1.5 The CIFF$^\neg$ proof procedure . . . 121
5.1.6 Ground Integrity Constraints . . . 123
5.2 Related work and comparison . . . 127
5.2.1 Comparison with $\mathcal{A}$-System . . . 128
5.2.2 Comparison with Answer Sets Programming . . . 130
5.2.3 Experimental results . . . 132
5.3 Conclusions . . . 139

6 Web Site Verification and Repair **141**
6.1 A Motivating Example . . . 142
6.2 A Formal Language for Expressing Web Checking Rules . . . 143
6.3 A Xcerpt-like grammar for positive web checking rules . . . 145
6.4 The translation process for positive rules . . . 146
6.4.1 XML representation . . . 146
6.4.2 The translation function . . . 147
6.5 Adding negation to the translation process . . . 151
6.5.1 Translation function . . . 151
6.6 Analysis for checking . . . 157
6.7 Running the System . . . 158
6.7.1 A preliminary experimentation . . . 159
6.8 A Web Repairing Framework . . . 160
6.8.1 Abductive logic programs for repairing . . . 161
6.9 Running the CIFFWEB System for repairing . . . 165
6.9.1 Abductively Generated Errors . . . 166
6.10 A more complex running example . . . 167
6.11 Analysis for repairing . . . 168
6.12 Related Work, Future Work and Conclusions . . . 170
6.13 Full theater example . . . 172
6.13.1 XML data . . . 172
6.13.2 XML translation . . . 172
6.13.3 Web checking rules . . . 173
6.13.4 Abductive logic program for checking . . . 178
6.13.5 Abductive logic program for repairing . . . 181

7 Conclusions **185**

Bibliography **187**

List of Figures

5.1 The CIFF System: main computational cycle 107
5.2 The CIFF System: modules interactions 110

6.1 The CIFFWEB System: JAVA translator GUI 158

List of Tables

4.1 CIFF proof rules 81
4.2 CIFF$^{\neg}$ proof rules 96

5.1 `N-Queens` results (first solution) 134
5.2 `Hamiltonian cycles` results (all solutions) 135
5.3 `Graph coloring` results (first solution). 136
5.4 `Scalability` results (**Test1**). 137
5.5 `Scalability` results (**Test2**). 137
5.6 `Scalability` results (**Test3**). 137
5.7 `Scalability` results (**Test4**). 138
5.8 `Scalability` results (**Test5**). 138
5.9 `Scalability` results (**Test6**). 138

6.1 CIFFWEB scalability results for checking 160

Chapter 1

Introduction

Nowadays, computers are massively used in almost every human activity to accomplish an infinite variety of tasks: from intensive data processing to 3D modeling passing through a web search for interesting movies played near home.
This has been a big revolution in modern society and it has been due both to the huge improvements in computer hardware in terms of costs, dimensions and performances and the big advances in theoretical computer science whose main applications are the "middlewares" (programming languages, internet protocols and so on) upon which the user-end applications are built.
However, the mostly used "middlewares" (e.g. C and C++ programming languages), to date, allow for writing user-end applications in a machine-oriented language and, on their side, user-end applications are often written to accomplish a specific instance of a more general problem.

In the last decade, as the user-end applications have grown in number and complexity, the need emerged of simplifying the building and the maintenance of user-end applications. This need has been even more evidenced by the exponential growth of the World Wide Web which, on the back-end side, needs for computational schemas able to handle huge amounts of heterogeneous data. Thus, a new generation of "middlewares" (e.g. Java and UML) have been developed and used for this purpose.
This generation of commonly used "middlewares", however, even if improving the previous stage, are still machine-oriented and tailored towards specific user-end applications.

In the meanwhile, almost since the beginning of computer science, a not very popular set of "middlewares" has been developed and refined so far: *declarative languages*. The main features of a declarative language are to be human-oriented and tailored towards general problem solving.
The main idea is that a declarative language provides a formal *logic* (a well-defined syntax and a clear semantics associated with) plus a *control* mechanism for that logic (i.e. a "reasoner" which evaluates formulae on that logic), while the application developer has only to *model* its problem under the logic of the language. It will be the control mechanism inside the language which will evaluate the particular instances of the problem with respect to the given logic.
The straight use of logic for defining the syntax of declarative languages, makes them very suitable tools for *knowledge representation*, i.e. for representing objects and their relations of the modeled *world* in an intuitive human-oriented syntax close to natural langauge. For example in Prolog, without any doubt the most important and influential example of declarative language which gave rise to the *Logic Programming* field, to express that *a father of a person is a male parent of that person*, it is enough to use the following simple and human-tailored logic formula:

$$father(X, Y) \leftarrow parent(X, Y), male(X)$$

If we add to the above general statement, the particular "world" instance that *John is a male* and *John is a parent of Mary*, represented as:

$male(John)$
$parent(John, Mary)$,

the embedded Prolog reasoner is able to *deduce* that $John$ is the father of $Mary$.
However, after an initial enthusiasm in the middle 70's, the popularity of declarative languages shrank dramatically due to the performance gap of declarative applications with respect to hand-tailored applications.

However, there are a number of classic problems and applications which would benefit of a declarative approach (planning, scheduling, combinatorial problems) and many other problems which are emerging, in particular concerning the Semantic Web vision. Indeed, the Semantic Web vision relies upon web specification and querying languages (XML/XML Schema, RDF/RDF Schema, OWL, XQuery, XPath, Xcerpt and so on) equipped with a human-tailored logic syntax (plus a not always well-defined semantics). Hence the use of declarative languages as their "control" mechanism is an intuitive, even if non-straightforward choice.
On the other hand, hardware improvements together with the refinements of declarative languages, make the use of them a concrete choice in developing a wide range of applications.

Over the years, declarative languages have been improved in many directions. Prolog itself has been refined and extended in a number of ways. The main extensions of Prolog, enhancing its expressiveness as a tool for knowledge representation, concern *parallelism and concurrency* (giving rise to *distributed logic programming*, with communication protocols as typical applications), *arithmetical constraints* (giving rise to *constraint logic programming*, with combinatorial applications as typical applications) and *abduction* (giving rise to *abductive logic programming*, with diagnosis and repairing applications as typical applications).

Our work is placed exactly in this setting, as the main contributions of our thesis are:

- the definition of the *CIFF proof procedure*, a general-purpose abductive extension of logic programming (including arithmetical constraints) which provides interesting advances in the expressiveness of the abductive framework together with the proof of its soundness with respect to a semantics for logic programming, namely the *three-valued completion semantics*,
- the *CIFF System*, a robust and efficient implementation, on top of Prolog, of the CIFF proof procedure, and
- the *CIFFWEB System*, an automatic tool for detecting and (possibly) repairing errors of XML web sites with respect to *web checking rules*, i.e. web requirements expressed in Xcerpt, a well-known XML query language with a very human-tailored syntax.

1.1 Abductive Logic Programming and CIFF

The notion of *abduction* was introduced by the philosopher Pierce in [144] where he identified three main forms of reasoning:

Deduction an analytic process based on the application of general rules to particular cases, with the inference of a result;

Induction synthetic reasoning which infers the rule from the case and the result;

Abduction another form of synthetic inference, inferring the case from a rule and a result.

Peirce further characterized abduction as the "probational adoption of a hypothesis" as explanation for observed facts (results) according to known laws. "It is however a weak form of inference, because we cannot say that we believe in the truth of the explanation, but only that it may be true" [144].
Abduction is widely used in common-sense, daily reasoning. For instance in diagnosis, to reason from effect to cause, as noted e.g. in [41].

In logic, the abductive task can be formalized as the task of finding a set of hypothesis Δ which, together with a logic theory P, representing the known laws, allows to infer a set of formulae Q, representing the observations (or the results).

The following well-known example of abductive reasoning was given by Pearl in [141]. Consider to observe that, walking in the garden, the shoes become wet. A simple explanation for this observation is that the grass is wet. Being a sunny day, further explanations are that either the sprinkler was on or it rained during the night. This common-sense process is exactly an abductive process. The explanations have been inferred abductively from the observations and the "rules", stating for example that "if it rained last night then the grass has to be wet". A logical formulation of this "world" could be obtained by a theory P consisting of:

$$\begin{array}{lcl} grass_is_wet & \leftarrow & rained_last_night \\ grass_is_wet & \leftarrow & sprinkler_was_on \\ shoes_are_wet & \leftarrow & grass_is_wet \end{array}$$

In this setting, the observation $Q = shoes_are_wet$ can be explained by both $rained_last_night$ and $sprinkler_was_on$ alternatively.

Typically, in an abductive framework, there is a further component: a set of *integrity constraints* IC, i.e. a set of formulae which have to be "satisfied" by Δ in order to declare such Δ an *acceptable* set of hypothesis for inferring Q. In the above example, an integrity constraint could be

$$rained_last_night \rightarrow false$$

With the above integrity constraint, any abductive explanation containing $rained_last_night$ is forbidden.

Abduction is a form of *nonmonotonic* reasoning because explanations which are consistent with an abductive framework, may become inconsistent adding new information to the framework.

More formally, the abductive task can be defined as the problem of finding a set of formulae Δ (*abductive explanations* for Q) such that:

1. $P \cup \Delta \models Q$,
2. $P \cup \Delta$ is consistent and
3. $P \cup \Delta$ satisfies IC

where $\models$ stands for a particular logic semantics.

The abductive task can be easily instantiated to logic programming, giving rise to *Abductive Logic Programming (ALP)*. In general an abductive framework in logic programming is a tuple $\langle P, A, IC \rangle$ where P is a logic program, IC is a set of first order closed formulae and A is a set of predicates declared as abducibles. Each atom in Δ must have a predicate $a \in A$. In the above example the set of abducible predicates is:

$$A = \{rained_last_night, sprinkler_was_on\}$$

ALP can also be integrated with *Constraint Logic Programming*, giving rise to *Abductive Logic Programming with Constraints (ALPC)*. An abductive framework with constraints is a tuple $\langle P, A, IC \rangle_{\Re}$ where $\Re$ is a constraint structure providing relations and functions (typically arithmetical relations and functions) on a domain $D(\Re)$, evaluated under the specific semantics of $\Re$. In this way an abductive framework with constraints combines the expressiveness of ALP and the expressiveness of the Constraints.

A number of *abductive proof procedures* for ALP have been proposed in the literature, e.g. the Kakas-Mancarella proof procedure [102, 100], the SLDNFA proof procedure [61] and the IFF proof procedure [82]. Abductive proof procedures, in general, differ in the expressiveness of their specific frameworks and in what rely upon various semantics for logic programming, the most common being the (generalized) stable models semantics [102] and the (three-valued) completion semantics [118].

In recent years, several proof procedures for ALPC have also been proposed, including ACLP [109] and the A-System [111].

Abduction (and in particular ALP and ALPC), has been used in a wide range of applications.
Abduction can be used to generate causal explanations for *fault diagnosis*, as seen for example in [147, 56]. In medical diagnosis, for example, the observations are the symptoms and the abductive process finds the possible causes (diseases) of those symptoms [152].
Abduction has been used for improving robotic *vision* [57, 165] where the observations are the raw data descriptions obtained from the robot visual sensors and the abductive process hypotheses on which objects effectively "see" the robot.
Also *scheduling* [109] and *planning* can be easily modeled by means of abduction. A plan can be viewed as a set of hypothetical actions (and subgoals) to be performed (and achieved) in order to reach the final goal state. The main approach to abductive planning is based on the *event calculus*, a logical framework for reasoning about actions and changes proposed by Sergot and Kowalski in [115]. Abductive planning has been studied by several authors [74, 135, 164].
Database updates is another important application of abduction [170, 101, 19]. In this setting the observations are the update requests and the abductive explanations are the transactions that satisfy those requests.
Finally, in recent years, abduction has been studied has a main reasoning paradigm for modeling *intelligent* and *autonomous agents*, a field that has captured great attention in the last decades [148, 53, 149, 98, 105, 31]. An intelligent agent can be defined as an actor which is capable of observing, reasoning and acting upon a (dynamical) environment.

The main contribution of this thesis is the *CIFF proof procedure* for ALPC. CIFF (see [71, 72, 70, 125, 73] for preliminary versions and applications) is a proof procedure based on *rewriting*, i.e. its computational core is based on a set of *proof rules* which rewrite a formula into an equivalent formula under the three-valued completion semantics. CIFF is an extension of the IFF proof procedure [82], which, with respect to other abductive proof procedures, shows two main features: (1) handling of (existentially and universally quantified) variables and (2) handling of integrity constraints in implicative form:

$$L_1 \wedge \cdots \wedge L_m \;\rightarrow\; A_1 \vee \cdots \vee A_n.$$

This form of integrity constraint makes possible the use of forward reasoning in addition to the classical Prolog-style backward reasoning, allowing for a flexible computational tool for modeling a wider class of problems. In particular forward reasoning with integrity constraints in implicative form can capture the *condition-action rule* type of behavior, which is very typical and useful in dynamic settings [157].

The CIFF proof procedure extends IFF in three ways, namely (1) by integrating abductive reasoning with constraint solving, (2) by relaxing the allowedness conditions on suitable inputs given in [82], in order to be able to handle a wider class of problems, and (3) by adding a different type of negation treatment in integrity constraints, namely the *negation as failure* (NAF) [48].
The third extension was similarly proposed in [157] as an extension for the IFF proof procedure. To briefly explain the NAF extension let us consider a very simple integrity constraint like the following:

$$\neg L \;\rightarrow\; A.$$

Normally, in CIFF, the negation is treated as classical negation, and the IC is treated equivalently to the disjunction $A \vee L$. Hence, in order to satisfy the IC either A or L must hold. However, often such an IC is required to produce condition-action or reactive behavior, where the intended interpretation is:

if there is evidence that $\neg L$ holds, then A must hold too,

or, more appropriately,

if there is no evidence that L holds, then A must hold.

This is exactly what the NAF view allows, avoiding some unintuitive abductive answers. Consider a "world" where an agent has to clean in a dangerous environment. The idea is that if it is a cleaning day and the alarm has not been activated, the agent must dust. This could be represented by the integrity constraint

$$cleaning_day \wedge \neg sound_alarm \;\rightarrow\; dust$$

A logical interpretation of negation, provided that today is a *cleaning day*, allows the agent to either *dust* or *sound the alarm* (even if no dangerous situation arises) in order to satisfy its rule of behavior! In the NAF view, the agent avoids to dust only if the alarm has been sounded for some other reason. As we will see, this feature enhances the expressive power of the framework especially for modeling condition-action and reactivity rules.

A main contribution of this thesis, is also the proof of soundness of CIFF with respect to the three-valued completion semantics.

The CIFF proof procedure has been implemented in Prolog in the CIFF System, which is available at *www.di.unipi.it/∼terreni/research.php*.
The system is a mature Prolog system for which much time has been spent in exploiting Prolog algorithms and data structures for improving the overall efficiency of the system, but maintaining a clear mapping between the specification and the implementation.
We have compared empirically CIFF and the CIFF System to other related systems, namely the A-System [111, 139], that is the closest system from both a theoretical and an implementative viewpoint, and two state-of-the-art answer sets solvers SMODELS [136, 166] and DLV [68, 119].
Answer sets solvers are very popular tools which implement the (answer sets) semantics [90]. Their computational schema is very distinct from the computational schema of logic programming and they gave rise to the branch of *Answer Sets Programming (ASP)*. However, ASP shares with ALPC the objective of modeling dynamic and non-monotonic settings in a declarative (and thus human-oriented) way.
The comparison both evidences how CIFF enriches the expressiveness of the abductive framework and both evidences the similarities and the differences between ALPC and ASP. In particular we outline how CIFF, with its NAF extension, is a step towards a unifying framework for ALPC and ASP.
The experimental results on the CIFF System show that (1) the CIFF System and the other systems have comparable performances and (2) the CIFF System has some unique features, e.g. its incorporation of NAF in integrity constraints and its handling of variables taking values on unbound domains.

The features of earlier versions of the procedure have been exploited in various application domains. In [105] CIFF has been used as the computational core for modeling an agent's planning, reactivity and temporal reasoning capabilities, based on a variant of the abductive event calculus [115, 133]. A prototype version of the CIFF system has also been studied for abductive planning [124, 71].

1.2 CIFF for repairing XML Web sites instances

We strongly believe that declarative languages such as Prolog [54], if they are well integrated with the web, will play a crucial role as the computational paradigms in the Semantic Web vision, as noted, e.g., in [182].
The increasing interest about the Semantic Web technologies [175, 13] seems to be the right place for declarative approaches. In particular web specification languages from XML/XMLSchema [177] to OWL [176] passing through RDF/RDFSchema [174] need expressive computational counterparts allowing more and more reasoning capabilities.

Abduction too, as it is a very suitable form of reasoning for diagnosis and repairing, could play a prominent role in that context, as noted, e.g., in [39], where some abductive tasks over ontologies have been individuated.

In this thesis we concentrate on how abduction could be used as a computational mechanism for maintaining and repairing XML/XHTML web sites instances.
The exponential growth of the WWW raises the question of maintaining and automatically repairing web sites, in particular when the designers of these sites require them to exhibit certain properties at both structural and data level. The capability of maintaining and repairing web sites is also important to ensure the success of the Semantic Web [175, 13] vision. Nevertheless, there is limited work on specifying, verifying and repairing web sites at a semantic level. Notable exceptions, at least for specifying and checking web sites, are represented by works such as [173] (which mainly inspired our work) [62] and [78]; the XLINKIT framework [40] and the GVERDI-R system [10, 23].
Searching the web, it is easy to encounter web pages containing errors in their structure and/or their data. We argue that, in most cases, considering an XML/XHTML web site instance, the errors can be divided into two main categories: structural errors and content-related (data) errors. Structural errors are those errors concerning the presence and/or absence of tag elements and relations amongst tag elements in the pages. For example, if a tag *tag*1 is intended to be a child of a tag *tag*2, the occurrence in the web site of a *tag*1 instance outside the scope of a *tag*2 instance is a structural error. Data errors, instead, are about the in-tag data content of tag elements. For example a *tag*3 could be imposed to hold a number greater than 100.
Consider the following extracts of two XML pages, representing, in a theater company web site instance, a list of shows produced by that company and the list of directors of that company:

```
%%%directorindex.xml
<directorlist>
    <director>John</director>
    <director>Mary</director>
</directorlist>
```

```
%%%showindex.xml
<showlist>
    <show>
        <showname>Epiloghi</showname>
        <year>2001</year>
    </show>
</showlist>
```

We could specify a number of rules which any web site instance should fulfill. For example, we could specify that a correct *structure* of a `show` tag in the first page must contain both a `showname` tag element and a `dir_by` tag element as its child. In this case we have a *structural error*, due to the lack of a `dir_by` tag element in the show of the list. Also, we could specify that all the shows in the list must be produced since the year 2000. In this case we have a *data error* if the value inside an `year` tag is less than 2000.
Requirements (and thus errors) can involve more than one web page. For example, a possible requirement for the theater company specification may be that

each director must direct at least one show

The above requirement can lead to content-related errors which involve both pages because the *content* of each `director` tag, e.g. `John` has to be matched to the content of at least one `dir_by` tag inside a `show` tag.

Here, we propose the CIFFWEB (prototype) tool, which, roughly speaking, uses CIFF as the computational core for verifying and, possibly, repairing web sites against sets of requirements which have to be fulfilled by a web site instance. A preliminary version of the tool is shortly described in [127].
We argue that the CIFF proof procedure has very useful features to be applied in a web reasoning scenario: implicative integrity constraints which could act as condition-action rules, handling of unbound variables which could directly represent missing data, and arithmetical constraint solving capabilities.

In our framework, we define an expressive characterization of rules for checking web sites' errors by using (a fragment of) the well-known semi-structured data query language Xcerpt [37, 38]. With respect to the other semi-structured query languages (like XQuery [27], XPath [47]) which all propose a path-oriented approach for querying semi-structured data, Xcerpt is a rule-based language which relies upon a (partial) pattern matching mechanism allowing to easily express complex queries in a natural and human-tailored syntax. Xcerpt shares many features of logic programming, for example its use of variables as place-holders and unification. However, to the best of our knowledge, it lacks (1) a clear semantics for negation constructs and (2) an implemented tool for running Xcerpt programs and evaluating Xcerpt queries. A by-product of this Chapter is the provision of both (1) and (2) for a fraction of Xcerpt, namely the subset of this language that we adopt for expressing *web checking rules*.
Then, we map formally the chosen fragment of Xcerpt for expressing checking rules into *programs for checking*, i.e. abductive logic programs with constraints that can be fed as input to the general-purpose CIFF proof procedure. By mapping web checking rules onto abductive logic programs with constraints and deploying CIFF for determining fulfillment (or identify violation) of the rules, we inherit the soundness properties of CIFF thus obtaining a sound concrete tool for web checking.
At the end of the translation process the CIFF System can be successfully used to reason upon the (translation of the) web checking rules finding those XML/XHTML instances not fulfilling the rules, and representing errors as abducibles in abductive logic programs. The CIFFWEB tool also integrates a JAVA translator from web checking rules to both programs for checking and program for repairing.

However, abductive reasoning seems to be very suitable not only for identifying errors in a web site instance but also for *suggesting* possible repair actions for them. In this respect, abducibles may represent not only an error instance fired by an XML/XHTML instance violating a rule r (for checking) but also possible modifications (repairs) to that XML/XHTML data such that both r is fulfilled and no other rules are violated.

Following these observations, we have identified some type of errors, arising from the violation of web checking rules, which are suitable for abductive repair, and we have done a further mapping from web checking rules to another type of abductive logic program with constraints: *programs for repairing*. Again, we use programs for repairing as input for CIFF which either determines the fulfillment of the rules, or for suggesting appropriate repair actions. The soundness properties of CIFF are inherited also in the repair framework, thus obtaining a sound concrete tool for web repairing.
To our knowledge, the repairing feature is a novel feature with respect to the other existing tools for web verification.

1.3 Overview of the thesis

The first two chapters of the thesis contain background notions. In Chapter 2 we present a brief overview of Logic Programming, considering its extensions and its alternatives related to our work. The abductive extension of Logic Programming, i.e. ALP, is left as the subject of Chapter 3, where we present a brief overview of the various abductive approaches in the literature, including several abductive proof procedures (with constraints).

Chapter 4 is arguably the main chapter of the thesis: the CIFF proof procedure, together with its NAF extension, is formally defined and its soundness with respect to the three-valued completion semantics is proved.
In Chapter 5, the CIFF System is described, pointing out the main algorithms and data structures used in order to make the system efficient. This chapter ends with a comparison of CIFF, both at theoretical and at implementative level, with other, related, existing tools.

The CIFFWEB tool for web sites verification and repair is the main subject of Chapter 6. The web checking rules are formally defined as well as the translation function to abductive logic programs with constraints.

Finally in Chapter 7, we conclude our work, pointing out the future work.

Chapter 2

Preliminaries

This chapter summarizes the basic background about Logic Programming (LP) which, roughly speaking, can be described as the application of (a part of) mathematical logic in computer programming. In this view of logic programming, which can be traced at least as far back as John McCarthy's advice-taker proposal [130], logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver.
Fifty years of research have produced a large amount of logic-based formalisms to improve both the computational aspect of logic programming and its expressiveness. Surely, the most influential result, which made logic programming as it is understood nowadays, was the definition of the Prolog language [54, 116].
Since the birth of Prolog, logic programming has been extended in so many ways that a comprehensive survey of the logic programming is almost an impossible task. *Abductive logic programming* (ALP) which is the central field of this thesis is such an extension of logic programming and it will be described in detail in the next chapter. In this chapter we focus on the core of logic programming, briefly describing its foundations and the extensions which are closer to our work.
We introduce first order languages together with the notions of interpretations and models of first order theories. In sections 2.2 and 2.3 we introduce logic programming, defining its syntax and its classical semantics. Then we present the most influential alternative semantics for negation in logic programming. Section 2.5 briefly introduces an important extension of logic programming, namely Constraint Logic Programming (CLP). Finally we present a short overview of a distinct Logic Programming framework, namely Answer Sets Programming (ASP). For a detailed overview on these subjects, refer to [123], [118], [17], [88], [94], [87].

2.1 First Order Logic

First order logic has two basic aspects: syntax and semantics. We start defining the syntax, i.e. defining a *first order language*.
A *first order language* is based upon a *signature* (or an *alphabet*), which is defined as follows.

Definition 2.1. *A* signature *is composed of the following classes of symbols:*

1. Variables*: we denote variables by* $X, Y, Z...$
2. Constants*: we denote constants by* $a, b, c...$
3. Function symbols*: we denote function symbols by* $f, g, h...$
4. Predicate (or relation) symbols*: we denote predicate symbols by* $p, q, r...$
5. Two special symbols*:* true *and* false

6. Connectives*:* $\neg$ *(negation),* $\vee$ *(disjunction),* $\wedge$ *(conjunction),* $\rightarrow$*(implication),* $\leftrightarrow$ *(equivalence)*

7. Quantifiers*:* $\exists$ *(there exists),* $\forall$ *(for all)*

8. Punctuation symbols*: '(', ')', '[', ']', ','.*

□

The sets of connectives, quantifiers, punctuation symbols and the two special symbols are fixed. We assume that the set of variables is infinite and also fixed. These classes of symbols are called *logical symbols*. Constants, function symbols and predicate symbols may vary and they are called *non-logical symbols*. A first-order language is determined by its non-logical symbols. In this thesis we also assume that the set of function symbols in a first-order language is infinite unless stated explicitly otherwise.

Each function symbol and predicate symbol has a fixed *arity*, that is the number of its arguments. A function (predicate) symbol with n arguments is said to be a n-ary function (predicate) symbol. Constants can be regarded as 0-ary function symbols.

An important class of strings of symbols over a given alphabet is the class of *terms*.

Definition 2.2. *A* term *is:*

1. *a variable,*

2. *a constant, or*

3. *a compound term* $f(t_1, \ldots, t_n)$ *where* f *is a* n*-ary function symbol and* $t_1, \ldots, t_n$ *are terms.*

□

A tuple of variables $X_1, X_2, X_3 \ldots$ (resp. terms $t_1, t_2, t_3 \ldots$) is denoted as $\vec{X}$ (resp. $\vec{t}$). For example $f(\vec{t})$ is a shorthand for $f(t_1, \ldots, t_n)$.

Over the class of terms is built the class of *formulas*.

Definition 2.3. *A* formula *is:*

1. *a special symbol (*true *or* false*),*

2. $p(t_1, \ldots, t_n)$ *where* p *is a* n*-ary predicate symbol and* $t_1, \ldots, t_n$ *are terms,*

3. $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$, $\neg\phi$, *where* ϕ, ψ *are formulae,*

4. $\exists X.\phi$ *and* $\forall X.\phi$ *where* ϕ *is a formula and* X *is a variable.*

We say that a formula of the first two categories is an atomic formula *or simply an* atom*.*

A literal *is either* A *or* $\neg A$*,* A *being an atom. A* positive literal *is an atom. A* negative literal *is the negation of an atom.*
Given a literal L*, we denote by* $Pred(L)$ *the* n*-ary predicate of* L*.* □

We are ready to give the formal definition of *first order language*.

Definition 2.4. *A* first order language L *based on a given alphabet* A *is the set of all the formulae that can be constructed from that alphabet.* □

A formula or a term are called *ground* when they contain no variables. If they contain at least a variable they are called *non-ground*.
In a formula $\exists X.\phi$ (respectively $\forall X.\phi$), the quantifier $\exists$ (resp. $\forall$) *binds* X in ϕ. The *scope* of $\exists X$ (resp. $\forall X$) is defined to be ϕ. A variable in a formula ϕ which is not *bound* by a quantifier is called a *free* variable.

Definition 2.5 (Closed formula). *A formula ϕ is* closed *when there is no free variable X occurring in ϕ. Let $X_1, \dots, X_n$ be the free variables in a formula ϕ, we write $\exists(\phi)$ for $\exists X_1, \dots, \exists X_n(\phi)$ (the* existential closure *of ϕ) and we write $\forall(\phi)$ for $\forall X_1, \dots, \forall X_n(\phi)$ (the* universal closure *of ϕ). A closed formula is also called a* sentence. □

Definition 2.6 (Theory). *We define a* theory *T of a first order language L as a set of closed formulae of L. Each formula $F \in T$ is called an* axiom. □

2.1.1 Interpretations and Models

The semantics of first orders logic relies upon the concepts of *interpretations* and *models* which give the meaning to formulae of first order languages.

Definition 2.7 (Pre-interpretation). *A* pre-interpretation *J of a first order language L is composed of:*

- *a non-empty set of objects D called the* domain *of J,*
- *a set J_C containing, for each constant in L, an assignment of an element in D,*
- *a set J_F containing, for each n-ary function symbol in L, a mapping from D^n to D.*

When needed, we write J^D to highlight the domain D of the pre-interpretation J. □

Definition 2.8 (Interpretation). *An* interpretation *I of a first order language L consists of a pre-interpretation J^D of L together with a set I_P containing, for each n-ary predicate symbol in L, a mapping from D^n to the set $\{true, false\}$. We also say that I is based on J^D* □

Definition 2.9 (Variable assignment). *Let J^D be a pre-interpretation of a first order language L. A* variable assignment *α with respect to J assigns an element in D to each variable in L.*
Given a formula F of L, we denote by $F[X \mapsto d]$ the formula F' of L obtained by replacing all the occurrences of the variable X in F by the domain element d. □

Definition 2.10 (Semantics of terms). *Let I be an interpretation (based on J^D) of a first order language L, let α be a variable assignment and let t be a term. The* meaning *α_I of t is defined as follows:*

- *if t is a constant $c \in L$ then $\alpha_I(t) = J_C(c)$,*
- *if t is a variable $X \in L$ then $\alpha_I(t) = \alpha(X)$,*
- *if t is a compound term $f(t_1, \dots, t_n)$ then $\alpha_I(t) = f_J(\alpha_I(t_1), \dots, \alpha_I(t_n))$, where $f_J = J_F(f)$*

Notice that the semantics of a compound term is obtained by applying the function f_J to the meanings of its principal subterms, which are obtained by recursive application of the above definition.

We introduce now the semantics of formulae of a first order language L which associates a formula F to a truth value ($true$, $false$). In the following definition, the notation $I \models_\alpha Q$ means "Q is $true$ with respect to the interpretation I and the variable assignment α". In the same way $I \not\models_\alpha Q$ means "Q is $false$ with respect to the interpretation I and the variable assignment α".

Definition 2.11 (Semantics of formulae). *Let I be an interpretation (based on J^D) of a first order language L and let α be a variable assignment. The* meaning *of a formula $Q \in L$ is defined as follows:*

- *for each n-ary predicate symbol $p \in L$, given $p_I = I_P(p)$, we have that:*
  - *$I \models_\alpha Q = p(t_1, \dots, t_n)$ iff $p_I(\alpha_I(t_1), \dots, \alpha_I(t_n)) = true$*

  - $I \not\models_\alpha Q = p(t_1, \ldots, t_n)$ *iff* $p_I(\alpha_I(t_1), \ldots, \alpha_I(t_n)) = false$

- $I \models_\alpha Q = (\neg F)$ *iff* $I \not\models_\alpha F$
- $I \models_\alpha Q = (F \wedge G)$ *iff* $I \models_\alpha F$ *and* $I \models_\alpha G$
- $I \models_\alpha Q = (F \vee G)$ *iff* $I \models_\alpha F$ *or* $I \models_\alpha G$
- $I \models_\alpha Q = (F \rightarrow G)$ *iff* $I \not\models_\alpha F$ *or* $I \models_\alpha G$
- $I \models_\alpha Q = (F \leftrightarrow G)$ *iff* $I \models_\alpha (F \rightarrow G)$ *and* $I \models_\alpha (G \rightarrow F)$
- $I \models_\alpha Q = (\forall X.F)$ *iff* $I \models_{\alpha[X \mapsto d]} F$ *for each* $d \in D$
- $I \models_\alpha Q = (\exists X.F)$ *iff* $I \models_{\alpha[X \mapsto d]} F$ *for some* $d \in D$

where F, G *are formulae in* L *and* X *is a variable in* L.

□

It is easy to see that the semantics of a closed formula does not depend on a variable assignment α, but it depends only on the interpretation I. Hence, we can speak unambiguously of a semantics of a closed formula Q with respect to an interpretation I. I.e. we denote by $I \models Q$ (resp. $I \not\models Q$) the fact that a closed formula Q is *true* (resp. $false$) with respect to an interpretation I.

The introduction of the syntax and the semantics of formulae of a first order language L allows us to have a tool for describing and characterizing "worlds". While the language L describes the universe of a discourse, a "world", it is natural to ask whether a set S of closed formulae and an interpretation I give a proper account of this "world". This is the case if all the formulae of S are true with respect to I.

Definition 2.12 (Model). *Let* S *be a set of closed formulae of a first order language* L. *We say that* M *is a* model *of* S *if* M *is an interpretation of* L *such that for each formula* $F \in S$*:*

$$M \models F$$

We denote that M *is a model for* S *by* $M \models S$ *and that* M *is not a model for* S *by* $M \not\models S$.

□

Obviously there are infinitely many interpretations I for language L. However it may happen that none of them is a model of a set of closed formulae S (e.g. $S = F \wedge \neg F$). Conversely, it may happen that all of them are models of S (e.g. $S = F \vee \neg F$). We have the following classification of sets of closed formulae S.

Definition 2.13. *Let* S *be a set of closed formulae of a first order language* L. *We say that*

- S *is* consistent[1] *if there exists an interpretation* I *of* L *such that* $I \models S$*,*
- S *is* valid *if for every interpretation* I *of* L*,* $I \models S$*,*
- S *is* inconsistent *if there is no interpretation* I *of* L *such that* $I \models S$*,*
- S *is* nonvalid *if there exists an interpretation* I *of* L *such that* $I \not\models S$.

□

Given a set of closed formulae S, it is also interesting to characterize formulae F which can be "derived" from S. I.e. formulae F that are *true* in every model of S. This is the idea behind the concept of *logical consequence*.

[1] We use the word *consistent* and not *satisfiable* as the literature on Logic Programming generally does, because *satisfiable* will be used in another context in next chapters.

Definition 2.14 (Logical Consequence)**.** *Let S be a set of closed formulae and F be a closed formula of a first order language L. We say that F is a* logical consequence *of S (denoted again as $S \models F$) if, for every interpretation I of L, we have that:*

$$if\ \ (I \models S)\ then\ (I \models F)$$

□

Proposition 2.1. *Let S be a set of closed formulae and F be a closed formula of a first order language L. Then F is a logical consequence of S if and only if $S \cup \{\neg F\}$ is inconsistent.* □

Another important concept about the semantics of formulae is the concept of *logical equivalence*.

Definition 2.15 (Logical Equivalence)**.** *Let F and G be two formulae. We say that F and G are* logically equivalent *(denoted by $F \equiv G$) if and only if F and G have the same truth value for each interpretation I and variable assignment α.* □

2.1.2 Substitutions and unification

In this section we present two fundamental concepts which are the basis of the operational mechanisms in Logic Programming: *substitutions* and *unification*.

Definition 2.16 (Substitution)**.** *A* substitution *σ is a finite set of the form $\{X_1/t_1, \ldots, X_n/t_n\}$ where each X_i is a variable (distinct from each other) and each t_i is a term distinct from X_i. Each element X_1/t_1 is called a* binding *for X_i. If all the t_i are ground terms, then σ is called a* ground substitution*. The substitution given by the empty set is called the* empty substitution *and is denoted as ϵ.* □

Definition 2.17. *An* expression *E is either a term or a formula. A* simple expression *is either a term or an atom.*

Definition 2.18 (Instance)**.** *Let $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ be a substitution and let E be an expression. The expression $E\sigma$ is called the instance of E by σ and it is obtained by simultaneously replacing each occurrence of all the variables $X_1, \ldots, X_n$ in E by the corresponding terms $t_1, \ldots, t_n$. An instance containing no variables is called a* ground instance*.* □

Definition 2.19 (Variant)**.** *Let E and F be expressions. We say that E and F are* variants *if there exist substitutions σ and θ such that $E = F\sigma$ and $F = E\theta$. We also say E is a variant of F and F is a variant of E.*

□

Substitutions can be *composed* and they have elementary properties as follows.

Definition 2.20. *Let $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ and $\theta = \{Y_1/v_1, \ldots, Y_m/v_m\}$ be substitutions. The* composed substitution *$\sigma\theta$ is defined as the set:*

$$\{X_i/t_i\theta \mid i \in [1 \ldots n] \ \wedge\ X_i \neq t_i\theta\} \cup \{Y_j/v_j \mid j \in [1 \ldots m] \ \wedge\ Y_j \notin \{X_1, \ldots, X_n\}\}$$

□

Proposition 2.2. *Let σ, θ, γ be substitutions and let E be an expression. The:*

- $\sigma\epsilon = \epsilon\sigma = \sigma$
- $(E\sigma)\theta = E(\sigma\theta)$
- $(\theta\sigma)\gamma = \theta(\sigma\gamma)$

□

Example 2.1. *The followings are simple examples of substitutions:*

$$\begin{array}{lcl} p(f(X,Z),f(Y,a))\{X/a,Y/Z,W/b\} & = & p(f(a,Z),f(Z,a)) \\ p(X,Y)\{X/f(Y),Y/b\} & = & p(f(Y),b) \end{array}$$

The following, instead, is a simple example of a composition of substitutions:

$$\{X/f(Z),Y/W\}\{X/a,Z/a,W/Y\} \quad = \quad \{X/f(a),Z/a,W/Y\}$$

□

Substitutions are needed for defining a central procedural mechanism of logic programming: *unification.*

Definition 2.21. *Let S be a set of simple expressions and let be σ a substitution. σ is a* unifier *of S if $S\sigma$ is a singleton. σ is the* most general unifier (mgu) *for S if and only if for each unifier θ of S, there exists a substitution γ such that $\theta = \sigma\gamma$.* □

The search of a unifier (and in particular an mgu) of two expressions E and F, can be viewed as the process of solving the equation $E = F$. More generally, given a set of equations $\{E_1 = F_1, \ldots, E_n = F_n\}$, θ is a unifier of this set if $E_i\theta = F_i\theta$ for each $i \in [1, n]$. For example $\theta = \{X/a, Y/a\}$ is a unifier of the equation $f(X, g(Y)) = f(a, g(X))$.
Here we present the Martelli-Montanari algorithm [129] for finding an mgu of a set of equations. It is based on the concept of equations in *solved form*.

Definition 2.22. *A set of equations $\{X_1 = t_1, \ldots, X_n = t_n\}$ is in* solved form *if $X_1, \ldots, X_n$ are distinct variables none of which appear in $t_1, \ldots, t_n$.* □

A set of equations in solved form has the following interesting property, whose proof can be found in [137].

Proposition 2.3. *Let $\{X_1 = t_1, \ldots, X_n = t_n\}$ be a set of equations in solved form. Then $\{X_1/t_1, \ldots, X_n/t_n\}$ is an mgu of the solved form.* □

It is also important to introduce the concept of equivalence between sets of equations.

Definition 2.23. *Two sets of equations are said to be* equivalent *if they have the same set of unifiers.* □

The Martelli-Montanari algorithm, given a set of equations, gives out (if possible) an equivalent set of equations in solved form. It is clear that, by definition of two equivalent sets of equations, an mgu of the solved form is also an mgu of the non-solved form.

Definition 2.24. *[Unification Algorithm]*

Input: *a set of equations S_0.*

begin
$S := S_0$
repeat

select *an arbitrary $s = t$ in S*
case $s = t$ ***of***
(1) $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$: ***replace*** *by $s_1 = t_1, \ldots, s_n = t_n$*

(2) $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$: ***fail***

(3) $X = X$: ***delete*** *the equation*

(4) $t = X$ where t is not a variable: ***replace*** *by $X = t$*

(5) $X = t$ where X does not occur in t and X appears elsewhere in S:
replace *all the occurrences of X in S by t*

(6) $X = t$ where $X \neq t$ and X occurs in t: ***fail****.*
until *no action is possible in S*
end

Output*: S*

□

The following theorem states the soundness and the completeness of the above unification algorithm, see [137] for the proof.

Theorem 2.1 (Unification Theorem)**.** *Let S be a set of equations. The unification algorithm of definition 2.24 applied on S terminates and returns an equivalent solved form of S of* ***failure*** *if no such solved form exists.* □

Example 2.2. *The following set $\{f(X, g(Y)) = f(g(Z), Z)\}$ has a solved form, indeed:*

$\{f(X, g(Y)) = f(g(Z), Z)\} \Rightarrow$ (1)
$\{X = g(Z), g(Y) = Z\} \Rightarrow$ (4)
$\{X = g(Z), Z = g(Y)\} \Rightarrow$ (6)
$\{X = g(g(Y)), Z = g(Y)\}$

The following set $\{f(X, g(X)) = f(Z, Z)\}$, instead, does not have a solved form:

$\{f(X, g(X)) = f(Z, Z)\} \Rightarrow$ (1)
$\{X = Z, g(X) = Z\} \Rightarrow$ (4)
$\{X = Z, Z = g(Z)\} \Rightarrow$ (5)
fail

This is because Z is a proper subterm of $g(Z)$. □

2.2 Definite Logic Programming

The very basic idea behind Logic Programming is to use a computer to draw conclusions from declarative descriptions. Those descriptions, called *logic programs*, consist of finite sets of logic formulae. In order to achieve logic systems which exhibit interesting theoretical properties and which could be computationally attractive, it was clear that restrictions on the logic formulae were needed.
In this section we introduce the language of the *definite logic programs* which is the core of logic programming as it is intended today. The main limitation in a definite logic program is the lack of *negation*: only positive "objects" can be described. The absence of negation, however, allows the definition of a clear and universally accepted declarative semantics for definite logic programs: the *least Herbrand model* semantics as well as its procedural counterpart: the SLD-resolution [116].

2.2.1 Definite clauses, programs and goals

A particular type of declarative sentence describing both *facts* and *rules* about a domain and which is used to compose logic programs is the *clause*. In particular we restrict our attention to *definite clauses*.

Definition 2.25 (Definite clause)**.** *A* definite clause *is a a first order formula of the form:*

$$\forall (H \vee \neg A_1 \vee \ldots \vee \neg A_n)$$

where H and each A_i are atoms. A definite clause can be represented in implicative form (we use the notation $\leftarrow$ instead of $\rightarrow$ for convenience) as follows:

$$H \leftarrow A_1, \ldots, A_n$$

.

□

H is called the *head* of the clause and $B = L_1, \ldots, L_n$ the *body* of the clause. The empty head is equivalent to $false$ whereas the empty body is equivalent to $true$. In the first case we say that the clause is a *denial* whereas in the latter case we say that the clause is a *fact*. The empty clause is denoted as □.

Definition 2.26 (Definite Logic Program)**.** *A* definite logic program *is a finite set of definite clauses.* □

Example 2.3. *Let us consider the following sentences:*

1. *John is a parent of Mary*
2. *John is a male person*
3. *A father of a person is a male and he is a parent of that person.*

The following definite logic program can be used to express above sentences.

$parent(john, mary).$
$male(john).$
$father(X, Y) \leftarrow parent(X, Y), male(X).$ □

A logic program is a description of a "world" and it is used to draw conclusions about it, i.e. it is used to determine whether a certain sentence is a logical consequence of the program or not. In logic programming such a sentence is in the form of a *goal*.

Definition 2.27. *[Definite Goal] A* definite goal *(or a* definite query*) G is a clause of the form:*

$$\leftarrow A_1, \ldots, A_n$$

where each A_i is an atom and it is called a subgoal *of G.* □

In the sequel we will drop the word "definite" if it is clear from the context.

Example 2.4. *Consider the example 2.3. The goal*

$\leftarrow father(john, mary)$

is a logical consequence of the program. □

2.2.2 Semantics of Definite Logic Programming

As noted in the previous section, logic programs are used to check whether a goal is a logical consequence of a program or not. Thus, it is natural to characterize the *declarative* semantics of a logic program as the set of its logical consequences. In the case of definite logic programming, where both logic programs and goals are definite, it is possible to characterize such a set as the *least Herbrand model*. Furthermore, there is also an *operational* (or *procedural*) semantics that has been defined for definite logic programming: it is based on the *SLD-resolution* [116] principle. Both semantics are universally accepted as the "correct" semantics for definite logic programming and an equivalence result between the two has been shown. For further details and all the proofs of propositions and theorems in this section see, for example, [123, 137, 14].

The *least Herbrand model* semantics

The *least Herbrand model* semantics is based upon a special class of models of definite logic programs: the class of *Herbrand models*. The idea is to abstract from the actual meanings of the functors and the constants (0-ary functors) of the language and to focus on those interpretations (*Herbrand interpretations*) whose domain is the set of variable-free terms and the meaning of a ground term is the term itself.

Definition 2.28 (Herbrand Universe)**.** *Let L be a first order language. The* Herbrand Universe *U_L of L is the set of all ground terms which can be constructed out from the function symbols and the constants of L.* □

Definition 2.29 (Herbrand Base)**.** *Let L be a first order language. The Herbrand Base B_L of L is the set of all ground atoms which can be constructed from the predicate symbols and the ground terms in U_L.* □

Both Herbrand universe and Herbrand base are usually defined over a given logic program P. In this case it is assumed that the alphabet of the language is composed exactly of those symbols appearing in P.

Example 2.5. *Let us consider the following logic program P_{ODD}:*

$odd(s(0))$.
$odd(s(s(X))) \leftarrow odd(X)$.

In this case the Herbrand universe U_P and the Herbrand base B_P look as follows:

$$U_P = \{0, s(0), s(s(0)), \ldots\}$$

$$B_P = \{odd(0), odd(s(0)), odd(s(s(0))), \ldots\}$$

□

Definition 2.30. *[Herbrand Pre-interpretation and Herbrand Interpretation] A* Herbrand pre-interpretation *J for a first order language L is the pre-interpretation of L defined as follows:*

- *the domain is the Herbrand Universe U_L,*
- *J_C is the identity function for each $c \in L$,*
- *J_F is the mapping $U_L^n \mapsto U_L : (t_1, \ldots, t_n) \mapsto f(t_1, \ldots, t_n)$, for each n-ary function symbol f in L.*

A Herbrand interpretation *I for a first order language L is any interpretation based on the Herbrand pre-interpretation J for L.* □

Thus, Herbrand interpretations for a logic program P have fixed meanings for the constants and the functors: in order to specify a Herbrand interpretation I it suffices to list, for each n-ary predicate symbol p in P, the n-tuples $\langle t_1, \ldots, t_n \rangle$ of ground terms such that $I \models p(t_1, \ldots, t_n)$. In practice an Herbrand interpretation is a subset of the Herbrand base B_P.

Example 2.6. *The followings are Herbrand interpretations for the logic program P in the example 2.5:*

$$\begin{aligned} I_1 &= \{\emptyset\} \\ I_2 &= \{odd(s(0))\} \\ I_3 &= \{odd(s^n(0)) \mid n \in \{1, 3, 5, 7, \ldots\}\} \\ I_4 &= \{B_P\} \end{aligned}$$

□

Definition 2.31 (Herbrand Model). *A* Herbrand model *M for a set of closed formulae S of a first order language L is a Herbrand interpretation I for L such that:*

$$M \models S$$

□

Example 2.7. *Consider again the examples 2.5 and 2.6. The Herbrand interpretation I_1 is not a Herbrand model for P since $odd(s(0))$ is not true in I_1. Conversely $I_4 = B_P$ is clearly a Herbrand model for P. Let us consider I_2. It is not a model of P because there is a ground instance of the rule $odd(s(s(X))) \leftarrow odd(X)$, namely $odd(s(s(s(0)))) \leftarrow odd(s(0))$, such that all the premises are true. However $odd(s(s(s(0))))$ is not belonging to I_2. Finally I_3 is a Herbrand model of P. Indeed $odd(s(0))$ is true in I_3. Let us consider any ground instance $odd(s(s(t))) \leftarrow odd(t)$ of the rule, with $t \in U_P$. If $odd(t) \notin I_3$ the ground instance is obviously true. Otherwise we have that obviously also $odd(s(s(t))) \in I_3$ by definition.* □

In the above example $I_4 = B_P$ was a Herbrand model of P. Indeed, for each logic program P, B_P is a Herbrand model because it contains all the ground instances of each predicate of P. However, by its definition, B_P is not an "interesting" model. An interesting model should not contain more instances of those which effectively follow from a program P. This interesting model is the *least Herbrand model*, which characterizes the semantics of a definite logic program.
We have the following interesting property about Herbrand models.

Proposition 2.4. *Let P be a definite logic program and let $\mathcal{M}$ be a non-empty set of Herbrand models for P. Then the intersection M_P of all the Herbrand models in $\mathcal{M}$ is an Herbrand model.* □

Definition 2.32. *The* least Herbrand model *M_P of a logic program P is the intersection of all the Herbrand models of P.* □

Example 2.8. *The least Herbrand model of the program P in the example 2.5 is the Herbrand interpretation I_3 seen in the example 2.6 i.e.:*
$I_3 = \{odd(s^n(0)) \mid n \in \{1, 3, 5, 7, \ldots\}\}$ □

The fact that the least Herbrand model M_P contains effectively all the elements of B_P which follow from P was shown by Kowalski and Van Emden in [69] which characterized M_P as the set of all the logical consequences of P.

Theorem 2.2. *Let P be a definite logic program (with Herbrand base B_P). Then*

$$M_P = \{A \in B_P \mid P \models A\}$$

□

There is also a constructive characterization of the least Herbrand model which we present now.

Definition 2.33 (Grounded Logic Program). *Let P be a logic program, we define the* grounded logic program *$ground(P)$ as the logic program obtained by replacing all the non-ground clauses of P by all their ground instances.* □

Let us consider the set $\mathcal{H}$ of all the Herbrand interpretations of a logic program P. We introduce the *immediate consequence operator*.

Definition 2.34 (Immediate Consequence Operator). *Let P be a definite logic program and let I be a Herbrand interpretation of P. The mapping $T_P : \mathcal{H} \mapsto \mathcal{H}$ is defined as follows:*

$$T_P(I) = \{A \mid A \leftarrow A_1, \ldots, A_n \in ground(P) \text{ and } A_1, \ldots, A_n \subseteq I\}$$

□

To clarify the interest of T_P we need to (very briefly) recall the fixpoint theory. Further information on this topic can be found for example in [123, 169].
A set S is *partially ordered* if there exists a *partial order* $\leq$ on its elements. A partially ordered set S has a *least upper bound* $lub(S) = \top$ (i.e. $\forall s \in S.s \leq \top$) and a *greatest lower bound* $glb(S) = \bot$ (i.e. $\forall s \in S.\bot \leq s$).
A *complete lattice* L is a partially ordered set such that for each subset L' of L there exist both $glb(L')$ and $lub(L')$. Given a mapping $T : L \mapsto L$, an element $T \in L$ is a said a *fixpoint* of T if and only if $T(x) = x$. In a complete lattice L are ensured to exist both a *least fixpoint* $lfp(T)$, i.e. a fixpoint such that for each fixpoint $x \in L$ it holds that $x \leq lfp(T)$, and a and a *greatest fixpoint* $gfp(T)$, i.e. a fixpoint such that for each fixpoint $x \in L$ it holds that $gfp(T) \leq x$.

Given a mapping $T : L \mapsto L$ with L a complete lattice, $T \uparrow \alpha$ represents α iterative applications of T starting from $\bot$ whereas $T \downarrow \alpha$ represents α iterative applications of T starting from $\top$. We define the followings ($<$ is the standard order in the set of ordinal numbers):

- $T \uparrow 0 = \bot$
- $T \uparrow \alpha = T(T \uparrow (\alpha - 1))$, if α is a successor ordinal
- $T \uparrow \alpha = lub\{T \uparrow \beta : \beta < \alpha\}$, if α is a limit ordinal
- $T \downarrow 0 = \top$
- $T \downarrow \alpha = T(T \downarrow (\alpha - 1))$, if α is a successor ordinal
- $T \downarrow \alpha = glb\{T \downarrow \beta : \beta < \alpha\}$, if α is a limit ordinal

Furthermore we have that if T is *continuous*, $lfp(T) = T \uparrow \omega$ (where ω is the smallest limit ordinal). Conversely, $gfp(T) = T \downarrow \omega$ does not hold in general.

The interest in the fixpoint theory is given by two facts: (1) the set $\mathcal{H}$ of the Herbrand interpretations of a logic program P is a *compete lattice* and we define $lub(\mathcal{H}) = \oslash$ and $glb(\mathcal{H}) = B_P$ and (2) the immediate consequence operator T_P is continuous. Thus, T_P has a least fixpoint.
We have the following result which characterizes the Herbrand models of a definite logic program P in terms of T_P.

Proposition 2.5. *Let P be a definite logic program and let I a Herbrand interpretation. Then*

$$I \models P \text{ if and only if } T_P(I) \subseteq I.$$

□

Finally, we have the constructive characterization of the least Herbrand model as shown in the following result.

Proposition 2.6. *Let P be a definite logic program. Then*

$$M_P = lfp(T_P) = T_P \uparrow \omega$$

□

It is very interesting to note how the immediate consequence operator reflects the procedural interpretation of a clause. Indeed, the immediate consequence operator also represents a link between the declarative semantics of a definite logic program and the procedural semantics which is the topic of the next section.

2.2.3 SLD-resolution

As said in previous sections, the logic programming paradigm is based upon the concept of *query answering*, i.e. answering whether a given query (goal) G can be drawn from a logic program P.
While the least Herbrand model is a declarative characterization of all the logical consequences of a definite logic program P, the *SLD-resolution*, proposed by Kowalski in [116], is the inference rule which is used to check whether a given query G is a logical consequence of P or not. SLD-resolution is a refinement of the original procedure based on the resolution principle defined by Robinson [154] and it is the foundation of the operational semantics of definite logic programming. Its name stands for Linear resolution with Selection function for Definite clauses and it was given in [18].

The first step is to define what is a *correct answer* to a goal (query).

Definition 2.35. *Let P be a definite logic program, let G be a definite goal and let σ a substitution for the variables occurring in G. We say that σ is a correct answer for G with respect to P if and only if*

$$P \models \forall(G\sigma)$$

□

Intuitively, to prove that a definite goal G is a logical consequence of a definite logic program P the SLD-resolution process tries to compute a contradiction from the assumption that $\neg G$ holds.
In the following definitions we assume to have *selection function* $\mathcal{S}$, i.e. a function which selects a literal in a goal G.
The SLD-resolution inference rule can be stated as follows.

Definition 2.36 (SLD-Resolution)**.** *Let P be a definite logic program and let S be a selection function. The* SLD-resolution *inference rule is defined as:*

$$\frac{G = \leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_m \qquad \mathcal{S}(G) = A_i \qquad C = (B_0 \leftarrow B_1, \ldots, B_n)}{\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_m)\theta}$$

where C is a variant of a clause in P and θ is the mgu of A_i and B_0. □

Definition 2.37 (SLD-Derivation)**.** *Let P be a definite logic program, let G be a definite goal and let $\mathcal{S}$ be a selection function. An* SLD-derivation *of $P \cup \{G\}$ via $\mathcal{S}$ is a (finite or infinite) sequence of definite goals:*

$$G_0 = G \leadsto^{C_1} G_1 \ldots G_{n-1} \leadsto^{C_n} G_n \ldots$$

such that each G_{i+1} is derived directly from G_i by a single step of SLD-resolution via $\mathcal{S}$ and a variant C_i of a program clause. □

Definition 2.38 (SLD-Refutation)**.** *Let P be a definite logic program, let G be a definite goal and let $\mathcal{S}$ be a selection function. An* SLD-refutation *of $P \cup \{G\}$ via $\mathcal{S}$ is an SLD-derivation of $P \cup \{G\}$ via $\mathcal{S}$ such that the last goal in the SLD-derivation is the empty clause $\Box$. If $G_n = \Box$ we say that the SLD-refutation has* length n. □

Example 2.9. *Consider again the logic program P in the example 2.3 and the goal $G = \leftarrow father(john, mary)$. We have the following* SLD-refutation.

$$G_0 = \leftarrow father(john, mary)$$
$$\Big| \; C_1 = father(X_0,Y_0) \leftarrow parent(X_0,Y_0), male(X_0);\ \theta_1 = \{X_0/john, Y_0/mary\}$$
$$G_1 = \leftarrow parent(john, mary), male(john)$$
$$\Big| \; C_2 = parent(john,mary);\ \theta_2 = \epsilon$$
$$G_2 = \leftarrow male(john)$$
$$\Big| \; C_3 = male(john);\ \theta_3 = \epsilon$$
$$\square$$

□

Definition 2.39 (SLD Computed Answer)**.** *Let P be a definite logic program, let G be a definite goal and let $\mathcal{S}$ be a selection function. An* SLD computed answer *θ for $P \cup \{G\}$ via $\mathcal{S}$ is the substitution obtained by restricting the composed substitution $\theta_1\theta_2 \dots \theta_n$ to the variables occurring in G, where $\theta_1, \dots, \theta_n$ is the sequence of mgu's used in an SLD-refutation of $P \cup \{G\}$ via $\mathcal{S}$.* □

Example 2.10. *Consider again the logic program P in the example 2.3 and the goal $G = \leftarrow father(X, mary)$. We have the following* SLD-refutation*.*

$$G_0 = \leftarrow father(X, mary)$$
$$\Big| \; C_1 = father(X_0,Y_0) \leftarrow parent(X_0,Y_0), male(X_0);\ \theta_1 = \{X_0/X, Y_0/mary\}$$
$$G_1 = \leftarrow parent(X, mary), male(X)$$
$$\Big| \; C_2 = parent(john,mary);\ \theta_2 = \{X/john\}$$
$$G_2 = \leftarrow male(john)$$
$$\Big| \; C_3 = male(john);\ \theta_3 = \epsilon$$
$$\square$$

In this case the composed substitution is:

$$\theta_1\theta_2\theta_3 is \{X_0/X, Y_0/mary\}\{X/john\}\epsilon = \{X_0/john, X/john, Y_0/mary\}$$

Hence, the SLD-computed answer *of this example is $\theta = \{X/john\}$.*

□

Definition 2.40 (Failed SLD Derivation)**.** *A* failed SLD derivation *of $P \cup \{G\}$ is a finite SLD derivation whose last element is not empty and it cannot be resolved with any clause in P.*

□

The SLD-resolution is *sound* and *complete* with respect to the class of definite logic programs.

Theorem 2.3 (SLD Soundness)**.** *Let P be a definite logic program, let G be a definite goal and let $\mathcal{S}$ be a selection function. Then every SLD computed answer for $P \cup \{G\}$ via $\mathcal{S}$ is a correct answer for $P \cup \{G\}$.* □

Theorem 2.4 (SLD Completeness)**.** *Let P be a definite logic program, let G be a definite goal and let $\mathcal{S}$ be a selection function. If $P \models \forall(G\sigma)$, then there exists an SLD-refutation of $P \cup \{G\}$ via $\mathcal{S}$ with a computed answer θ such that $G\sigma$ is an instance of $G\theta$* □

The SLD-resolution process allows to answer to queries by means of the application a single inference step. For this reason the SLD-resolution is easily adaptable for computerized automation and it led to the development of the PROLOG system (PROgrammation en LOGic) by Colmerauer (as described in [54]).

A very important result is the independence of an SLD-refutation with respect to the choice of a selection function.

Theorem 2.5. *Let P be a definite logic program and let G be a definite goal. Suppose there is an SLD-refutation of $P \cup \{G\}$ with computed answer σ. Then, for any selection function $\mathcal{S}$ there exists an SLD-refutation of $P \cup \{G\}$ via $\mathcal{S}$ with computed answer σ' such that $G\sigma'$ is a variant of $G\sigma$.*

□

The following *SLD-tree* is a typical representation of SLD-derivations by means of a tree structure.

Definition 2.41. *Let P be a definite logic program, let G be a definite goal and let $\mathcal{S}$ be a selection function. An* SLD-tree *for $P \cup \{G\}$ via $\mathcal{S}$ is a tree satisfying the following properties:*

1. *each node of the tree is a (possibly empty) definite goal,*
2. *the root node is G,*
3. *let $A_1, \ldots, A_m, \ldots, A_n$ with $n \geq 1$ be a node in the tree and suppose that A_m is the selected atom by $\mathcal{S}$; then for each clause $H \leftarrow B_1, \ldots, B_k$ in P such that H and A_m are unifiable with mgu θ, the node has a child*

$$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_k, A_{m+1}, \ldots, A_n)\theta$$

4. *nodes which are the empty clause have no children.*

□

Most Prolog systems uses a depth-first strategy to build an SLD-tree with respect to a program P and a goal G always selecting the leftmost leaf of the current tree. Note that due to the theorem 2.5 we do not have to consider alternative selection functions in building an SLD-tree: this reduces dramatically the tree size.
Each branch of an SLD-tree is a derivation of $P \cup \{G\}$. Branches corresponding to successful derivations are called *success branches*, branches corresponding to infinite derivations are called *infinite branches* and branches corresponding to failed derivations are called *failure branches* (we will represent failures with ■).

Finally, we show that the declarative and the procedural characterizations of the semantics of definite logic programs coincide.

Definition 2.42. *Let P be a definite logic program. The success set S_P of P is defined by the set of atoms $A \in B_P$ such that there exists an SLD-refutation for $P \cup \{\leftarrow A\}$* □

Theorem 2.6. *Let P be a definite logic program and let $A \in B_P$. Then the following are equivalent:*

- *$A \in S_P$;*
- *$A \in T_P \uparrow \omega$;*
- *every SLD-tree for $P \cup \{\leftarrow A\}$ contains an SLD-refutation;*
- *$P \models A$.*

□

2.3 Negation in Logic Programming

Logic Programming is a declarative programming paradigm especially useful for representing knowledge at a very high level and for reasoning about it. Definite logic programs are able to express positive knowledge as well as the least Herbrand model semantics and the SLD-resolution are, respectively, the declarative and the operational semantics with which positive conclusions can be drawn from definite programs. However the lack of negative knowledge both in the goals and in the programs is a main issue in representing knowledge.
The introduction of negative knowledge brings with it a number of complications which nowadays have not been addressed completely yet.
In this section we show the "classical" approach of negation in logic programming. First we introduce the *Negation As Failure (NAF)* rule and the *completion* of a logic program which allow to infer negative conclusions from definite logic programs. Then we introduce *normal logic programs* which allow for negation in the body of their clauses. Finally we briefly present the SLDNF-resolution which extends SLD-resolution by addressing negation. Again, for further details and all the proofs of propositions and theorems in this section see, for example, [123, 137, 14].
In the next section we will see other approaches to address the problem of negation in logic programming.

2.3.1 The Negation As Failure (NAF) rule

Reiter in [151] was probably the first who defined an inference rule for treating negation: the *Closed World Assumption (CWA)* rule. The idea is very simple: if a ground atom A is not a logical consequence of a normal logic program P then the CWA rule simply infers $\neg A$.
The CWA rule was introduced in database context [151] where it is often a very natural rule to use: if an information is not explicitly present in the database then it is taken to be *false*. Indeed, databases can be seen as logic programs whose clauses are all ground facts. In this case any information is either represented by a fact in the program or it is considered *false*. In general, applying the CWA rule to a logic program P is not so simple. Indeed, due to the undecidability of the validity problem of first order logic, there is no algorithm which takes an arbitrary A as input and responds in a finite amount of time with the answer whether A is or is not a logical consequence of P. If A is not a logical consequence of P, its proof can loop forever.
The above observation lead us to restrict the attention to those A whose attempted proofs *fail finitely*.

Definition 2.43. *Let P be a definite logic program P and let G be a definite goal. An SLD-tree for $P \cup \{G\}$ which is finite and contains no success branches is called a* finitely failed SLD-tree *for $P \cup \{G\}$.* □

The *Negation As Failure (NAF)* proposed by Clark [48] and implemented in most Prolog systems, is based on the finitely failed SLD-tree.

Definition 2.44. *Let P be a definite logic program P and let A be an atom. The* Negation As Failure (NAF) *rule is defined as:*

$$\textit{if } P \cup \{A\} \textit{ has a finitely failed SLD-tree then } P \models \neg A.$$

□

The main problem with the NAF rule is that it unsound! This is because, given a definite logic program P we cannot infer a negative literal $\neg A$ because the program $P \cup \{A\}$ is always consistent. Indeed, its Herbrand base B_P is a Herbrand model for P.
From a procedural viewpoint, this is reflected in that SLD-resolution is no able to infer negative literals.
However, Clark [48] proposed to solve that issue introducing the *completion* of a logic program P which allows to infer negative conclusions from it. It is the subject of the next section.

2.3.2 Completion of a logic program

The *completion* of a logic program is a syntactical transformation of a logic program replacing implicative clauses ($\leftarrow$) by *if-and-only-if* statements ($\leftrightarrow$). This transformation allows to draw negative conclusions from a logic program as depicted in the following example.

Example 2.11. *Consider again the logic program in the Example 2.3. Suppose to replace the clause*
$father(X, Y) \leftarrow parent(X, Y), male(X).$ *by* $father(X, Y) \leftrightarrow parent(X, Y), male(X).$
The first form states that if *X is a male parent of Y* then *X is father of Y. The second form instead states that X is father of Y* if and only if *X is a male parent of Y. The latter implies that if X does not satisfy the conditions, then X is* not *father of Y. The same cannot be drawn from the implicative form.* □

The above idea is the basis of the *completion* of a logic program.

Definition 2.45 (Completed Definite Logic Program)**.** *Let P be a definite logic program and let = be a binary predicate symbol not occurring in P and whose intended interpretation is the identity relation. The Complete Definite Logic Program Comp(P) is obtained from P by the following transformations.*

1. *For each n-ary predicate p occurring in P replace each clause C of p of the form*

$$C = [p(t_1, \dots, t_n) \leftarrow A_1, \dots, A_m$$

   by the formula

$$F = [p(X_1, \dots, X_n) \leftarrow B]$$

   where

$$B = \exists \vec{Y}.[(X_1 = t_1), \dots, (X_1 = t_m), A_1, \dots, A_m)]$$

   where $\vec{Y_i}$ *are the variables occurring in the clauses C and* $X_1, \dots, X_n$ *are distinct variables do not occurring in C.*

2. *Let* $F_1, \dots, F_k$ *be all the formulae obtained by the previous step for an n-ary predicate p. Replace all* $F_1, \dots, F_k$ *by*

   - $p(X_1, \dots, X_n) \leftrightarrow B_1 \dots B_k$ *if* $k > 0$
   - $p(X_1, \dots, X_n) \leftrightarrow false$ *if* $k = 0$

3. *Add the following axioms to the above obtained set of formulae:*

   - $c \neq d$ *for all pairs c, d of distinct constants in L*
   - $\forall(f(X_1, \dots, X_n) \neq g(Y_1, \dots, Y_m))$ *for all pairs f, g of distinct function symbols in L*
   - $\forall(f(X_1, \dots, X_n) \neq c$ *for each function symbol f and constant c in L*
   - $\forall(t \neq X)$ *for each term t containing X and different from X*
   - $\forall((X_1 \neq Y_1) \vee \dots \vee (X_n \neq Y_n) \rightarrow f(X_1, \dots, X_n) \neq f(Y_1, \dots, Y_n))$ *for each function symbol f in L*
   - $\forall(X = X)$
   - $\forall(X = Y \rightarrow Y = X)$
   - $\forall(X = Y \wedge Y = Z \rightarrow X = Z)$
   - $\forall((X_1 = Y_1) \wedge \dots \wedge (X_n = Y_n) \rightarrow f(X_1, \dots, X_n) = f(Y_1, \dots, Y_n))$ *for each function symbol f in L*

- $\forall((X_1 = Y_1) \land \ldots \land (X_n = Y_n) \rightarrow p(X_1, \ldots, X_n) \rightarrow p(Y_1, \ldots, Y_n))$ *for each predicate symbol* p *in* L *(including* $=$*).*

Those axioms define the meaning of the predicate $=$ *and they are called the* Clark Equality Theory (CET)

□

Example 2.12. *Let* P *be the logic program of Example 2.3 augmented by the fact* $parent(ann, mary)$.

The completion $Comp(P)$ *of the logic program* P *is (we don't add explicitly CET):*

$$\begin{array}{rcl} father(X,Y) & \leftrightarrow & \exists V, W.[X = V, Y = W, parent(V,W), male(V)] \\ parent(X,Y) & \leftrightarrow & [X = john, Y = mary] \lor [X = ann, Y = mary] \\ male(X) & \leftrightarrow & [X = john] \end{array}$$

□

The *completion* of a definite logic program allows for a logical justification of the NAF rule whose soundness has been shown by Clark in [48].

Theorem 2.7. *Let* P *be a definite logic program and let* A *be an atom. If* $P \cup \{A\}$ *has a finitely failed SLD tree then* $Comp(P) \models \neg A$. □

NAF is also complete with respect to definite logic programs.

Theorem 2.8. *Let* P *be a definite logic program and let* A *be an atom. If* $Comp(P) \models \neg A$ *then* $P \cup \{A\}$ *has a finitely failed (fair) SLD tree.* □

The concept of *fairness* in SLD trees is defined as follows.

Definition 2.46. *An SLD-derivation is* fair *if it is either failed or, for every atom* A *in the derivation, (some further instantiated version of)* A *is selected within a finite number of steps.*

An SLD-tree is fair *if every branch of the tree is a fair SLD-derivation.*

□

2.3.3 SLDNF for Definite Logic Programs

The completion of a logic program allows to infer negative conclusions from it. Hence we could think of combining the SLD resolution with the NAF rule in order to get a procedural mechanisms for answering to *normal goals.*

Definition 2.47. *[Normal Goal] A* normal goal *(or a* normal query*)* G *is a clause of the form:*

$$\leftarrow L_1, \ldots, L_n$$

where each L_i *is a literal and it is called a* subgoal *of* G. □

In the remainder we will drop the word "normal", if it is clear from the context. In order to evidence the use of the NAF rule for handling negation, we denote a literal L as an atom A or as its negative counterpart *not* A where the symbol *not* denotes the use of NAF.
We are now ready to present the *SLDNF resolution for definite programs.*

Definition 2.48. *[SLDNF Resolution for Definite Programs] Let* P *be a definite logic program, let* G *be a normal goal and let* S *be a selection function. An* SLDNF derivation *of* $P \cup \{G\}$ *via* S *is a finite or infinite sequence of normal goals:*

$$G_0 = G \leadsto^{C_1} G_1 \ldots G_{n-1} \leadsto^{C_n} G_n \ldots$$

such that each G_{i+1} *is derived directly from* G_i *by one of the followings:*

1. *if $\mathcal{S}(G_i)$ is an atom, then G_{i+1} is obtained from G_i by a single step of SLD-resolution with a variant C_i of a program clause;*
2. *if $\mathcal{S}(G_i) = not\ A$ and the goal $\leftarrow A$ has a finitely failed SLD tree, then G_{i+1} is obtained from G_i by removing A (and C_i is empty)*

□

The basic idea in SLDNF resolution is that, when a negative literal *not* A is selected, a new SLD derivation, called a *consistency derivation* starts from A. Each consistency derivation has to fail finitely in order to prove *not* A.

Example 2.13. *Consider the following logic program P, where the predicate $on(X, Y)$ stands for "a block X is on the top of a block Y":*

$on(c, b)$.
$on(b, a)$.
An SLDNF refutation of the goal

$\leftarrow on(X, Y), not\ on(Z, X)$

is the following:

$$\leftarrow on(X, Y), not\ on(Z, X)$$
$$\Big|\ C_1 = on(c,b)\ \ \theta_1 = \{X/c, Y/b\}$$
$$\leftarrow not\ on(Z, c) \qquad\qquad \leftarrow on(Z, c)$$
$$\Big|\ C_2 = \epsilon\ \ \theta_2 = \epsilon \qquad\qquad \Big|$$
$$\square \qquad\qquad \blacksquare$$

The SLDNF computed answer is $\theta = \{X/c, Y/b\}$. As we can see in the picture, to prove not $on(Z, c)$, a new SLD derivation is fired starting from $\leftarrow on(Z, c)$. As the latter gaol fails finitely (because there is no fact in the program which unifies to it), then not $on(Z, c)$ succeeds.
If at the first we have would selected $on(b, a)$, the resulting goal $\leftarrow not\ on(Z, b)$ would have failed due to the presence of $on(c, b)$ in P.

□

The soundness of SLDNF is shown by the following theorem.

Theorem 2.9. *Let P be a definite logic program and let G a normal goal. If $P \cup \{G\}$ has an SLDNF refutation with the computed answer θ, then*

$$Comp(P) \models \forall(G\theta)$$

□

However the SLDNF is not complete. This is due to the quantification of variables in the negative literals. Let us consider the goal $G = \leftarrow$ *not* $on(X, Y)$ with the previous "blocks" example. We expect several answers for this goal, for example $\{X/a, Y/b\}$. However the only SLDNF derivation for $P \cup \{G\}$ fails because there is an SLD refutation for the $\leftarrow on(X, Y)$. The problem is that variables into negative literals are universally quantified and a consistency derivation is a "universal test" which does not bind the variables.
To ensure (a restricted form of) completeness we have to use a *safe* selection function.

Definition 2.49 (Safe Selection Function)**.** *A selection function S is* safe *is it selects either an atom or a ground negative literal.* □

Here, we don't present the completeness theorem of SLDNF for definite logic programs, however we assume in the sequel that a selection function S will be always safe.

2.3.4 Normal Logic Programming

Being able to infer negative conclusions from a program, it is natural to extend the syntax of programs allowing for negative information in the bodies of the clauses. Such programs are called *normal logic programs* and their clauses are *normal clauses*.

Definition 2.50 (Normal Clause and Normal Logic Program)**.** *A* normal clause *is a clause of the form:*

$$H \leftarrow L_1, \ldots, L_n$$

where H is an atom and each L_i is a literal.

A normal logic program *is a finite set of normal clauses.*

□

Example 2.14. *We could extend the logic program $P = \{on(c,b).\ on(b,a).\}$ of the Example 2.13 with the clauses:*

$blocked(X) \leftarrow on(Y,X).$
$on_top(X) \leftarrow not\ blocked(X).$

stating that (1) "if a block Y is on a block X then X is blocked" and (2) "if a block X is not blocked then it is the top of a tower of blocks".

□

In the literature there are a number of proposals for how normal logic programs (and more generally negation in logic programs) can be interpreted. In this section we continue the "classical" approach, i.e. we extend the approach of both completion and SLDNF to normal logic programs. We will see later other approaches to negation.
The notion of completion of a logic program is still valid for normal logic programs.

Example 2.15. *2.16 The completion of the program P of the Example 2.14 is as follows:*

$$\begin{array}{rcl} on(X,Y) & \leftrightarrow & [X=c, Y=b] \vee [X=b, Y=a] \\ blocked(X) & \leftrightarrow & \exists V, W.[X=W, on(V,W)] \\ on_top(X) & \leftrightarrow & \exists V.[X=V, not\ blocked(V)] \end{array}$$

However, the completion of normal logic programs is not always consistent. Consider for example the program P:

$$p \leftarrow not\ p$$

The completion of P is clearly inconsistent, then no Herbrand model can be found for it. The idea is to restrict our attention to normal logic programs whose completion is consistent.
We present here some important classes of consistent normal logic programs: *call-consistent*, *stratified* and *locally stratified* normal logic programs. We need the notion of *dependency graph* of a logic program.

Definition 2.51 (Dependency Graph)**.** *A dependency graph G_P of a logic program P is defined as follows:*

- *the nodes of G_P are the predicate symbols occurring in P*
- *for each clause in P of the form $p(t_1,\ldots,t_n) \leftarrow B$ such that $q(v_1,\ldots,v_m) \in B$, the graph G_P has a directed edge labeled '+' from the node for the n-ary predicate p to the node for the m-ary predicate q*
- *for each clause in P of the form $p(t_1,\ldots,t_n) \leftarrow B$ such that not $q(v_1,\ldots,v_m) \in B$, the graph G_P has a directed edge labeled '−' from the node for the n-ary predicate p to the node for the m-ary predicate q*

□

The first important class of logic programs is the *call-consistent* class.

Definition 2.52 (Call-Consistent Logic Program). *A logic program P is* call-consistent *if and only if its dependency graph G_P contains no cycle having an odd number of edges.*

□

The next classes are all subsets of the call-consistent class and they are all based on the notion of *stratification*.

Definition 2.53 (Stratification). *Let P be a logic program. A partition $\pi_1, \ldots, \pi_n$ of the set of all predicate symbols of P is called a* stratification *of P if for each clause $H \leftarrow B$ and for any $p \in \pi_k, i \in [1, n]$ then if $p(\vec{t})$ occurs in H we have that*

- *if $q(\vec{v})$ occurs as an atom in B then $q \in \pi_i, i \in [1, k]$*
- *if not $q(\vec{v})$ occurs as a negative literal in B then $q \in \pi_i, i \in [1, k-1]$*

If a predicate p does not occur in the head of any clause of P, then $p \in \pi_1$

□

Definition 2.54 (Stratified Logic Program). *A logic program P is* stratified *if it admits a stratification $\pi_1, \ldots, \pi_n$, i.e. $P = \pi_1 \cup \ldots \cup \pi_n$.*

Example 2.16. *Let us consider again the program P of the Example 2.14. The following partition π_1, π_2 is a stratification:*

π_1: $on(c, b).$
$on(b, a)$
$blocked(X) \leftarrow on(Y, X).$
π_2 : $on_top(X) \leftarrow not\ blocked(X).$

□

We can extend the notion of stratification of a logic program P considering the grounded logic program $ground(P)$.

Definition 2.55 (Local Stratification). *Let P be a logic program with its Herbrand base B_P. A local stratification stratum is a function which maps the set B_P to the set of countable ordinals and such that for each $a \in B_P$:*

$$stratum(not\ a) = stratum(a) + 1$$

□

Definition 2.56. *Let P be a logic program. A clause $H \leftarrow B$ of P is* locally stratified *with respect to a local stratification stratum if for each ground instance $H_G \leftarrow B_G$, for each literal l in B_G we have that:*

$$stratum(H_G) \geq stratum(l)$$

The whole program P is locally stratified *if there exists a local stratification stratum such that for each clause C in P, C is locally stratified with respect to stratum.*

Again we can easily observe that a stratified logic program is a locally stratified logic program.

A program belonging to one of the above classes is ensured to have a Herbrand model, as shown by Sato in [159].

Theorem 2.10. *If P is a call-consistent logic program, then $Comp(P)$ has a Herbrand model.*

□

Having a Herbrand model we could think about a restatement of the least Herbrand model for consistent normal logic programs. The *immediate consequence operator* T_P is redefined as follows:

Definition 2.57 (Immediate Consequence Operator)**.** *Let P be a normal logic program and let I be a Herbrand interpretation of P.*

$$T_P(I) = \{A \mid A \leftarrow L_1, \ldots, L_n \in ground(P) \;\; and \;\; \forall i \in [1, n], I \models L_i\}$$

□

We can extend the result of Proposition 2.5 to consistent normal logic programs.

Proposition 2.7. *Let P be a normal logic program and let I be a Herbrand interpretation. Then*

$$I \models P \text{ if and only if } T_P(I) \subseteq I.$$

□

The following result (originally made for definite logic programs) by Apt and Van Emden [18] characterizes the Herbrand models of $Comp(P)$ in terms of the operator T_P.

Theorem 2.11. *For every Herbrand interpretation I, $I \models Comp(P)$ if and only if $T_P(I) = I$.* □

Given a definite logic program P, it is natural to choose its least Herbrand model as the *standard* model of P.
Switching to normal logic programs, the situation is very different because the existence of a least Herbrand model is not ensured. Call-consistent logic programs are ensured to have *minimal* Herbrand models (with respect to sets inclusion) as shown by Dung in [65]. The problem is that call-consistent normal logic program can have more than one minimal Herbrand model.

Example 2.17. *Let us consider the following stratified logic program P:*
$p \leftarrow not\ q$
There are two minimal Herbrand models of P:

$$M_1 = \{p\} \qquad\qquad M_1 = \{q\}$$

□

Having more than one minimal Herbrand model, we need a characterization of what an intended model should contain. This characterization is given by the concept of *supported* interpretation.

Definition 2.58. *Let P be a normal logic program and let I be a Herbrand interpretation. I is said to be* supported *if $\forall H \in I$, I is such that:*

$$\exists L_1, \ldots, L_n.(H \leftarrow L_1, \ldots, L_n \in ground(P) \text{ and } I \models L_1, \ldots, L_n)$$

If I is a model of P, it is called a supported *model of P.*

□

We have also the following result.

Proposition 2.8. *Let P be a normal logic program and let I be a Herbrand interpretation. I is a supported model of P if and only if $T_P(I) = I$.* □

For the class of stratified logic programs, we can define a supported model as follows.

Let $T_P \uparrow \omega(I)$, where I is a Herbrand interpretation, denote the limit of the sequence:

$$\begin{array}{rcl} T_P \uparrow 0(I) & = & I \\ T_P \uparrow (n+1)(I) & = & T_P(T_P \uparrow n(I)) \cup T_P \uparrow n(I) \end{array}$$

Given a stratified program $P = \pi_1 \cup \ldots \cup \pi_n$, it is possible to define a Herbrand model of P as follows:

$$\begin{array}{rcl} M_1 & = & T_{\pi_1} \uparrow \omega(\oslash) \\ M_2 & = & T_{\pi_2} \uparrow \omega(M_1) \\ & \vdots & \\ M_n & = & T_{\pi_n} \uparrow \omega(M_{n-1}) \end{array}$$

The supported model $M_P = M_n$ is called the *standard* model of P because, as shown in [16], that M_P is a *minimal* Herbrand model of P and it is not depend on the stratification of P.

Example 2.18. *Let us consider again the program P of the Example 2.14. The standard model $M_P = M_2$ is obtained as follows:*

$$\begin{array}{ll} M_1 = & \{on(c,b), on(b,a), blocked(b), blocked(a)\} \\ M_2 = & \{on_top(c)\} \cup M_1 \end{array}$$

However another (rather counterintuitive) minimal Herbrand model is:

$$M = \{on(c,b), on(b,a), blocked(b), blocked(a), blocked(c)\}$$

□

2.3.5 SLDNF resolution for Normal Logic Programs

As we have adapted the semantics of the least Herbrand model, through the minimal Herbrand models, to (a subclass of) normal logic programs, we could extend the SLDNF resolution in order to answering goals with respect to normal logic programs.
We do not present the details of the SLDNF resolution for normal logic programs because it is quite long and it is not central to our dissertation. However, the procedure is very similar to SLDNF for definite logic programs: the key difference is that when a negative literal *not* A is selected, then a new (consistency) SLD derivation is fired, starting from $\leftarrow A$. If A has an SLD refutation with the empty substitution as the computed answer then *not* A fails finitely; else if A fails finitely then *not* A succeeds. During the derivation starting from $\leftarrow A$, another negative literal can be selected, thus another SLD derivation is fired.

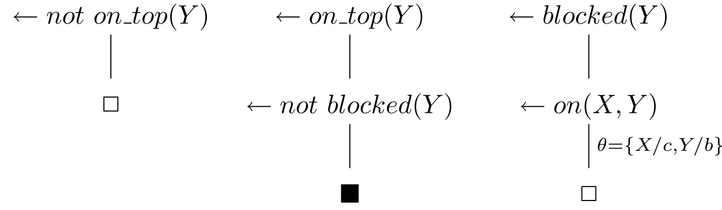
Example 2.19. *Let us consider again the program P of the Example 2.14 and the goal $G = \leftarrow$ not $on_top(b)$. The following is an SLDNF refutation for $P \cup \{G\}$.*

$$\begin{array}{ccc} \leftarrow not\ on_top(b) & \leftarrow on_top(b) & \leftarrow blocked(b) \\ | & | & | \\ \square & \leftarrow not\ blocked(b) & \leftarrow on(X,b) \\ & | & |\ \theta = \{X/c\} \\ & \blacksquare & \square \end{array}$$

Note that the SLD refutation starting from $\leftarrow blocked(b)$ has the empty substitution as computed answer. Then $\leftarrow on_top(b)$ fails finitely. □

The importance of having the empty substitution as computed answer for a consistency SLD derivation starting from $\leftarrow A$ and fired in order to prove $\leftarrow$ *not* A is crucial for the soundness of the SLDNF resolution for normal logic programs.

Example 2.20. *Let us consider again the program P of the Example 2.14 and the goal $G = \leftarrow$ not $on_top(Y)$. Consider drop the "empty substitution" condition from SLDNF resolution. The following would be an SLDNF refutation for $P \cup \{G\}$.*



In this case, having an SLD refutation for $\leftarrow blocked(Y)$ *(with computed answer* $\{Y/b\}$*), we draw that* for each *block* Y, Y *is not on the top. This is obviously unsound.*

$\Box$

In order to get soundness, SLDNF should be implemented with a safe selection function. However, SLDNF implemented with a safe selection function still shows problems of *completeness*. A main problem of completeness is represented by goals composed only of negative non-ground literals. This is the problem of *floundering*.
There is no optimal solution to avoid floundering because in general the problem of checking statically if, given a selection function, a program will lead to floundering is undecidable. A solution which limits the expressiveness of the procedure is to restrict the input programs to allowed programs, i.e. programs which have sufficient syntactical conditions ensuring the absence of floundering. One simple (and weak) condition is to impose that variables appearing in a negative literal in a clause, appear also in positive literal of the body of the clause.

Finally, we state the soundness theorem for the SLDNF resolution for normal logic programs.

Theorem 2.12. *Let* P *be a normal logic program, let* $\leftarrow G$ *be a normal goal and let* $\mathcal{S}$ *be a safe selection function.*

- *if the SLDNF resolution with respect to* $P \cup \{G\}$ *has a refutation with computed answer* θ *then*
$$Comp(P) \models \forall(G\theta)$$
- *if the SLDNF resolution with respect to* $P \cup \{G\}$ *fails finitely then*
$$Comp(P) \models \forall(\neg G)$$

$\Box$

2.4 Alternative Semantics of Normal Logic Programming

The completion semantics and the SLDNF-resolution do not resolve entirely the problem of negation which has been the subject of a number of researches during the last thirty years.
A main issue is to extend the notions seen in previous section to the whole class of normal logic programs. One way this has been addressed is by switching to *three-valued models*. Intuitively, in this setting a term can have three truth-values *true*, $false$ or *unknown*. These semantics allows for defining models of a certain domain with a partial knowledge about the truth values of the elements of the domain. These models are called *partial models* and can be seen as a generalization of the two valued models (*total models*) in which all the elements have a defined truth value.
However, the completion semantics seen in previous section is not the only way of interpreting normal logic programs. Alternative semantics for normal logic programs which in some contexts better capture the meaning of the negative information have been proposed. Probably, the most relevant alternative approach is the *stable models semantics* approach [88] (which is based on the concept of *supported* interpretations, see Definition 2.58).

In front of the huge amount of research on this topic there is no unanimity on which is the best semantics for normal logic programs. The objective here, is not to do a comprehensive survey

on this subject, and we present briefly the approaches that we will use in our dissertation. In particular we will present the *three-valued completion semantics* (extending the notions seen in the previous section to a three-valued logic), *stable models semantics* and finally the *well founded semantics* which extends the stable models semantics to three-valued models. For further details and for further semantics, see [123], [118], [17], [85], [86], [155], [25], [88]. Another approach for defining a semantics for negation is the argumentation approach (see for example, [65] and [30]) which we do not include in this survey as it is not so relevant for our scopes.

2.4.1 Three-valued completion semantics

The three-valued completion semantics was proposed by Fitting in [79]. Important results for this semantics was subsequently given by Kunen in [118].

In the three-valued completion semantics an element of the Herbrand base of a logic program P can have three possible values: *true*, *false* and *unknown*. This approach is based on a three valued logic due to Kleene [112] in which three values are assumed: 0 representing *false*, 1 representing *true* and $\frac{1}{2}$ representing *unknown*.
The three-valued completion semantics ensures the existence of a least (three-valued) Herbrand model for any normal logic program P.

We introduce now a mapping $|\cdot|$ from B_P to the set $\{0, \frac{1}{2}, 1\}$ which defines the meaning of ground formulae as follows:

$$\begin{array}{lcl} \mid \neg F \mid & = & 1- \mid F \mid \\ \mid F \wedge G \mid & = & min(\mid F \mid, \mid G \mid) \\ \mid F \leftarrow G \mid & = & \begin{cases} 1 & \text{if } \mid F \mid \ \geq \ \mid G \mid \\ 0 & \text{if } \mid F \mid \ < \ \mid G \mid \end{cases} \end{array}$$

Note that $\leftarrow$ receives a two-valued interpretation, i.e. a formula $F \leftarrow G$ can be either *true* or *false*. When a ground formula evaluates to 1 we say that it is *true relative to* $|\cdot|$, when it evaluates to 0 we say that it is *false relative to* $|\cdot|$. The mapping $|\cdot|$ can be conveniently presented in the form of a three-valued Herbrand interpretation.

Definition 2.59. *Let P be a logic program with its Herbrand base B_P. A pair $I = (I^+, I^-)$, with $I^+, I^- \subseteq B_P$, is called a* three-valued Herbrand interpretation. *I^+ are atoms assumed to be true, and I^- are atoms assumed to be false. All the atoms in the set $B_P - (I^+ \cup I^-)$ are assumed to be unknown.*
Moreover we say that:

- *I is* total *if $I^+ \cup I^- = B_P$*
- *I is* consistent *if $I^+ \cap I^- = \oslash$*

□

Note that every standard Herbrand interpretation I, as in definition 2.30 can be identified with the total, consistent three-valued Herbrand interpretation $(I, B_P - I)$.

Definition 2.60 (Information Ordering)**.** *Let I, J be three-valued Herbrand interpretations. The* information ordering $\sqsubseteq$ *is the natural ordering defined as follows:*

$$I \sqsubseteq J \textit{ if and only if } I^+ \subseteq J^+ \textit{ and } I^- \subseteq J^-$$

□

We introduce now the three-valued immediate consequence operator Φ which is intuitively the three-valued counterpart of the operator T_P.

Definition 2.61. *Let P be a logic program and let I be a three-valued Herbrand interpretation. We define*

$$\Phi_P(I) = (T, F)$$

where

$T = \{H \mid \exists L_1, \ldots, L_n.(H \leftarrow L_1, \ldots, L_n \in ground(P) \; and \, L \; is \; true \; in \; I)\}$
$F = \{H \mid \forall L_1, \ldots, L_n.(if \, H \leftarrow L_1, \ldots, L_n \in ground(P) \; then \, L_1, \ldots, L_n \; is \, false \; in \, I)\}$

□

The following result summarizes the interesting properties of Φ.

Proposition 2.9. *Let P be a logic program.*

1. *if I is a consistent three-valued Herbrand interpretation then $\Phi_P(I)$ is a consistent three-valued Herbrand interpretation;*
2. *Φ is monotonic;*
3. *Φ, in general, is not continuous.*

□

The next step is to define the meaning of the completion in the three-valued logic. For this purpose we also need to assign meaning both to the disjunction and equivalence connectives and to quantifiers.

$$| F \vee G | = max(| F |, | G |)$$
$$| F \leftrightarrow G | = \begin{cases} 0 & \text{if } | F | = | G | \\ 1 & \text{if } | F | \neq | G | \end{cases}$$

Note that $\leftrightarrow$ receives a two-valued interpretation as for $\leftarrow$. Quantifiers are interpreted in the standard way.
The new interpretation of connectives and quantifiers allows us to determine when a first-order formula F is *true* in a three-valued interpretation I, written as $I \models_3 F$. We have the following results.

Proposition 2.10. *Let P be a logic program. For every three-valued Herbrand interpretation I we have that*

$$I \models_3 Comp(P) \; \textit{if and only if} \; \Phi_P(I) = I$$

Moreover we have that $lfp(\Phi_P)$ is a consistent three-valued model of $Comp(P)$.

□

The three-valued completion semantics is defined for each normal logic program P by ensuring the existence of a three-valued model for its completion $Comp(P)$.
The last result of Kunen [118] is an important result showing that the truth value of a formula with respect to $Comp(P)$ can be determined by iterating a *finite* number of times the operator Φ_P starting from the empty three-valued Herbrand interpretation $(\oslash, \oslash)$.

Proposition 2.11. *Let P be a logic program and let F be a formula not containing $\leftarrow$ and $\leftrightarrow$. Then*

$$Comp(P) \models_3 F \; \textit{if and only if} \; \Phi_P((\oslash, \oslash)) \uparrow n \models_3 F \; \textit{for some finite} \; n.$$

□

Example 2.21. *Consider the normal logic program $P = p \leftarrow not\ p$. We have that $Comp(P)$ is inconsistent with respect to the standard two-valued semantics. However it is consistent in the three-valued completion semantics. We have that:*

$$\Phi_P((\oslash, \oslash)) = (\oslash, \oslash)$$

which indeed is a model of P.

□

The SLDNF resolution is still sound with respect to the three-valued completion semantics. Moreover, if we consider only allowed normal logic programs (i.e. programs such that variables occurring in a negative literal of a clause C also appear in a positive literal in the body of C), the SLDNF resolution is also complete.

Example 2.22. *Let us consider the following (allowed) logic program P together with its completion:*

$$\begin{array}{llllll} P: & p \leftarrow q. & Comp(P): & p & \leftrightarrow & [q \vee not\ q] \\ & p \leftarrow not\ q. & & q & \leftrightarrow & q \\ & q \leftarrow q. & & & & \end{array}$$

The SLDNF resolution, with respect to the goal $G = p$. loops infinitely. However we have that p is a logical consequence of $Comp(P)$ in standard completion semantics but it is not in the three-valued completion semantics. This is because in three-valued logic $q \vee \neg q$ is not equivalent to true. □

2.4.2 Stable models semantics

The *stable models semantics* for normal logic programs was proposed by Gelfond and Lifschitz in [88] and it introduces a different characterization of negation in normal logic programs from that of the completion semantics. It is based on the following definition of the *reduct* Π_P^I of a logic program P with respect to an interpretation I.

Definition 2.62. *Let P be a logic program and I a set of atoms. Consider the grounded logic program $ground(P)$. The* reduct *Π_P^I of P with respect to I is the logic program obtained from $ground(P)$ by deleting:*

- *all the clauses $H \leftarrow B$ of $ground(P)$ such that not $p(\vec{t}) \in B$ and $p(\vec{t}) \in I$*
- *all the negative literals in the body of the remaining clauses*

□

Clearly, each reduct of a logic program P is a definite logic program, hence it admits a least Herbrand model.

Definition 2.63 (Stable Model Operator)**.** *Let P be a logic program and I a set of atoms. We define the* Stable Model operator *St_P with respect to P as the least Herbrand model of the reduct Π_P^I, i.e.:*

$$St_P(I) = M_{\Pi_P^I}$$

□

Definition 2.64 (Stable Model)**.** *Let P be a logic program and I an Herbrand interpretation. We say that I is a stable model of P if and only if*

$$I = St_P(I)$$

□

Example 2.23. *Let us consider the logic program P:*

$p \leftarrow not\ q.$
$q \leftarrow q.$

The unique stable model of P is $M_1 = \{p\}$. □

The intuition behind the definition of a stable model is to consider the above P and I respectively as a set of *rules* and a set of *beliefs* for a *rational* entity (usually called an *agent* in the literature). Then any clause that has a literal *not* A in its body, with $A \in I$ can be deleted as it is useless for the agent. Moreover any literal *not* A with $A \notin I$ is trivial with respect to the agent beliefs, thus can be removed. This yields the reduct Π_P^I. Then if I happens to be precisely the set of atoms which follow logically from this simplified set of premise, then the set I of beliefs is *stable*. Hence, citing [88], "stable models are possible sets of beliefs an agent might hold".

Note that stable models are supported models but the contrary does not hold. For example the program P of the Example 2.23 has two supported models, namely $M_1 = \{p\}$ and $M_2 = \{q\}$, but only a stable model, namely M_1.

In general, a normal logic program P can have one, none or many stable models.
Consider again the logic program $P = p \leftarrow not\ p$. It has no stable models because the interpretations $I = \oslash$ and $I = \{p\}$ which are the only possible interpretations of P, do not coincide with the least Herbrand models of the corresponding reducts of P.

On the other hand, consider the logic program P defined as:

$$p \leftarrow not\ q$$
$$q \leftarrow not\ p$$

The stable models for P are both $M_1 = \{p\}$ and $M_2 = \{q\}$.

A class of logic programs for which the existence of a unique stable model is ensured is the class of *locally stratified* logic programs for which Gelfond and Lifschitz in [88] presented the following result.

Theorem 2.13. *If a logic program P is locally stratified then it has a unique stable model.* □

The relationships between stable models and the standard model defined for stratified logic programs, is shown by the following result.

Theorem 2.14. *Let P be a stratified logic program. Its standard model M_P coincides with the unique stable model of P.* □

Finally, we have the following result, shown by Dung in [65] which gives a sufficient condition for the existence of at least a stable model of logic program P.

Theorem 2.15. *If a logic program P is call-consistent then it has at least one stable model.* □

2.4.3 Well-founded semantics

The well-founded semantics was proposed by Van Gelder, Ross and Schlipf in [86]. This is a widely accepted semantics for normal logic programs based on three-valued models.
The well-founded semantics, as we will see, has strong relationships with the stable models semantics. In this brief presentation we follow the fixpoint characterization given in [85]. This characterization is based on the stable model operator St.
Given a logic program P and a set of atoms I, the operator $St_P^2(I) = St_P(St_P(I))$ is defined as the Stable Model operator applied twice. The following result, shown in [85], holds for St^2.

Proposition 2.12. *Let P be a logic program. The operator St^2_P is monotonic. Then the least fixpoint $lfp(St^2_P) = I_P$ exists as well as the greatest fixpoint $gfp(St^2_P)$. Moreover we have that*

$$gfp(St^2_P) = St_P(I_P)$$

The well-founded model of a logic program P is defined as follows.

Definition 2.65 (Well-founded model)**.** *Let P be a logic program with its Herbrand base B_P and let I_P be the least fixpoint of St^2_P. We define the* well-founded *model W_P of P as follows:*

1. *for each atom $a \in B_P$ such that $a \in I_P$, a is true in W_P*
2. *for each atom $a \in B_P$ such that $a \notin St_P(I_P)$, a is $false$ in W_P*
3. *for each other atom $a \in B_P$, a is unknown in W_P*

Example 2.24. *Let us consider the following program P:*

$p \leftarrow not\ q$
$q \leftarrow not\ p$

We have that $lfp(St^2_P) = I_P = \oslash$ and $St_P(I_P) = \{p, q\}$. Then both p and q are unknown in the well founded model of P, i.e.:

$$W_P = (\oslash, \oslash)$$

□

In [86] the following result has been shown.

Proposition 2.13. *Let P be a logic program. Then W_P exists and it is unique.*

The last result was given by [86] and it shows the relation between stable models and well-founded models.

Theorem 2.16. *Let P be a logic program. If P has a unique stable model, then it coincides with the well-founded model of P* □

Roughly speaking in the well founded model W_P of a logic program P all the elements of B_P which are $true$ in all the stable models of P are also $true$ while all the elements which are $false$ in all the stable models of P are also $false$ in W_P. All the other elements of B_P are *unknown*. Thus, if a logic program has a unique stable model then it coincides with W_P and this holds for the class of the locally stratified logic programs.

The well founded semantics, with the three-valued completion semantics, is defined for any normal logic program P but the two semantics are different. First, the well founded semantics relies upon two valued connectives rather than three-valued ones. Furthermore, given a logic program P, it is not ensured that the well-founded semantics and the three-valued completion coincide.
Consider the logic program $P = p \leftarrow p$. We have that P has a unique stable model which is the empty set of atoms $\oslash$. It coincides with the well-founded model because $lfp(St^2_P) = \oslash$ and $gfp(St^2_P) = \oslash$. Thus p is $false$ in W_P. The three valued completion semantics instead interprets p as *unknown*.

2.5 Constraint Logic Programming (CLP)

Constraint Logic Programming (CLP) [15], [94], [95] is an extension of logic programming in which the underlying language has a predefined part, namely the *constraint domain*. A constraint domain provides a set of relations, called *constraints* which can be embedded into the body of the clauses of a logic program. Those relations are then collected, during a computation, in a *constraint store* and it is the domain dependent *constraint solver* which provides the possible solutions to the set of constraints in the store. The solutions are then integrated with the solutions obtained by the classical resolution-based algorithm of logic programming. The real issue is that having an a-priori fixed domain, the constraint solver could provide an algorithm which is typically some order of magnitude faster than resolution. Moreover, constraint solvers can handle some types of reasoning, e.g. arithmetic reasoning, which is difficult or impossible to dealt with by resolution. Important constraint domains are e.g. real numbers arithmetic, term equations, and finite domain variables reasoning.
Hence, the CLP framework merges the declarative benefits of logic programming and the computational benefits of dedicated algorithms for some predefined domains. We denote as CLP($\mathcal{C}$) the CLP language based on the constraint domain $\mathcal{C}$.

Definition 2.66 (Structure)**.** *Let Σ be a signature. A Σ-structure $\mathcal{D}$ consists of a set D of objects and of an assignment of functions and relations on D to the symbols of Σ respecting the arities of those symbols.* □

Definition 2.67 (Constraint Domain and Constraint Solver)**.** *A constraint domain $\mathcal{C}$ is a tuple $\langle \Sigma, \mathcal{D}, L, T\rangle$ where Σ is a signature of functions and relations (each with an assigned arity), D is a Σ-structure, i.e. an assignment of the functions and relations in Σ on a set D (respecting the arities of the symbols), L is the (first-order) class of formulae (*constraints*) which can be expressed and T is an axiomatization of (some) properties of $\mathcal{D}$ (*constraint theory*).*

A constraint solver $solv_{\mathcal{C}}$ is a function which maps each formula in L to $\{true, false, unknown\}$. □

Most of the constraint domains have to satisfy some basic properties (which we assume to hold for a constraint domain $\mathcal{C}= \langle \Sigma, \mathcal{D}, L, T\rangle$ throughout our thesis):

- the binary predicate symbol $=$ is contained in Σ and it is interpreted as the identity in $\mathcal{D}$;
- the class of constraints L is closed under variable renaming, conjunction and existential quantification;
- $\mathcal{D}$ is a model for the constraint theory T, i.e. all the formulae in T are true under $\mathcal{D}$;
- a solver $solv_{\mathcal{C}}$ agrees with T, i.e. for each constraint $c \in L$, if $solv_{\mathcal{C}}(c) = true$ then $T \models \exists c$, and if $solv_{\mathcal{C}}(c) = false$ then $T \models \neg\exists c$.

Example 2.25. *A typical example of a constraint domain $\mathcal{C}$ is the* finite domain *constraint domain $\mathcal{FD}$. In this settings, the signature Σ is composed of the set of classical arithmetical functions $(+, -, *, \ldots)$ and relations $(<, \leq, > \ldots)$; the Σ-structure D assigns the elements of Σ to elements of the set of integers $\mathbb{Z}$ interpreting the elements in Σ in the standard way; L is the language of integer expressions where each variable takes values in a finite integer interval and finally T is the classical set of axioms on integers.*

□

Another very desirable property of a constraint domain is the *satisfaction completeness* of its constraint theory.

Definition 2.68. *Let $\mathcal{C} = \langle \Sigma, \mathcal{D}, L, T\rangle$ be a constraint domain. The constraint theory T is said to be* satisfaction complete *if for each constraint $c \in L$ either $T \models \exists c$ or $T \models \neg\exists c$.*

Moreover we say that a solver $solv_{\mathcal{C}}$ *is* complete *whenever* $solv_{\mathcal{C}}(c) = true$ *if and only if* $T \models \exists c$ *and* $solv_{\mathcal{C}}(c) = false$ *if and only if* $T \models \neg\exists c$.

□

A CLP($\mathcal{C}$) program is a set of clauses of the form

$$H \leftarrow C_1, \ldots, C_n, L_1, \ldots, L_m$$

where the key difference with respect to a normal clause is that each C_i is a constraint of $\mathcal{C}$. We refer to the constraints in the body of a rule as *constraint atoms*. The predicates not related to the constraint domain are referred to as the *user-defined* predicates.

Example 2.26. *The following is a CLP($\mathcal{FD}$) clause.*

$$p(X) \leftarrow Y > 2, q(X, Y)$$

□

The semantics of a CLP($\mathcal{C}$) program can be defined in both an operational and a declarative way as for logic programs. Those semantics are simply extensions of the semantics of logic programs taking into account that constraint atoms have a predefined semantics and their valuation is left to the constraint solver.
We briefly present only the declarative semantics for definite CLP programs, i.e. CLP programs which do not have any negative literal in the bodies of the clauses. We follow mainly [95] to which we demand for the proofs of the theorems.

Definition 2.69. *Let P be a definite CLP($\mathcal{C}$) program and let* $\mathcal{C} = \langle \Sigma, \mathcal{D}, L, T \rangle$. *A $\mathcal{C}$-interpretation $I_{\mathcal{C}}$ for P is an Herbrand interpretation which agrees with $\mathcal{D}$ on the interpretation of the symbols in $\mathcal{C}$.* □

As the meaning of the constraints is fixed by $\mathcal{C}$ then we may represent a $\mathcal{C}$-interpretation $I_{\mathcal{C}}$ simply by the subset of the user-defined atoms.

Definition 2.70. *A $\mathcal{C}$-model $M_{\mathcal{C}}$ for P is a $\mathcal{C}$-interpretation which is a model of P.*

□

As for definite logic programs, we have the following result.

Theorem 2.17. *Each definite CLP($\mathcal{C}$) program P has a $\mathcal{C}$-model. We denote the least $\mathcal{C}$-model of P as $lm(P, \mathcal{C})$.* □

Hence, $lm(P, \mathcal{C})$ is the declarative characterization of the semantics of a CLP($\mathcal{C}$) program P.
Operationally, we can obtain a similar result extending the SLD-procedure to handle constraint atoms. If a constraint atom is *selected* then its valuation is delegated to the constraint solver which also builds incrementally a *constraint store* for the global valuation of the selected constraints. We do not show here the procedure as it is out of our scope. However all the theoretical results stated for logic programs can be extended to CLP programs. In particular we have the equivalence between the operational and declarative semantics for definite CLP programs.
Moreover, the other semantics for logic programming can also be extended to CLP logic programming in a similar way. In our thesis we denote as

$$\models_{3(\mathcal{C})}$$

the notion of entailment in the three-valued completion semantics for a CLP($\mathcal{C}$) program.

2.6 Answer Sets Programming (ASP)

In the last decade, another logic programming paradigm has become one of the more popular frameworks especially for knowledge representation: the *Answer Sets Programming (ASP)* paradigm [25, 128, 87, 24].

Answer sets programming is based on an extended form of logic programs, namely *disjunctive logic programs*. But before of defining disjunctive logic programs, a note on negation is needed. As seen in previous sections, a literal L in the body of a normal clause is either of the form A or $not\ A$ (where A is an atom) and the syntactical connective *not* is intended as *Negation As Failure*. Thus, in normal logic programming we never have the *evidence* of negative information. I.e. we cannot say $\neg A$ holds because we have the *evidence* that A is $false$. Rather we say that $\neg A$ holds because A fails finitely.
In disjunctive logic programming it is possible to represent the two forms of negation: *not* , denoting Negation As Failure, and $\neg$, denoting classical negation.

A disjunctive logic program P is a set of clauses of the form:

$$L_1 \vee \ldots \vee L_k \leftarrow L_{k+1}, \ldots, L_m, not\ \ L_{m+1}, \ldots, not\ \ L_n$$

where each L_i is either an atom A or its classical negation $\neg A$. A disjunctive logic program P such that $k \in [0, 1]$ for each clause of P is called an *extended logic program*. An extended logic program is a disjunctive logic program without disjunctions in the heads of the clauses.
With respect to normal logic programs, disjunctive logic programs introduce two important differences. The first one is the presence of two forms of negations as depicted above. Note that in disjunctive logic program a *literal* is now A or its classical negation $\neg A$ and not A or $not\ A$. The second difference is that the head of a clause can be a disjunction. Intuitively if the body of a clause is satisfied then at least a disjunct of the head must be satisfied too, in order to satisfy the whole clause.

Disjunctive logic programs are an extension of normal logic programs. A normal logic program P is a disjunctive logic program without disjunctions (i.e. an extended logic program) and such that each L_i in each clause is an atom (i.e. there is no classical negation in P). In the sequel we denote as a *definite disjunctive logic program*, a disjunctive logic program without negation as failure (*not*) in each of its clauses.

Now we define what is an *answer set* starting from definite disjunctive logic programs. We denote the set of ground literals which can be formed from a disjunctive logic program P as $Lit(P)$.

Definition 2.71. *Let P be a definite disjunctive logic program. An answer set of P is a smallest (with respect to set-theoretic inclusion) set of literals S such that:*

- *for any clause $L_1 \vee \ldots \vee L_k \leftarrow L_{k+1}, \ldots, L_m \in ground(P)$, if $\{L_{k+1}, \ldots, L_m\} \subseteq S$ then for some $i \in [1, k]$, $L_i \in S$*
- *if S contains a pair of literals $L, \neg L$ then $S = Lit(P)$*

□

To extend the definition of answer set to the whole class of disjunctive logic programs we need to adapt the notion of *reduct* Π_P^S with respect to a disjunctive logic program P and a set of literals S.

Definition 2.72. *Let P be a disjunctive logic program and S a set of literals. The* reduct Π_P^S *of P with respect to S is the disjunctive logic program obtained from $ground(P)$ by deleting:*

- *all the clauses*
  $$L_1 \vee \ldots \vee L_k \leftarrow L_{k+1}, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n$$
  such that $\{L_{m+1}, \ldots, L_n\} \cap S = \oslash$ and

- *all occurrences of the set not* $L_{m+1}, \ldots, not\ L_n$ *from the remaining clauses*

□

Obviously, each reduct Π of a disjunctive logic program P is a disjunctive logic program without *not*.

Definition 2.73 (Answer Set)**.** *Let P be a disjunctive logic program. A set of literals S is an answer set of P if S is an answer set of Π_P^S.* □

The semantics of a disjunctive logic program is characterized by its answer sets. There can be one, zero or many answer sets for a program P. Note that an answer set is a generalization of a stable model. If we consider a normal logic program P, the stable models of P and the answer sets of P coincide.

Example 2.27. *Let us consider the following normal logic program P_1:*

$p \leftarrow not\ q$
$q \leftarrow not\ p$

The sets $M_1 = \{p\}$ and $M_2 = \{q\}$ are both the stable models of P_1 and the answer sets of P_1. Consider now the following program P_2:

$p \leftarrow \neg q$
$q \leftarrow \neg p$

The unique answer set of P_2 is $M = \oslash$, while stable models are not defined because P_2 is not a normal logic program.

□

Interestingly, disjunctive logic programs can be reduced to normal logic programs.

Definition 2.74. *Let P be a disjunctive logic program. Let p be a predicate occurring in P and let $p*$ be a predicate of the same arity of p such that p^* does not occur in P. Given a negative literal $L = \neg p(t_1, \ldots, t_n)$ we say that the atom $p^*(t_1, \ldots, t_n)$ is the* positive version *of L denoted as L^+. The positive version of an atom A is A itself.*

We define the positive form *of P, the normal logic program P^+ derived from P as follows:*

- *for each clause in P of the form:*

$$L_1 \vee \ldots \vee L_k \leftarrow L_{k+1}, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$$

  there is a clause in P^+ of the form:

$$L_1^+ \vee \ldots \vee L_k^+ \leftarrow L_{k+1}^+, \ldots, L_m^+, not\ L_{m+1}^+, \ldots, not\ L_n^+$$

- *for each predicate p occurring in P there is a clause in P^+ of the form:*

$$\leftarrow p(t_1, \ldots, t_n), p^*(t_1, \ldots, t_n)$$

□

Intuitively a positive version of a disjunctive logic program P is obtained by replacing negative literals with atoms whose predicates do not occur in P. The added clauses, then, give the intended meaning to those new predicates, i.e. two atoms $p(t_1, \ldots, t_n), p^*(t_1, \ldots, t_n)$ lead to inconsistency if they are both *true*.
We have the following result shown in [89] for extended logic programs which can be easily adapted [90] for disjunctive logic programs.

Proposition 2.14. *Let P be a disjunctive logic program. A set $S \subset Lit(P)$ is an answer set of P if and only if S^+ is an answer set of P^+,where S^+ stands for the set consisting of the positive version of each literal in S.* □

For further details, results on answer sets and further semantics of disjunctive logic programs, see [25], [134], [87].

Here, we give some notions about the procedural counterpart of the answer sets semantics, i.e. how answer sets can be computed. Apart from the presence of disjunctive heads and classical negation, the classical SLDNF-resolution is inadequate even for computing stable models with respect to normal logic programs. The problem is the presence of multiple stable models for many stratified logic programs.
To deal with the answer sets semantics a number of answer sets solvers have been developed, e.g. the DLV system [68], SMODELS [136], DeReS [43], Cmodels [91]. All of them use a computational mechanism very distinct from SLD-resolution.

Answer sets solvers rely upon a grounded version of a program and then try to build answer sets following a bottom-up approach. Typically, classical negation is implemented using the positive form of a program. Briefly a general algorithm for generating answer sets starts from the empty set of literals S and determines the set of literals $Det(S)$ which can be inferred deterministically from S. For example if a rule of the program is $p \leftarrow$ then it belongs to each reduct of P and so it will belong to $Det(S)$. Then a checking procedure is fired on $Det(S)$ to verify either if it is inconsistent or if it is an answer set of P (in which case it is returned as an answer). Otherwise a new literal is selected and the algorithm builds two new sets: $S \cup L$ and $S \cup \mathit{not}\ L$ on which the main procedure is fired recursively. When the checking procedure detects an inconsistency or an answer set the main algorithm backtracks to find other solutions. The algorithm ends when there are no more literals to be selected. Thus all the answer sets of a program are found through backtracking.
Clearly, there are a lot of optimization techniques for both smartly selecting literals and grounding the program. In particular database techniques are very suitable for optimizing the grounding process [76].
Answer sets solvers have captured a great attention in the logic programming and artificial intelligence fields and they are used to model and to solve a wide range of applications and problems, such as planning, diagnosis, semantic web reasoner and so on [24, 120, 22, 119, 161]. From a theoretical viewpoint they have a solid background, they have a good expressiveness and the underlying answer sets semantics is a very intuitive semantics for negation. From a computational viewpoint the answer sets solvers show good performance being able to handle easily hundreds of thousands of ground clauses and they are ensured to terminate and to compute minimal answers, i.e. answers such that each of their subsets is no more an answer (and this is obtained directly from the definition of answer set and stable model). The drawbacks of this approach is that until now the programs have to be *function-free* (apart from some built-in functions) and the ground domain has to be finite. Another possible drawback is that they are not goal oriented but they are model generators, i.e. the whole model has to be built even if the query could be easily answered using a goal oriented approach.
The best known answer sets solvers are the DLV system [68, 119] and the SMODELS [136, 166] which offer both good performance and good expressiveness. The main differences between the two are that DLV system handles disjunctive logic programs while SMODELS handles only extended logic programs (even if some constructs of the system reduces the expressiveness gap, as noted in [120]). Other differences regard the heuristics used in the systems and the grounding of the program which in the case of the SMODELS is done in a preprocessing phase through a separate process while in the case of DLV is done during the answer sets building phase. Furthermore, both systems show interesting extensions as the extension for handling *weak constraints* (DLV system, [119]) and the extension for handling *weight and cardinality constraints* (SMODELS, [166]).

Chapter 3

Abductive Logic Programming with Constraints (ALPC)

In this chapter we present *abduction* and in particular we focus on how abductive reasoning has been formalized in computational logic through the many proof procedures which have been proposed in the literature. Abduction has a wide range of applications and also a huge amount of literature has been produced in the last decades. The aim of this chapter is to give an overview of abductive reasoning in computational logic but we do not claim to cover either all the aspects of abduction or all the approaches to abduction followed in the literature. Rather we mainly focus on the closest approaches to the main contribution of this thesis, i.e. the *CIFF abductive proof procedure*.

We start from an informal introduction of abductive reasoning and then we show how it is formalized in logic and which are its applications. In Sections 3.4 and 3.5 we do a brief survey of Abductive Logic Programming (ALP) together with the main abductive proof procedures. Finally, in Sections 3.6 and 3.7 we present the ALP extension of CIFF, i.e. Abductive Logic Programming with Constraints (ALPC), together with its proof procedures.

For further information about the general aspects of abduction, mainly focusing on computational logic, see the two abductive surveys of Kakas, Toni and Kowalski [99, 97] and the one of Kakas and Denecker [96] which we mainly follow in this presentation.

3.1 Abductive reasoning

The notion of *abduction* was introduced by the philosopher Pierce in [144] where he identified three main forms of reasoning:

Deduction an analytic process based on the application of general rules to particular cases, with the inference of a result;

Induction synthetic reasoning which infers the rule from the case and the result;

Abduction another form of synthetic inference, inferring the case from a rule and a result.

Peirce further characterized abduction as the "probational adoption of a hypothesis" as explanation for observed facts (results) according to known laws. "It is however a weak form of inference, because we cannot say that we believe in the truth of the explanation, but only that it may be true" [144].
Abduction is widely used in common-sense, daily reasoning. For instance in diagnosis, to reason from effect to cause, as noted e.g. in [41]. The following well-known example of abductive reasoning was given by Pearl in [141] and it has been represented in [99, 97, 96]. Suppose we observe that, walking in the garden, the shoes are become wet. A simple explanation to this fact is that the

grass is wet. Being a sunny day, further explanations are that either the sprinkler was on or it rained during the night. This common-sense process can be viewed as an abductive process. The explanations have been inferred abductively from the observations and the "rules", stating for example that "if it rained last night then the grass has to be wet". This example can be easily formalized in logic as follows.

Example 3.1. *Consider the following logic theory T which models the "rules" of the domain of our interest.*

$$\begin{array}{lll} grass_is_wet & \leftarrow & rained_last_night \\ grass_is_wet & \leftarrow & sprinkler_was_on \\ shoes_are_wet & \leftarrow & grass_is_wet \end{array}$$

In this setting, the observation $shoes_are_wet$ can be explained by both $rained_last_night$ and $sprinkler_was_on$ alternatively. Each of the two hypothesis together with T implies the given observation. □

Abduction consists of computing such explanations from observations. It is a form of *nonmonotonic* reasoning because explanations which are consistent with one state of a knowledge base may become inconsistent by adding new information. In the above example, if we know, later on, that it did not rain last night, the explanation $rained_last_night$ turns out to be false, while the explanation $sprinkler_was_on$ becomes the only right explanation. The existence of multiple explanations is a general characteristic of abductive reasoning and the selection of "preferred" explanations is an important issue.

3.2 Abduction in logic

We formalize the abductive task presented informally presented in the previous section.

Definition 3.1. *[Abductive Task (1)] Let T be a first order theory and let G be a formula. We define the* abductive task *as the problem of finding a set of formulae Δ (*abductive explanations *for G) such that:*

1. *$T \cup \Delta \models G$ and*
2. *$T \cup \Delta$ is consistent.*

□

In general, it is often needed to put additional restrictions on Δ in order to restrict the possible causes to observed effects to a domain-specific predefined class of formulae called *abducibles*. In particular Poole shown in [146] that hypotheses can be further restricted, without loss of generality, to be atoms (which in a sense "name" those formulas) whose predicates are explicitly indicated as *abducibles*. In our thesis we assume that the class of abducible predicates is always given.

Example 3.2. *Let us consider the logic theory of the Example 3.1. Let $G = \{shoes_are_wet\}$ be the goal and let $\mathcal{A} = \{rained_last_night, sprinkler_was_on\}$ be the set of abducibles.*
There are three sets of explanations for G, namely

$$\begin{array}{ll} \Delta_1 = & \{rained_last_night\} \\ \Delta_2 = & \{sprinkler_was_on\} \\ \Delta_3 = & \{sprinkler_was_on, rained_last_night\} \end{array}$$

□

In the literature a number of additional criteria have been proposed, to restrict the number of candidate abductive explanations, see e.g. [97, 145, 34, 41, 163]. Among these, the most used method is the introduction of *integrity constraints* to the knowledge base. The basic idea is that only certain states of the knowledge base are the desired states. Integrity constraints are formulae which help in enforcing the adoption of abductive explanations leading to those desired states.

The definition 3.1 can be extended through the use of integrity constraints.

Definition 3.2. *[Abductive Task (2)] Let T be a first order theory, let G be a formula and let I a set of closed formulae (*integrity constraints*). We define the* abductive task *as the problem of finding a set of formulae Δ (*abductive explanations *for G) such that:*

1. $T \cup \Delta \models G$,
2. $T \cup \Delta$ *is consistent and*
3. $T \cup \Delta$ *satisfies* I

□

There are several ways for defining what it means for a knowledge base (in our case $T \cup \Delta$) to *satisfy* a set of integrity constraints I, e.g. see [156, 153, 102, 122]. The three most used views are the *consistency* view, the *theoremhood* view and the *metalevel* view. The *consistency* view requires that $T \cup \Delta$ satisfies I if and only if $T \cup \Delta \cup I$ is consistent. I.e.:

$$T \cup \Delta \not\models \neg I$$

Alternatively, the *theoremhood* view requires that: $T \cup \Delta$ satisfies I if and only if $T \cup \Delta \models I$. I.e.:

$$\forall \phi \in I.(T \cup \Delta \models \phi).$$

In general, the theoremhood view is much stronger than the consistency view: if $T \cup \Delta \models I$ then $T \cup \Delta \cup I$ is consistent, while the contrary does not hold. Under the consistency view, it is enough that a model M of $T \cup \Delta$ is such that $M \models I$ in order to satisfy the integrity constraints. In the theoremhood view each model M of $T \cup \Delta$ must be such that $M \models I$.
The *metalevel* view, instead, regards integrity constraints as *metalevel* or *epistemic* statements about the knowledge base, i.e. they are understood at a different level from the statements in the knowledge base. They specify what must be true about the knowledge base rather than what is true about the world modeled by the knowledge base. In the *metalevel* view, the integrity constraints are used for "selecting" from the models of $T \cup \Delta$, all such models M such that $M \models I$.
Thus, the *metalevel* view is stronger than the *consistency* view but weaker than the *theoremhood* view.

Example 3.3. *Let us consider again the theory of the Example 3.1. Suppose we add the following integrity constraint:*

$rained_last_night \rightarrow false$

In this case, independently from the view in which the integrity constraint is understood, all the abductive explanations containing the abducible $rained_last_night$ are forbidden because the adoption of that abducible would lead to an inconsistent state. □

Cox and Pietrzykowski, in [57], identify other desirable properties of abductive explanations, in particular they should be *basic* and *minimal.*

An abductive explanation is *basic* if it is not explainable in terms of other explanations. Suppose that $\{grass_is_wet\}$ is an abducible in the example 3.1. It would be an abductive explanation for the observation $shoes_are_wet$, but it would not be basic because it is explainable in terms of either $\{rained_last_night\}$ or $\{sprinkler_was_on\}$ which instead are both basic.

An abductive explanation is *minimal* if it is not subsumed by another one. Consider again the example 3.1. The basic explanation

$$\Delta_3 = \{rained_last_night, sprinkler_was_on\}$$

is not minimal while Δ_1 and Δ_2 are.

3.3 Abductive Reasoning in Applications

Deduction has been considered for years the main reasoning paradigm for declarative problem solving and the birth of logic programming and logic applications strengthened this conviction. However it can be shown easily that abduction can be a more appropriate choice in many applications, in particular when there is incomplete knowledge about the modeled world and the application task is to find configurations of the objects of the world which respect certain conditions.
An intuitive example, borrowed from [96] is the task of compiling a timetable of the lectures at a university. Typically we have some important types of well-defined objects in this domain such as *lectures*, *classrooms*, $professors$, *time_slots*, ... Moreover, it is often the case additional requirements to be satisfied, e.g. a certain professor cannot teach on a certain day of the week; a certain classroom is not available in a certain time slot and so on. The process of computing the tables mapping the lectures both to the classrooms and to the time slots, which we could represent through the objects $room_of_lecture$ and $time_of_lecture$, can clearly be formulated as an abductive task. Indeed, the instances of the $root_of_lecture$ and $time_of_lecture$ objects are not given a priori, thus finding a set of such instances which satisfies the given conditions is not a deductive process but an abductive process.

Indeed, during the last years abduction has been used in a wide range of applications.

Abduction can be used to generate causal explanations for *fault diagnosis*, as seen for example in [147, 56]. In this setting the theory describes the “abnormal” behavior (i.e. the *faulty* cases) of a system and the task is to find a set of hypotheses for observed abnormal behaviors. In medical diagnosis, for example, the observations are the symptoms and the abductive process finds the possible causes (diseases) to those symptoms [152].

Abduction has been used for improving robotic *vision* [57]. The observations are the raw data descriptions obtained from the robot visual sensors and the abductive process hypotheses on which objects effectively “see” the robot. Some very interesting work on this topic has been done by Shanahan [165] which proposes hierarchical use of abduction from raw data to real-world objects passing through intermediate levels of shapes growing in complexity.

Another application of abduction is natural language understanding [83, 41] where an abductive process is used to interpret ambiguous sentences. A related abductive framework is *legal reasoning* [160] where abduction is exploited through a case-based reasoning.

As seen in the sketched example at the beginning of this section, *scheduling* [109] and *planning* can be easily modeled by means of abduction. A planning can be viewed as a set of hypothetical actions (and subgoals) to be performed (and achieved) in order to reach the final goal state. The main approach to abductive planning is based on the *event calculus*, a logical framework for reasoning about actions and changes proposed by Sergot and Kowalski in [115] and then revised and extended by Miller and Shanahan in [133]. Abductive planning has been studied by several authors [74, 135, 164] and also a prototype version of the CIFF system has been studied for abductive planning [124, 71].

Database updates is another important application of abduction [170, 101, 19]. In this setting the observations are the update requests and the abductive explanations are the transactions that satisfy those requests. Database updates can be seen as an instance of the more general *knowledge assimilation* framework [113, 103]. In this case, we have an initial knowledge base KB and some new information I to be integrated in it obtaining a new knowledge base KB'. If the new information cannot be deduced from KB, then explanations Δ to I can be found by means of abduction, thus accommodating the new data into the new knowledge base. I.e., $KB' = KB \cup I \cup \Delta$.

In recent years, abduction has been studied as a main reasoning paradigm for modeling *intelligent* and *autonomous agents*, a field that has captured great attention in the last decades [148, 53, 149]. An intelligent agent can be defined as an actor which is capable of observing, reasoning and acting upon a (dynamical) environment. Kowalski and Sadri proposed in [114] an agent architecture

based on abduction which shows a reactive behavior and such that the well-known Beliefs-Desires-Intentions (BDI) [148] agent architecture can be seen as a special case. More recently, within the SOCS European Project [167], further developments of a logic-based agent architecture have been proposed in the Knowledge-Goals-Plans (KGP) model [98, 105, 31]. This agent model shows a (formal) computational-logic agent architecture based on the event calculus which embraces reactivity, goal decision, planning and temporal reasoning, and abduction plays an important role in most of the agent features. The SOCS European Project, proposed also an architecture of a social infrastructure for agents based on abduction [132, 5], thus entering the field of *multi-agent systems* [58, 114]. Here, abduction is used to check the fulfillment of communication protocols between agents through the use of *social* integrity constraints [5].

An important application field of abduction is the *integration of information* field. In this case abduction is used to integrate information obtained from heterogenous sources in such a way that the resulting information is as most as possible coherent to the various sources. In particular the integration of databases, see e.g. [20] and the integration of web sources, see e.g. [63], has been intensively studied.

Finally, abduction is being considered as an important paradigm in the World Wide Web, in particular for the Web 2.0 reasoner. As noted above, integration information is an important challenge over the Internet due to the exponential growth of web sources. Also abductive techniques of multi-agent systems have been used over the net, as for example in [4, 3] for verifying properties of Web Services. Moreover, the increasing interest about the Semantic Web technologies [175, 13] seems to be the right place for exploiting abductive reasoning. In particular web specification languages from XML/XMLSchema [177] to OWL [176] passing through RDF/RDFSchema [174] need expressive computational counterparts allowing more and more reasoning capabilities. Some abductive tasks over ontologies have been individuated in [39]. An abductive application for reasoning about XML/XHTML web sites properties will be presented in this dissertation in Chapter 6.

This brief and non-exhaustive overview of abductive application should give the idea of the interest in that reasoning paradigm. From the next section we restrict our view to abduction in logic programming.

3.4 Abductive Logic Programming (ALP)

The abductive task as in definition 3.2 can be easily instantiated to logic programming.

Definition 3.3 (Abductive Framework)**.** *An* abductive framework *is the tuple $\langle P, A, IC\rangle$ where P is a normal logic program (the abductive theory), A is a set of predicates defined as* abducibles[1] *and IC is the set of* integrity constraints *of the form of first order closed formulae.*

□

In our dissertation we assume, without loss of generality (as shown, e.g. in [97]) that given an abductive framework $\langle P, A, IC\rangle$, there is no clause in P whose head is an abducible atom. This ensures that abductive explanations are *basic* as they cannot depend on other predicates.

In the literature there are various characterizations of the set IC of integrity constraints. In general we assume that they are in a disjunctive form:

$$L_1 \vee \ldots \vee L_n$$

or in *denial* form, i.e.:

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

[1] In the sequel we will do an abuse of notation denoting as A both the set of abducible predicates and the set of atoms which can be obtained from A in the current framework.

or in a more general *implicative* form

$$L_1 \wedge \ldots \wedge L_n \rightarrow A_1 \vee \ldots \vee A_m.$$

where each L_i is a literal and each A_i is an atom.[2] We use the $\rightarrow$ symbol instead of $\leftarrow$ symbol to distinguish between program clauses ($\leftarrow$) and integrity constraints ($\rightarrow$). We will follow this notation throughout the thesis.

In the literature various declarative semantics for abductive logic programs have been proposed , all based upon the various semantics of logic programming seen in chapter 2. The idea is to characterize the meaning of abductive logic programs through *generalized models*, which extend the notion of models in logic programming taking into account the abducibles and the integrity constraints in an abductive framework $\langle P, A, IC \rangle$. The generalized models, defined in terms of a set $\Delta \subseteq A$ of ground abducible atoms, will be the models which solve the abductive task in the sense of definition 3.2.
Kakas and Mancarella first proposed in [102] the *generalized stable models semantics* for abductive logic programs.

Definition 3.4 (Generalized Stable Model)**.** *Let $\langle P, A, IC \rangle$ be an abductive framework and let $\Delta \subseteq A$ be a set of ground abducible atoms. Then we define a set of ground atoms $M(\Delta)$ to be a* generalized stable model *of $\langle P, A, IC \rangle$if and only if*

- *$M(\Delta)$ is a stable model of $P \cup \Delta$ and*
- *$M(\Delta) \models IC$*

□

Note that in the generalized stable models semantics, the satisfaction of the integrity constraints is inside the definition of the generalized model itself and it is similar to the *metalevel* view in the sense that integrity constraints are considered not at the same level of the program P. The following examples shows that the *metalevel* view is stronger than the *consistency* view and it is weaker than the *theoremhood* view.

Example 3.4. *Let $\langle P, A, IC \rangle$ be an abductive framework as follows:*

$$\begin{array}{rl} P: & p \leftarrow a \\ & q \leftarrow r \\ A: & \{a\} \\ IC: & p \rightarrow r \end{array}$$

Consider the set $\Delta = \{a\}$. The only stable model of $P \cup \Delta$ is $M = \{a, p\}$ which is not a generalized stable model of $\langle P, A, IC \rangle$ because it does not entail the set IC due to the absence of r. However, the set $M' = \{a, p, q, r\}$ is indeed a model of $P \cup \Delta$ which satisfies IC. So we have that $P \cup \Delta$ satisfies the integrity constraints under the consistency view. However M' is not a generalized stable model of $\langle P, A, IC \rangle$ because it is not a stable model of $P \cup \Delta$. So we have that the metalevel view is stronger than the consistency view. □

Example 3.5. *Let $\langle P, A, IC \rangle$ be an abductive framework as follows:*

$$\begin{array}{rl} P: & p \leftarrow a \\ & q \leftarrow r, s \\ A: & \{a\} \\ IC: & q \rightarrow r \end{array}$$

[2]Recall that since we have introduced the Extended Logic Programming in Section 2.6, we follow the notation which denotes NAF by *not* and strong negation by $\neg$. Hence a literal, here, is either of the form A or *not* A where A is an atom.

Consider the set $\Delta = \{a\}$. *We have that the set* $M(\Delta) = \{a, p\}$ *is a generalized stable model of* $\langle P, A, IC\rangle$ *because it is a stable model of* $P \cup \Delta$ *and it entails* $q \rightarrow r$. *However, also the set* $\{a, p, q\}$ *is a model of* $P \cup \Delta$ *but it does not entails the integrity constraint due to the absence of* r. *Hence the set* $P \cup \Delta$ *does not satisfies IC under the theoremhood view proving that it is stronger than the metalevel view.*

□

The generalization of the stable model semantics to abductive logic programs can be applied to the other model-theoretic semantics of logic programming, by considering only those models of $P \cup \Delta$ of the appropriate kind, e.g. well-founded models and so on, in which the integrity constraints are entailed.

Thus we can formalize the abductive task in logic programming with the definition of abductive answer.

Definition 3.5 (Abductive answer)**.** *An* abductive answer *to a query* Q *with respect to an abductive logic program* $\langle P,\ A,\ IC\rangle$ *is a pair* $\langle \Delta, \sigma\rangle$*, where* Δ *is a finite set of ground abducible atoms and* σ *is a substitution for the free variables occurring in* Q*, such that:*

- $P \cup \Delta \models_{LP} Q\sigma$ *and*
- $P \cup \Delta\ satisfies\ IC$

where $\models_{LP}$ *stands for the chosen semantics for logic programming.* □

In the literature have been proposed some extensions to the declarative semantics for extended logic programs in order to embrace abduction. E.g. in [143] an extension of the well-founded semantics for extended logic programs is presented, but integrity constraints have not been taken into account.

However, a declarative semantics without an operational counterpart which allows for computing (efficiently) abductive answers is not enough. Thus, a number of proposals have been presented in the literature trying to fill this gap. A brief review of the most important approaches is the subject of the next section.

3.5 Abductive proof procedures

Switching from the declarative semantics of abductive logic programming to concrete algorithms for computing abductive answers, the first thing which arises is that the abductive task is computationally very expensive and this has been intensively studied, e.g. in [162, 66, 67]. For example it has been proved in [67] that the abductive task is a hard task in the case of function-free programs and that it is undecidable for programs with functions.

Nevertheless in literature there are many proposals of *proof procedures* for abductive logic programming giving an effective computational counterpart to the abductive theory.
Here we present the most influential approaches sketching their operational behavior. In particular we will see the Kakas and Mancarella proof procedure (Section 3.5.1), the SLDNFA proof procedure (Section 3.5.3), and the IFF proof procedure (Section 3.5.2). Other interesting approaches, less related to our work are briefly surveyed in Section 3.5.4.

3.5.1 The Kakas and Mancarella procedure

The Kakas and Mancarella abductive proof procedure [102, 100] (the *KM-procedure* for short) is surely one of the first and the most influential approaches in the field. The KM-procedure is defined under the generalized stable model semantics. It is a generalization of an earlier abductive proof procedure proposed by Eshghi and Kowalski in [75] which in turn is an abductive extension of the SLDNF.

The Eshghi and Kowalski proof procedure (*EK-procedure* for short) proposes an abductive treatment for NAF in order to extend the SLDNF algorithm to calculate stable models of a wider class of logic programs than the standard SLDNF can do.

The idea in the EK-procedure is to transform a normal logic program P into an abductive framework $\langle P^*, A^*, IC^* \rangle$ as follows.
For each predicate p occurring in P, we define a *fresh* predicate p^* (i.e. such that p^* does not occur in P). Those fresh predicates are defined as *abducible* predicates and they comprise the set A^*.
The intuitive meaning of p^* is *not* p (similar to the transformation seen in section 2.6 for reducing disjunctive logic programs to normal logic programs) and the next step is to obtain the definite logic program P^* from P by replacing all the occurrences of *not* $p(\vec{t})$ in P by $p^*(\vec{t})$ for each predicate p.
The intuitive meaning of the predicates in A^* is characterized by the set of integrity constraints I^* containing, for each predicate p^* in A two closed formulae. The first formula states that $p(\vec{X})$ and $p^*(\vec{X})$ cannot appear together in an interpretation for any $\vec{X}$, i.e.

$$\forall \vec{X}.[p(\vec{X}), p^*(\vec{X}) \rightarrow].$$

The second formula states that for any interpretation I either $p(\vec{X})$ or $p^*(\vec{X})$ must belongs to I for each $\vec{X}$. This is to ensure a two-valued interpretation and that formula is of the form:

$$\forall X.[p(\vec{t}) \vee p^*(\vec{t})]$$

.
The abductive framework is then $\langle P^*, A^*, IC^* \rangle$ where the * symbols indicate the particular form of the components as described above.

The EK-procedure works in two phases like the SLDNF proof procedure and it builds incrementally an abductive answer Δ for a goal G, starting from an initial Δ_0 usually empty. The first phase is the *abductive* phase based on standard SLD resolution. In this phase, when an abducible atom $p^*(\vec{t})$ not in the current set of abductive hypotheses Δ is encountered, it is added to Δ and a *consistency* phase is fired in order to check that the integrity constraint $p(\vec{t}), p^*(\vec{t}) \rightarrow$ is not violated. To do so, the procedure checks that $p(\vec{t})$ fails finitely. If during a consistency phase, another abducible atom $q^*(\vec{s})$ not in the current Δ is encountered, the procedure fires another abductive phase starting from $q(\vec{s})$, in order to check the integrity constraint $q(\vec{s}) \vee q^*(\vec{s})$. Note that the current Δ is used in the two phases in order to remember which abducibles have been assumed so far avoiding to start new proofs for them. This use of the current abductive hypotheses Δ makes the EK-procedure able to compute stable models of normal logic programs for which the SLDNF procedure loops forever.

Example 3.6. *Let us consider the normal logic program:*

$$P: \quad \begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \end{array}$$

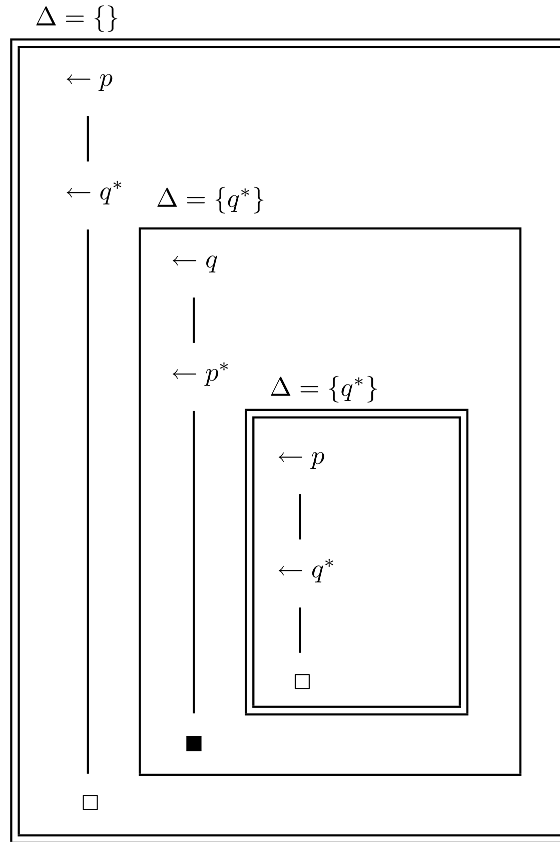
and the goal $G = \{\leftarrow p\}$. The only SLDNF derivation loops as follows:

$$\begin{array}{ccccc} \leftarrow p & \leftarrow q & \leftarrow p & \leftarrow q & \\ | & | & | & | & \dots \\ \leftarrow not\ q & \leftarrow not\ p & \leftarrow not\ q & \leftarrow not\ p & \end{array}$$

Let us consider now the abductive framework $\langle P^*, A^*, IC^* \rangle$ obtained from P:

$$P^*: \begin{array}{l} p \leftarrow q^* \\ q \leftarrow p^* \end{array} \qquad IC^*: \begin{array}{l} p, p^* \rightarrow;\ p \vee p^* \\ q, q^* \rightarrow;\ q \vee q^* \end{array} \qquad A^* = \{p^*, q^*\}$$

The EK-procedure, starting from $G = \leftarrow p$ and $\Delta_0 = \oslash$ succeeds with the answer $\Delta = \{q^*\}$ (we represent abductive phases with double-lined boxes and consistency phases with single-lined boxes):



The above behavior is obtained through the use of Δ during the computation: in the innermost box, the procedure succeeds because q^* is in the current Δ and no further checks are needed. Thus q fails finitely and finally p, in the outermost box, succeeds.
Note that the set $\{p, q^*\}$ represents the only stable model for P containing p, i.e. the set $\{p, not\ q\}$. The same behavior of the two procedures is obtained for the goal $G = \leftarrow q$, i.e. while SLDNF loops forever, the EK-procedure succeeds with the answer $\Delta = \{p^*\}$ which represents the stable model $\{q, not\ p\}$ of P.

□

The EK-procedure however is quite limited. A main limitation is that the selection function cannot select non-ground abducibles. It is the same limitation regarding negative literals in SLDNF.
Moreover the EK-procedure, in general, is not sound for the whole class of normal logic programs, as proved in [75], but it can be proved sound, following the results of Dung in [64], with respect to the class of the call-consistent logic programs.

The KM-procedure generalizes the abductive framework $\langle P^*, A^*, IC^* \rangle$ of the EK-procedure allowing for expressing both positive abducibles (not only the predicates introduced for NAF) and domain-dependent integrity constraints.
The new abductive framework for the KM-procedure is:

$$\langle P^*, A^* \cup A, IC^* \cup IC \rangle$$

where A and IC are the new components with respect to the framework for the EK-procedure. An integrity constraint in IC must be in denial form with at least an abducible atom in its body.

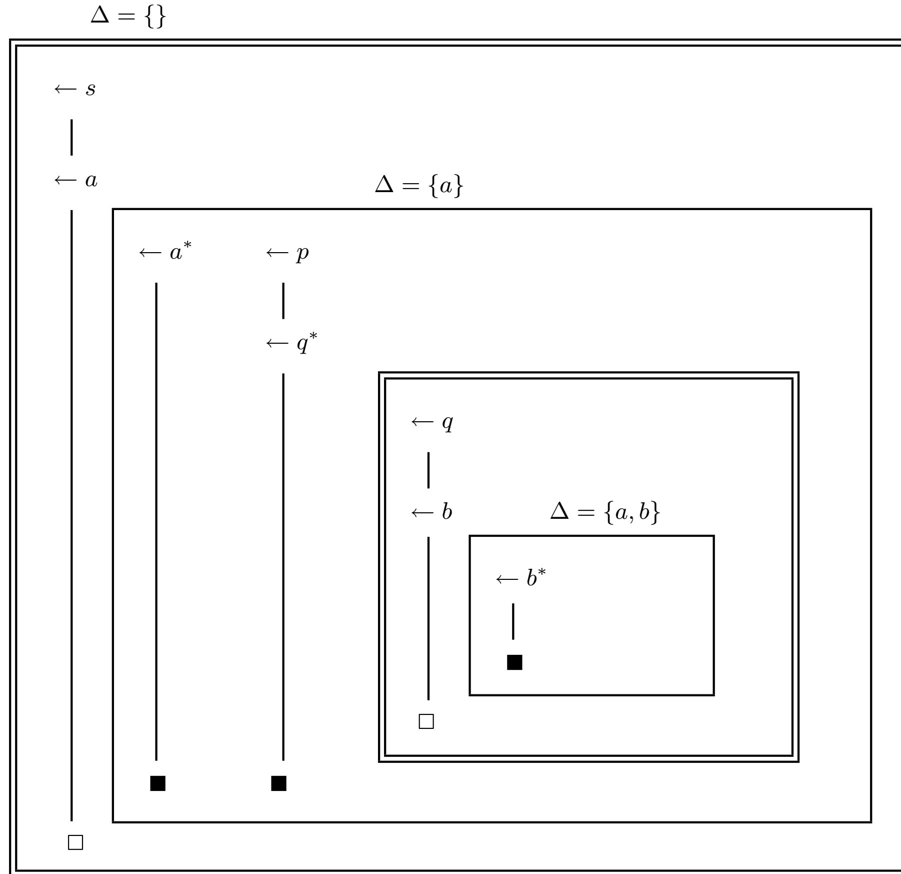
The KM-procedure retains the computational structure of the EK-procedure adding some machinery for handling the new components. In particular, while the presence of user-defined abducibles does not impose modifications to the EK-procedure, the presence of domain-dependent integrity constraints does.

The idea is that when the procedure abduces an atom $a(\vec{t})$ (resp. $a^*(\vec{t})$, it must be checked against the whole set of integrity constraints which contain $a(\vec{t})$ (resp. $a^*(\vec{t})$) in their body. This is the motivation to enforce the presence of an abducible in the body of each integrity constraint.

Example 3.7. *Let us consider the following abductive framework:*

$$
\begin{array}{llllllll}
P^*: & s \leftarrow a & IC: & a, p \rightarrow & IC^*: & p, p^* \rightarrow; p \vee p^* & & A = \{a, b\} \\
 & p \leftarrow q^* & & a, q \rightarrow & & q, q^* \rightarrow; q \vee q^* & & A^* = \{p^*, q^*, a^*, s^*, b^*\} \\
 & q \leftarrow b & & & & a, a^* \rightarrow; a \vee b^* & & \\
 & & & & & b, b^* \rightarrow; b \vee b^* & & \\
 & & & & & s, s^* \rightarrow; s \vee s^* & &
\end{array}
$$

The following is the computation of the KM-procedure with respect to the goal $G = \{s\}$:



As we can see, the KM-procedure, after it abduces a, checks for all the constraints containing a: first it checks for the finite failure of a^* (trivial) and then it continues checking for p which in turn needs the abduction of b in its proof of finite failure.

□

The KM-procedure, like the EK-procedure, is able to compute stable models of a normal logic program. Due to the possibility of representing user-defined abducibles and domain-dependent integrity constraints in the abductive framework, the declarative semantics proposed for the KM-procedure in [102] is the generalized stable models semantics. Indeed that semantics was inspired by the proof procedure itself. However, the KM-procedure suffers from the same limitations as the EK-procedure regarding the selection of only ground abducibles and the soundness problems for

non call-consistent logic programs. To avoid the latter problem, Toni [172] proposed a three-valued argumentation semantics.
The importance of the KM-procedure was not only at the theoretical level. Both its expressiveness and its computational mechanism (relatively simple to be implemented as a Prolog meta-interpreter), made the KM-procedure a widely used proof procedure in several abductive fields such as databases updates [101], truth maintenance [104] and knowledge assimilation [103].
More recently the KM-procedure has been extended in various ways. In [126], the KM-procedure has been extended allowing for a limited class of integrity constraints in implicative form giving some reactivity behavior to the procedure which can be needed for example in an agent setting. We will return to reactivity issues in section 3.5.3.
Other important extensions to the KM-procedure are the *ALIAS* proof procedure [45] for multi-agent systems and the *ACLP* proof procedure [109] for abductive logic programming with constraints. Both extensions will be briefly presented respectively in section 3.5.4 and 3.7.1.

3.5.2 The SLDNFA procedure

The SLDNFA proof procedure was proposed by Denecker and De Schreye in [60] and subsequently refined in [61].
As its name suggests, the SLDNFA procedure is an extension of the SLDNF proof procedure to the treatment of abduction which has two main relevant aspects: (1) it has been proved sound and complete with respect to the three-valued completion semantics; and (2) it can produce *non-ground* abductive answers to a query.

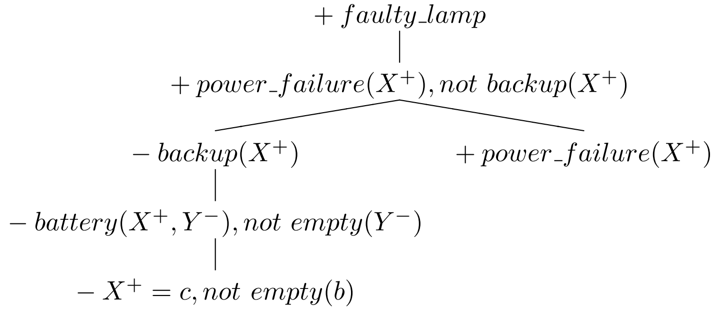
The computational schema of the SLDNFA is quite complicated because it integrates with the SLDNF schema the rules to manage non-ground abducibles. The main difficulties arise from the fact that abducibles can contain either existentially or universally quantified variables, thus presenting almost the same problems of variables in negative literals as in the SLDNF.
We sketch the SLDNFA proof procedure and we give later an example of computation.
In an SLDNFA computation for a normal goal G there is an explicit distinction between *positive goals* (subgoals which have to succeed to prove G) *negative goals* (subgoals which have to fail finitely to prove G). Variables in *positive goals* are called *positive variables*, and they are implicitly free/existentially quantified variables. Variables occurring only in *negative goals* are called *negative variables*, and they are implicitly universally quantified variables.
Positive goals are computed through standard SLD-resolution in which abducible atoms cannot be selected: they are collected incrementally and they will be part of the abductive answer. If a negative literal *not* $q(\vec{t})$ is selected, then a new computation branch is fired, starting from the *negative goal* $q(\vec{t})$ which has to fail finitely. Conversely, if a negative literal *not* $p(\vec{t})$ is encountered in a negative goal, then a new computation branch is fired, starting from the *positive goal* $p(\vec{t})$ which has to succeed.
Negative variables are those variables (implicitly universally quantified) introduced by the application of SLD resolution on negative goals. They are ensured to occur only in negative goals because the selection function of the SLDNFA can select only ground negative literals in negative goals, thus preventing switching a non-ground negative goal to a positive goal.
However it may happen that abducible atoms in negative goals contain negative variables. This is handled by resolving those abducibles against each resolvent in each positive goal thus checking it against all the abduced atoms. For example consider a negative goal $G = \leftarrow a(X), p(X)$ where X is a negative variable. If there are two abduced atoms, namely $a(t1), a(t2)$, then $a(X)$ is resolved to both of them producing two new negative goals, namely $\leftarrow p(t1)$ and $\leftarrow p(t2)$, which have both to fail finitely.

Interconnections between positive and negative variables in an SLDNFA computation represent the main technical challenges solved by the formal definition of the procedure. The following example shows the computational behavior of the SLDNFA procedure on the well-known *lamp* example.

Example 3.8. *Let us consider the following abductive framework* $\langle P, A, IC \rangle$.

$$\begin{array}{ll} P: & faulty_lamp \leftarrow power_failure(X), not\ backup(X) \\ & backup(X) \leftarrow battery(X,Y), not\ empty(Y) \\ & lamp(l) \leftarrow \\ & battery(c,b) \leftarrow \\ A: & \{power_failure, empty\} \\ IC: & \oslash \end{array}$$

The SLDNFA-refutation of $G = \leftarrow faulty_lamp$ *is shown below. Positive goals and positive variables are marked with* $+$*, while negative goals and negative variables are marked with* $-$*.*



The first step is SLD-resolution on G*, then the only thing to do (because* $power_failure$ *is abducible) is to switch* not $backup(X^+)$ *to a negative goal opening a new branch starting* $-backup(X^+)$ *which has to fail finitely.*
The third step is again an SLD-resolution step, but this is done on a negative goal: the new variable Y *is a negative variable. The negative goal has to fail for each value of* Y^- *which is bound by the* $battery(X^+, Y^-)$ *atom. In this case the only clause in the program is* $battery(c, b)$ *and the resolution step returns the node* $X^+ = c, not\ empty(b)$*. The equality* $X^+ = c$ *in a negative goal is called* irreducible *and it acts as a constraint of the form* $X \neq c$ *in the extracted abductive answer. The procedure does not go further on a branch with an irreducible equality. The abductive answer extracted from this SLDNFA refutation is in the following general non-ground form:*

$$X \neq c \wedge power_failure(X)$$

Note that the positive variable X^+ *has not been unified to* c *because unifying them we would have lost all the abductive answers containing* $power_failure(X^+) \wedge X^+ \neq c$*.*

□

In the SLDNFA abductive framework, integrity constraints have not been stated explicitly. However, as argued in [61], integrity constraints in denial form can be implicitly added to the framework. Consider an atom $\forall X.p(X)$ which has to be satisfied and a fresh predicate *violated*. By adding both the clause *violated* $\leftarrow$ *not* $p(X)$ to a normal logic program and the atom *not violated* to a query, the SLDNFA procedure computes the abductive solutions to the query satisfying $\forall X.p(X)$ under the theoremhood view.

The SLDNFA proof procedure has been extended to the SLDNFA$_+$ in [61] addressing both the minimality of the abductive answers and the construction of more ground abductive solutions. In the example 3.8, the solution $power_failure(c)$, $empty(b)$ is a correct solution not computed by the SLDNFA procedure. The SLDNFA$_+$ computes that solution through a different treatment of irreducible equalities. However, it has been shown [60] that those ground answers can be derived from SLDNFA non-ground computed answers, thus ensuring the completeness of the procedure.

The SLDNFA procedure and its extensions have been used in many abductive applications, from temporal reasoning and planning [59] to scheduling and constraint satisfaction problems [140, 142]. In [26], the ALCN description logic [21] has been mapped to abductive logic programming and the SLDNFA has been used experimentally as the abductive engine.

An implementation of the system has been given in [138] which also integrates a finite domain constraint solver. It is the precursor of the $\mathcal{A}$-System [111] which we will present in section 3.7.2.

3.5.3 The IFF proof procedure

The *IFF* proof procedure was proposed by Fung and Kowalski in [82, 81] and it is the work on which our CIFF proof procedure is based. It is an abductive proof procedure based on *rewriting rules* and it takes many ideas from an earlier abductive proof procedure proposed by Console, Dupré and Torasso in [55] which was the first procedure based on the completion semantics.
Like the SLDNFA, the IFF proof procedure has been proved sound and complete with respect to the three-valued completion semantics [82, 81]. Indeed the two proof procedures, besides being developed independently, show the same theoretical results. Moreover, the SLDNFA$_+$ computes the same abductive answers as IFF. However, the computational schema of the IFF proof procedure is simpler than the SLDNFA schema and its formalization inspired some refinements to the SLDNFA including the SLDNFA$_+$ extension.

An IFF abductive framework is of the form $\langle P,\ A,\ IC\rangle$ where IC is a set of integrity constraints in implicative form. The integrity constraints have to be satisfied under the theoremhood view. Moreover, the IFF procedure needs a refinement of the standard completion of P, $Comp(P)$. That refinement is the *abductive completion*. The idea, borrowed from [55], is that an abductive proof procedure based on the completion semantics, in order to operate as intended, should leave the abducible predicates "open", i.e. without a completed definition. Consider an abducible predicate a: by the assumption made in Section 3.4, a should not appear in the head of any clause of P, thus its completed definition should be $a \leftrightarrow false$. Obviously it is not the intended meaning of a as an abducible predicate. Moreover we have that in the language of $\langle P,\ A,\ IC\rangle$, some non-abducible predicates may occur only in the integrity constraints IC but not in P. Those predicates, being non-abducibles, need a completed definition. In particular they should be equivalent to $false$ having no definitions in P. The formal definition of *abductive completion* is as follows.

Definition 3.6 (Selective Completion)**.** *Let P be a normal logic program and let A be a set of predicates. We define the selective completion of P with respect to A, namely $Comp_A(P)$ as:*

$$Comp_A(P) = Comp(P) - \{a(\vec{t}) \leftrightarrow B \mid a(\vec{t}) \leftrightarrow B \in Comp(P) \text{ and } a \in A\}$$

□

Definition 3.7 (Abductive Completion)**.** *Let $\langle P, A, IC\rangle$ be an abductive framework. We define the* abductive completion *of P with respect to $\langle P, A, IC\rangle$, namely $Comp^{IC}_A(P)$ as:*

$$Comp^{IC}_A(P) = Comp_A(P) \cup \{p(\vec{t}) \leftrightarrow false \mid p \text{ occurs in } IC \text{ but not in } P \cup A\}$$

□

In the following, given an abductive framework $\langle P,\ A,\ IC\rangle$, we will denote the abductive completion of P by *Th*.

Example 3.9. *Consider the following abductive framework $\langle P,\ A,\ IC\rangle$ based on the Example 3.1.*

$$\begin{array}{ll} P: & grass_is_wet \leftarrow rained_last_night \\ & grass_is_wet \leftarrow sprinkler_was_on \\ & shoes_are_wet \leftarrow grass_is_wet \\ A: & \{rained_last_night, sprinkler_was_on\} \\ IC: & rained_last_night \rightarrow false \\ & cloudy_last_night \rightarrow rained_last_night \end{array}$$

The abductive completion Th is as follows:

$$\begin{array}{ll} Th: & grass_is_wet \leftrightarrow rained_last_night \vee sprinkler_was_on \\ & shoes_are_wet \leftrightarrow grass_is_wet \\ & cloudy_last_night \leftrightarrow false \end{array}$$

As we can see, the predicate cloudy_last_night is completed with $false$ because it occurs only in the second integrity constraint while the abducible predicates have no definition in Th.

□

An IFF computation iteratively rewrites a formula into another formula through a set of *rewriting rules* which ensure the equivalence of the two formulae with respect to the three-valued completion semantics. An IFF formula is a first order formula in disjunctive form and abductive answers can be extracted from each consistent disjunct (or *node*) such that no rewrite rule can be applied to it. A node is a conjunction of three types of elements: (1) atoms; (2) implications (derived from either integrity constraints or negative literals which are represented in implicative form, i.e. *not a* is represented as $a \rightarrow false$); and (3) disjunctions (mainly obtained by resolution steps between atoms and their completed definitions). A variable appearing either in an atom or in a disjunction is implicitly free/existentially quantified with the scope being the entire disjunct, while all other variables are implicitly universally quantified with the scope being the implication in which they occur.
An IFF computation starts with a single node composed of a query conjoined to the integrity constraints. The rewrite rules all replace a node in a formula with a (possibly single) disjunction of successor nodes. The main IFF rewrite rules are: *unfolding* which resolves an atom with its definition; *propagation* which resolves an atom in a node with an atom in the body of an implication; *splitting* which distributes a n-ary disjunction to a set of n successor nodes. The other interesting rules are all for managing the equalities in a node distinguishing between free/existentially quantified variables and universally quantified ones. One such rule is the *case analysis* rule which handles equalities with free/existential variables occurring in the body of an implication due to a propagation step. Given an equality $X = t$, the case analysis rule replace the node with two successor nodes: one in which $X = t$ is assumed to hold and one in which it is assumed to be false. The other equality rules are mainly based upon the Martelli-Montanari unification algorithm [129].

Example 3.10. *Consider the following abductive framework $\langle P, A, IC\rangle$ together with the abductive completion Th of P.*

$$\begin{array}{rl}
P: & faulty_lamp \leftarrow power_failure(X), not\ backup(X) \\
 & backup(X) \leftarrow battery(X,Y), not\ empty(Y) \\
 & lamp(l) \leftarrow \\
 & battery(c,b) \leftarrow \\
A: & \{power_failure, faulty_lamp, empty\} \\
IC: & \oslash \\
Th: & faulty_lamp \leftrightarrow \exists X.[power_failure(X), not\ backup(X)] \\
 & backup(X) \leftrightarrow \exists Y,Z.[X = Z, battery(Z,Y), not\ empty(Y)] \\
 & lamp(X) \leftrightarrow X = l \\
 & battery(X,Y) \leftrightarrow X = c, Y = b
\end{array}$$

The goal is $\leftarrow faulty_lamp$ which also represents the initial formula F_0. An IFF computation is the following rewrite sequence.

$$\begin{array}{ll}
F_0: & faulty_lamp \\
F_1: & power_failure(X),\ [backup(X) \rightarrow false] \\
F_2: & power_failure(X),\ [X = Z, battery(Z,Y), (empty(Y) \rightarrow false) \rightarrow false] \\
F_3: & power_failure(X),\ [battery(X,Y), (empty(Y) \rightarrow false) \rightarrow false] \\
F_4: & power_failure(X),\ [battery(X,Y), \rightarrow empty(Y)] \\
F_5: & power_failure(X),\ [X = c, Y = b, \rightarrow empty(Y)] \\
F_6: & power_failure(X),\ [X = c \rightarrow empty(b)] \\
F_7: & power_failure(X),\ [X = c, [true \rightarrow empty(b)]] \ \vee\ [X = c \rightarrow false] \\
F_8: & [power_failure(X), X = c, [true \rightarrow empty(b)]] \ \vee\ [power_failure(X), [X = c \rightarrow false]] \\
F_9: & [power_failure(X), X = c, empty(b)] \ \vee\ [power_failure(X), [X = c \rightarrow false]] \\
F_{10}: & [power_failure(c), empty(b)] \ \vee\ [power_failure(X), [X = c \rightarrow false]]
\end{array}$$

The abductive answer $power_failure(X), X \neq c$ *can be extracted from the second disjunct of the formulae* F_8, F_9 *and* F_{10}*, while the abductive answer* $power_failure(c), empty(b)$ *can be extracted from the first disjunct of* F_{10}*. Note that those answers correspond to the answers generated by the* $SLDNFA_+$ *on the same example,.*
Note also the use of case analysis *on the formula* F_6*. In this case* X *is existentially quantified and the subsequent formulae cover the two cases: in one case* X *is unified with* c *reaching the solution* $power_failure(c) \wedge empty(b)$*; otherwise the implication succeeds (falsifying its body) and the* $power_failure(X) \wedge X \neq c$ *abductive answer is reached.*

□

The IFF procedure is very powerful in handling variables. The main limitation is that, during a computation, universally quantified variables (i.e. variables appearing only in the body of an integrity constraint) must occur in a positive atom of that body. This condition is very similar to the safeness condition on the selection function of the SLDNFA procedure but in the IFF procedure is imposed through *allowedness* conditions on the syntax of normal logic programs. We will see the definition of allowedness for the IFF procedure in the next chapter, when we go into the formal details of the CIFF proof procedure.

The IFF procedure has been used in many abductive applications. For example in [173], the IFF procedure has been used as the computational engine for managing web information. However, the IFF procedure has been used in particular in (multi-)agent settings [114, 81] exploiting the expressiveness of the event calculus [115].
Indeed the use of integrity constraints in implicative form make an IFF abductive framework a good choice for modeling agents. In this respect an important extension of the IFF procedure was proposed by Sadri and Toni in [157]. This extension allows for expressing two types of negation in the body of integrity constraints: strong negation and Negation As Failure. In particular, the NAF view, allows for a better modeling of *reactive rules*. Assume we have a rule of behavior stating that "if today is a *cleaning day* and there no *alarm* has been sounded, then we have to *dust*". This rule can be modeled through the following integrity constraint ϕ:

$$cleaning_day, \neg sound_alarm \rightarrow dust.$$

Assume today is a *cleaning day*. If no alarm has been sounded, then ϕ is satisfied by proving *dust*. However, using classical negation, ϕ can also be satisfied by sounding the alarm! This is because ϕ is logically equivalent to:

$$cleaning_day \rightarrow dust \vee sound_alarm.$$

When integrity constraints can model rules of behavior of an agent, and in particular when the head of integrity constraints can model the actions an agent has to perform in order to react to some conditions of the agent's world, situations like the one described above should be avoided. A NAF view of negative literals in the body of integrity constraints can help in modeling reactive rules. Using NAF, we can modify ϕ as follows:

$$cleaning_day, not\ sound_alarm \rightarrow dust.$$

Now, the integrity constraint capture the intended meaning of the rule, i.e. whenever it is proved that today is a *cleaning_day* and whenever *sound_alarm* fails, then *dust*. Note that this is the unique way for satisfying ϕ, as ϕ is no longer equivalent to

$$cleaning_day \rightarrow dust \vee sound_alarm.$$

This extension of the IFF procedure has been proven sound with respect to the three-valued completion semantics as the computed abductive answers are a subset of the answers computed by the original IFF procedure. This extension has been mainly used in (multi-)agent settings and databases updates [157, 158]. We propose a similar extension of the CIFF proof procedure in the next chapter.

3.5.4 Other computational approaches to abduction

The abductive procedures seen in the previous section are those procedures which mainly inspired (and relate to) our work. In this section we give a brief, non-exhaustive survey of other computational approaches to abduction.

The Lin-You procedure

The Lin-You abductive proof procedure was proposed in [121] and it is based on rewriting rules like the IFF procedure. It has been proven sound and complete with respect to the partial stable models semantics [155] and, like the KM-procedure, it is able to compute only ground abductive answers. An interesting feature of the Lin-You proof procedure is the presence of rules for *loop handling*. Roughly speaking, the procedure traces the literals taken into account during the computation (generalizing, in a sense, what the KM-procedure only does for abducible atoms). When a literal occurs twice in a trace, the procedure either fails or succeed in that trace according to the semantics. Some experimental results of the Lin-You procedure have been shown in [121].

The ALIAS framework

The ALIAS framework [45, 46] is an interesting abductive framework used in multi-agent systems applications. The novel idea of ALIAS is to distribute the abductive computation over the agents involved in the system implementing a coordination protocol for that scope. The global knowledge shared by all agents is represented by a set of abducibles posted in a global space. Each agent is equipped with an abductive procedure mostly inspired by the KM-procedure and when local abduction is performed, the global knowledge can be updated by the agent provided the consistency of that update is checked. This framework has been studied mainly for problems arising in multi-agent settings, in particular competition and cooperation [46].

The ABDUAL system

The ABDUAL system [9] is based upon the well-founded semantics for extended logic programs and it is based on SLG-resolution [42, 168], i.e. SLDNF resolution with a tabling mechanism for storing intermediate solutions. An ABDUAL abductive framework is a tuple $\langle P, A, IC \rangle$ where P is an extended logic program and IC is a set of integrity constraint in denial form. The idea is to build a *dual* program of P and evaluate the abductive query with respect to P augmented with its dual. For each defined predicate p in P, the dual program contains a rule of the form $not\ p(\vec{t}) \leftarrow B$ where B encodes all the *failure conditions* for $p(\vec{t})$. For example, given a predicate p defined by $p \leftarrow not\ q$ and $p \leftarrow not\ r$, the dual program contains the rule $not\ p \leftarrow q, r$.
The ABDUAL system handles only grounded extended logic programs and it has been proven sound with respect to the well-founded semantics for extended logic programs. ABDUAL has been implemented under the XSB Prolog [150], which is a special Prolog integrating a tabling mechanism for SLG-resolution. The ABDUAL system has been used in many abductive applications, e.g. diagnosis [84] and knowledge assimilation [8].

Abduction in Answer Sets Programming

An abductive framework $\langle P, A, IC \rangle$ can be embedded in the answer set programming paradigm seen in Section 2.6. This is done through a transformation of the framework similar to the transformation performed in the KM-procedure, i.e. the introduction of predicates of the form p^* (see Section 3.5.1). An accurate description of that transformation is given in [28]. The resulting abductive framework can be fed as input to any answer set solver, such as SMODELS [136] and DLV [68]. The main drawback of this approach is the restriction to ground abductive answers, even if some work for avoiding that limitation has been done [29], at least at the theoretical level.

3.6 ALP + CLP = ALPC

The integration of abductive logic programming (ALP) with constraint logic programming (CLP) (see Section 2.5) gives rise to the *Abductive Logic Programming with Constraints* (ALPC) framework which combines the knowledge representation capabilities of ALP with the computational enhancements of CLP. The ALPC framework is the underlying framework of our CIFF proof procedure.
The relationship between ALP and CLP has been recognized quite early. An overview of these first integrations and relationships can be found in [97].
One way to integrate special predicates typical of constraint logic programming is to interpret such predicates as abducibles and to constrain them with a set of integrity constraints which give the intended semantics. For example the $<$ predicate, typical in numerical constraints, can be seen as an abducible predicate constrained by:

$$\forall X, Y, Z.(X < Y, Y < Z \rightarrow X < Z)$$
$$\forall X, Y.\neg(X < Y \wedge Y < X)$$
$$\forall X.\neg(X < X)$$

The problem in this approach, followed for example in [74], is that abductive logic programming is not very suitable for embedding simplifications like the following:

$$\forall X.((2 < X \wedge 3 < X) \equiv 3 < X)$$

Due to this fact, this way of integrating CLP and ALP has not been continued, with the notable exception of PROCALOG [180, 117].

The other way to integrate ALP and CLP is to consider the two as autonomous entities cooperating in a unique framework for solving a common goal. One of the first attempts in this direction was done in [49] where abductive reasoning was integrated into the concurrent constraint language *ccFD*. However, the most known and influential integration of ALP and CLP under this approach is probably the *ACLP* procedure [106, 110, 109] proposed by Kakas and Michael.
The ACLP procedure, which will be discussed in more detail in section 3.7.1, extends the KM-procedure with a constraint solver over finite domain regarded as an underlying "black box" entity. The abductive answers are composed of both a set of abducibles and a set of constraints (a *constraint store*) which are ensured to be satisfiable by the solver. In this setting an abducible atom $a(X)$ in an abductive answer represents the set of ground abductive answers given by all the possible substitutions of X satisfying the constraint store.

Another hybrid approach of integrating ALP and CLP is by using *Constraint Handling Rules* (CHR) [80], a generic and flexible platform for defining constraint solvers. The idea is to integrate abductive logic programming with a set of constraint handling rules for defining the meaning of special predicates (similar to the first approach) and then delegate to a CHR system the evaluation of those rules (similar to the second approach). This has been done, e.g., in [32, 33].

In our work we will use the second approach, i.e. we will integrate an underlying black-box constraint solver for evaluating constraints into a top-level abductive proof procedure. This approach is surely the one most followed because it allows for using the expressiveness of the abduction and the power of constraint solving in a straightforward way. In the next sections we give a brief overview of the state-of-the-art systems: the above mentioned ACLP procedure and the $\mathcal{A}$-System [111, 139].

Here we focus on the theoretical aspects of Abductive Logic Programming with Constraints, restating the notion of abductive answer seen in Section 3.4 for embracing constraints.
The main idea is that *abductive logic programming with constraints* extends abductive logic programming in the same way constraint logic programming extends logic programming.

We recall some notions about constraint logic programming seen in section 2.5.

The CLP framework is defined over a particular structure $\Re$ consisting of a domain $D(\Re)$, and a set of constraint predicates which includes equality, together with an assignment of relations on $D(\Re)$ for each constraint predicate. The structure $\Re$ is equipped with a notion of $\Re$-satisfiability. Given a set of constraint atoms C, the fact that C is $\Re$-satisfiable will be denoted as $\models_\Re C$. Moreover we denote as $\sigma \models_\Re C$, the fact that the grounding σ of the variables of C over $D(\Re)$ satisfies C, i.e. C is $\Re$-satisfied via σ.

An abductive framework with constraints (or an *abductive logic program with constraints*) is a tuple $\langle P,\ A,\ IC\rangle_\Re$ such that all the components are defined as above but now constraint atoms might occur in the body of clauses of P and integrity constraints of IC. Also queries for abductive logic programs with constraints might include constraint atoms.

The semantics of CLP is obtained by combining the logic programming semantics $\models_{LP}$ and the notion of $\Re$-satisfiability (see [94]). We denote this semantic notion as $\models_{LP(\Re)}$ and we use it in the notion of abductive answer with respect to an abductive logic program with constraints.

Definition 3.8 (Abductive answer with constraints)**.** *An* abductive answer with constraints *to a query Q with respect to an abductive logic program with constraints $\langle P,\ A,\ IC\rangle_\Re$ is a tuple $\langle \Delta, \sigma, \Gamma\rangle$, where Δ is a finite set of abducible atoms, σ is a ground substitution for the (existentially quantified) variables occurring in Q and Γ is a set of constraint atoms such that*

1. *there exists a ground substitution σ' for the variables occurring in $\Gamma\sigma$ such that $\sigma' \models_\Re \Gamma\sigma$ and*

2. *for each ground substitution σ' for the variables occurring in $\Gamma\sigma$ such that $\sigma' \models_\Re \Gamma\sigma$, there exists a ground substitution σ'' for the variables occurring in $Q \cup \Delta \cup \Gamma$, with $\sigma\sigma' \subseteq \sigma''$, such that:*

   - $P \cup \Delta\sigma'' \models_{LP(\Re)} Q\sigma''$ *and*
   - $P \cup \Delta\sigma''$ *satisfies IC.*

Example 3.11. *Consider the following abductive logic program with constraints (here we assume that $<$ is a constraint predicate of $\Re$ with the expected semantics):*

$$
\begin{array}{ll}
P: & p(X) \leftarrow q(T_1, T_2) \wedge T_1 < X \wedge X < 8 \\
 & q(X_1, X_2) \leftarrow s(X_1, a) \\
A: & \{r, s\} \\
IC: & r(Z) \ \rightarrow \ p(Z)
\end{array}
$$

An abductive answer with constraints for the query $Q = r(6)$ is

$$\langle \{r(6), s(T_1, a)\}, \{T_1 < 6\}\rangle$$

Intuitively, given the query $r(6)$, the integrity constraint in IC would fire and force the atom $p(6)$ to hold, which in turn requires $s(T_1, a)$ for some $T_1 < 6$ to be true.

Considering a non-ground version of the query, for example $Q = r(Y)$, we would have obtained an abductive answer with constraints:

$$\langle \{r(Y), s(T_1, a)\}, \{T_1 < Y, Y < 8\}\rangle.$$

□

3.7 Abductive proof procedures with constraints

Abductive Logic Programming with Constraints is a relatively new field. Thus, despite its many attractive features (the expressiveness of abductive logic programming plus the computational efficiency of constraint logic programming) which make it suitable for a number of applications, not so many concrete proposal have been presented to date. Here we briefly present the two approaches most related to our work, namely the ACLP proof procedure and the A-System. Other interesting approaches, not covered here, are the PROCALOG [180, 117] approach and the abductive approach based on Constraints Handling Rules which was followed in [32, 33] and more recently in the *SCIFF* proof procedure [7], an abductive proof procedure based on the IFF procedure which is specifically designed to verify properties of multi-agent behavior protocols.

3.7.1 The ACLP proof procedure

The ACLP procedure [106, 110, 109] is probably the best known attempt of integrating ALP and CLP embedding a black-box constraint solver in a higher-level abductive proof procedure, namely the KM-procedure.
The ACLP abductive framework with constraints is of the form $\langle P^*, A^* \cup A, IC^* \cup IC \rangle_{\Re}$ where the three components are specified as in the KM-procedure (see Section 3.5.1) while constraints are expressed in the finite domain constraint domain $D(\Re)$. Obviously, with respect to the KM-procedure, P^* is now obtained from a constraint normal logic program and constraint atoms can also occur in the body of integrity constraints.
The computational schema of ACLP follows the interleaving of abductive and consistency phases seen in the KM-procedure but when constraint atoms are introduced by resolution steps, they are delegated to the constraint solver which accumulates them into a constraint store Γ. The constraint solver also cooperates withthe abductive reasoner by checking for the satisfiability of the current Γ and eventually simplifying it or pruning those branches of the abductive search tree containing an inconsistent Γ.

The ACLP procedure has been proven sound with respect to the generalized stable models semantics (extended for abductive framework with constraints in the sense of definition 3.8). Note that the ACLP procedure is able to compute non-ground abductive answers but only with respect to variables involved in the constraint store: indeed the ACLP procedure retains the KM-procedure's limitations on the selection of non-ground literals.

An implementation of the procedure, the *ACLP system* [109], has been done as a Prolog meta-interpreter on top of the CLP language of ECL^{i}PSe [178] through the use of its constraint solver over finite domains. However, the architecture of the system is quite general and it can be easily adapted to other constraint solvers. The experimental results in [109] show the expressive power and the good computational performances of the system.
The merit of the ACLP system also include the first large application of an abductive system in an industrial context: the scheduling of the crew for the flights of Cyprus Airways [107, 108]. The ACLP system was able to efficiently produce solutions that were of good quality according to the company's experts.

3.7.2 The $\mathcal{A}$-System

The $\mathcal{A}$-System was proposed by Kakas, Van Nuffelen and Denecker in [111, 139]. It was conceived as a reformulation of the SLDNFA procedure (with the integration of a constraint solver à la ACLP) as a rewriting procedure, thus showing in a concrete way the close relationships between SLDNFA and IFF. Indeed this procedure can be thought as an hybrid of the IFF procedure, the SLDNFA procedure, and the ACLP procedure.

The abductive framework with constraints of the $\mathcal{A}$-System is of the form $\langle P, A, IC \rangle_{\Re}$ where P is a constraint normal logic program, IC is a set of integrity constraints in denial form and the constraint domain $D(\Re)$ is the finite domain. The $\mathcal{A}$-System retains the distinction between the

positive and negative goals done in the SLDNFA, but it represents explicitly integrity constraints in the abductive framework and it embeds a finite domain constraint solver as in ACLP.
The $\mathcal{A}$-System rules, which substantially implement the SLDNFA formalization (apart from finite domain constraints), are formulated in an IFF fashion, even if they differ in particular for the treatment of integrity constraints which in the IFF are in implicative form.
Compared with the IFF, no allowedness conditions are given, but the *safety* of a selection is determined at run-time checking the quantification of the variables, thus avoiding the selection of *floundering patterns*, i.e. patterns which would lead to floundering. In particular, in the body of a denial, neither negative literals nor constraints atoms with universally quantified variables can be selected. Those conditions, roughly speaking, reformulate in a dynamic way the allowedness conditions of the IFF procedure and they are very similar to the approach used in our CIFF proof procedure.
The $\mathcal{A}$-System has been proven in [139] sound and complete with respect to the three-valued completion semantics. However the completeness result does not take into account floundering patterns, thus it is substantially equivalent to the IFF completeness result.

The importance of the $\mathcal{A}$-System, which is the most related abductive proof procedure to our CIFF, can be found mainly in two aspects. The first one is that it is the first approach which combines an abductive proof procedure which is able to compute non-ground abductive answers with a finite domain constraint solver. The second one is that it is probably the first procedure whose implementation was made as a meta-interpreter of the SICStus Prolog [1], and which explores a number of algorithms to improve efficiency. The $\mathcal{A}$-System implementation, indeed, makes use both of complex data structures for storing the state of an abductive computation and of *heuristics* on the choice points of a computation.

This approach, as reported in [111, 139], has given very good computational results compared to simpler abductive implementations (e.g. ACLP) which basically make use of the left-most Prolog strategy and "standard" Prolog data structures. The main application of the $\mathcal{A}$-System has been the coherent integration of databases [20, 139].

A comparison between the $\mathcal{A}$-System and our CIFF proof procedure will be done in Chapter 5.

Chapter 4

The CIFF Proof Procedure and the CIFF$^\neg$ extension

This chapter is the main chapter of the thesis. Here we define the CIFF proof procedure and the CIFF$^\neg$ extension and we prove the soundness of both procedures.

In Section 4.1 we describe the CIFF framework, constraining the concepts seen in the previous chapters to the concrete choices done for CIFF. In Section 4.1, we define formally the CIFF proof procedure: we start describing the CIFF framework, then we present in detail the CIFF proof rules which are the computational core of CIFF and finally we present the concepts of CIFF derivation and CIFF extracted answer. The main contribution of Section 4.2 is the proof of the soundness of the CIFF proof procedure, i.e. we show that a CIFF extracted answer is indeed an abductive answer with constraints with respect to a CIFF framework and a given query.
Finally, in Section 4.3, we define the CIFF$^\neg$ proof procedure, a useful extension which provides a NAF treatment of negative literals in the CIFF integrity constraints (based on the ideas of [157]), and we prove its soundness.
A comparison with other existing tools and proof procedures will be done in the next Chapter, after the presentation of the CIFF (and CIFF$^\neg$) implementation.

4.1 The CIFF proof procedure

In Chapter 3, we gave a brief overview of abductive logic programming (with constraints), indicating some of the syntactical and semantical variants used in the literature. In this section we describe the concrete choices done for defining the CIFF proof procedure.

It is useful to recall some notions from the previous chapters. We start from the definition of *abductive logic program* constraining it to the syntax used in CIFF.
An *abductive logic program* is a tuple $\langle P, A, IC \rangle$ where:

- P is a *normal logic program*, namely a set of *clauses* of the form:

  $$p(\vec{X}) \leftarrow l_1(\vec{Y_1}), \ldots, l_n(\vec{Y_n})$$

  where $p(\vec{X})$ is an atom and each $l_i(\vec{Y_i})$ is a literal, i.e. an atom $a(\vec{Y})$ or the negation of an atom $a(\vec{Y})$, represented as *not* $a(\vec{Y})$. We refer to $p(\vec{X})$ as the *head* of the clause and to $l_1(\vec{Y_1}) \wedge \ldots \wedge l_n(\vec{Y_n})$ as the *body* of the clause. A predicate p occurring in the head of at least one clause in P is called a *defined predicate* and the set of clauses in P such that p occurs in their heads is called the *definition set* of p.

  Any variable in a clause is implicitly universally quantified with the scope the entire clause.

- A is a set of predicates, referred to as *abducible predicates*. Atoms whose predicate is an abducible predicate are referred to as *abducible atoms* or simply *abducibles*. Abducible atoms must not occur in the head of any clause of P (without loss of generality, see [97]).
- IC is a set of *integrity constraints* which are *implications* of the form:

$$l_1(\vec{X_1}) \wedge \ldots \wedge l_n(\vec{X_n}) \;\rightarrow\; a_1(\vec{Y_1}) \vee \ldots \vee a_m(\vec{Y_m}) \qquad n, m \geq 1$$

  Each of the $l_i(\vec{X_i})$ is a literal (as defined above) while each of the $a_i(\vec{Y_i})$ is an atom. We refer to $l_1(\vec{X_1}) \wedge \ldots \wedge l_n(\vec{X_n})$ as the *body* and to $a_1(\vec{Y_1}) \vee \ldots \vee a_m(\vec{Y_m})$ as the *head* of the integrity constraint.

  Any variable in an integrity constraint is implicitly universally quantified with scope the entire implication.

As we will see, negation in the body of an integrity constraint is treated as classical negation by the CIFF proof procedure. To evidence this behavior we use the symbol $\neg$ in a CIFF framework instead of the symbol *not* used in an abductive logic program with constraints.

An abductive logic program can be usefully extended to handle constraint predicates in the same way constraint logic programming (CLP) extends logic programming. The CLP framework is defined over a particular structure $\Re$ consisting of a domain $D(\Re)$, and a set of constraint predicates which includes equality ($=$) and inequality ($\neq$), together with an assignment of relations on $D(\Re)$ for each constraint predicate. We will refer to the set of constraint predicates in $\Re$ as the *constraint signature* (of $\Re$), and to atoms of the constraint predicates as *constraint atoms* (for $\Re$).
The structure $\Re$ is equipped with a notion of $\Re$-satisfiability. Given a set of (possibly non-ground) constraint atoms C, the fact that C is $\Re$-satisfiable will be denoted as $\models_{\Re} C$. Moreover we denote as $\sigma \models_{\Re} C$ the fact that the grounding σ of the variables of C over $D(\Re)$ satisfies C, i.e. C is $\Re$-satisfied via σ.

An *abductive logic program with constraints* is a tuple $\langle P, A, IC\rangle_{\Re}$ with all components defined as above but where constraint atoms for $\Re$ might occur in the body of clauses of P and of integrity constraints of IC.

The semantics of abductive logic programming with constraints is strictly related to the notion of *abductive answer with constraints* for a query Q which is a conjunction of (constraint) literals. The following definition is a modification of Definition 3.8 where the integrity constraints are satisfied under the *theoremhood* view (see Section 3.2 and Section 3.4 for further details). The *theoremhood* view is the notion of integrity constraint satisfaction which is used in our CIFF proof procedure. In the following definition, $\models_{LP(\Re)}$ is the chosen semantics of logic programming (LP) combined with the chosen constraint structure ($\Re$).

Definition 4.1 (Abductive answer with constraints)**.** *An* abductive answer with constraints *to a query Q with respect to an abductive logic program with constraints $\langle P, A, IC\rangle_{\Re}$ is a tuple $\langle \Delta, \sigma, \Gamma\rangle$, where Δ is a finite set of abducible atoms, σ is a ground substitution for the (existentially quantified) variables occurring in Q and Γ is a set of constraint atoms such that*

1. *there exists a ground substitution σ' for the variables occurring in $\Gamma\sigma$ such that $\sigma' \models_{\Re} \Gamma\sigma$ and*
2. *for each ground substitution σ' for the variables occurring in $\Gamma\sigma$ such that $\sigma' \models_{\Re} \Gamma\sigma$, there exists a ground substitution σ'' for the variables occurring in $Q \cup \Delta \cup \Gamma$, with $\sigma\sigma' \subseteq \sigma''$, such that:*
   - $P \cup \Delta\sigma'' \models_{LP(\Re)} Q\sigma''$ *and*
   - $P \cup \Delta\sigma'' \models_{LP(\Re)} IC$.

□

Example 4.1. *Consider the following abductive logic program with constraints (here we assume that $<$ is a constraint predicate of $\Re$ with the expected semantics):*

$$
\begin{array}{ll}
P: & p(X) \leftarrow q(T_1,T_2) \wedge T_1 < X \wedge X < 8 \\
& q(X_1,X_2) \leftarrow s(X_1,a) \\
A: & \{r,s\} \\
IC: & r(Z) \ \rightarrow \ p(Z)
\end{array}
$$

An abductive answer with constraints for the query $Q = r(6)$ is

$$\langle \{r(6), s(T_1,a)\}, \{T_1 < 6\} \rangle$$

Intuitively, given the query $r(6)$, the integrity constraint in IC would fire and force the atom $p(6)$ to hold, which in turn requires $s(T_1,a)$ for some $T_1 < 6$ to be true.

Considering a non-ground version of the query, for example $Q = r(Y)$, we would have obtained an abductive answer with constraints:

$$\langle \{r(Y), s(T_1,a)\}, \{T_1 < Y, Y < 8\} \rangle.$$

□

We are now going to formally introduce the CIFF proof procedure. The language of CIFF is the same as that of an abductive logic program with constraints, but we assume to have the following special symbols:

- the special atom $\bot$ which represents *false*;
- the special atom $\top$ which represents *true*.

These will be used, in particular, to represent the empty body ($\top$) and the empty head ($\bot$) of an integrity constraint.
The CIFF framework relies upon the availability of a concrete CLP structure $\Re$ over arithmetical domains equipped at least with the set $\{<, \leq, >, \geq, =, \neq\}$ of constraint predicates whose intended semantics is the expected one. The set of constraint predicates is assumed to be closed under complement. When needed, we will denote by $\overline{Con}$ the complement of the constraint atom Con (e.g. $\overline{X < 3}$ is $X \geq 3$). We also assume that the constraint domain offers a set of functions like $+, -, *\ldots$ whose semantics is again the expected one.
The structure $\Re$ is a *black box* component in the definition of the CIFF proof procedure: for handling constraint atoms and evaluating constraint functions, we rely upon an underlying *constraint solver* over $\Re$ which is assumed to be both sound and complete with respect to $\models_\Re$. In particular we will assume that, given a constraint atom Con and its complement $\overline{Con}$, the formulae $Con \vee \overline{Con}$ and $Con \rightarrow Con$ are tautologies with respect to the constraint solver semantics. We do not commit to any concrete implementation of a constraint solver, hence the range of the admissible arguments to constraint predicates ($D(\Re)$) depends on the specifics of the chosen constraint solver.

The semantics of the CIFF proof procedure is defined in terms of definition 4.1 where (1) the constraint structure $\Re$ is defined as above, and (2) the semantics of logic programming is the three-valued completion semantics [118] (we denote as $\models_{3(\Re)}$ the notion of $\models_{LP(\Re)}$ with respect to that semantics). We refer to an abductive answer with constraints as a *CIFF abductive answer*. Recall that the three-valued completion semantics embeds the Clark Equality Theory [48], denoted by *CET*, which handles equalities over Herbrand terms.

The CIFF proof procedure operates on a set of *iff-definitions* obtained from the *completion* [48] of the defined predicates $p_1, \ldots, p_n$ in the language of $\langle P,\ A,\ IC \rangle_\Re$.
The completion of a predicate p with respect to $\langle P,\ A,\ IC \rangle_\Re$ is defined as follows. Assume that the following set of clauses is the definition set p in $\langle P,\ A,\ IC \rangle_\Re$:

$$\begin{array}{rcl} p(\vec{t_1}) & \leftarrow & D_1 \\ & \vdots & \\ p(\vec{t_k}) & \leftarrow & D_k \end{array}$$

where each D_i is a conjunction of literals and constraint atoms. The *iff-definition* of p is of the form:

$$p(\vec{X}) \;\leftrightarrow\; [\vec{X} = \vec{t_1} \wedge D_1] \vee \cdots \vee [\vec{X} = \vec{t_k} \wedge D_k]$$

where $\vec{X}$ is a vector of *fresh* variables (not occurring in any D_i or t_i) implicitly *universally* quantified with scope the entire iff-definition, and all other variables are implicitly *existentially* quantified with scope the right-hand side disjunct in which it occurs.
Notice that the equality symbol $=$ is overloaded as it denotes both equality in the constraints and equality over Herbrand terms. When using the latter, in the remainder, we will talk about equality atoms and equality predicate.
If p is a non-abducible, non-constraint, non-equality atom and it does not occur in the head of any clause of P its iff-definition is of the form:

$$p(\vec{X}) \;\leftrightarrow\; \bot.$$

Definition 4.2 (CIFF Theory and CIFF Framework)**.** *Let $\langle P, A, IC\rangle_{\Re}$ be an abductive logic program with constraints. The* CIFF theory *Th relative to $\langle P, A, IC\rangle_{\Re}$ is the set of all the iff-definitions of each non-abducible, non-constraint predicate in the language of $\langle P, A, IC\rangle_{\Re}$. Moreover we say that a* CIFF framework *is the tuple $\langle Th, A, IC\rangle_{\Re}$.*

□

Example 4.2. *Let us consider the following abductive logic program with constraints $\langle P, A, IC\rangle_{\Re}$:*

$$\begin{array}{ll} P: & p(T) \leftarrow s(T) \\ & p(W) \leftarrow W{<}8 \\ A: & \{s\} \\ IC: & r(T) \wedge s(T) \;\rightarrow\; p(T) \end{array}$$

The resulting CIFF theory Th is:

$$\begin{array}{ll} p(X) & \leftrightarrow [X = T \wedge s(T)] \vee [X = W \wedge W{<}8] \\ r(Y) & \leftrightarrow \bot. \end{array}$$

With explicit quantification, the theory Th would be:

$$\begin{array}{lll} \forall X & (p(X) & \leftrightarrow [\exists T(X = T \wedge s(T)] \vee [\exists W(X = W \wedge W{<}8)]) \\ \forall Y & (r(Y) & \leftrightarrow \bot). \end{array}$$

To improve readability and unless otherwise stated, in the remainder we will write CIFF theories with implicit variable quantification. Note that Th includes an iff-definition for r even though r occurs only in the integrity constraints IC. Moreover, there is no iff-definition for the abducible predicate s.

□

Definition 4.3 (CIFF query)**.** *A* CIFF query *Q is a conjunction of literals, possibly including constraint literals. All the variables in a CIFF query Q are implicitly existentially quantified over Q.* □

Allowedness. [82] require frameworks for their IFF proof procedure to meet a number of so-called *allowedness conditions* to be able to guarantee the correct operation of their proof procedure. These conditions are designed to avoid problematic patterns of quantification which can lead to *floundering* [123]. Informally, the floundering problem arises when a universally quantified variable occurring in a clause occurs nowhere in the body except possibly in a negative literal or in an abducible atom.
The IFF proof procedure for abductive logic programming (without constraints) has the following allowedness conditions:

- an integrity constraint $A \rightarrow B$ is allowed iff every variable in it also occurs in an atomic conjunct within its body A;
- an iff-definition $p(\vec{X}) \leftrightarrow D_1 \vee \cdots \vee D_n$ is allowed iff every variable, other than those in $\vec{X}$, occurring in a disjunct D_i, also occurs inside a non-equality atomic conjunct within the same D_i;
- a query is allowed iff every variable in it also occurs in an atomic conjunct within the query itself.

As stated in [82], the allowedness conditions ensure that floundering is avoided.
Our CIFF frameworks $\langle Th,\ A,\ IC \rangle_\Re$ must also be *allowed* in order to guarantee the correct operation of CIFF. Unfortunately, it is difficult to formulate appropriate allowedness conditions that guarantee correct execution of the proof procedure without imposing too many unnecessary restrictions. This is a well-known problem, which is further aggravated for languages that include constraint predicates. Our proposal is to relax the above allowedness conditions, and instead to check *dynamically*, i.e. at runtime, the risk of floundering. In particular, we relax the allowedness conditions as follows.

Definition 4.4 (Allowed CIFF framework and Allowed CIFF query)**.**
A CIFF framework $\langle Th,\ A,\ IC \rangle_\Re$ *is* allowed *if the following syntactical conditions are satisfied:*

- *an iff-definition* $p(\vec{X}) \leftrightarrow D_1 \vee \cdots \vee D_n$ *is allowed iff every variable, other than those in* $\vec{X}$*, occurring in a disjunct* D_i*, also occurs inside an atomic conjunct within the same* D_i*.*

Notice that in the case of CIFF, there are no restrictions concerning integrity constraints.

A CIFF query Q *is* allowed *iff every variable in it also occurs in an atomic conjunct within the query itself.*

□

Example 4.3. *The following CIFF framework and the following CIFF query are allowed (*P_1 *is the original normal logic program with constraints):*

$$
\begin{array}{ll}
P_1: & p(Z) \\
 & p(Y) \leftarrow not\ q(Y) \\
Th_1: & p(X) \leftrightarrow [X = Z] \vee [X = Y \wedge \neg q(Y)] \\
A_1: & \oslash \\
IC_1: & Z = W \ \rightarrow \ s(Z, W) \\
Q_1: & q(Z) \wedge Y = c
\end{array}
$$

It is worth noticing that neither the above CIFF framework nor Q_1 *are IFF allowed, besides the fact that there are no constraints in them. The following CIFF framework, instead, is not allowed:*

$$
\begin{array}{ll}
Th_2: & p(X) \leftrightarrow [X = Z \wedge \neg q(Z, Y)] \\
A_2: & \oslash \\
IC_2: & q(Z) \ \rightarrow \ s(Z, W)
\end{array}
$$

The allowedness violation is due to the variable Y *in* Th_2 *which occurs only in a negative literal.*

□

In the remainder of this thesis, we will always assume that CIFF frameworks and CIFF queries are allowed.

4.1.1 CIFF proof rules

The CIFF proof procedure is a rewriting procedure, consisting of a number of *CIFF proof rules*, each of which replaces a *CIFF formula* by another one.
In the remainder, a negative literal $L = \neg A$, everywhere in a CIFF framework, in a CIFF query, or in a CIFF formula, will be written in implicative form, i.e.:
$\neg L$ is written as $L \rightarrow \bot$
Hence, in this context a literal is either an atom A or an implication $A \rightarrow \bot$.
A special case of such implication is given by the next definition.

Definition 4.5 (CIFF Inequality). *A* CIFF inequality *is an implication of the form*

$$X = t \rightarrow \bot$$

where X is an existentially quantified variable *and t is a term* not in the form of a universally quantified variable *and such that X does not occur in t*[1]. □

Definition 4.6 (CIFF formula, CIFF node and CIFF conjunct). *A* CIFF formula *F is a disjunction*

$$N_1 \vee \ldots \vee N_n \qquad n \geq 0.$$

If $n = 0$, the disjunction is equivalent to $\bot$.
Each disjunct N_i is a CIFF node *which is of the form:*

$$C_1 \wedge \ldots \wedge C_m \qquad m \geq 0.$$

If $m = 0$, the conjunction is equivalent to $\top$. Each conjunct C_i is a CIFF conjunct *and it can be of the form of:*

- *an atom (*atomic CIFF conjunct*),*
- *an implication (*implicative CIFF conjunct*, including negative literals) or*
- *a disjunction of conjunctions of literals (*disjunctive CIFF conjunct*)*

where implications are of the form:

$$L_1 \wedge \ldots \wedge L_t \rightarrow A_1 \vee \ldots \vee A_s \qquad s, t \geq 1,$$

where each L_i is a literal (possibly $\bot$ or $\top$) and each A_i is an atom (possibly $\bot$ or $\top$).
In a CIFF node *N, variables which appear either in an atomic CIFF conjunct or in a disjunctive CIFF conjunct are implicitly existentially quantified with scope N. All the remaining variables, i.e. variables occurring only in implicative CIFF conjuncts, are implicitly universally quantified with the scope being the implication in which they appear.*

Finally a CIFF node N can have an associated label *λ. We will denote a node N labeled by λ as $\lambda : N$.* □

We are now going to present the *CIFF proof rules*. In doing that, we treat a CIFF node as a *(multi)set* of CIFF conjuncts and a CIFF formula as a *(multi)set* of CIFF nodes. I.e. we represent a CIFF formula $F = N_1 \vee \ldots \vee N_n$ as

$$\{N_1, \ldots, N_n\}$$

[1]Here, the symbol = may represent either equality over Herbrand or equality over constraints.

where each N_i is a CIFF node, of the form $C_1 \wedge \ldots \wedge C_m$ represented by

$$\{C_1, \ldots, C_m\}$$

where each C_j is a CIFF conjunct.

Example 4.4. *Let us consider the following abductive logic program with constraints* $\langle P, A, IC \rangle_{\Re}$*:*

$$\begin{array}{rl} P: & p \leftarrow a \\ & p \leftarrow b \\ A: & \{a, b, c\} \\ IC: & a \rightarrow c \end{array}$$

The CIFF formula $p \wedge (a \rightarrow c)$ *(composed of a single node) is represented by:*

$$\{\{p, (a \rightarrow c)\}\}$$

The CIFF formula $[a \wedge (a \rightarrow c)] \vee [b \wedge (a \rightarrow c)]$*, composed of two CIFF nodes* $N_1 = a \wedge (a \rightarrow c)$ *and* $N_2 = b \wedge (a \rightarrow c)$ *is represented by:*

$$\{\{a, (a \rightarrow c)\}, \{b, (a \rightarrow c)\}\}.$$

□

Each *CIFF proof rule*[2] operates over a node N within a formula F and it will result in a new formula F'. A rule is presented in the following form:

Rule name	**Input:** F, N **Output** F'
Given:	a set of CIFF conjuncts χ in N
Conditions:	a set of conditions over χ and N
Action:	{**replace**, **replace_all**, **add**, **delete**} Cs'; **mark** λ

The **Given** part identifies a (possibly empty) set of conjuncts χ in N within F. A rule ϕ can be applied on a set χ of conjuncts of N satisfying the stated **Conditions**. We call such a set χ a *rule input* for ϕ. Finally, the **Action** part defines both a new set of conjuncts Cs' and an action (**replace**, **replace_all**, **add**, **delete** or **mark**) which states, as described below, how F' is obtained from F through Cs'. In the remainder we will omit to specify the **Input** part and the **Output** part.
Given a rule ϕ as above, we denote by

$$F \xrightarrow[\phi]{N,\chi} F'$$

the *application* of rule ϕ with **Input** F, N, **Given** χ, and **Output** F'.
Abstracting from the particular action, F' is always derived from F replacing the node N by a set of nodes $\mathcal{N}$, i.e.:

$$F' = F - \{N\} \cup \mathcal{N}$$

We refer to $\mathcal{N}$ as the *CIFF successor nodes* of N and we refer to each node $N' \in \mathcal{N}$ as a *CIFF successor node* of N. Each type of action defines $\mathcal{N}$ as follows:

[2]In the remainder, when we want to refer to a CIFF framework, a CIFF node, a CIFF formula and so on, we drop the prefix "CIFF" if it is clear from the context.

replace:	$\mathcal{N} = \{(N - \chi) \cup Cs'\}$
replace_all:	$\mathcal{N} = \{[(N - \chi) \cup \{D_1\}], \ldots, [(N - \chi) \cup \{D_k\}]\}$
	where $Cs' = \{D_1 \vee \ldots \vee D_k\}$
add:	$\mathcal{N} = \{N \cup Cs'\}$
delete:	$\mathcal{N} = \{N - Cs'\}$
mark:	$\mathcal{N} = \{\lambda : N\}$

The **mark** action does not change the elements in N but it marks the node N with the label λ[3]. All the actions, apart from the **replace_all** action, replace N by a single successor node.
In the **replace_all** action, Cs' consists of a single conjunct in disjunctive form, i.e. $Cs' = \{D_1 \vee \ldots \vee D_k\}$. This action adds to F a set $\mathcal{N}$ of k successor nodes, each of them obtained from N by deleting χ and by adding a single disjunct D_i.

We are now ready to specify the proof rules in detail.

In the presentation we are going to write $\vec{t} = \vec{s}$ as a shorthand for $t_1 = s_1 \wedge \cdots \wedge t_k = s_k$ (with the implicit assumption that the two vectors have the same length), and $[\vec{X}/\vec{t}]$ for the substitution $[X_1/t_1, \ldots, X_k/t_k]$. Note that X and Y will always represent variables.

Furthermore, in our presentation of the proof rules, we abstract away from the order of conjuncts in the body of an implication by writing the body of implications with the "critical" conjunct in the first position.

Recall that, in writing the proof rules, we use implicit variable quantification described in Definition 4.6.

The first proof rule replaces an atomic conjunct in a node N by its iff-definition:

R1 - Unfolding atoms

Given:	$p(\vec{t})$
Conditions:	$[p(\vec{X}) \leftrightarrow D_1 \vee \cdots \vee D_n] \in Th$
Action:	**replace** $\{\ (D_1 \vee \cdots \vee D_n)[\vec{X}/\vec{t}]\ \}$

Note that any variable in $D_1 \vee \cdots \vee D_n$ is implicitly existentially quantified in the resulting formula F'.
We assume that variable renaming may be applied so that all existential variables have distinct names in the resulting CIFF node.

Unfolding could be applied also to atoms occurring in the body of an implication yielding one new implication for every disjunct in the corresponding iff-definition:

R2 - Unfolding within implications

Given:	$(p(\vec{t}) \wedge B) \rightarrow H$
Conditions:	$[p(\vec{X}) \leftrightarrow D_1 \vee \cdots \vee D_n] \in Th$
Action:	**replace** $\{\ [(D_1[\vec{X}/\vec{t}] \wedge B) \rightarrow H], \ldots, [(D_n[\vec{X}/\vec{t}] \wedge B) \rightarrow H]\ \}$

Observe that, within F', any variable in any D_i becomes universally quantified with scope the implication in which it occurs. Also in rule **R2**, a *renaming* is assumed.

The next rule is the *propagation* rule, which allows us to resolve an atom in the body of an implication in N with a matching atomic conjunct also in N.

R3 - Propagation

Given:	$[(p(\vec{t}) \wedge B) \rightarrow H], \quad p(\vec{s})$
Conditions:	
Action:	**add** $\{\ (\vec{t} = \vec{s} \wedge B) \rightarrow H\ \}$

[3] As we will see later, λ can only be the label *undefined* in this section, while it could be the labels $undefined$ or NAF in section 4.3. When clear from the context, we will represent a CIFF node omitting its labels.

The *splitting* rule is the only rule performing a **replace_all** action. Roughly speaking it distributes a disjunction over a conjunction.

R4 - Splitting	
Given:	$D_1 \vee \cdots \vee D_n$
Conditions:	
Action:	**replace_all** $\{ D_1 \vee \ldots \vee D_n \}$

The following *factoring* rule can be used to generate two cases, one in which the given abducible atoms unify and one in which they do not:

R5 - Factoring	
Given:	$p(\vec{t}), \quad p(\vec{s})$
Conditions:	p abducible
Action:	**replace** $\{ [p(\vec{t}) \wedge p(\vec{s}) \wedge (\vec{t} = \vec{s} \rightarrow \bot)] \vee [p(\vec{t}) \wedge \vec{t} = \vec{s}] \}$

The next set of CIFF proof rules are the *constraint* rules. They manage constraint atoms and they are, in a sense, the interface to the constraint solver. They also deal with equalities and CIFF inequalities (see Definition 4.5) which can be delegated to the constraint solver if their arguments are in the constraint domain $D(\Re)$. The formal definition of the proof rules is quite complex, hence we first introduce some useful definitions.

Definition 4.7 (Basic c-atom)**.** *A* basic c-atom *is either a constraint atom whose predicate is distinct from* $=$*, or an atom of the form* $A = B$ *where* A *and* B *are not both variables.* □

As an example, $X > 3$ and $X = Y + 2$ are both basic c-atoms, whereas $X = Y$ and $X = a$ are not (where $a \notin D(\Re)$).

Definition 4.8 (basic c-conjunct and constraint variable)**.** *A* basic c-conjunct *is a basic c-atom which occurs as a CIFF conjunct in a node.*
A constraint variable *is a variable occurring in a basic c-conjunct.* □

Note that a constraint variable is always an existentially quantified variable with its scope the entire CIFF node in which it occurs. This is because it must appear in a basic c-conjunct (i.e. outside an implication).

Definition 4.9 (c-atom and c-conjunct)**.** *A* c-atom *is a constraint atom (including equality atoms) such that all the variables occurring in it are constraint variables.*
A c-conjunct *is a c-atom which occurs as a CIFF conjunct in a node.* □

We are now ready to present the first *constraint* proof rule.

R6 - Case analysis for constraints	
Given:	$(Con \wedge A) \rightarrow B$
Conditions:	Con is a c-atom
Action:	**replace** $\{ [Con \wedge (A \rightarrow B)] \vee \overline{Con} \}$

Observe that as Con is a c-atom, all the variables occurring in it are constraint variables, thus they are existentially quantified.

The next rule provides the actual *constraint solving* step itself. It may be applied to any set of c-conjuncts in a node, but to guarantee soundness, eventually, it has to be applied to the set of *all* c-conjuncts in a node. To simplify presentation, we assume that the constraint solver will fail whenever it is presented with an ill-defined constraint such as, say, $bob \leq 5$ (in the case of an arithmetic solver). For inputs that are "well-typed", however, such a situation never arises.

R7 - Constraint solving	
Given:	$Con_1, \ldots, Con_n$
Conditions:	each Con_i is a c-conjunct;
	$\{Con_1, \ldots, Con_n\}$ is not $\Re$-satisfiable
Action:	**replace** $\{ \perp \}$

The next proof rules deal with equalities (which are not constraint atoms to be handled by the constraint solver) and they rely upon the following rewrite rules which essentially implement the term reduction part of the unification algorithm of [129]:

(1) Replace $f(t_1, \ldots, t_k) = f(s_1, \ldots, s_k)$ by $t_1 = s_1 \wedge \cdots \wedge t_k = s_k$.

(2) Replace $f(t_1, \ldots, t_k) = g(s_1, \ldots, s_l)$ by $\perp$ if f and g are distinct or $k \neq l$.

(3) Replace $t = t$ by $\top$.

(4) Replace $X = t$ by $\perp$ if t contains X.

(5) Replace $t = X$ by $X = t$ if X is a variable and t is not.

(6) Replace $Y = X$ by $X = Y$ if X is a universally quantified variable and Y is not.

In the following *equality rewriting* rules, we denote as $\mathcal{E}(e)$ the result of applying the above rewrite rules (1)-(6) to the equality e. If no rewrite rule can be applied then $\mathcal{E}(e) = e$.

R8 - Equality rewriting in atoms	
Given:	$t_1 = t_2$
Conditions:	
Action:	**replace** $\{ \mathcal{E}(t_1 = t_2) \}$

R9 - Equality rewriting in implications	
Given:	$(t_1 = t_2 \wedge B) \rightarrow H$
Conditions:	
Action:	**replace** $\{ (\mathcal{E}(t_1 = t_2) \wedge B) \rightarrow H \}$

The following two *substitution rules* propagate equalities to the rest of the node. In the first case we assume that $N = (X = t \wedge Rest)$.

R10 - Substitution in atoms	
Given:	$X = t, \quad Rest$
Conditions:	$X \notin t$
Action:	**replace** $\{ X = t, \quad (Rest[X/t]) \}$

R11 - Substitution in implications	
Given:	$(X = t \wedge B) \rightarrow H$
Conditions:	X universally quantified; $X \notin t$;
Action:	**replace** $\{ (B \rightarrow H)[X/t] \}$

Note that if B is empty, then $(B \rightarrow H)[X/t]$ should be read as $(\top \rightarrow H)[X/t]$.

If none of the *equality rewriting* or *substitution* rules are applicable, then an equality in the body of an implication may give rise to a *case analysis:*

R12 - Case analysis for equalities	
Given:	$(X = t \wedge B) \rightarrow H$
Conditions:	$(X = t \wedge B) \rightarrow H$ is not of the form $X = t \rightarrow \perp$; $X \notin t$;
	X is existentially quantified; $X = t$ is not a c-atom;
	t is not a universally quantified variable
Action:	**replace** $\{ [X = t \wedge (B \rightarrow H)] \vee [X = t \rightarrow \perp] \}$

Note that the variables which occur in t become existentially quantified in the first disjunct while in the second disjunct each variable in t maintains its original quantification.
The first condition of the rule avoids applying *case analysis* if the implication $(X = t \wedge B) \rightarrow H$ is of the form $X = t \rightarrow \bot$. This is because, if it were applied, the resulting first disjunct would become $[X = t \wedge (\top \rightarrow \bot)]$ which is trivially *false*, while the second disjunct would become $X = t \rightarrow \bot$ itself. The other conditions guarantee that none of the earlier rules are applicable.

The next rule moves negative literals in the body of an implication to the head of that implication:

R13 - Negation rewriting	
Given:	$((A \rightarrow \bot) \wedge B) \rightarrow H$
Conditions:	
Action:	**replace** $\{ B \rightarrow (A \vee H) \}$

Note that if B is empty, then $B \rightarrow (A \vee H)$ should be read as $\top \rightarrow (A \vee H)$.

The following are *logical simplification* rules.

R14 - Logical simplification #1	
Given:	$\top$
Conditions:	
Action:	**delete** $\{ \top \}$

R15 - Logical simplification #2	
Given:	$(\top \wedge B) \rightarrow H$
Conditions:	B is not empty
Action:	**replace** $\{ B \rightarrow H \}$

R16 - Logical simplification #3	
Given:	$\bot \rightarrow H$
Conditions:	
Action:	**delete** $\{ \bot \rightarrow H \}$

R17 - Logical simplification #4	
Given:	$\top \rightarrow H$
Conditions:	H does not contain any universally quantified variable
Action:	**replace** $\{ H \}$

Note that the last simplification rule replaces an implication with an empty body, with its head as a CIFF conjunct. This is done only if no universally quantified variables occur in the head, otherwise we would have some universally quantified variables outside implications in a node. For example, suppose we applied the rule on $\top \rightarrow a(f(Y))$ where Y is universally quantified and a is abducible. We would obtain $a(f(Y))$ as a conjunct in a node, thus leading to two main problems: (1) the variable quantification cannot be implicit and, even worse, (2) the semantics should be extended to the case of *infinitely* many instantiations of abducible atoms in an abductive answer. The case where H does have a universally quantified variable is dealt with by the *Dynamic Allowedness* rule, which is used to identify nodes with problematic quantification patterns, which could lead to floundering:

R18 - Dynamic allowedness (DA)	
Given:	$B \rightarrow H$
Conditions:	either $B = \top$ or B consists of constraint atoms alone; no other rule applies to the implication
Action:	**mark** *undefined*

Due to the definition of the other CIFF proof rules, the implication $B \rightarrow H$ to which **DA** is applied, falls in one of the following cases:

1. $B = \top$ and there is a universally quantified variable in H;
2. there is a c-atom in B with an universally quantified variable occurring in it.

With the **DA** rule, we avoid to obtain infinitely many abducible atoms in an abductive answer. For example, let us consider an implication of the form $X > Y \rightarrow H$ such that X is universally quantified. Depending on $D(\Re)$, there could be infinitely many instances of X satisfying the c-atom and CIFF should handle *all* those cases. However, we believe that **DA** could be relaxed, in particular for those implications falling in the case 2 above. Consider, for example, the following implication:

$$X > 3 \wedge X < 100 \rightarrow a(X)$$

where X is universally quantified and a is an abducible predicate. If $D(\Re)$ is the set of all integers, there is a finite set of abducible atoms satisfying the implication, i.e. the set $\{a(4), a(5), \ldots, a(99)\}$. However, **DA** marks a node with this implication as undefined due to the presence of X. The relaxation of **DA** is not in the scope of this thesis.

4.1.2 CIFF derivation and answer extraction

The CIFF proof rules are the building blocks of a *CIFF derivation* which defines the process of computing answers with respect to a framework $\langle Th,\ A,\ IC\rangle_{\Re}$ and a query Q.
Prior to defining a CIFF derivation formally, we introduce some useful definitions.

Definition 4.10 (Failure and undefined CIFF nodes)**.** *A CIFF node N which contains $\bot$ as an atomic CIFF conjunct is called a* failure CIFF node*. A CIFF node N marked as undefined is called an* undefined CIFF node*.* □

Definition 4.11 (CIFF selection function)**.** *Let F be a CIFF formula. We define a* CIFF selection function *$\mathcal{S}$ as a function such that:*

$$\mathcal{S}(F) = \langle N, \phi, \chi\rangle$$

where N is a CIFF node in F, ϕ is a CIFF proof rule and χ is a set of CIFF conjuncts in N such that χ is a rule input *for ϕ.*

□

We are now ready to define a *CIFF pre-derivation* and a *CIFF branch.*

Definition 4.12 (CIFF Pre-derivation and initial formula)**.**
Let $\langle Th,\ A,\ IC\rangle_{\Re}$ be a CIFF framework, let Q be a query and let $\mathcal{S}$ be a CIFF selection function*. A* CIFF pre-derivation *for Q with respect to $\langle Th,\ A,\ IC\rangle_{\Re}$ and $\mathcal{S}$ is a (finite or infinite) sequence of CIFF formulae $F_1, F_2, \ldots, F_i, F_{i+1}\ldots$ such that each F_{i+1} is obtained from F_i through $\mathcal{S}$ as follows:*

- *$F_1 = \{N_1\} = \{Q \cup IC\}$, where Q and IC are treated as sets of CIFF conjuncts, (we will refer to F_1 as the* initial formula *of a CIFF pre-derivation)*
- *$\mathcal{S}(F_i) = \langle N_i, \phi_i, \chi_i\rangle$ such that N_i is neither an undefined CIFF node nor a failure CIFF node and*
- $F_i \xrightarrow[\phi_i]{N_i, \chi_i} F_{i+1}$

□

The construction of a pre-derivation can be interpreted as the construction of an or-tree rooted at N_1 and whose nodes are CIFF nodes. Roughly speaking, the whole or-tree can be seen as a search tree for answers to the query. Note that all the variables in the query are existentially quantified in N_1 because the allowedness conditions of Definition 4.4 impose that each variable in Q occurs in an atomic conjunct of Q.
CIFF formulas F_i in a pre-derivation correspond to successive frontiers of the search tree. Each derivation step is done by applying (through $\mathcal{S}$) the *selected* proof rule on a set χ of CIFF conjuncts within a node N in a frontier. The resulting frontier is obtained by replacing N by the set of *successor nodes* $\mathcal{N}$.

Definition 4.13 (Successor Nodes in a CIFF pre-derivation)**.** *Let $\mathcal{D}$ be a CIFF pre-derivation for a query Q with respect to a CIFF framework $\langle Th,\ A,\ IC\rangle_{\Re}$ and a selection function $\mathcal{S}$.*
We say that $\mathcal{N}$ is the set of successor nodes of N in $\mathcal{D}$*, if*

- $\mathcal{S}(F_i) = \langle N, \phi_i, \chi_i\rangle$,
- $F_i \xrightarrow[\phi_i]{N,\chi_i} F_{i+1}$, *and*
- *for each N' in $\mathcal{N}$, $N' \in F_{i+1}$ and $N' \notin F_i$.*

Moreover we say that a node N' in $\mathcal{N}$ is a successor node of N in $\mathcal{D}$*.* □

Definition 4.14 (CIFF branch)**.** *Given a CIFF pre-derivation $\mathcal{D} = F_1, F_2, \ldots, F_i, F_{i+1}\ldots$, a* CIFF branch $\mathcal{B}$ *in $\mathcal{D}$ is a (finite or infinite) sequence of CIFF nodes $N_1, N_2, \ldots, N_i, N_{i+1}\ldots$ such that each $N_i \in F_i$ and each N_{i+1} is a CIFF successor node of N_i in $\mathcal{D}$.* □

The next step, finally, is the definition of a CIFF derivation.

Definition 4.15 (CIFF derivation)**.** *Let $\langle Th,\ A,\ IC\rangle_{\Re}$ be a CIFF framework, let Q be a query and let $\mathcal{S}$ be a* CIFF selection function*. A* CIFF derivation *$\mathcal{D}$ for Q with respect to $\langle Th,\ A,\ IC\rangle_{\Re}$ is a CIFF pre-derivation $F_1, F_2, \ldots$ such that for each CIFF branch $\mathcal{B}$ in $\mathcal{D}$ if*

- $\mathcal{S}(F_i) = \langle N_i, \phi, \chi\rangle$,
- $\mathcal{S}(F_j) = \langle N_j, \phi, \chi\rangle$,
- $N_i \in \mathcal{B}$,
- $N_j \in \mathcal{B}$ *and*
- $i \neq j$

*then $\phi \notin \{$**Propagation**, **Factoring**, **Equality rewriting in atoms**, **Equality rewriting in implications**, **Substitution in atoms**$\}$.* □

Informally, a derivation is a pre-derivation such that in each branch certain proof rules can be applied only once to a given set of selected CIFF conjuncts. This is because those rules can produce loops if they are applied repeatedly to the same set of conjuncts. The concept of successor nodes in a pre-derivation is valid also for a derivation.

Example 4.5. *Consider the following framework $\langle Th, A, IC\rangle_{\Re}$:*

$$\begin{array}{ll} Th: & p \leftrightarrow \bot \\ A: & \{a\} \\ IC: & p \ \rightarrow \ a \end{array}$$

The following is a pre-derivation $\mathcal{D}$ for the query $Q = p$.

$$\begin{array}{lll} F_1 = & \{\{p, [p \rightarrow a]\}\} & \textit{[\textbf{Init}]} \\ F_2 = & \{\{p, [p \rightarrow a], [\top \rightarrow a]\}\} & \textit{[\textbf{R3}]} \\ F_3 = & \{\{p, [p \rightarrow a], [\top \rightarrow a], [\top \rightarrow a]\}\} & \textit{[\textbf{R3}]} \\ & \vdots & \end{array}$$

*The **Propagation** rule **R3** can be applied repeatedly to the integrity constraint giving rise to an infinite pre-derivation which should be avoided in a derivation*[4]. □

Definition 4.16 (Successor CIFF Derivation)**.** *Let* $\mathcal{D}= F_1, \ldots, F_i$ *be a CIFF derivation, let* $\mathcal{S}$ *be a CIFF selection function and let* $N \in F_i$. *We say that* $D' = F_1, \ldots, F_{i+1}$ *is a* successor CIFF derivation via N of $\mathcal{D}$ *if*

- $\mathcal{S}(F_i) = \langle N, \phi_i, \chi_i \rangle$,
- $F_i \xrightarrow[\phi_i]{N, \chi_i} F_{i+1}$, *and*
- D' *is a CIFF derivation,*

□

Definition 4.17 (Leaf and successful CIFF nodes)**.** *Let* $\mathcal{D}= F_1, \ldots, F_i$ *be a CIFF derivation. A CIFF node* N *in* F_i *is a* leaf CIFF node *if*

- *it is a* failure CIFF node *or*
- *it is an* undefined CIFF node *or*
- *there exists no successor CIFF derivation via* N *of* $\mathcal{D}$.

A leaf node *which is neither a failure CIFF node nor an undefined CIFF node is called a* successful CIFF node. □

We are now ready to introduce the following classifications of *CIFF branches* and *CIFF derivations.*

Definition 4.18 (Failure, undefined and successful CIFF branches)**.** *Let* $\mathcal{D}$ *be a CIFF derivation and let* $B= N_1, \ldots, N_k$ *be a CIFF branch in* $\mathcal{D}$. *We say that* $\mathcal{B}$ *is*

- *a* successful CIFF branch *if* N_k *is a successful CIFF node;*
- *a* failure CIFF branch *if* N_k *is a failure CIFF node;*
- *an* undefined CIFF branch *if* N_k *is an undefined CIFF node.*

□

Definition 4.19 (Failure and Successful CIFF Derivations)**.** *Let* $\mathcal{D}$ *be a CIFF derivation.* $\mathcal{D}$ *is called a* successful CIFF derivation *if it contains at least one* successful CIFF branch. $\mathcal{D}$ *is called a* failure CIFF derivation *if all its branches are* failure CIFF branches. □

Intuitively, an abductive answer to a query Q can be extracted from a successful node of a *successful derivation.* Formally:

[4] The example shows the need of *multisets* for representing correctly CIFF formulae and CIFF nodes.

Table 4.1: CIFF proof rules

R1	Unfolding atoms
R2	Unfolding in implications
R3	Propagation
R4	Splitting
R5	Factoring
R6	Case analysis for constraints
R7	Constraint solving
R8	Equality rewriting in atoms
R9	Equality rewriting in implications
R10	Substitution in atoms
R11	Substitution in implications
R12	Case analysis for equalities
R13	Negation rewriting
R14	Logical Simplification #1
R15	Logical Simplification #2
R16	Logical Simplification #3
R17	Logical Simplification #4
R18	Dynamic Allowedness

Definition 4.20 (CIFF Extracted Answer)**.** *Let $\langle Th, A, IC\rangle_{\Re}$ be a CIFF framework and let Q be a CIFF query. Let $\mathcal{D}$ be a successful CIFF derivation for Q with respect to $\langle Th, A, IC\rangle_{\Re}$. A* CIFF extracted answer *from a successful node N of $\mathcal{D}$ is a pair*

$$\langle \Delta, C\rangle$$

where Δ is the set of abducible atomic conjuncts in N, and $C = \langle \Gamma, E, DE\rangle$ where:

- *Γ is the set of all the c-conjuncts in N,*
- *E is the set of all the equality atoms in N,*
- *DE is the set of all the CIFF inequalities in N.*

□

The soundness of the CIFF proof procedure with respect to the notion of $\Re$-satisfiability and the three-valued completion semantics is the subject of the next section. The idea is to show that CIFF extracted answers correspond to abductive answers with constraints in the sense of Definition 4.1.

Example 4.6. *Consider the following framework $\langle Th, A, IC\rangle_{\Re}$, obtained from the abductive logic program with constraints of Example 3.11, and the following query Q:*

$$\begin{array}{ll} Th: & p(T) \leftrightarrow T = X \wedge q(T_1, T_2) \wedge T_1 < X \wedge X < 8 \\ & q(X, Y) \leftrightarrow X = X_1 \wedge Y = X_2 \wedge s(X_1, a) \\ A: & \{r, s\} \\ IC: & r(Z) \;\rightarrow\; p(Z) \\ Q: & r(Y) \end{array}$$

The following is a CIFF derivation $\mathcal{D}$ for Q with respect to $\langle Th, A, IC\rangle_{\Re}$ (the CIFF proof rules are summarized in Table 4.1):

$F_1 = \{\{r(Y), [r(Z) \rightarrow p(Z)]\}\}$ ***[Init]***
$F_2 = \{\{r(Y), [Z = Y \rightarrow p(X)], [r(Z) \rightarrow p(Z)]\}\}$ ***[R3]***
$F_3 = \{\{r(Y), [\top \rightarrow p(Y)], [r(Z) \rightarrow p(Z)]\}\}$ ***[R11]***
$F_4 = \{\{r(Y), p(Y), [r(Z) \rightarrow p(Z)]\}\}$ ***[R17]***
$F_5 = \{\{r(Y), Y = X, q(T_1, T_2), T_1 < X, X < 8, [r(Z) \rightarrow p(Z)]\}\}$ ***[R1]***
$F_6 = \{\{r(X), Y = X, q(T_1, T_2), T_1 < X, X < 8, [r(Z) \rightarrow p(Z)]\}\}$ ***[R10]***
$F_7 = \{\{r(X), Y = X, T_1 = V, T_2 = W, s(T_1, a), T_1 < X, X < 8, [r(Z) \rightarrow p(Z)]\}\}$ ***[R1]***
$F_8 = \{\{r(X), Y = X, T_1 = V, T_2 = W, s(V, a), V < X, X < 8, [r(Z) \rightarrow p(Z)]\}\}$ ***[R10]***

No more new rules can be applied to the only node in F_8 and this is neither a failure node nor an undefined node. Hence, it is a successful node *from which we extract the following answer:*

$$\langle \{r(X), s(V, a)\}, C \rangle$$

where $C = \langle \Gamma, E, DE \rangle$ is:

$$\begin{aligned} \Gamma &: \{Y = X, T_1 = V, V < X, X < 8\} \\ E &: \{T_2 = W\} \\ DE &: \oslash \end{aligned}$$

Indeed, the abductive answer with constraints *given in Example 3.11 (modulo variable renaming and instantiation of T due to the ground query in Example 3.11).* □

We represent the example 3.11 of abductive logic programming with constraints.

Example 4.7. *Consider the following framework $\langle Th, A, IC \rangle_{\Re}$, and the following query Q:*

$$\begin{aligned} Th &: p(X) \leftrightarrow X = Z \wedge a(Z) \wedge Z < 5 \\ A &: \{a\} \\ IC &: a(2) \rightarrow \bot \\ Q &: p(Y) \end{aligned}$$

The following is a CIFF derivation $\mathcal{D}$ for Q with respect to $\langle Th, A, IC \rangle_{\Re}$:

$F_1 = \{\{p(Y), [a(2) \rightarrow \bot]\}\}$ ***[Init]***
$F_2 = \{\{Y = Z, a(Z), Z < 5, [a(2) \rightarrow \bot]\}\}$ ***[R1]***
$F_3 = \{\{Y = Z, a(Z), Z < 5, [2 = Z \rightarrow \bot]\}\}$ ***[R3]***
$F_4 = \{\{Y = Z, a(Z), Z < 5, [Z = 2 \rightarrow \bot]\}\}$ ***[R9]***
$F_5 = \{\{Y = Z, a(Z), Z < 5, [Z \neq 2 \vee [Z = 2, (\top \rightarrow \bot)]]\}\}$ ***[R6]***
$F_6 = \{\{\{Y = Z, a(Z), Z < 5, Z \neq 2\}, \{Y = Z, a(Z), Z < 5, Z = 2, (\top \rightarrow \bot)\}\}$ ***[R4]***
$F_6 = \{\{\{Y = Z, a(Z), Z < 5, Z \neq 2\}, \{Y = Z, a(Z), Z < 5, Z = 2, \bot\}\}$ ***[R17]***

No more new rules can be applied to the two nodes in F_6. The first node is neither a failure node nor an undefined node. Hence, it is a successful node *from which we extract the following answer:*

$$\langle \{a(Z)\}, \langle \{Y = Z, Z < 5, Z \neq 2\}, \oslash, \oslash \rangle \rangle$$

□

4.2 Soundness of the CIFF Proof Procedure

As anticipated in the previous section, the CIFF proof procedure is sound with respect to the three-valued completion semantics, i.e. each CIFF extracted answer is indeed a CIFF correct answer in the sense of Definition 4.1. All the results stated in this section are based upon the results given in [81] for the IFF proof procedure.
The main result of this section is the following theorem.

Theorem 4.1 (CIFF Soundness). *Let $\langle P, A, IC\rangle_{\Re}$ be an abductive logic program with constraints such that the corresponding CIFF framework is $\langle Th, A, IC\rangle_{\Re}$. Let $\langle\Delta, C\rangle$, where $C = \langle\Gamma, E, DE\rangle$, be a CIFF extracted answer from a successful CIFF node in a CIFF derivation with respect to $\langle Th, A, IC\rangle_{\Re}$ and a CIFF query Q.*

Then there exists a ground substitution σ such that $\langle\Delta, \sigma, \Gamma\rangle$ is an abductive answer with constraints to Q wrt $\langle P, A, IC\rangle_{\Re}$.

□

The proof of the theorem relies upon the following propositions. The first proposition shows that given a CIFF extracted answer $\langle\Delta, C\rangle$ there exists a substitution which satisfies all the constraint atoms, equality atoms and CIFF inequalities in C.

Proposition 4.1. *Let $\langle\Delta, C\rangle$ be a CIFF extracted answer from a successful CIFF node N, where $C = \langle\Gamma, E, DE\rangle$.*

Then:

1. *there exists a ground substitution θ such that $\theta \models_{3(\Re)} \Gamma$, and*
2. *for each such ground substitution θ, there exists a ground substitution σ such that*

$$\theta\sigma \models_{3(\Re)} \Gamma \cup E \cup DE$$

□

Proof of Proposition 4.1. To prove the first part of the proposition we need the semantics of the constraint solver while to prove the second part we need the *Clark Equality Theory (CET)*. Both are embedded in our semantics ($\models_{3(\Re)}$) and we will write explicitly $\models_{\Re}$ and $\models_{CET}$, respectively, instead of $\models_{3(\Re)}$ where appropriate.

1. Γ is the set of c-conjuncts in N, and this is a successful node. Then the **Constraint solving** rule **R7** cannot be applied to N. Thus, by the assumption of having a sound and complete constraint solver, we have that Γ is not an unsatisfiable set of constraints, i.e. we can always obtain a ground substitution θ such that:

   $$\models_{\Re} \Gamma\theta$$

   and so

   $$\theta \models_{\Re} \Gamma.$$

2. Let us consider $F = (E \cup DE)\theta$. Equalities in E are of the form

   $$X_i = t_i \qquad (1 \leq i \leq n, n \geq 0)$$

   where each X_i is an existentially quantified variable and t_i is a term (containing neither universally quantified variables nor X_i itself). The scope of each variable in E is the whole CIFF node N and each X_i does not appear elsewhere in the node due to the exhaustive application of the **Equality rewriting in atoms** rule **R8**.

   The inequalities in DE are of the form

   $$X_j = t_j \rightarrow \bot \qquad (n < j \leq m, m \geq 0)$$

where each X_j is an existentially quantified variable appearing also in E (due to the **Substitution in atoms** rule **R10**) and t_j is a term not in the form of a universally quantified variable.

The ground substitution θ contains an assignment to all the constraint variables occurring in $(E \cup DE)$. This is because (i) all the equalities in E which are c-conjuncts are also in Γ and (ii) there is no CIFF inequality in DE of the form $X_i = t_i \rightarrow \bot$ where $X_i = t_i$ is a c-atom because the **Case analysis for constraints** rule **R6** replaced any such CIFF inequality with a c-conjunct of the form $X_i \neq t_i$.

Note that also CIFF inequalities of the form $X = Y \rightarrow \bot$ such that X is a constraint variable and Y is not (or viceversa) are not a problem. This is because X has been substituted by a ground term c by θ and there is no equality of the form $Y = c$ in E: otherwise that equality should belong to Γ and Y would be a constraint variable too.

Finally, the proposition is proven by finding a ground substitution σ such that $\models_{CET} F\sigma$ and this can be done following the proof in [81], as follows.

First we assign a value to each existentially quantified variable X_j in $DE\theta$. We do this by using a fresh function symbol g_j, i.e. the function symbol g_j does not appear in the CIFF branch whose leaf is N (we assume here that we have an infinite number of distinct function symbols in our language). Then we choose a constant c and we assign $g_j(c)$ to X_j. We define $G = F\sigma_I$ where σ_I is the ground substitution composed of the above assignments.

The second step is to assign to each variable X_i in $(E\theta)\sigma_I$ its corresponding term $s_i = t_i\sigma_I$.

Finally, for each remaining existentially quantified variable, we use another fresh function and a constant c to make assignments as was done for CIFF inequalities.

The whole set of assignments so far obtained is the ground substitution σ which proves the proposition. This is because, after $\theta\sigma$ has been applied, each equality originally in E is of the form $t = t$ and each CIFF inequality originally in DE is of the form $f(t) = g(t) \rightarrow \bot$ which are obviously entailed by CET.

We have:

$$\sigma \models_{3(\Re)} (E \cup DE)\theta$$

and thus, being $\theta \models_{\Re} \Gamma$, we have:

$$\theta\sigma \models_{3(\Re)} \Gamma \cup E \cup DE$$

$\square$

Example 4.8. *Given* $\Gamma = \{2 \leq T, T < 4\}$, $E = \{X = f(Y), Z = g(V)\}$ *and* $DE = \{(Y = h(W,V)) \rightarrow \bot\}$, *we have that both* $\theta_1 = \{T/2\}$ *and* $\theta_2 = \{T/3\}$ *satisfy* Γ *and they contain all the possible assignments for* T *(given that* $D(\Re)$ *is the set of all integers). We can obtain a ground substitution* $\theta_1\sigma$ *(with* $\sigma = \sigma_{DE} \cup \sigma_E$*) as follows:*

1. $\sigma_{DE} = \{Y/r(c)\}$ *obtaining* $S_1 = ((E \cup DE)\theta)\sigma_{DE} =$
   $\{X = f(r(c)), Z = g(V), (r(c) = h(W,V)) \rightarrow \bot\}$

2. *the second step is to assign the corresponding terms to* X *and* Z *obtaining*

$$S_2 = \{f(r(c)) = f(r(c)), g(V) = g(V), (r(c) = h(W,V)) \rightarrow \bot\}$$

3. finally we assign new terms with fresh functions to the remaining existentially quantified variable V, e.g. $\sigma_E = \{V/t(c)\}$ obtaining

$$S_3 = \{f(r(c)) = f(r(c)), g(t(c)) = g(t(c)), (r(c) = h(W, t(c))) \rightarrow \bot\}$$

The set S_3 is clearly entailed by CET. Note that we do not care about the universally quantified variable W in S_3. This is because

$$(r(c) = h(W, t(c))) \rightarrow \bot$$

is entailed by CET for any assignment to W, due to the fact that r and h are distinct function symbols.

Similarly, we can obtain another ground substitution using θ_2. □

The next proposition directly extends the above result to the set Δ of a CIFF extracted answer.

Proposition 4.2. *Let $\langle \Delta, C \rangle$ be a CIFF extracted answer from a successful CIFF node N where $C = \langle \Gamma, E, DE \rangle$. For each ground substitution σ' such that $\sigma' \models_{3(\Re)} \Gamma \cup E \cup DE$, there exists a ground substitution σ which extends σ' for the variables that are in Δ but not in C such that*

1. $\sigma' \subseteq \sigma$
2. $\Delta\sigma \models_{3(\Re)} \Delta \cup \Gamma \cup E \cup DE$

□

Proof of Proposition 4.2. Let us consider the set $\Delta\sigma'$. There can be existentially quantified variables in Δ not assigned by σ' because they do not appear in C. Then it is enough to choose arbitrary ground terms to assign to those variables to obtain a substitution σ such that $\sigma' \subseteq \sigma$, which proves the proposition. □

The third proposition shows that the CIFF proof rules are indeed equivalence preserving rules with respect to the three-valued completion semantics. This a basic requirement to prove the soundness of CIFF.

Proposition 4.3 (Equivalence Preservation)**.** *Given an abductive logic program with constraints $\langle P,\ A,\ IC \rangle_\Re$, a CIFF node N and a set of CIFF successor nodes $\mathcal{N}$ obtained by applying a CIFF proof rule ϕ to N, it holds that:*

$$P \models_{3(\Re)} N \quad \textit{iff} \quad P \models_{3(\Re)} \mathcal{N}^\vee$$

where $\mathcal{N}^\vee$ is the disjunction of the nodes in $\mathcal{N}$. □

Proof of Proposition 4.3. We prove the proposition considering each of the CIFF proof rules in turn. Recall that, apart from the **Splitting** rule, for each proof rule the set $\mathcal{N}$ of successor nodes of N is a singleton, i.e. $\mathcal{N} = \{N'\}$.

R1 - Unfolding atoms. This rule applies a resolution step on a defined atom $p(\vec{t})$ in N and its iff-definition in Th:

$$p(\vec{X}) \leftrightarrow (D_1 \vee \cdots \vee D_n)$$

Hence, the atom $p(\vec{t})$ is replaced in N' by

$$(D_1 \vee \cdots \vee D_n)[\vec{X}/\vec{t}]$$

The replacement is obviously equivalence preserving with respect to P and $\models_{3(\Re)}$.

R2 - Unfolding within implication. This rule resolves a defined atom $p(\vec{t})$ with its iff-definition in Th:

$$p(\vec{X}) \leftrightarrow (D_1 \vee \cdots \vee D_n)$$

as in the previous rule. The result is a set of implications in N' replacing the original implication, each one containing one of the disjuncts $D_i\theta$, with $1 \leq i \leq n$ where $\theta = [\vec{X}/\vec{t}]$. Without loss of generality, suppose that the original implication is of the form

$$(p(\vec{t}[\vec{W},\vec{Y}]) \wedge R[\vec{W},\vec{Y}]) \rightarrow H[\vec{W},\vec{Y}]$$

where R is a conjunction of literals and H is a disjunction of atoms. We use the notation $E[\vec{Y}]$ to say that $\vec{Y}$ may occur in E for a generic E. Suppose that all and only the variables in $\vec{W}$ occur also in another non-implicative CIFF conjunct (recall that in a CIFF node variables appearing only within an implication are implicitly universally quantified with scope the implication itself and variables appearing outside an implication are existentially quantified with scope the whole node). Making the quantification explicit, the implication becomes:

$$\exists\vec{W}\forall\vec{Y}(p(\vec{t}[\vec{W},\vec{Y}]) \wedge R[\vec{W},\vec{Y}] \rightarrow H[\vec{W},\vec{Y}])$$

To simplify the presentation, in the following we assume that $\vec{W}$ and $\vec{Y}$ may occur everywhere in the implication without denoting it explicitly. Applying resolution we obtain:

$$\exists\vec{W}\forall\vec{Y}((\exists\vec{Z}_1 D'_1\theta \vee \cdots \vee \exists\vec{Z}_n D'_n\theta) \wedge R \rightarrow H)$$

where each D_i is of the form $\exists\vec{Z}_i D'_i$ and the vectors $\vec{Z}_i$ of existentially quantified variables arise from the iff-definition. Thus we have:

$$\begin{array}{ll}
\exists\vec{W}\forall\vec{Y}((\exists\vec{Z}_1(D'_1\theta) \vee \cdots \vee \exists\vec{Z}_n(D'_n\theta)) \wedge R \rightarrow H) & \equiv \\
\exists\vec{W}\forall\vec{Y}(\neg(\exists\vec{Z}_1(D'_1\theta) \vee \cdots \vee \exists\vec{Z}_n(D'_n\theta)) \vee \neg R \vee H) & \equiv \\
\exists\vec{W}\forall\vec{Y}((\neg(\exists\vec{Z}_1(D'_1\theta)) \wedge \cdots \wedge \neg(\exists\vec{Z}_n(D'_n\theta))) \vee \neg R \vee H) & \equiv \\
\exists\vec{W}\forall\vec{Y}((\neg(\exists\vec{Z}_1(D'_1\theta)) \vee \neg R \vee H) \wedge \cdots \wedge (\neg(\exists\vec{Z}_n(D'_n\theta)) \vee \neg R \vee H)) & \equiv \\
\exists\vec{W}(\forall\vec{Y}(\neg(\exists\vec{Z}_1(D'_1\theta)) \vee \neg R \vee H) \wedge \cdots \wedge \forall\vec{Y}(\neg(\exists\vec{Z}_n(D'_n\theta)) \vee \neg R \vee H)) & \equiv \\
\exists\vec{W}(\forall\vec{Y},\vec{Z}_1(\neg D'_1\theta \vee \neg R \vee H) \wedge \cdots \wedge \forall\vec{Y},\vec{Z}_n(\neg D'_n\theta \vee \neg R \vee H)) & \equiv \\
\exists\vec{W}(\forall\vec{Y},\vec{Z}_1(D'_1\theta \wedge R \rightarrow H) \wedge \cdots \wedge \forall\vec{Y},\vec{Z}_n(D'_n\theta \wedge R \rightarrow H)) &
\end{array}$$

Note that the variables $\vec{Z}_i$ in the new implications are universally quantified with scope the implication in which they occur. So with our convention for implicit quantification, the last sentence is:

$$(D_1\theta \wedge R \rightarrow H) \wedge \cdots \wedge (D_n\theta \wedge R \rightarrow H).$$

R3 - Propagation. This rule uses an atomic CIFF conjunct $p(\vec{s})$ and an atom $p(\vec{t})$ within an implication of the form $(p(\vec{t}) \wedge B) \rightarrow H$ and it adds in N' an implication of the form:

$$\vec{t} = \vec{s} \wedge B \rightarrow H$$

It is obvious that, due to the fact that the second implication is a consequence of the CIFF conjunct and the implication and both remain in N', the **Propagation** rule is equivalence preserving.

R4 - Splitting. This rule uses a disjunctive CIFF conjunct of the form $D = D_1 \vee \ldots \vee D_k$ and builds a set of CIFF successor nodes $\mathcal{N} = \{N_1, \ldots, N_k\}$ such that in each N_i the conjunct D is replaced by D_i.
It is obvious that the **Splitting** rule is equivalence preserving because it is an operation of disjunctive distribution over a conjunction, i.e. is a case of the tautology:

$$A \wedge (D_1 \vee \ldots \vee D_k) \equiv (A \vee D_1) \wedge \ldots \wedge (A \vee D_k)$$

R5 - Factoring. This rule uses two atomic CIFF conjuncts of the form $p(\vec{t})$ and $p(\vec{s})$ and it replaces them in N' by a disjunction of the form:

$$(p(\vec{s}) \wedge p(\vec{t}) \wedge (\vec{t} = \vec{s} \rightarrow \bot)) \vee (p(\vec{t}) \wedge \vec{t} = \vec{s})$$

To show that the rule is equivalence preserving, consider the tautology

$$(\vec{t} = \vec{s} \rightarrow \bot) \vee \vec{t} = \vec{s}$$

We have that

$$\begin{array}{ll}
p(\vec{t}) \wedge p(\vec{s}) & \equiv \\
p(\vec{t}) \wedge p(\vec{s}) \wedge ((\vec{t} = \vec{s} \rightarrow \bot) \vee \vec{t} = \vec{s}) & \equiv \\
(p(\vec{t}) \wedge p(\vec{s}) \wedge (\vec{t} = \vec{s} \rightarrow \bot)) \vee (p(\vec{t}) \wedge p(\vec{s}) \wedge \vec{t} = \vec{s}) & \equiv \\
(p(\vec{t}) \wedge p(\vec{s}) \wedge (\vec{t} = \vec{s} \rightarrow \bot)) \vee (p(\vec{t}) \wedge p(\vec{t}) \wedge \vec{t} = \vec{s}) & \equiv \\
(p(\vec{t}) \wedge p(\vec{s}) \wedge (\vec{t} = \vec{s} \rightarrow \bot)) \vee (p(\vec{t}) \wedge \vec{t} = \vec{s}) &
\end{array}$$

R6 - Case Analysis for constraints.
Recall that variables in Con are all existentially quantified and that the constraint domain is assumed to be closed under complement, i.e. the complement $\overline{Con}$ of a constraint atom Con is a constraint atom.

$$\begin{array}{ll}
(Con \wedge A) \rightarrow B & \equiv \\
Con \rightarrow (A \rightarrow B) & \equiv \\
(Con \rightarrow Con) \wedge (Con \rightarrow (A \rightarrow B)) & \equiv \\
Con \rightarrow (Con \wedge (A \rightarrow B)) & \equiv \\
\neg Con \vee (Con \wedge (A \rightarrow B)) & \equiv \\
\overline{Con} \vee (Con \wedge (A \rightarrow B)) &
\end{array}$$

Variable quantification need not be taken into account here because each variable occurring in Con must be existentially quantified in order for the rule to be applied to it. Hence the quantification of those variables remain unchanged in the two resulting disjuncts.

R7 - Constraint solving. This rules replaces a set $\{Con_1, \ldots, Con_k\}$ of constraint atoms, which occur as CIFF conjuncts in N, by $\bot$ in N', provided the constraint solver evaluates them as unsatisfiable. By the assumption that the constraint solver is sound and complete, the rule is obviously equivalence preserving.

R8 - Equality rewriting in atoms and **R9 - Equality rewriting in implications**. These rules are directly borrowed from the Martelli-Montanari unification algorithm. The equivalence preserving is proven by the soundness of this algorithm [129].

R10 - Substitution in atoms and **R11 - Substitution in implications**. These rules simply propagate an equality either to the whole node or to the implication in which it occurs. Again they are obviously equivalence preserving rules.

R12 - Case Analysis for equality. The equivalence preservation of this rule requires some carefulness due to the quantification of the variables involved. First of all notice that if no variable in the **Given** formula is universally quantified the proof is trivial. For simplicity we provide the full proof for the case in which the **Given** formula contains only one universally quantified variable and no other existentially quantified variables except X. The proof can be then easily adapted to the general case. With this simplification, we need to prove that the following two formulae are equivalent (where implicit quantifications are made explicit).

F1 $\exists X\, \forall Y((X = t \wedge B) \rightarrow H)$

F2 $[\exists X, Y(X = t \wedge (B \rightarrow H))] \vee [\exists X \forall Y(X = t \rightarrow \bot)]$

We do a *proof by cases*, using the following two (complementary) hypotheses:

Hyp1 $\neg\exists X \exists Y (X = t)$.
Hyp2 $\exists X \exists Y (X = t)$

The equivalence under **Hyp1** is trivial.

Assume **Hyp2** holds. Let s be a ground value for X such that

$\exists Y (s = t)$.

and let ϑ be the ground substitution for X and Y such that $X\vartheta = s$ and $(X = t)\vartheta$. Notice that, by CET, given s such a ground substitution is unique. Consider now the formulae obtained from **F1** and **F2** by substituting X by s

F1(s) $\forall Y ((s = t \wedge B) \rightarrow H)$
F2(s) $[\exists Y (s = t \wedge (B \rightarrow H))] \vee [\forall Y (s = t \rightarrow \bot)]$

It is not difficult to see that **F1(s)** is equivalent to

$(B \rightarrow H)\vartheta$

since for any ground instantiation of Y other than $Y\vartheta$ the implication $((s = t \wedge B) \rightarrow H)$ is trivially true.

Consider now **F2(s)**. The second disjunct is false by **Hyp2** whereas the first disjunct is clearly equivalent to $(B \rightarrow H)\vartheta$ due to the uniqueness of ϑ.

R13 - Negation rewriting. This rule uses common logical equivalences:

$$\begin{array}{ll} ((A \rightarrow \bot) \wedge B) \rightarrow H & \equiv \\ B \rightarrow \neg(A \rightarrow \bot) \vee H & \equiv \\ B \rightarrow \neg(\neg A \vee \bot) \vee H & \equiv \\ B \rightarrow (A \wedge \top) \vee H & \equiv \\ B \rightarrow A \vee H & \end{array}$$

R14, R15, R16, R17 - Logical simplification #1 - #4 rules. All the four simplification rules are again obviously equivalence preserving rules as they use common logical equivalences.

R18 - Dynamic Allowedness. This rule does not change the elements of a node N. Hence, given that $N' = N$, ignoring the marking, the equivalence preservation is proven. □

The following corollary follows immediately from Proposition 4.3.

Corollary 4.1 (Equivalence Preservation of CIFF Formulae)**.** *Let $\langle P, A, IC\rangle_\Re$ be an abductive logic program with constraints, F a CIFF formula and $\mathcal{S}$ any CIFF selection function. Let $\mathcal{S}(F) = \langle N, \phi, \chi\rangle$ and F' the result of applying ϕ to N in F. Then:*

$$P \cup IC \models_{3(\Re)} F \quad \textit{iff} \quad P \cup IC \models_{3(\Re)} F', \textit{ i.e.}$$

$$P \cup IC \models_{3(\Re)} (F \leftrightarrow F')$$

.

□

Proof of Corollary 4.1. The proof is an immediate consequence of Proposition 4.3, because for any CIFF formula F' obtained from F through the application of a CIFF proof rule ϕ on a node N, we have that

$$F' = F - \{N\} \cup \mathcal{N}$$

where $\mathcal{N}$ is the set of successor nodes of N with respect to ϕ. □

We are now ready to prove the soundness theorem 4.1.

Proof of Theorem 4.1. Let us consider a CIFF successful node N. By definition of CIFF extracted answer, the node N from which $\langle \Delta, C\rangle$ is extracted, is a conjunction of the form

$$\Delta \wedge \Gamma \wedge E \wedge DE \wedge Rest$$

where $C = \langle \Gamma, E, DE\rangle$ and $Rest$ is a conjunction of CIFF conjuncts.
Propositions 4.1 and 4.2 ensure the existence of a ground substitution $\overline{\sigma}$ such that:

$$\Delta\overline{\sigma} \models_{3(\Re)} \Delta \cup \Gamma \cup E \cup DE.$$

Let X the set of variables occurring in Q and let θ the restriction of $\overline{\sigma}$ over the variables in X. Let γ be a ground substitution for all the variables occurring in $Q\theta$. Let $\sigma = \theta\gamma$. It is straightforward that
$\Delta\theta\gamma \models_{3(\Re)} \Delta \cup \Gamma \cup E \cup DE$
as the substitution γ does not involve any variable in $\Delta \cup \Gamma \cup E \cup DE$.

To prove that $\langle \Delta, \sigma, \Gamma\rangle$ is an abductive answer with constraint, we need that:

1. there exists a ground substitution σ' for the variables occurring in $\Gamma\sigma$ such that $\sigma' \models_{\Re} \Gamma\sigma$ and
2. for each ground substitution σ' for the variables occurring in $\Gamma\sigma$ such that $\sigma' \models_{\Re} \Gamma\sigma$, there exists a ground substitution σ'' for the variables occurring in $Q \cup \Delta \cup \Gamma$, with $\sigma\sigma' \subseteq \sigma''$, such that:
   - $P \cup \Delta\sigma'' \models_{LP(\Re)} Q\sigma''$ and
   - $P \cup \Delta\sigma'' \models_{LP(\Re)} IC$.

Again, Propositions 4.1 and 4.2 ensure that

- there exists a ground substitution σ' for the variables occurring in $\Gamma\sigma$ such that $\sigma' \models_{\Re} \Gamma\sigma$ and such that, for each ground substitution σ' and
- for each ground substitution σ' for the variables occurring in $\Gamma\sigma$ such that $\sigma' \models_{\Re} \Gamma\sigma$, there exists a ground substitution σ'' for the variables occurring in $Q \cup \Delta \cup \Gamma$, with $\sigma\sigma' \subseteq \sigma''$, such that:

$$\Delta\sigma'' \models_{3(\Re)} \Delta \cup \Gamma \cup E \cup DE \qquad (+)$$

If we prove that $\Delta\sigma'' \models_{3(\Re)} Rest$, we have that

$$P \cup \Delta\sigma'' \models_{3(\Re)} N. \qquad (*)$$

From this, by induction and by Proposition 4.3, we will obtain

- $P \cup \Delta\sigma'' \models_{3(\Re)} Q\sigma''$, and
- $P \cup \Delta\sigma'' \models_{3(\Re)} IC$,

thus proving that $\langle \Delta, C\rangle$ is an abductive answer with constraints to Q with respect to $\langle P, A, IC\rangle_{\Re}$.

We now prove $(*)$. It is obvious that:

$$P \cup \Delta\sigma'' \models_{3(\Re)} \Delta \cup \Gamma \cup E \cup DE$$

by $(+)$ above. We need to show that:

$$P \cup \Delta\sigma'' \models_{3(\Re)} Rest.$$

Let us consider the structure of $Rest$. Due to the exhaustive application of CIFF proof rules, a CIFF conjunct in $Rest$ cannot be any of the following:

- a disjunction (due to the exhaustive application of **Splitting**);
- a defined atom (due to the exhaustive application of **Unfolding atoms**);
- either $\top$ or $\bot$ (due to the exhaustive application of **Logical simplification (#1 - #4)** and the fact that N is not a failure node, respectively);
- an implication whose body contains a defined atom (due to the exhaustive application of **Unfolding in implications**);
- an implication with a negative literal in the body (due to the exhaustive application of **Negation rewriting**);
- an implication with $\top$ or $\bot$ in the body (due to the exhaustive application of **Logical simplification (#1 - #4)**);
- an implication with only equalities or constraint atoms in the body (due to the exhaustive application of **Case analysis for equalities**, **Case analysis for constraints**, **Substitution in implications** and **Dynamic Allowedness**).

Thus, each CIFF conjunct in *Rest* is an implication whose body contains at least an abducible atom. We denote as $A_a \subseteq \Delta$ the set of abducible atoms in Δ whose predicate is a. Consider an implication $I \in Rest$ of the form $a(\vec{t}) \wedge B \rightarrow H$ where a is an abducible predicate and $\vec{t}$ may contain universally quantified variables.
Either $A_a = \oslash$ or not. If $A_a = \oslash$ then it trivially holds that $P \cup \Delta\sigma'' \models_{3(\Re)} I$ because the body of I falsified.
The case $A_a \neq \oslash$ is more interesting. Assume $A_a = a(\vec{s_1}), \ldots, a(\vec{s_k})$. Due to the fact that a has no definition in P, $a(\vec{s_1})\sigma'', \ldots, a(\vec{s_k})\sigma''$ represent all and only the instances of $a(\vec{t})$ which are entailed by $P \cup \Delta\sigma''$ with respect to the three-valued completion semantics.
Hence, if $\vec{t} = \vec{s}\sigma''$, where $\vec{s}$ is such that $a(\vec{s})\sigma'' \notin A_a$, it trivially holds that $P \cup \Delta\sigma'' \models_{3(\Re)} I$, because the body of I falsified.
Consider now the case $\vec{t} = \vec{s}\sigma''$, where $\vec{s}$ is such that $a(\vec{s})\sigma'' \in A_a$. Because N is a CIFF successful node, **Propagation** has been exhaustively applied in the CIFF branch $\mathcal{B}$ whose leaf node is N. This means that for each $a(\vec{s_i})\sigma'' \in A_a$, an implication I' of the form

$$\vec{t} = \vec{s_i}\sigma'' \wedge B \rightarrow H$$

occurs in at least a node $N_i \in \mathcal{B}$ (otherwise **Propagation** is still applicable and N is not a successful node). Then, if B of the body does not contain other abducibles, the implication I' is not in *Rest* and has been reduced to a conjunction in N.
Otherwise, if B contains another abducible atom, the process is applied again on it. Because a successful branch is finite, the proof is obtained by induction on the number of abducible atoms in B.

Hence, it holds that:

$$P \cup \Delta\sigma'' \models_{3(\Re)} Rest$$

and

$$P \cup \Delta\sigma'' \models_{3(\Re)} N$$

Let us consider the CIFF branch $\mathcal{B}$ whose leaf node is N, i.e. the branch $\mathcal{B} = N_1 = Q \wedge IC, N_2, \ldots, N_l = N$ with $l \geq 1$. If we prove that for each pair of nodes N_i and N_{i+1} belonging to $\mathcal{B}$ it holds that if

$$P \cup \Delta\sigma'' \models_{3(\Re)} N_{i+1}$$

then

$$P \cup \Delta\sigma'' \models_{3(\Re)} N_i$$

we have, by induction, that

$$P \cup \Delta\sigma'' \models_{3(\Re)} Q\sigma'' \wedge IC$$

Suppose $P \cup \Delta\sigma'' \models_{3(\Re)} N_{i+1}$, for some i. Due to the definition of CIFF branch, each node $N_{i+1} \in \mathcal{B}$ is one of the successor nodes of N_i. If N_{i+1} is obtained by N_i by applying a CIFF proof rule distinct from the **Splitting** rule, if follows immediately that

$$P \cup \Delta\sigma'' \models_{3(\Re)} N_i$$

given that N_{i+1} is the only successor node of N_i and thus, from Proposition 4.3, we have that $N_i \equiv N_{i+1}$. If the **Splitting** rule has been applied, however, then the node N_i is of the form

$$RestNode \wedge (D_1 \vee \ldots \vee D_n)$$

and N_{i+1} is of the form

$$(RestNode \wedge D_i) \qquad \text{for some } i \in [1, n].$$

It is obvious that the latter formula entails the former.

Summarizing, we have that

$$P \cup \Delta\sigma'' \models_{3(\Re)} Q\sigma'' \wedge IC$$

which implies that
$P \cup \Delta\sigma'' \models_{3(\Re)} Q\sigma''$, and
$P \cup \Delta\sigma'' \models_{3(\Re)} IC$. □

The CIFF soundness in Theorem 4.1 concerns only those branches of a CIFF successful derivation whose leaf node is a CIFF successful node. It implies that abductive answers with constraints can be obtained also by those derivations which contain failure and undefined branches but which have at least a successful branch. We also prove the following notion of soundness regarding failure CIFF derivations.

Theorem 4.2 (Soundness of failure)**.** *Let $\langle P,\ A,\ IC\rangle_\Re$ be an abductive logic program with constraints such that the corresponding CIFF framework is $\langle Th, A, IC\rangle_\Re$. Let $\mathcal{D}$ be a failure CIFF derivation with respect to $\langle Th, A, IC\rangle_\Re$ and a query Q. Then:*

$$P \cup IC \models_{3(\Re)} \neg Q$$

□

Proof of Theorem 4.2. From the definition of failure CIFF derivation, $\mathcal{D}$ is a derivation starting with $Q \cup IC$ and such that all its leaf nodes are CIFF failure nodes which are equivalent to $\bot$. Hence, due to Corollary 4.1 and the transitivity of the equivalence, it follows immediately that:

$$P \cup IC \models_{3(\Re)} (Q \wedge IC) \leftrightarrow \bot$$

Because IC occurs in both the left and the right hand side of the statement, we have that

$$P \cup IC \models_{3(\Re)} Q \leftrightarrow \bot$$

and thus

$$P \cup IC \models_{3(\Re)} \neg Q.$$

□

Note that there is a class of CIFF derivations for which a soundness result cannot be stated, i.e. all the derivations containing only undefined and failure branches. The meaning of such CIFF derivations is that for each branch, no CIFF answer can be extracted, but there are some branches (undefined branches) for which neither failure nor success is ensured. The presence of an undefined branch is due to the application of the **Dynamic Allowedness** rule and, as we have seen at the end of Section 4.1.1, this could lead to infinite sets of abducibles in the answers.

The IFF proof procedure was proven complete with respect to the three-valued completion semantics and the class of allowed IFF frameworks [81], with the only requirement of a *fair* selection function, i.e. a selection function that does not always select a node in a looping branch. For example, suppose we have an abducible predicate a and two predicates p and q defined as

$$q \leftrightarrow p \vee a$$
$$p \leftrightarrow p.$$

Consider the query q. After the unfolding of q, the IFF proof procedure would return the abductive answer a if the second disjunct is eventually selected, but it loops forever in the other case. A *fair* selection function ensures that the second disjunct is eventually selected during a derivation.
In the case of CIFF, tackling the allowedness problem dynamically, we could obtain undefined derivations, even with a *fair* selection function. Hence a completeness theorem for CIFF allowed frameworks cannot be formulated. But we can make some remarks regarding completeness.

- To make CIFF "more" complete, the CIFF selection function should select the **Dynamic Allowedness** rule with the lowest priority: in this way we avoid labeling a node as $undefined$ if the node can yet become a failure node.

- For the class of IFF allowed frameworks, the results for the IFF proof procedure are directly inherited by CIFF because there are no constraint atoms and the CIFF proof rules which do not manage constraint atoms are the same as for IFF (obviously assuming a *fair* selection function).

- We argue that the completeness theorem for the IFF procedure can be extended to CIFF frameworks $\langle Th, A, IC \rangle_{\Re}$ and queries Q which are IFF allowed and such that each variable X appearing in a constraint atom in:

  - a definition in Th,
  - an implication in IC, or
  - in Q,

  is such that X occurs in a positive non-equality atom in the same definition, in the same implication or in the query respectively. This is because in this way, the **Dynamic Allowedness** rule never applies as variables are always "bound" by positive non-equalities atoms.

4.3 The CIFF$^{\neg}$ proof procedure

As remarked in Chapter 3, integrity constraints play an important role in the abductive process. The CIFF proof procedure allows for combining both backward reasoning by means of iff-definition and forward reasoning via integrity constraints in implicative form which can be used for modeling reactive and condition-action rules in a natural way. This feature makes CIFF a powerful mechanism for building computational models in a wide range of applications, e.g., agents [124, 105, 158] and database updates [157]. However, in some cases the classical treatment of negation in integrity constraints in CIFF can lead to non-intuitive answers being computed.

Example 4.9. *Consider the following CIFF framework $\langle Th, A, IC\rangle_{\Re}$ which could be thought of as a (very simple) control program for a cleaning agent in a dangerous environment. The idea is that if it is a cleaning day and the alarm has not been activated, the agent must dust. The alarm sounds if the temperature exceeds 40 degrees and in that case the agent must evacuate the building.*

$$
\begin{array}{rl}
Th: & \oslash \\
A: & \{cleaning_day, sound_alarm, temperature, dust, evacuate\} \\
IC: & cleaning_day \wedge \neg sound_alarm \rightarrow dust \\
& temperature(X) \wedge X > 40 \rightarrow sound_alarm \\
& sound_alarm \rightarrow evacuate
\end{array}
$$

Given the observation (query) $Q = cleaning_day$, we have the following CIFF derivation:

$$
\begin{array}{lll}
F_1 = \{N_1\} = & \{\{cleaning_day, & \\
& (cleaning_day \wedge (sound_alarm \rightarrow \bot)) \rightarrow dust, & \\
& (temperature(X) \wedge X > 40) \rightarrow sound_alarm, & \\
& sound_alarm \rightarrow evacuate\}\} & \textbf{[Init]} \\
F_2 = \{N_2\} = & \{\{(sound_alarm \rightarrow \bot) \rightarrow dust\} \cup N_1\} & \textbf{[R3]} \\
F_3 = \{N_3\} = & \{\{\top \rightarrow (dust \vee sound_alarm)\} \cup N_1\} & \textbf{[R13]} \\
F_4 = \{N_4\} = & \{\{dust \vee sound_alarm\} \cup N_1\} & \textbf{[R17]} \\
F_5 = \{N_5, N_6\} = & \{\{dust\} \cup N_1, \{sound_alarm \cup N_1\}\} & \textbf{[R4]} \\
F_6 = \{N_5, N_7\} = & \{N_5, N_6 \cup \{\top \rightarrow evacuate\}\} & \textbf{[R3]} \\
F_7 = \{N_5, N_8\} = & \{N_5, N_6 \cup \{evacuate\}\} & \textbf{[R17]}
\end{array}
$$

From the nodes N_5 and N_8 we can extract two distinct answers:

$$Ans_1 = \langle\{cleaning_day, dust\}, \langle\oslash, \oslash, \oslash\rangle\rangle$$

$$Ans_2 = \langle\{cleaning_day, sound_alarm, evacuate\}, \langle\oslash, \oslash, \oslash\rangle\rangle$$

It is clear that Ans_2 is a counter-intuitive answer. The initial cleaning_day observation propagates *the first integrity constraint and then, after a* ***Negation rewriting*** *rule, there are two ways to succeed: dust or sound_alarm. In the second case the agent sounds the alarm to succeed even if the temperature does not exceed 40 degrees producing the CIFF extracted answer Ans_2. Note that Ans_2 is counter-intuitive, but it is correct with respect to the CIFF semantics.* □

Hence, we have investigated a different way of treating negation within (the implications derived from) integrity constraints. Following the approach proposed in [157], we have defined a modified CIFF proof procedure, which we call CIFF$^{\neg}$, which makes use of *negation as failure* (NAF) in integrity constraints.
The basic idea behind the CIFF$^{\neg}$ proof procedure is that negative literals in the body of integrity constraints should not be moved into the head, by **Negation rewriting**, but rather they should be "checked", to see if they hold taking into account the current abduced atoms. The intention is to avoid performing new abductions for proving that negative literals in the body of integrity constraints hold.

We now describe the CIFF$^{\neg}$ proof procedure in detail. Almost all definitions used for the original CIFF proof procedure still remain valid for the CIFF$^{\neg}$ proof procedure. In the sequel when we refer to CIFF$^{\neg}$ node, CIFF$^{\neg}$ derivation, etc. we will assume that they have the same definitions as the corresponding definitions for CIFF, i.e. CIFF node, CIFF derivation, etc. apart from those definitions which are explicitly changed. Being the NAF treatment of negation in integrity constraints, the main feature of CIFF$^{\neg}$, we use now the symbol *not* in a CIFF$^{\neg}$ framework instead of $\neg$.

CIFF$^{\neg}$ frameworks, nodes, conjuncts and queries are the same as CIFF frameworks, nodes, conjuncts and queries with the following differences:

- an implication in a CIFF$^\neg$ node can be either *marked* (by using the symbol $*$) or *unmarked* (an ordinary CIFF implication),
- atoms in a CIFF$^\neg$ node can be arguments of a special (meta-)predicate *prv*[5].

If an implication is *marked*, the negative literals in its body have to be treated with NAF, while if the implication is *unmarked* they have to be treated as in CIFF. Every CIFF$^\neg$ proof rule can be applied to marked implications, except for the **Negation rewriting** rule. In this way we prevent the moving of negative literals into the head of marked implications.

To handle marked implications, the CIFF$^\neg$ proof procedure has the **NAF rewriting** rule which rewrites negative literals in the body of a marked implication through the use of the *prv* special (meta-)predicate. A $prv(p(\vec{t}))$ atom stands for "$p(\vec{t})$ is provably true" and it will be "checked" by new CIFF$^\neg$ rules which avoid new abductions during the "checking" process. Note that a *prv* atom cannot be handled by any CIFF proof rule because neither it has a definition in Th, nor it is an abducible predicate.
The **NAF rewriting** rule is the following.

R19 - NAF rewriting	
Given:	$*[((p_1(\vec{t_1}) \rightarrow \bot) \wedge \ldots \wedge (p_n(\vec{t_n}) \rightarrow \bot)] \rightarrow B)$
Conditions:	none
Action:	**replace** $\{\ \top \rightarrow [prv(p_1(\vec{t_1})) \vee \ldots \vee prv(p_n(\vec{t_n})) \vee$
	$[*(p_1(\vec{t_1}) \rightarrow \bot) \wedge \ldots \wedge *(p_n(\vec{t_n}) \rightarrow \bot) \wedge B]]\ \}$

Accordingly, we change the conditions in the **Negation Rewriting** rule.

R13' - Negation rewriting	
Given:	$((A \rightarrow \bot) \wedge B) \rightarrow H$
Conditions:	$((A \rightarrow \bot) \wedge B) \rightarrow H$ is unmarked
Action:	**replace** $\{\ B \rightarrow (A \vee H)\ \}$

Note that all the other proof rules involving implications can be applied to marked or unmarked implications as well.

Example 4.10. *Let us consider again Example 4.9. Suppose the integrity constraint*

$$*(cleaning_day \wedge (sound_alarm \rightarrow \bot)\ \rightarrow\ dust)$$

*is marked in the initial formula F_1. In this case we cannot apply the **Negation rewriting** rule to the only node of N_2, in order to obtain the formula F_3, but we must apply the **NAF rewriting** rule to N_2 obtaining:*

$$F_3 = \{N_3\} = \\ \{\{\top\ \rightarrow\ [prv(sound_alarm) \vee [*(sound_alarm\ \rightarrow\ \bot)\ \wedge\ dust]]\} \cup N_1\}\quad \textit{[R13]}$$

Intuitively, this means that either the alarm has been sounded or the agent must dust! □

The problem now, is how to manage the *prv* atoms which occur in a node N to which no other rules can be applied. The idea is that these atoms have to be proved in a way that (1) the abduced atoms are taken into account and eventually used in the proof, and (2) no new abductions are performed to support the proof. In order to obtain this behavior, we introduce the following CIFF$^\neg$ proof rules.

[5] The special (meta-)predicate *prv* cannot appear in a CIFF$^\neg$ framework. It may only appear in a CIFF$^\neg$ node due to the application of the CIFF$^\neg$ rules shown later.

R20 - NAF Switch	
Given:	
Conditions:	N is unmarked; $\overline{\Delta}$ is the set of all abducible atoms occurring as CIFF$^\neg$ conjuncts in N
Action:	**mark** $NAF(\overline{\Delta})$

The **NAF Switch** rule simply marks a node N as a NAF node with the set of abduced atoms $\overline{\Delta}$. This allows to record the set of abducibles which can be used to prove the *prv* atoms. This rule will only be used when no other CIFF$^\neg$ proof rules can be applied to a node, as we will see later. We will refer to $\overline{\Delta}$ such that $NAF(\overline{\Delta})$ marks a node as the *marking set* of that node. Note that the marking set of a node may be empty.

The next rules are all applicable only to nodes marked as NAF nodes.

R21 - Provable rewriting	
Given:	$prv(p_1(\vec{t_1})) \wedge \ldots \wedge prv(p_k(\vec{t_k}))$
Conditions:	$prv(p_1(\vec{t_1})) \wedge \ldots \wedge prv(p_k(\vec{t_k}))$ are all the *prv* atoms in N; N is marked with $NAF(\overline{\Delta})$
Action:	**replace** $\{ \ p_1(\vec{t_1}) \wedge \ldots \wedge p_k(\vec{t_k}) \ \}$

The **Provable rewriting** rule simply drops the special (meta-)predicates *prv*, in order to make their arguments manageable by other CIFF$^\neg$ proof rules.

R22 - NAF Factoring #1	
Given:	$p(\vec{s})$
Conditions:	p abducible; N is marked with $NAF(\overline{\Delta})$; there is no atom in $\overline{\Delta}$ whose predicate is p
Action:	**replace** $\{ \ \bot \ \}$

R23 - NAF Factoring #2	
Given:	$p(\vec{s})$
Conditions:	p abducible; N is marked with $NAF(\overline{\Delta})$; $p(\vec{s}) \notin \overline{\Delta}$; $p(\vec{t_1}), \ldots, p(\vec{t_k})$ are all the atoms in $\overline{\Delta}$ whose predicate is p
Action:	**replace_all** $\{ \ \vec{s} = \vec{t_1} \vee \ldots \vee \vec{s} = \vec{t_k} \ \}$

The **NAF Factoring** rules are the rules which allow both to take into account "old" abducibles (those in $\overline{\Delta}$) and to forbid new abductions. This is done, in **NAF Factoring #1**, by failing when a new abducible atom $p(\vec{s})$ occurs in a NAF marked node such that there is no atom in $\overline{\Delta}$ whose predicate is p. This is because the only way to prove $p(\vec{s})$ would be to perform a new abduction. **NAF Factoring #2**, instead, handles an abducible atom $p(\vec{s})$ such that there are k atoms in $\overline{\Delta}$ whose predicate is p. The idea is to get k successor nodes each one with an equality atom of the form $\vec{s} = \vec{t_i}$, for each $p(\vec{t_i}) \in \overline{\Delta}$, replacing the abducible atom $p(\vec{s})$. This ensures that all the ways in which $p(\vec{s})$ can be unified with an atom in $\overline{\Delta}$ are taken into account and, moreover, no new abductions are done. This rule should not be applied to an abducible $p(\vec{s}) \in \overline{\Delta}$: otherwise that abducible would disappear from the node.

We have now presented all new CIFF$^\neg$ rules, summarized in Table 4.2.
Now we focus on the building of a CIFF$^\neg$ derivation and the answer extraction process, pointing out the differences with the original CIFF.

The first difference concerns a *CIFF$^\neg$ selection function* $\mathcal{S}$. What we require of $\mathcal{S}$ is that:

Table 4.2: CIFF$^\neg$ proof rules

R1	Unfolding atoms
R2	Unfolding in implications
R3	Propagation
R4	Splitting
R5	Factoring
R6	Case analysis for constraints
R7	Constraint solving
R8	Equality rewriting in atoms
R9	Equality rewriting in implications
R10	Substitution in atoms
R11	Substitution in implications
R12	Case analysis for equalities
R13	Negation rewriting
R14	Logical Simplification #1
R15	Logical Simplification #2
R16	Logical Simplification #3
R17	Logical Simplification #4
R18	Dynamic Allowedness
R19	NAF rewriting
R20	NAF switch
R21	Provable rewriting
R22	NAF factoring #1
R23	NAF factoring #2

- it must select the **NAF switch** rule with respect to a node N only if no other rule can be selected with respect to N (apart from **Dynamic Allowedness** for improving completeness, as remarked in section 4.2): in this way we ensure that all abductions are done prior to proving the *prv* atoms; and
- it must not select **Propagation** rule and **Factoring** rule for a node N if N is marked with $NAF(\overline{\Delta})$. In this way the abducibles introduced in a marked node N will be handled exclusively by the **NAF factoring** rules.

Definition 4.21 (CIFF$^\neg$ selection function)**.** *Let F be a CIFF$^\neg$ formula. We define a* CIFF$^\neg$ selection function *$\mathcal{S}$ as a function such that:*

$$\mathcal{S}(F) = \langle N, \phi, \chi \rangle$$

where N is a CIFF$^\neg$ node in F, ϕ is a CIFF$^\neg$ proof rule and χ is a set of CIFF$^\neg$ conjuncts in N such that χ is a rule input *for ϕ. Moreover $\mathcal{S}(F)$ must satisfy the following conditions:*

- *if $\mathcal{S}(F) = \langle N,$ **NAF switch**$, \oslash \rangle$ then there does not exist ϕ such that*
  - *$\phi \neq$ **NAF switch**,*
  - *$\phi \neq$ **Dynamic Allowedness**, and*
  - *there exists a* rule input *for ϕ in N*
- *if either $\mathcal{S}(F) = \langle N,$ **Propagation**$, \chi \rangle$ or $\mathcal{S}(F) = \langle N,$ **Factoring**$, \chi \rangle$ then N is* unmarked*.*

□

The next definition is the definition of a *CIFF¬ pre-derivation*, taking into account *marked* integrity constraints.

Definition 4.22 (CIFF¬ Pre-derivation and initial formula)**.** *Let $\langle Th,\ A,\ IC\rangle_{\Re}$ be a CIFF¬ framework, let Q be a query and let $\mathcal{S}$ be a* CIFF¬ selection function*. Let IC^* be the set of marked integrity constraints obtained by marking with $*$ all the integrity constraints in IC. A* CIFF¬ pre-derivation *for Q with respect to $\langle Th,\ A,\ IC\rangle_{\Re}$ and $\mathcal{S}$, is a (finite or infinite) sequence of CIFF¬ formulae $F_1, F_2, \ldots, F_i, F_{i+1} \ldots$ such that each F_{i+1} is obtained from F_i through $\mathcal{S}$ as follows:*

- *$F_1 = \{N_1\} = \{Q \cup IC^*\}$, where Q and IC^* are treated as sets of CIFF¬ conjuncts, (we will refer to F_1 as the* initial formula *of a CIFF¬ pre-derivation)*
- *$\mathcal{S}(F_i) = \langle N_i, \phi_i, \chi_i\rangle$ such that N_i is neither an undefined CIFF¬ node nor a failure CIFF¬ node and*
- $F_i \xrightarrow[\phi_i]{N_i, \chi_i} F_{i+1}$

□

The definitions of a *CIFF¬ branch* and a *CIFF¬ derivation* are unchanged with respect to the definition of a *CIFF branch* and a *CIFF derivation*, respectively (obviously using the definitions of CIFF¬ selection function and CIFF¬ pre-derivation).

The notion of extracted answer remains unchanged, apart from removing the marking of all the extracted CIFF¬ inequalities.

Example 4.11. *Let us consider Example 4.9. The following is a CIFF¬ derivation to the query $Q = cleaning_day$.*

$$
\begin{array}{lll}
F_1 = \{N_1\} = & \{\{cleaning_day, & \\
& (cleaning_day \wedge (sound_alarm\ \rightarrow\ \bot))\ \rightarrow\ dust, & \\
& (temperature(X) \wedge X > 40)\ \rightarrow\ sound_alarm, & \\
& sound_alarm\ \rightarrow\ evacuate\}\} & \textbf{[Init]} \\
F_2 = \{N_2\} = & \{\{(sound_alarm\ \rightarrow\ \bot)\ \rightarrow\ dust\} \cup N_1\} & \textbf{[R3]} \\
F_3 = \{N_3\} = & \{\{\top \rightarrow [prv(sound_alarm) \vee & \\
& [*(sound_alarm\ \rightarrow\ \bot)\ \wedge\ dust]]\} \cup N_1\} & \textbf{[R19]} \\
F_4 = \{N_4\} = & \{\{[prv(sound_alarm) \vee & \\
& [*(sound_alarm\ \rightarrow\ \bot)\ \wedge\ dust]]\} \cup N_1\} & \textbf{[R17]} \\
F_5 = \{N_6, N_5\} = & \{\{N_1 \cup \{prv(sound_alarm)\}\}, & \\
& \{N_1 \cup \{*(sound_alarm\ \rightarrow\ \bot)\ \wedge\ dust\}\}\} & \textbf{[R4]} \\
F_6 = \{N_7, N_5\} = & \{NAF(cleaning_day) : N_6, N_5\} & \textbf{[R20]} \\
F_7 = \{N_8, N_5\} = & \{NAF(cleaning_day) : (N_1 \cup \{sound_alarm\}), N_5\} & \textbf{[R21]} \\
F_8 = \{N_9, N_5\} = & \{NAF(cleaning_day) : (N_1 \cup \{\bot\}), N_5\} & \textbf{[R22]} \\
F_9 = \{N_{10}, N_5\} = & \{N_9, NAF(cleaning_day, dust) : N_5\} & \textbf{[R20]}
\end{array}
$$

From the above CIFF¬ derivation we can only extract the following answer from node N_{10}:

$$Ans_1 = \langle\{cleaning_day, dust\}, \langle\emptyset, \emptyset, \emptyset\rangle\rangle$$

*Note that the unintuitive answer Ans_2 of Example 4.9 cannot be extracted from the CIFF¬ derivation because in node N_8 the abducible sound_alarm is not in the marking set, thus **NAF factoring #1** rule leads to a failure node. Note also that due to the definition of a CIFF¬ selection function, the abducible sound_alarm in node N_8 cannot be propagated to the (marked) implication $sound_alarm \rightarrow evacuate$ because node N_8 is marked with $NAF(cleaning_day)$.*

□

The following proposition shows the intuition that a successful CIFF$^\neg$ node does not contain abducible atoms, occurring as CIFF$^\neg$ conjuncts, which do not belong to the marking set of that node. This means that no abduction is performed for proving the *prv* atoms.

Proposition 4.4. *Let $\langle Th, A, IC\rangle_\Re$ be a CIFF$^\neg$ framework and let $\mathcal{D}$ be a successful CIFF$^\neg$ derivation for a query Q. LetN be a successful CIFF$^\neg$ node in $\mathcal{D}$ and let $\langle \Delta, C\rangle$ be the CIFF$^\neg$ extracted answer from N.*
Then N is marked with $NAF(\Delta)$ and each abducible atom $a(\vec{t})$ occurring as a CIFF conjunct in N belongs to Δ. □

Proof of Proposition 4.4. Consider the successful branch B= $N_1, \ldots, N_s = N$ in $\mathcal{D}$. The fact that N is marked with $NAF(\Delta')$ for a certain Δ' is straightforward because N is a successful node, hence the **NAF switch** rule must have been applied at a certain node N_k in $\mathcal{B}$ with $1 \leq k < s$. Moreover there is no CIFF$^\neg$ rule which unmarks a node. The fact that each abducible in N belongs to Δ' is again straightforward because for each abducible $a(\vec{t})$ occurring in a node N_i with $i > k$ in $\mathcal{B}$, there exists a node N_j with $j > i$ in $\mathcal{B}$ which is obtained by applying the **NAF factoring #2** rule to the node N_{j-1}. As that rule removes from the node the abducible $a(\vec{t})$, then $\Delta' - \Delta = \oslash$ and the statement is proved. Note that the rule **NAF factoring #1** cannot be applied in $\mathcal{B}$ because N is a successful node. □

4.4 Soundness of the CIFF$^\neg$ proof procedure

Intuitively, as seen in the previous examples, the CIFF$^\neg$ proof procedure is sound with respect to the three-valued completion semantics because given a CIFF$^\neg$ framework and a query, the set of CIFF$^\neg$ extracted answers is a subset of the set of answers found by the original CIFF proof procedure. Hence, the soundness of CIFF ensures the soundness of CIFF$^\neg$.

However, to prove formally this soundness result is not a straightforward task because the added rules are not equivalence preserving with respect to the three-valued completion semantics. Intuitively, as the new rules invalidate some sound CIFF answers, it is obvious that they cannot be equivalence preserving.
The proof of soundness is done in two main steps:

- first, we define the CIFF$^\vee$ proof procedure by simply adding a new *equivalence preserving* rule to the set of CIFF proof rules; proving the equivalence preservation of the new rule, the CIFF$^\vee$ proof procedure directly inherits the soundness results of CIFF;
- then we prove that any CIFF$^\neg$ extracted answer can be extracted from a CIFF$^\vee$ derivation.

The CIFF$^\vee$ proof procedure

The CIFF$^\vee$ proof procedure is defined by adding the following rule to the set of CIFF proof rules in Table 4.1.

R24 - Extending disjunction	
Given:	$p_1(\vec{t_1}) \vee \cdots \vee p_n(\vec{t_n}) \vee D$
Conditions:	each $p_i(\vec{t_i})$ is an atom, $n \geq 1$
Action:	**replace** { $p_1(\vec{t_1}) \vee \cdots \vee p_n(\vec{t_n}) \vee D \vee$ $[(p_1(\vec{t_1}) \rightarrow \bot) \wedge \cdots \wedge (p_n(\vec{t_n}) \rightarrow \bot) \wedge D]$ }

All the CIFF definitions seen in section 4.1 remain valid for the CIFF$^\vee$ proof procedure (obviously considering the new rule in the set of rules when needed). The following example shows the use of the new rule.

Example 4.12. *Consider the following CIFF framework* $\langle Th, A, IC\rangle_{\Re}$:

$$\begin{array}{rl} Th: & q(X) \leftrightarrow X = a \\ & p(X) \leftrightarrow r(X) \wedge \neg q(X) \\ A: & \{s, r\} \\ IC: & p(X) \rightarrow s \end{array}$$

The following is a CIFF$^{\vee}$ *derivation* $\mathcal{D}$ *for the query* $r(Y)$.

$$\begin{array}{rll} F_1 = \{N_1\} = & \{r(Y), [p(X) \rightarrow s] & \textit{[}\textbf{\textit{Init}}\textit{]} \\ F_2 = \{N_2\} = & \{r(Y), [r(X) \wedge (q(X) \rightarrow \bot) \rightarrow s] & \textit{[}\textbf{\textit{R2}}\textit{]} \\ F_3 = \{N_3\} = & \{r(Y), [r(X) \rightarrow (q(X) \vee s)] & \textit{[}\textbf{\textit{R13}}\textit{]} \\ F_4 = \{N_4\} = & \{N_3 \cup \{X = Y \rightarrow (q(X) \vee s)\} & \textit{[}\textbf{\textit{R3}}\textit{]} \\ F_5 = \{N_5\} = & \{N_3 \cup \{\top \rightarrow (q(Y) \vee s)\} & \textit{[}\textbf{\textit{R11}}\textit{]} \\ F_6 = \{N_6\} = & \{N_3 \cup \{q(Y) \vee s\} & \textit{[}\textbf{\textit{R17}}\textit{]} \\ F_7 = \{N_7\} = & \{N_3 \cup \{q(Y) \vee s \vee (q(Y) \rightarrow \bot \wedge s)\} & \textit{[}\textbf{\textit{R24}}\textit{]} \\ & \vdots & \end{array}$$

□

The only CIFF$^{\vee}$ proof rule whose equivalence preservation needs to be proved is the **Extending disjunction** rule, as all other CIFF$^{\vee}$ proof rules are also CIFF rules.

Proposition 4.5 (Equivalence preservation of the **Extending disjunction** rule)**.** *Given an abductive logic program with constraints* $\langle P, A, IC\rangle_{\Re}$, *a CIFF*$^{\vee}$ *node* N *and a set of CIFF*$^{\vee}$ *successor nodes* $\mathcal{N}$ *obtained by applying the* ***Extending disjunction*** *rule to* N, *it holds that:*

$$P \models_{3(\Re)} N \quad \textit{iff} \quad P \models_{3(\Re)} \mathcal{N}^{\vee}$$

where $\mathcal{N}^{\vee}$ *is the disjunction of the nodes in* $\mathcal{N}$. □

Proof of Proposition 4.5. We show generally that the equivalence

$$A \vee B \equiv A \vee B \vee (\neg A \wedge B)$$

holds with respect to the three-valued completion semantics. The following truth table demonstrates this:

A	B	$\neg A$	$A \vee B$	$\neg A \wedge B$	$A \vee B \vee (\neg A \wedge B)$
0	0	1	0	0	0
0	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
0	1	1	1	1	1
$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{2}$
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\frac{1}{2}$	1	$\frac{1}{2}$	1	$\frac{1}{2}$	1
1	0	0	1	0	1
1	$\frac{1}{2}$	0	1	0	1
1	1	0	1	0	1

Recall that in the three-valued completion semantics, each sentence can have three truth values: 0 representing $false$, $\frac{1}{2}$ representing $undefined$ and 1 representing $true$. We denote by $\mid F \mid$ the truth value of a sentence F. In the table above, we have used the information that given two sentences F and G, we have that:

$$\begin{array}{lcl} \mid \neg F \mid & = & 1- \mid F \mid \\ \mid F \wedge G \mid & = & min(\mid F \mid, \mid G \mid) \\ \mid F \vee G \mid & = & max(\mid F \mid, \mid G \mid) \end{array}$$

□ □

Due to the above proposition, the soundness theorem 4.1 also applies to the CIFF$^\vee$ proof procedure.

Now we turn to the second part of the proof of soundness of the CIFF$^\neg$ proof procedure, i.e. given a CIFF$^\neg$ extracted answer, we show how to obtain the same CIFF$^\vee$ extracted answer. The idea is to map each rule used to build the CIFF$^\neg$ branch from which the answer has been extracted, to a set of CIFF$^\vee$ rules in the building of a CIFF$^\vee$ derivation with respect to the same framework and the same query. We introduce the following definitions.

Definition 4.23. *Let $\mathcal{D}$ be a CIFF (respectively a CIFF$^\neg$ or a CIFF$^\vee$) derivation.*
We say that $N_k = N, \ldots, N_{k+m} = N'$ is a CIFF (CIFF$^\neg$/CIFF$^\vee$) derivation sequence *with respect to $\mathcal{D}$, and we denote it by $Seq(D, N, N')$ if and only if there exists a branch $\mathcal{B}$ in $\mathcal{D}$ such that $\mathcal{B}$ is of the form*

$$\ldots, N_k, \ldots, N_{k+m}, \ldots$$

In the sequel we will write simply $Seq(N, N')$ if $\mathcal{D}$ is clear from the context. □

Definition 4.24. *Let I^* be a set of marked implications. We say that I is the* unmarked set of *I^* if and only if*

- *I contains only unmarked implications, and*
- *an unmarked implication belongs to I if and only if an identical implication marked with $*$ belongs to I^*.*

□

Definition 4.25. *Let N be a CIFF$^\neg$ node and let I^* be the set of all the marked implications in N. We say that the* unmarked node of N, *denoted by $\overline{N}$, is the node obtained from N by replacing I^* by its unmarked set I.* □

The following proposition, intuitively, shows that the CIFF$^\neg$ rules applied to nodes marked as $NAF(\Delta)$ nodes in a branch of a CIFF$^\neg$ derivation, can be mapped to standard CIFF proof rules.

Proposition 4.6. *Let $\langle Th, A, IC\rangle_\Re$ be a CIFF$^\neg$ framework and let Q be a query. Let $N = Ps \wedge I^* \wedge RestNode$ be a CIFF$^\neg$ node within a CIFF$^\neg$ derivation $\mathcal{D}$ for Q such that:*

- *$Ps = prv(p_1(\vec{t_1})) \wedge \ldots \wedge prv(p_n(\vec{t_n}))$ is the conjunction of all the prv atoms in N,*
- *I^* is the conjunction of all the marked implications in N,*
- *no rules apart from **NAF Switch** and **Dynamic Allowedness** can be applied to N.*

Let N_s be a successful CIFF$^\neg$ node in a branch $\mathcal{B} = N_1, \ldots, N_k, \ldots, N_s$ of $\mathcal{D}$ such that $N_k = N$. Then we have the following:

- *$N_{k+2} = p_1(\vec{t_1}) \wedge \ldots \wedge p_n(\vec{t_n}) \wedge I^* \wedge RestNode$, and*
- *there exists a CIFF derivation D' such that*
  - *for each CIFF$^\neg$ node N_j with $j \in [k+3, s-1]$, there exists a finite CIFF derivation sequence $Seq(D', \overline{N_j}, \overline{N_{j+1}})$*
  - *for each unmarked CIFF$^\neg$ node $\overline{N_j}$ with $j \in [k+3, s-1]$, there exists a finite CIFF derivation sequence $Seq(D', \overline{N_j}, \overline{N_{j+1}})$*

□

Proof of Proposition 4.6. By assumption, no CIFF¬ rule can be applied to N_k apart from **NAF Switch** and **Dynamic Allowedness**. But **Dynamic Allowedness** cannot be applied because, by assumption, $\mathcal{B}$ is a successful branch. Hence, **NAF Switch** is the only rule applicable to N_k obtaining N_{k+1} which simply is N_k marked with $NAF(\Delta_N)$, where Δ_N is the set of all the abducible atoms occurring as CIFF¬ conjuncts in N. Now, the only rule applicable to N_{k+1} is the **Provable rewriting** rule. Applying it to N_{k+1} we obtain, by definition:

$$N_{k+2} = p_1(\vec{t_1}) \wedge \ldots \wedge p_n(\vec{t_n}) \wedge I^* \wedge Rest$$

The second part of the proof is done by considering that each N_{j+1} is obtained from N_j by the application of a CIFF¬ proof rule ϕ. We consider exhaustively all the cases, doing a mapping to a finite sequence of CIFF proof rules. Let N_j be a CIFF¬ node in $\mathcal{D}$ with $j \in [k+3, s-1]$.

Case1 (NAF Switch) - this rule cannot be applied to N_j because it is marked with $NAF(\Delta_N)$.

Case2 (NAF Rewriting) - this rule cannot be applied to N_j. This is because **NAF Rewriting** is applied to marked implications. By assumption no rule can be applied to a marked implication in N_k and the only rule introducing marked implications is **NAF Rewriting** itself.

Case3 (Provable rewriting) - this rule cannot be applied to N_j. This is because:

1. all the *prv* atoms in N_k are dropped in N_{k+2}, and
2. the only rule introducing *prv* atoms in a node is the **NAF Rewriting** rule which cannot be applied to any node N_i in $\mathcal{D}$ with $i \geq k$ (Case2).

Case4 (NAF Factoring #1) - this rule cannot be applied to N_j. This is because, by assumption, $\mathcal{B}$ is a successful branch.

Case5 (NAF Factoring #2) - by definition, we have that N_j is of the form

$$p(\vec{s}) \wedge Rest_j$$

and N_{j+1} is of the form

$$\vec{s} = \vec{t_i} \wedge Rest_j$$

where $p(\vec{s}) \notin \Delta_N$ and $p(\vec{t_i}) \in \Delta_N$.

By definition of Δ_N, we have that $p(\vec{t_i})$ occurs as a CIFF¬ conjunct in *Rest* in N. But it also occurs in $Rest_j$ because the only rule which can eliminate an abducible atom in *Rest* is the **Factoring** rule (**R5**) and, by assumption, no rule can be applied to *Rest*. Then if we apply first the **Factoring** rule using $\{p(\vec{t_i}), p(\vec{s})\}$ as its rule input, and then the **Splitting** rule, we obtain exactly N_{j+1}.

Case6 (Any other rule) - each CIFF¬ proof rule ϕ not considered in the previous cases, is also a CIFF proof rule. Hence we could map ϕ to itself.

The third statement, i.e. that for each unmarked CIFF¬ node $\overline{N_j}$ with $j \in [k+3, s-1]$, there exists a CIFF derivation sequence $Seq(\mathrm{D}', \overline{N_j}, \overline{N_{j+1}})$, can be proved similarly to the previous one. □

We are now ready to show the soundness theorem for the CIFF¬ proof procedure which is based on the following proposition.

Proposition 4.7. *Let $\langle Th, A, IC \rangle_{\Re}$ be a CIFF framework and let Q be a query. Let $\langle \Delta, C \rangle$ be a CIFF¬ extracted answer from a CIFF¬ derivation $\mathcal{D}$ for Q. Then there exists a CIFF$^\vee$ derivation $\mathcal{D}'$ for Q with a CIFF$^\vee$ successful node from which $\langle \Delta, C \rangle$ can be extracted.* □

Proof of Proposition 4.7. The initial formula in $\mathcal{D}$ is $F_1 = N_1 = Q \wedge IC^*$ by definition. Each successful branch B= $N_1, \ldots, N_s$ of $\mathcal{D}$ can be divided into two main CIFF$^\neg$ derivation sequences $S_1 = Seq(N_1, N_k)$ and $S_2 = (N_k, N_s)$ where N_k is a node of the form

$$Ps \wedge I^* \wedge Rest$$

such that

- $Ps = prv(p_1(\vec{t_1})) \wedge \ldots \wedge prv(p_n(\vec{t_n}))$ is the conjunction of all the prv atoms in N_k;
- I^* is the conjunction of all the marked implications in N_k and
- the **NAF Switch** rule is the only rule that can be applied to it.

Note that in a CIFF$^\neg$ successful branch $\mathcal{B}$ there is always such a node N_k because $\mathcal{B}$ is a successful branch and by both the definition of a CIFF$^\neg$ selection function and the conditions of the **NAF Switch** rule, that rule can be applied to an unmarked node if no other rule (apart from **Dynamic Allowedness** rule) applies to it. This condition is always reached by a successful branch. Moreover, this is the only node in the branch to which **NAF Switch** can be applied because the rule marks the node as a NAF node and there is no rule which unmarks a node.
Let us consider now a CIFF$^\vee$ derivation for Q. Its initial formula is $F_1 = N_1^\vee = Q \wedge IC$. We can build a CIFF$^\vee$ derivation sequence $S_1^\vee = Seq(N_1^\vee, N_k^\vee)$ such that $N_k^\vee$ is the unmarked node of N_k, i.e.

$$N_k^\vee = \overline{N_k} = p_1(\vec{t_1}) \wedge \ldots \wedge p_n(\vec{t_n}) \wedge I \wedge Rest$$

To build $S_1^\vee$ we can just apply exactly the same rules together with the same rule inputs used in S_1 (obviously each marked implication is replaced by an identical unmarked implication) with the following exceptions:

- for each **NAF Rewriting** rule applied to a marked implication of the form

  $*(((q_1(\vec{t_1}) \rightarrow \bot) \wedge \ldots \wedge (q_m(\vec{t_m}) \rightarrow \bot)) \rightarrow H)$

  obtaining

  $\top \rightarrow [prv(q_1(\vec{t_1})) \vee \ldots \vee prv(q_m(\vec{t_m})) \vee [*(q_1(\vec{t_1}) \rightarrow \bot) \wedge \ldots \wedge *(q_m(\vec{t_m}) \rightarrow \bot) \wedge H]]$

  we apply, instead, the **Logical simplification #4** rule and m times the **Negation rewriting** rule, obtaining

  $q_1(\vec{t_1}) \vee \ldots \vee q_m(\vec{t_m}) \vee H$

  and then the **Extending disjunction** rule obtaining

  $q_1(\vec{t_1}) \vee \cdots \vee q_m(\vec{t_m}) \vee H \vee [(q_1(\vec{t_1}) \rightarrow \bot) \wedge \cdots \wedge (q_m(\vec{t_m}) \rightarrow \bot) \wedge H]$

- for each **Splitting** rule applied to a disjunction of the form

  $prv(q_1(\vec{t_1})) \vee \ldots \vee prv(q_m(\vec{t_m})) \vee [*(q_1(\vec{t_1}) \rightarrow \bot) \wedge \ldots \wedge *(q_m(\vec{t_m}) \rightarrow \bot) \wedge H]$

  we apply, instead, the **Splitting** rule on the disjunction

  $q_1(\vec{t_1}) \vee \cdots \vee q_m(\vec{t_m}) \vee H \vee [(q_1(\vec{t_1}) \rightarrow \bot) \wedge \cdots \wedge (q_m(\vec{t_m}) \rightarrow \bot) \wedge H]$.

It is straightforward that, selecting the appropriate disjuncts when the **Splitting** rule is applied, we obtain a CIFF$^\vee$ derivation sequence $S_1^\vee = Seq(N_1^\vee, N_k^\vee)$ such that $N_k^\vee$ is of the form:

$$p_1(\vec{t_1}) \wedge \ldots \wedge p_n(\vec{t_n}) \wedge I \wedge Rest.$$

Now, from Proposition 4.6 we have that N_{k+2} must be of the form:

$$p_1(\vec{t_1}) \wedge \ldots \wedge p_n(\vec{t_n}) \wedge I^* \wedge Rest.$$

Again from Proposition 4.6, we have that there exists a CIFF derivation sequence $Seq(N_k^\vee, N_s^\vee)$ where $N_s^\vee$ is obtained from N_s by replacing each marked implication with an unmarked one. It is obvious that $N_s^\vee$ is a CIFF$^\vee$ successful node because N_s is a CIFF$^\neg$ successful node, thus no other rule can be applied to it and it does not contain any disjunction to which the **Extending disjunction** rule could be applied. By construction, the CIFF$^\vee$ extracted answer from $N_s^\vee$ is $\langle \Delta, C \rangle$. □

Theorem 4.3 (CIFF$^\neg$ Soundness)**.** *Let $\langle P, A, IC \rangle_\Re$ be an abductive logic program with constraints and let $\langle Th, A, IC \rangle_\Re$ be the corresponding CIFF$^\neg$ framework. Let $\langle \Delta, C \rangle$ be an extracted answer from a successful node N in a CIFF$^\neg$ derivation with respect to $\langle Th, A, IC \rangle_\Re$ and a query Q. Then, $\langle \Delta, C \rangle$ is an abductive answer with constraints to Q with respect to $\langle P, A, IC \rangle_\Re$.*

□

Proof of Theorem 4.3. Due to Proposition 4.7, we can build a CIFF$^\vee$ derivation from which we can extract $\langle \Delta, C \rangle$. Then, due to the soundness of CIFF$^\vee$, we obtain the result. □

The above result shows formally the soundness of the CIFF$^\neg$ proof procedure with respect to the three-valued completion semantics. As for CIFF, we do not show a completeness result for CIFF$^\neg$. Moreover, in the case of CIFF$^\neg$, it is difficult to define a completeness property because the CIFF$^\neg$ proof procedure is designed to avoid returning some sound (though counter-intuitive) abductive answers.
We will return to this topic, informally, in Section 5.2.2.

Chapter 5

The CIFF System

The CIFF System is the SICStus [1] Prolog implementation of the CIFF proof procedure. The version described here is version 4[1].
Previous versions [72, 70] of the CIFF System (until version 3.0), also implemented in SICStus Prolog [1], were developed under the SOCS European Project [167].

CIFF System 4.0 is not an update of version 3.0 but the system has been completely redesigned from scratch. The CIFF System 3.0 is a well-designed and robust system. Moreover, despite the high number of Prolog code-lines, its code is well-commented and simple to understand (and it is absolutely a good point for redesigning a system from scratch!). So, why did we implement the version 4.0 from scratch? Well, the main drawback of version 3.0 is one: its *efficiency*.

Efficiency is a very big drawback in CIFF System 3.0. Generally there are not so many expectations about efficiency of systems and applications written in Prolog but modern Prolog platforms provide efficient underlying mechanisms which together with well-crafted top-level algorithms can help in boosting drastically performance. Thus, the choice of Prolog can be reasonable choice also for efficiency, if good care is given to the algorithms implementing the core routines of a system. This was not the case with CIFF System 3.0 which was basically a very good functional prototype of a CIFF implementation. Each application of a CIFF proof rule in CIFF System 3.0 is computationally very expensive and when a CIFF node grows (in the order of tens of elements in a node, a situation occurring almost at every CIFF run), the computation becomes unacceptable in practice.

Apart from efficiency, the redesign in the CIFF System 4.0 includes a completely new NAF extension implementation. This is also because at the time of version 3.0, the CIFF$^{\neg}$ proof procedure was not still developed at theoretical level and the implementation was based on an adaptation of the NAF extension for the IFF proof procedure specified in the Toni-Sadri paper [157].

The work behind the new version was huge and it took almost two years. A great help in the redesign of the system was given by version 3.0 itself because some ideas of code-design have been reused and, more importantly, the older version has been heavily used as a sort of “oracle” to see the correctness of basic operations implemented with new algorithms and data structures. The CIFF System 3.0 was a big time-saving entity towards the new version.
At the very beginning, the main choice was whether to use Prolog or not. On one side, the drawbacks of Prolog are basically (1) the efficiency of the platform (even if nowadays it can be reasonable, as noticed above); (2) the coding style, which initially requires high flexibility of thinking, especially if one it is used to imperative programming and (3) the little Prolog “community” which implies not so much standard solutions and methodologies for Prolog applications. These points are very important if the application is composed of thousands of codelines. And the CIFF System is one of such applications!

[1]The CIFF System 4.0 can be downloaded from `http://www.di.unipi.it/~terreni/research.php`

On the other side, Prolog includes many interesting features for implementing CIFF. First of all, the CIFF proof procedure is an abductive logic programming with constraints proof procedure: it is natural to think of Prolog as the underlying platform for an implementation. Indeed Prolog is based on unification and resolution and also CIFF is based on such concepts at the core level. Choosing Prolog, these are for free and moreover it is assumed that they are robust and well-implemented. Moreover, modern Prolog platforms have a number of standard features and data structures (e.g. attribute variables [93]) which make possible to think about fast algorithms underlying the CIFF proof rules. The last issue regards *constraints*. Many Prolog platform present extensions to Constraint Logic Programming, in particular integrating constraint solvers for finite domain constraints. This is a key issue in choosing Prolog because interfacing the system with the constraint solver becomes a comparatively easy task.

At the end, the choice was to continue in using Prolog, in particular in using SICStus Prolog which is probably the best Prolog platform for robustness, efficiency and constraint solving. The other choice which would have been very interesting (also at theoretical level), would have been the implementation of CIFF as an abductive extension of the Warren Abstract Machine [179, 2]. But this would have been a mayor new contribution.

The rest of the chapter is organized as follows. In the next section we give an overview of the implementation presenting the main design choices and sketching the main algorithms. In Section 5.2 we give a brief comparison with other proof procedures and other systems, in particular with the most related system, the A-System [111, 139], and two answer set solvers, namely SMODELS [136, 166] and DLV [68, 119]. Finally in Section 5.3 conclusions are drawn together with some hints for future work.

5.1 The CIFF System: an Overview

The CIFF System 4.0 requires SICStus Prolog 3.11.2 (or newer versions of SICStus Prolog 3 release) and it starts by compiling, on the Prolog top-level shell, the `ciff.pl` file which, in turn, compiles all the CIFF System modules.

The CIFF System is composed of the following modules:

- the *main* module[2]: it provides the main predicate (`run_ciff/3`) for starting a CIFF computation and it manages the main computational cycle,
- the *flags* module: it provides all the CIFF flags a user can set to change the CIFF behavior (e.g. turning on/off the NAF module, the level of debugging info returned to the user and so on); all the flags are detailed in the CIFF manual [171],
- the *preprocess* module: it provides the translation from the user's abductive logic program with constraints to its internal representation,
- the *proc* module: it provides the predicate `sat/2` which, roughly speaking, implements the CIFF proof rules,
- the *proc-aux* module: it provides auxiliary predicates to `sat/2`,
- the *aux* module: it provides general auxiliary predicates,
- the *constraints* module: it provides the wrapper around the SICStus CLPFD constraint solver,
- the *debug* module: it provides predicates to return debugging info to the user, and

[2]Each "module" is implemented in a `.pl` file having the same name with a `ciff-` prefix. E.g. the *ciff-main* module is implemented by the `ciff-main.pl` file

- the *ground-ics* module: it provides an efficient algorithm for handling some classes of integrity constraints (see Section 5.1.6).

Once compiled the system, a CIFF computation starts with the `run_ciff/3` call:[3]

```
| ?- run_ciff(+ALPFiles,+Query,-Answer).
```

where `ALPFiles` is a list of `.alp` files which together represent an abductive logic program with constraints (ALPC) and `Answer` is the output variables which will be instantiated to the CIFF extracted answers (if any) to the list of literals provided in `Query`. The possibility of specifying more than one `.alp` file in the `ALPFiles` list is to facilitate the user in writing CIFF applications. A typical example is a two elements list where one `.alp` file contains the clauses and the integrity constraints which specify the problem and the other file contains the specification of the particular problem instance. In this way the first file could be reused for other instances.

A CIFF computation is composed of a preprocessing phase which both translates `ALPFiles` into its internal representation and initializes the system data structures, and an abductive processing loop implemented as a recursive call to the `sat/2` predicate. Each call to the `sat/2` represents an application of a CIFF proof rule and at the end of the loop `Answer` is instantiated either to a CIFF extracted answer or to the special `undefined` (if the **Dynamic Allowedness** rule has been applied). Further answers can be obtained through Prolog backtracking. If no answer has been found, the system fails returning the control to the user. The main CIFF computational cycle is described in the following picture.

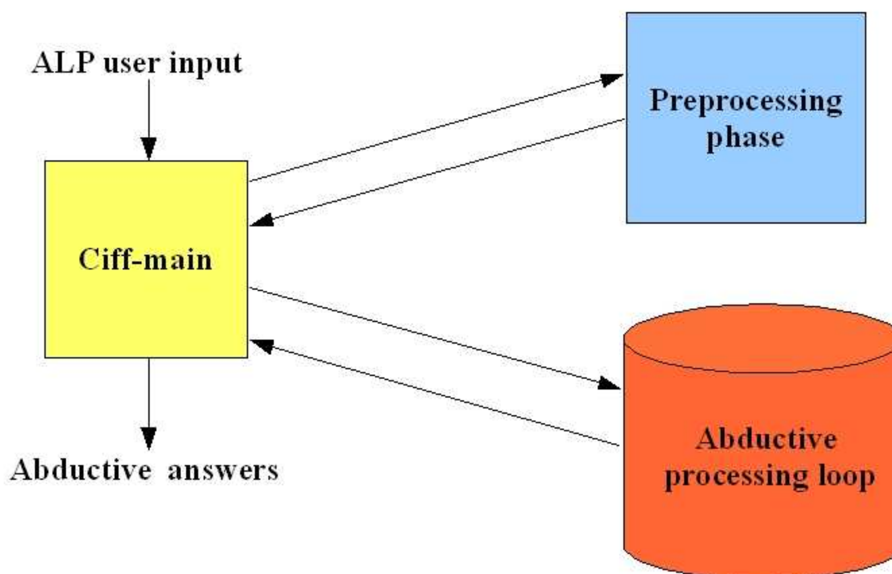


Figure 5.1: The CIFF System: main computational cycle

The preprocessing phase, detailed in Section 5.1.1, stores the iff-definitions derived from the user program in an internal representation in which the various elements are maintained in disjoint sets depending on their type: abducibles, defined atoms, constraints, equalities and so on. The same

[3] In what follows, we will use several Prolog standard notations: variable names start with capital letters, `!` is the *cut* operator, `[ ]` is the notation for lists and `\+` stands for negation. Moreover the notation `pred/n` stands for "the predicate `pred` with arity `n`" and finally `+,-` before a predicate argument `Arg` represent that `Arg` is an "input" argument and an "output" argument respectively.

type-based partition is performed on the first CIFF node of the computation, composed of the conjunction of the integrity constraints in `ALPFiles` and the `Query`.
This node is passed as the first input of the `sat(+State, -Answer)` predicate, where `State` is the *state* of the CIFF computation, which is composed of the representation of the current CIFF node plus other auxiliary run-time information, in particular for NAF (see below). A CIFF node is represented by a `state` atom of the form:

```
state(Diseqs,Constraints,Implications,DefinedAtoms,Abducibles,Disjunctions)
```

where the arguments represent the CIFF conjuncts of the node. They are maintained aggregated, depending on their "type" (and their names are quite self-explanatory): `Diseqs` represents the set of CIFF inequalities, `Constraints` represents the current finite domain constraint store, `Implications` the set implications, `DefinedAtoms` the set of defined atoms, `Abducibles` the set of currently abduced atoms and finally `Disjunctions` represents the set of disjunctive CIFF conjuncts.

Each element in `Disjunctions` represents a *choice point* of the computation and when the **Splitting** rule is applied on it, the current CIFF node is selected in a CIFF formula following the left-most criterion. When the computation on a certain node ends (instantiating `Answer` or failing), the next node is selected via Prolog backtracking. Delegating the management of the switching among CIFF nodes to the Prolog backtracking allows for maintaining in the current `State` only the information of the selected CIFF node and, for performance, it is the only possible practical choice.
Each `sat/2` clause represents the implementation of a CIFF proof rule and it has the following general structure:

1. a first part to search for a *rule input* in the node satisfying the applicability conditions of the implemented proof rule
2. a second part for applying the rule and updating the `State`
3. a recursive call to sat/2 with the updated `State`.

The search for a rule input is done by scanning sequentially the CIFF conjuncts in the current node. Thus, maintaining a type-based partition of the CIFF conjuncts throughout a computation plays a very important role in terms of efficiency and clarity of the code. This is because each CIFF proof rule operates on certain types of CIFF conjuncts (e.g. the **Unfolding atoms** rule operates on defined atoms, the **Factoring** rule operates over abducibles and so on) and that separation facilitates the search for a rule input among the CIFF conjuncts.
If such a rule input has been found, the concrete application of the rule is performed (relying upon the *procaux*, *aux* and *constraints* modules, also depending on the given proof rule), and the `State` is updated.

Implementing the proof rules as `sat/2` clauses, implies that, due to the Prolog semantics, the order of these clauses determines the priority given by the CIFF System to a certain proof rule during the computation. Roughly speaking, the system tries to apply the proof rule implemented in the first `sat` clause; if no rule input is found, then the following `sat` clause is tried. If no rule can be applied to a node and no inconsistencies have been detected, an abductive `Answer` is extracted and then returned to *main* module.
When all the *choice points* have been traversed, the CIFF computation ends returning the answer `no` to the user indicating that no more answers can be found.

The above description outlines the implementation of the CIFF selection function. We use "the standard" Prolog-like selection function, i.e. we always select the left-most CIFF node in a CIFF formula. It is not a *fair* selection function in the sense that it does not ensure completeness (see the end of Chapter 4 for further details), but it has been found to be the only possible practical choice in terms of efficiency (in both time and space) taking advantage of Prolog backtracking.

Concerning the order of selection of the proof rules in a CIFF node, this is determined by the order of the `sat` clauses. If a `sat` clause defining a CIFF proof rule, e.g. **Unfolding atoms (R1)**, is placed before the `sat` clause defining, e.g. **Propagation (R3)**, then the system tries to find a rule input for **R1** and if no such rule input can be found, then the system tries the same for **R3** and so on. Further details on this topic are given in Section 5.1.4.

However, some CIFF proof rules, in particular those rules regarding equalities (**Substitution in atoms**, **Equality rewriting in atoms** and so on) and the most part of the **Logical simplification** rules, are not implemented each one as a `sat/2` clause. Rather they are embedded in the other main CIFF proof rules, i.e. when a proof rule R, e.g. **Unfolding atoms**, is applied, the system first determines all the operations on the equalities and the logical simplifications arising from R and then it performs them immediately. Thus the updated `State` is not only derived from the application of R but also from the application of all the other embedded proof rules. This point, as detailed in Section 5.1.2, determines a great enhancement of performance and moreover it allows for a better readability of a CIFF computation for debugging purposes.

Until now we For now, we have not yet described the NAF module whose details will be given in Section 5.1.5. To give a global picture of the system, the implementation of the NAF extension, following the specifications given in Chapter 4, can be logically split in two parts: one part which handles the marked integrity constraints and the production of the *prv* atoms and the second part, i.e. from the application of the *NAF Switch* rule onward, checking the so-far produced *prv* atoms, avoiding new abductions through the *NAF factoring* rules.
The preprocessing phase changes according to whether the NAF extension is activated or not, producing marked or non-marked implications in the first node, respectively In the same way NAF proof rules (implemented again as `sat` clauses) are taken into account or not by the system. The NAF extension is activated through a CIFF flag and it requires a distinct preprocessing for integrity constraints according to the CIFF$^\neg$ specifications.
Moreover, to embrace the NAF extension, the `State` argument of `sat` is a structure of the following form:

```
state(Diseqs,Constraints,Implications,DefinedAtoms,Abducibles,Disjunctions):
 naf_state(DeltaNAF,NAFSwitch)
```

where `DeltaNAF` represents the marking set of a CIFF$^\neg$ node and `NAFSwitch` is a `1/0` flag indicating whether the *NAF Switch* rule has been applied or not.

The interactions of the modules in a CIFF computation is displayed in Figure 5.2.
In the rest of the chapter, we detail the main parts of the system together with the solution adopted, rather than describing the CIFF System "line-by-line" from begin to end. We start from the preprocessing phase together with a description of the user input and the returned answer. Then we describe, in Section 5.1.2 and in Section 5.1.3, the management of both variables (in particular their quantification) and constraints. In Section 5.1.4 we discuss the loop checking routines in the CIFF System. In Section 5.1.5 we describe the issues in the implementation of the NAF extension and finally in Section 5.1.6 we describe the *ground-ics* module containing an efficient algorithm for evaluating some classes of integrity constraints.

5.1.1 Input Programs, Preprocessing and Abductive Answers

As said in the previous section, a CIFF run starts with a `run_ciff(+ALPFiles,+Query,-Answer)` call where the `.alp` files in the `ALPFiles` list represent the user-defined abductive logic program with constraints.
Each abductive logic program with constraints (ALPC) consists of the following components, which could be placed in any position in any `.alp` file:

- Declarations of abducible predicates, using the predicate `abducible`. For example an abducible predicate `abd` with arity 2, is declared via

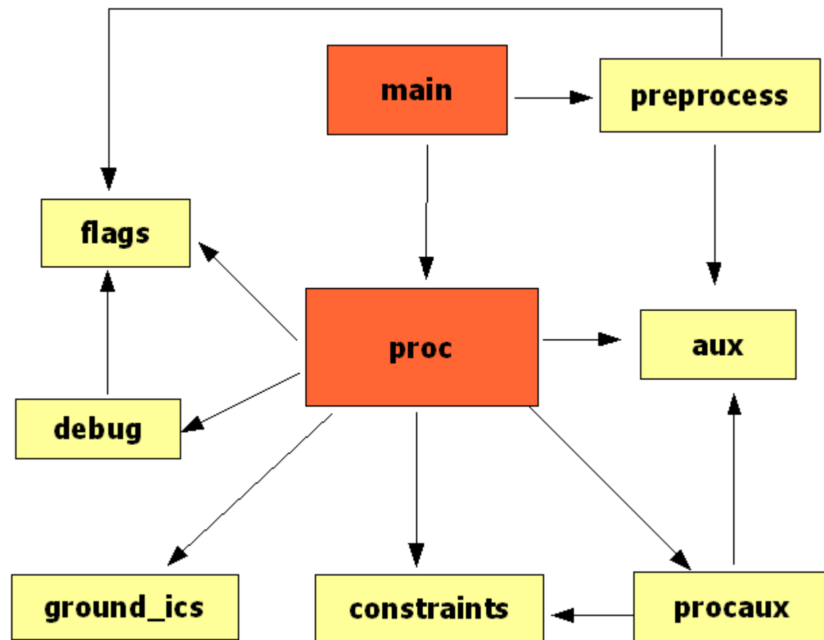


Figure 5.2: The CIFF System: modules interactions

```
abducible(abd(_,_)).
```

- Clauses, represented as

  ```
  A :- L1, ..., Ln.
  ```

- Integrity constraints, represented as

  ```
  [L1, ..., Lm] implies [A1, ..., An].
  ```

  where the left-hand side list represents a conjunction of CIFF literals while the right-hand side list represents a disjunction of CIFF atoms.

Equality/inequality atoms are defined via `=`, `\==`. Constraint predicates are `#=`, `#\=`, `#<`, `#>`, `#=<`, `#>=` together with the domain inclusion predicate `in` (e.g. `X in 1..100`). Finally, negative literals are of the form `not(Atom)` where `Atom` is an ordinary atom.

Example 5.1. *The following is the CIFF System representation of the Example 3.3.*

```
% Abducibles:
 abducible(sprinkler_was_on).
 abducible(rain_last_night).

% Definitions:
 grass_is_wet :- rained_last_night.
 grass_is_wet :- sprinkler_was_on.
 shoes_are_wet :- grass_is_wet.

% ICs:
 [rained_last_night] implies [false].
```

Let assume that the file is named `grass.alp`*. The following is the call to run CIFF on the query* *shoes_are_wet:*

```
run_ciff([grass],[shoes_are_wet],Answer).
```

□

Example 5.2. *The following is the CIFF System representation of the Example 3.8.*

```
% Abducibles:
 abducible(empty(_)).
 abducible(power_failure(_)).

% Definitions:
 lamp(a).
 battery(b,c).
 faulty_lamp :- power_failure(X), not(backup(X)).
 backup(X) :- battery(X,Y), not(empty(Y)).
```

Let assume that the file is named `lamp.alp`*. The following is the call to run CIFF on the query* *faulty_lamp:*

```
run_ciff([lamp],[faulty_lamp],Answer).
```

□

The *preprocess* module translates both `ALPFiles` and `Query` and stores them in their internal representation. We can logically split the user input in two main parts: a *static* part (iff-definitions) and a *dynamic* part (query plus integrity constraints). The main objectives in preprocessing are to separate elements of both input parts depending on their types (as discussed in the previous section) and to store the static part in order to efficiently retrieve the information when needed at run-time.
Each iff-definition is stored in the Prolog global State (through an *assertion*) in the form of

```
iff_def(+PredName, +Arguments, +Disjuncts)
```

where `PredName` is the name of the predicate, `Arguments` is a list of N distinct variables (where N is the arity of the predicate) and `Disjuncts` is the list of disjuncts in the form

```
disj(+Constraints,+Equalities,+Diseqs,+Implications,+DefinedAtoms,+Abds)
```

where the list of `Implications` is obtained transforming the negative literals in the clauses in implicative form whose internal representation will be discussed later.
For example the clause

```
p(X,Y,d) :- X #> 5, a(Y), q(Z), r(X), Y \== Z
```

is translated into (assuming that `a/1` is abducible and the above clause is the only clause for `p/3`):

```
iff_def(p,[X,Y,W],[disj([X#>5],[W=d],[Y\==Z],[],[q(Z),r(X)],[a(Y)])]).
```

Suppose now that during the computation, we have to *unfold* `p(2,b,d)`. To retrieve the iff-definition of tt p/3 we simply call:

```
iff_def(p,[2,b,d],-Disjuncts)
```

where `Disjuncts` in this case will be:

```
[disj([2#>5],[d=d],[b\==Z],[],[q(Z),r(2)],[a(b)])]
```

In this way we have both a direct access to the iff-definitions and a clear separation of the elements by their type in order to update efficiently the current `State` when needed.

Integrity constraints are not stored in the Prolog global state as they, together with the `Query`, will represent the first CIFF node of the computation. However, in the internal representation, their components are again separated depending on their type as follows:

```
body(BCons,BEqs,BAts,BAbds) implies head(HCons,HEqs,HDiseqs,HAts,HAbds)
```

Consider the integrity constraint

```
[not(c), q(Y), p(X), r(Z), X=Y, X\==Z] implies [false]
```

Its internal representation (assuming `r/1` is abducible) is

```
body([],[X=Y],[q(Y),p(X)],[r(Z)]) implies head([],[X=Z],[],[c],[])
```

This is because the preprocessing phase translates the negative elements into their positive form (including inequalities, transformed into equalities) and puts them into the head, eventually removing `false`[4].

The implicative form of negative literals in the clauses, e.g. `not(p)`, is as follows.

```
body([],[],[p],[]) implies head([],[],[],[false],[])
```

The above representation may seem quite complex, but it allows for maintaining a type-based separation, thus avoiding many run-time routines of type-checking. The dynamic part of the input represents the `State` argument of the `sat/2` predicate:

```
state(Diseqs,Constraints,Implications,DefinedAtoms,Abducibles,Disjunctions)
```

where all the components arise from the various conjuncts in the `Query`, apart `Implications` which is composed of the conjunction of all the preprocessed integrity constraints and the implicative form of the negative literals in the `Query`.

The last remarks of this section are about the form of an abductive `Answer`. A CIFF extracted answer is represented by a triple, namely a list of abducible atoms, a list of CIFF inequalities DE and finally a list of finite domain constraints Γ. The set of equalities E is not returned explicitly to the user because the final substitution (in E) is applied directly by the system.

```
Abducibles:Inequalities:Constraints:Labels
```

The `Labels` component, to be discussed in Section 5.1.3, contains a consistent assignment (or a *labeling*) of the constraint variables in `Constraints`.

Example 5.3. *The only CIFF extracted answer of the Example 5.1 is:*

```
[sprinkler_was_on]:[]:[]:[]
```

The two answers of the Example 5.2 are:

```
A1 =  [power_failure(b),empty(c)]:[]:[]:[]
A2 =  [power_failure(_A)]:[_A\==b]:[]:[]
```

□

[4] Note that, for now, we do not take into account the NAF extension. When we describe the NAF extension we will slightly change this representation because negative literals in the body of a marked integrity constraint have to stay in the body.

5.1.2 Variable Handling

Variables play a fundamental role in both the theory and the implementation of the CIFF proof procedure. The presence of both existentially quantified and universally quantified variables is difficult to manage at both levels.
We recall that in a CIFF node, a universal variable U is a variable which occurs *only* in an implication which also defines the scope of U, while an existential variable E is a variable occurring in *at least* a CIFF conjunct which is not an implication and the scope of E is the whole CIFF node.
In this section we focus on two issues: checking variable quantification at run-time and the implementation of the proof rules concerning equalities, i.e. **Equality rewriting** and **Substitution** rules. To simplify the presentation, we refer to those rules as *equality* rules.

The main issue in variable quantification is that existentially quantified variables may occur inside an implication due to a *propagation* rule application. The implementation of a CIFF proof rule managing implications must take into account that a variable can be either existentially or universally quantified, in order to avoid unintended behaviors. Thus, the CIFF System must know at run-time the quantification of a variable and moreover, as it is largely required throughout a CIFF computation, there is the need of efficient access to this information.
The solution adopted is to associate to existential variables the `existential/0` *attribute*. The mechanism of attributed variables ([93]), i.e., roughly speaking, "appending" some information (*attributes*) to a variable, allows for determining variable quantification in a fast and reliable way through a local access to the variable itself. The idea is very simple: all the variables occurring outside the `Implications` argument of the current `State` have the `existential` attributed associated with them; during the application of a CIFF proof rule, the `existential` attribute is associated to each variable moving from an implication to the outside. In this way at the end of each CIFF proof rule application, we have two disjoint sets of variables: the set of existentially quantified variables (with the `existential` attribute) and the set of universally quantified variables (without attributes).

As said in Section 5.1, *equality* rules are implemented at the end of the other main proof rules. For example, consider unfolding a CIFF conjunct `p(X,Y)` in a CIFF node where the iff-definition of `p/2` is:

```
iff_def(p, [Arg1,Arg2], [disj([],[Arg1=a,Arg2=b],[],[],[],[])]).
```

Following the CIFF specifications, unfolding will result in the set of equalities

```
X=Arg1, Y=Arg2, Arg1=a, Arg2=b.
```

Successively, this set has to be simplified and propagated to the whole node by applying a number of "variable" CIFF proof rules. In the CIFF System, instead, the latter operations are done all-in-one, simplifying locally the set of equalities resulting from the application of a CIFF proof rule and then propagating the resulting set of substitutions to the node (in case of existentially quantified variables) or to the implication (in case of universally quantified variables). In doing this, the system takes advantage of the underlying Prolog unification as much as possible.

In CIFF, *equality* rules rely upon a slightly modified version of the Martelli-Montanari unification algorithm [129] and the problem in using Prolog unification is again related to variable quantification: all the variables being existentially quantified, from a Prolog point of view, Martelli-Montanari conditions on variable quantification cannot be checked by Prolog itself.
Due to this, the implementation of the *equality* rules relies on:

- Prolog unification for all the *equality* rules not involving implications: this is because in that case all the variables are existentially quantified, hence the Prolog unification is able to manage them;

- an explicit implementation of the Martelli-Montanari algorithm (in the *proc-aux* module) otherwise.

Addressing *equality* rules at the end of the other main proof rules brings some important benefits. The first one is efficiency. Performing all at once those operations at the end of a main proof rule grants an immediate and local access to those variables which need to be unified/propagated and, moreover, the absence of dedicated `sat` clauses for those rules avoids searching for suitable equalities in the current CIFF node to be unified/propagated. To give an idea, in a typical abductive logic program, each main CIFF proof rule gives out the preconditions to the application of 4-5 *equality* rules: this would have required a corresponding number of CIFF iterations. Not surprisingly, this point together with taking advantage as much as possible of Prolog unification boosts significantly the performance of the system (execution times of CIFF System 3.0 were approximately halved).

Note also that the use of Prolog unification for existential variables makes it useless to maintain the equalities as CIFF conjuncts in the current `State`. This is because at the end of a proof rule all the possible substitutions have been exhaustively applied.

Finally, dropping *equality* rules from the main `sat` loop brings also a great enhancement in debugging. This is because for a human eye, even an expert human eye, it is much easier to follow the trace of a CIFF computation without a verbose application of *equality* rules. Consider again the example above. With a verbose application of *equality* rules we would have:

```
1 - p(X,Y)
2 - X=Arg1, Y=Arg2, Arg1=a, Arg2=b
3 - X=a, Y=Arg2, Arg1=a, Arg2=b
...
```

Instead, applying all-at-once the *equality* rules we simply obtain:

```
1 - p(X,Y)
2 - X=a, Y=b
```

It is obvious that in a CIFF node with tens of CIFF conjuncts and tens of variables a verbose application of *equality* rules leads to a untraceable computation.

5.1.3 Constraints Handling

The management of CLP constrains is another big implementation issue of the CIFF System. The SICStus CLP constraint solver for Finite Domains (*CLPFD solver*) is a state-of-the-art constraint solver [77] and its interface to the standard Prolog engine is quite simple and effective. However, as happens in general with constraint solver implementations, given a certain constraint store, the only way to know about its satisfiability is by performing an exhaustive checking (or *labeling*) of the involved constraint variables.
This implies that during a CIFF computation, the constraint store is not ensured to be consistent, unless *labeling* is performed.
Labeling is an expensive operation on big constraint stores, and so it should be used carefully during a CIFF computation. Applying frequently the *constraint solving* rule to exhaustively check the constraint store could not be a good practical solution because if on one side some CIFF branches can be cut if the store is unsatisfiable, on the other side the overhead due to labeling could be very big.
The choice taken in the implementation is to perform labeling every N CIFF proof rule applications, where N is modifiable by the user through a CIFF flag. A further check is always performed at the very end of a CIFF branch computation, when no other proof rules can be applied to the current node and before an answer is returned to the user.

Apart from efficiency issues, the main problem in using labeling during a CIFF computation is that labeling results in a complete grounding of the constraint variables, thus, the constraint store in the CIFF extracted answer would be totally ground.

To solve this issue, the simplest solution is to build, on-the-fly, a working copy of the constraint store with *fresh* constraint variables and then to perform labeling on that working copy. If the constraint store is unsatisfiable the current CIFF branch fails, otherwise the computation goes on with the original copy. But this does not work in practice because copying a constraint store is a very expensive operation in terms of both space and time and it is not reliable.

Instead, the solution adopted is able to perform labeling by maintaining a single copy of the constraint store by means of a backtracking algorithm which, when needed, checks the satisfiability of the CLP store through a labeling and then restores the non-ground values via a forced backtracking. Surprisingly, this is the best solution, to our knowledge, for efficiency and memory usage which allows both satisfiability checks and non-ground answers.

Now we sketch the implementation of that backtracking algorithm. We assume it is applied at the last CIFF step in a branch, before an abductive answer is returned to the user.

The following is a piece of the `sat` last clause.

```
%%%Final clause: returns an abudctive answer if the solver succeed
S1   sat( State, Answer) :-
S2       ...
S3       !,
S4       (label_flag(0) ; label_flag(1)),
S5       run_time_label(CVars),
S6       make_label_assocs(CVars,0,Assocs),
S7       ...
```

The symbol `;` at line `S4` represents a disjunction with the standard backtracking semantics of Prolog: if `label_flag(0)` succeeds, the disjunction becomes a backtracking point and `label_flag(1)` is evaluated in case of failure later on. The predicate `label_flag` represents whether labeling has been performed on this store (`label_flag(1)` succeeds) or not yet (`label_flag(0)` succeeds). It is asserted, with value `0`, in the Prolog global state at CIFF System initialization as follows:

```
assert(label_flag(0)).
```

The asserted value of `label_flag` is then used in `run_time_label(CVars)` which effectively performs the labeling on the constraint variables `CVars`. The predicate in line `S6` regards the added component in the implementation of a CIFF abductive answer: the association of a ground solution for the constraint variables as will be discussed later on.

The `run_time_label/1` predicate is defined as follows[5]

```
L1    run_time_label(CVars) :-
L2        labeling_mode(Mode),
L3        retract(label_flag(V)),
L4        if (V = 0) {
L5           if (labeling(Mode,CVars)) {
L6               assert_label_list(CVars,0),
L7               assert(label_flag(1)),
L8               !,
L9               fail
```

[5]In real Prolog code the `if-else` statement is represented by the operators `-> ;` whose semantics ensures that the second branch is not a backtracking point if the first branch succeeds. For readability we use an "imperative-like" syntax.

```
L10          } else {
L11             assert(label_flag(0)),
L12             !,
L13             fail
L14          }
L15       } else {
L16          assert(label_flag(0))
L17       }.
```

Mode in line L2 represents one of the labeling algorithms offered by the SICStus CLPFD solver [1]. It can be set through the labeling_mode CIFF flag.
In line L3 the value V of label_flag is checked (recall that in the sat clause both 0 and 1 are accepted by the system) and the atom is *retracted* from the global Prolog state.
If V = 0 (the initial situation), the labeling (line L5) is performed. If it fails, the label_flag(0) is asserted again and run_time_label fails too. The last assertion ensures that sat also fails because the disjunction in line S4 fails. If the labeling succeeds, then label_flag(1) is asserted, run_time_label fails again but when the control returns, via backtracking, to sat at line S4, it does not fail due to the presence of label_flag(1) in the Prolog global state. Hence run_time_label is traversed again, the test at line L4 fails because now the value of V is 1, and, before succeeding, label_flag(0) is asserted again in order to restore the initial situation.

Note that whenever the run_time_label clause fails, the variables in the constraint store are restored via Prolog backtracking. This happens also when the labeling succeeds.
In this way, if labeling succeeds, and then label_flag(1) is asserted, we are ensured that the current constraint store is satisfiable but constraint variables are not bound to ground values.

The assert_label_list/2 predicate in line L6 simply asserts in the Prolog global state the numeric values returned by labeling. Those assertions are done maintaining the positional order of the list of variables CVars and they are used by the make_label_assocs to retrieve the consistent assignment to be displayed to the user together with the abductive answer.

The other use of the algorithm (every N CIFF steps) is very similar. The only difference is that the assertions of the ground values are not performed and moreover, in the real code, there is a further flag for distinguishing whether it is the last step or an Nth step.

5.1.4 Loop Checking and CIFF Proof Rules Ordering

CIFF specifications do not ensure that a CIFF computation terminates. E.g., a CIFF computation for the query p with respect to the abductive logic program composed of the single clause $p \leftarrow p$, will loop forever applying *unfolding*. This example (and many other examples) has a theoretical justification: p is *undefined* under the three-valued completion semantics thus neither the query p nor the query $\neg p$ succeeds, leading to infinite looping.

However, when switching from theory to practice, further non-terminating sources arise which do not have a theoretical justification. In this section we address four types of non-termination sources:

- the order of selection of CIFF nodes in a CIFF formula;
- the order of selection of the CIFF proof rules;
- the order of selection of CIFF conjuncts in a CIFF node;
- CIFF proof rules which cannot be applied twice in a CIFF branch.

Both in the specifications and in the implementation, the number of leaf CIFF nodes in a CIFF formula can be incremented only by the application of **Splitting** on a disjunctive CIFF conjunct.

While, theoretically, any leaf node can be selected at any CIFF step, in the implementation we chose the *left-most* strategy. This implies that, sometimes, abductive answers can never be returned to the user because the system loops forever in a non-terminating branch. This choice follows the classical Prolog approach and it allows for the direct use of the underlying Prolog backtracking in case of failures. The benefits are in terms of efficiency and simplicity because no additional machinery is needed to keep trace of non-selected nodes, while the main drawback is obviously the potential non-termination. However, we argue that managing even small-medium size CIFF applications, this approach is the only computationally feasible approach because managing branching without relying upon Prolog backtracking is computationally very expensive in both time and space.

The second source of non-termination is the order of selection of the CIFF proof rules. Consider the abductive logic program consisting of the clause $p \leftarrow p$ and the integrity constraint $a \rightarrow false$ where a is an abducible predicate. A CIFF computation with the query p, a will loop forever if **Unfolding atoms** has a higher priority than **Propagation**. In this case the system applies **Unfolding atoms** infinitely many times on p, while propagating a to $a \rightarrow false$ would lead to an immediate failure.

This is a main issue in implementing CIFF because in general there is no evidence of which rule should be applied at each step in order to avoid non-termination, or at least to reduce the final size of a proof tree. In the example above, the right choice was **Propagation** and it seems that setting a high priority to **Propagation** rule can prevent some non-terminating situations. However, in general **Propagation** is a computationally expensive rule because it increments the set of implications in a node which are a main source of inefficiency due to the presence of universally quantified variables. Hence, in many applications it could be better to give a lower priority to **Propagation** and in general to rules managing implications.

As we said in Section 5.1 each CIFF proof rule is implemented as a `sat/2` clause in the *proc* module. This was a central choice in the design of CIFF and that solution implies that each proof rule has a predefined priority during a CIFF computation which corresponds to its order among the `sat` clauses. I.e. the system, at each iteration, tries to apply the proof rule represented by the first `sat` clause. If no rule input can be found in the node, the second `sat` clause is explored and so on. The last `sat` clause represents the fact that no rule can be applied on the node and then an abductive answer is returned to the user.

It is clear that in certain cases, if the order of the `sat` clauses is wrong, this solution does not prevent non-termination. Nevertheless we think that this is a good practical solution because on the one side it allows for having a modular implementation of the proof rules and on the other side it is possible to change the order of the rules simply by moving the `sat` clauses up and down in the *proc* module, thus allowing for a simple domain dependent tuning of the system.

The third potential non-termination source is the order of selection of CIFF conjuncts in a node. Consider an abductive logic program composed of the clauses $p \leftarrow p$ and $q \leftarrow false$. The query p, q will not terminate **Unfolding atoms** p forever. However, if CIFF unfolds q, the query fails immediately. Note that the CIFF proof rule is the same but in one case CIFF does not terminate and in the other case it does.

To prevent such behavior, the CIFF System implements a left-most selection strategy on the lists of typed CIFF conjuncts, but the insertion of new conjuncts is done at the end of the lists. Consider again the above example and let us suppose that the current `State` has the list `[p, q]` of defined atoms. First CIFF selects `p` as the rule input for **Unfolding atoms** because is the first element in the list. The application of the rule will result in inserting `p` in the list again, but this is done at the end of the list. Thus the new list of defined atoms will be `[q, p]`. At the next CIFF step, `q` will be selected to be unfolded, thus leading to failure. We believe that this strategy represent a good compromise between efficiency and loop prevention.

The fourth and last source of non-termination is represented by those CIFF proof rules which require explicit loop-checking in the specification of a CIFF derivation, i.e. **Factoring** and **Prop-**

agation[6].

We first consider **Factoring**. To implement loop-checking for **Factoring** we added two numbers to each abducible CIFF conjunct in the current `State`: a unique identifier and a *factoring counter*. I.e. each element in the `Abducibles` argument of `state` is of the form:

```
Abd:Id:FactCounter
```

where `Id` is a unique system-generated integer while `FactCounter` indicates the *least* abducible `Abd'` (i.e. the abducible with the least identifier) such that `Abd` and `Abd'` have not been factorized yet. The list of abducibles is maintained ordered with respect to the identifiers throughout the computation and when a new abducible is inserted in the list, its `FactCounter` is initialized to `1`. When **Factoring** is selected, the system selects a *pivot* abducible atom `Abd:Id:FactCounter` which is the abducible with the least `Id` such that `FactCounter < Id`. If no pivot is found then all the pairs of abducibles have already been factorized. Otherwise, if `Abd` is found then the least abducible `Abd':Id':FactCounter'` such that `FactCounter <= Id' < Id` is searched for. If there is such an `Abd'`, then `Abd` and `Abd'` are factorized and `FactCounter` is set to `Id'` else `FactCounter` is set to `Id + 1` and the next pivot is selected for **Factoring**.

Example 5.4. *Consider the following three abducibles in the current* `State` *and let us assume that no* ***Factoring*** *rule has been applied yet in the branch (*`FactCounter` *is set to* `1` *for all the abducibles):*

```
a(X):1:1
a(Y):2:1
a(Z):3:1
```

When ***Factoring*** *is selected for the first time,* `a(X)` *is not selected as a pivot because* `1 = 1`*. Instead* `a(Y)` *is selected and it is factorized to* `a(X)`*. This is because* `1 <= 1 < 2`*. Then the* `FactCounter` *of* `a(Y)` *is set to* `1 + 1 = 2`*. We obtain:*

```
a(X):1:1
a(Y):2:2
a(Z):3:1
```

At this stage only `a(Z)` *can be selected as a pivot and it is factorized first to* `a(X)` *and then to* `a(Y)` *obtaining:*

```
a(X):1:1
a(Y):2:2
a(Z):3:3
```

At this point no abducible can be selected as a pivot and, indeed, all the pairs of abducibles have been factorized.

Note that, given a pivot abducible, only those elements which precede it in the list are considered for **Factoring**. This is because each pair of abducibles have to be factorized only once in a CIFF branch. If for each pivot we considered all the abducibles in the list, each pair of abducibles would have been factorized twice.
Following the specifications, when two abducibles are factorized, e.g. `a(X)` and `a(Y)`, in one successor CIFF node the two abducibles are still in the list and the inequality `X ≡ Y` is added to the current `State`, while in the other successor CIFF node, the pivot abducible is deleted from the list and the substitution `Y = X` is applied to the whole `State`. If the two abducibles are both ground, then only one successor node is computed and if they are equal, the pivot is simply deleted.

[6] Note that in CIFF specifications, also other rules concerning equalities need loop-checking. This is obtained for-free in the implementation by performing them at the end of the other proof rules.

The loop checking for *propagation* is a bit more complex.
In order to perform a loop checking for **Propagation** which follows exactly the specifications, we need to maintain a data structure, for each implication in the current `State`, containing all the CIFF conjuncts (abducibles and defined atoms) to which that implication has been propagated. This data structure can be very big and checking whether an element occurs in it could be very expensive, and the situation is worsened by the presence of existentially quantified variables. Suppose we have a CIFF conjunct `p(X)` in the `State` and the clause `p(X) :- p(Y)` in the input program which, when applied by **Unfolding atoms**, introduces a fresh existential variable in the node. I.e. the atom `p(X)` is unfolded to `p(X1)`, then to `p(X2)` and so on. Each implication containing `p(Z)` in its body has to be propagated to each atom `p(Xi)` because the `Xi` variables are all distinct.

Obviously, maintaining for each implication in a node, that structure and checking against it each potential CIFF conjunct to be propagated to, implies a huge overhead in terms of efficiency.

Our implemented solution avoids such overhead, paying something in terms of completeness.
The two main ideas are that (1) in most practical cases, **Propagation** can be applied only to abducible CIFF conjuncts and (2) if **Propagation** is applied only to abducible CIFF conjuncts, the machinery introduced for **Factoring** can be reused for **Propagation**.

Due to the presence of the **Unfolding in implications** rule, if we do not apply **Propagation** against defined atoms, we can only loose *failure* branches, but not successful abductive answers. We do not prove this but we show several examples supporting that evidence.

Example 5.5. *Let $\langle P, A, IC\rangle_{\Re}$ be the following abductive framework with constraints:*

$$\begin{array}{lll} P: & p \leftarrow p & \\ A: & \{a\} & \\ IC: & [IC1] & p \rightarrow a \end{array}$$

*Let us consider the goal p. In this case, either applying or not applying **Propagation** to $[IC1]$ and p does not change the CIFF computed abductive answers. If we do not apply **Propagation**, CIFF loops forever due to the clause $p \leftarrow p$. Propagating p to $[IC1]$ will lead to the abduction of a. However CIFF still loops forever, thus computing no abductive answer.* □

Example 5.6. *Let $\langle P, A, IC\rangle_{\Re}$ be the following abductive framework with constraints:*

$$\begin{array}{lll} P: & p \leftarrow b & \\ A: & \{a, b\} & \\ IC: & [IC2] & p \rightarrow a \end{array}$$

*Let us consider the goal p. As in the example 5.5, either applying or not applying **Propagation** to $[IC1]$ and p does not change the CIFF computed abductive answers. The following is a CIFF computation in the first case:*

$$\begin{array}{ll} F_0: & p,\ [p \rightarrow a] \\ F_1: & p,\ [p \rightarrow a],\ [\top \rightarrow a] \\ F_2: & p,\ [p \rightarrow a],\ a \\ F_3: & b,\ [p \rightarrow a],\ a \\ F_4: & b,\ [b \rightarrow a],\ a \\ F_5: & b,\ [b \rightarrow a],\ a,\ [\top \rightarrow a] \\ F_6: & b,\ [b \rightarrow a],\ a,\ a \\ F_7: & b,\ [b \rightarrow a],\ a \end{array}$$

The abductive answer that we can extract from F_7 is $\langle \{a, b\}, \oslash\rangle$.

*The following is a CIFF computation if we do not apply **Propagation** against defined atoms as CIFF conjuncts.*

$$\begin{array}{ll} F_0: & p,\ [p \to a] \\ F_1: & b,\ [p \to a] \\ F_2: & b,\ [b \to a] \\ F_3: & b,\ [b \to a],\ [\top \to a] \\ F_4: & b,\ [b \to a],\ a \end{array}$$

The abductive answer that we can extract from F_4 *is again* $\langle\{a, b\}, \oslash\rangle$*. As we can see, not only is the answer the same as in the first case, but the latter computation is less expensive than the previous one, in terms of number of steps.* □

It is quite clear that, as exemplified above, the presence of the **Unfolding in implications** rule makes **Propagation** against defined atoms redundant in most cases. The only cases in which this is not true are shown by the following example.

Example 5.7. *Let* $\langle P, A, IC\rangle_{\Re}$ *be the following abductive framework with constraints:*

$$\begin{array}{lll} P: & p \leftarrow p & \\ A: & \oslash & \\ IC: & [IC2] & p \to false \end{array}$$

Let us consider the goal p*. In this case, if we apply* ***Propagation*** *to* $[IC1]$ *and* p*, CIFF fails immediately. If we do not apply* ***Propagation****, instead, CIFF loops forever due to the clause* $p \leftarrow p$*.* □

If we do not apply **Propagation** against defined atoms, we loose the above type of failures. Nevertheless the system benefits from huge efficiency enhancements because as we will see we can avoid the maintenance of a **Propagation** data structure for each implication in a node and this is a great value for most practical cases. Moreover we argue that most of the "lost" failure answers derive from "ill-defined" abductive logic programs (as in the case of the above example), which should be avoided.

Propagating only abducible CIFF conjuncts, allow us to build an efficient loop checking algorithm similar to the one for **Factoring**.
Each implication in the `Implications` list of the current `State` is of the form:

```
Implication:Id:PropCounter
```

where `Id` is a unique system-generated integer while `PropCounter` indicates the *least* item `Abd` in the `Abducibles` list such that `Abd` has not been propagated to `Implication` yet.
The added information is used in the same way as for **Factoring** but some notes are needed.

Let `BAbds` be the list of abducible atoms in the body of an implication. Only the first element of `BAbds` is considered for **Propagation**. This is an important computational enhancement and it is a safe optimization in the sense that no potential abductive answers may be lost. Consider, e.g., an implication of the form:

$$a1, a2 \to head$$

where $a1$ and $a2$ are abducibles. In order to empty the body, thus firing *head* as a CIFF conjunct in the node, both abducibles have to be eliminated and this can be done only by propagating them. It is obvious that not only is the order in which they are propagated meaningless in this respect, but if **Propagation** is applied to all the abducibles, it results in more CIFF steps and thus higher computational costs. In the above example, if we apply **Propagation** exhaustively we would obtain the following set of implications:

$$
\begin{aligned}
&a1, a2 \rightarrow head \\
&a1 \rightarrow head \\
&a2 \rightarrow head \\
&\top \rightarrow head \\
&\top \rightarrow head
\end{aligned}
$$

thus, potentially, *head* is fired twice in a CIFF branch. If instead we apply **Propagation** only to the first abducible we obtain:

$$
\begin{aligned}
&a1, a2 \rightarrow head \\
&a2 \rightarrow head \\
&\top \rightarrow head
\end{aligned}
$$

In this way, the computation is forced to produce a minimal set of implications firing *head* only once.

The `PropNumber` element is initialized to `1` for each `Implication` and empty `BAbds` are skipped by the loop checking algorithm, thus leaving `PropNumber` as `1`. Each time a **Propagation** is performed the newly generated implication has again `PropNumber` set to `1` because the first abducible is removed by **Propagation** itself and the new first element of `BAbds` can be potentially propagated to the whole set of abducible CIFF conjuncts. When **Unfolding in implication** is performed, instead, the newly generated implications have their `PropNumber'` element set to the value of `PropNumber` because the new abducibles in the body (if any) are put at the tail of `BAbds`, thus leaving unchanged the first abducible atom.

The last note concerns the behavior of the system against two identical abducible CIFF conjuncts in a node. Suppose we have an implication of the form $a \rightarrow h$ and two abducible atoms a and a in the current node. The loop checking algorithm for **Propagation** does not detect that the two abducibles are identical, thus opening the door for potential loops. To avoid this, there are two solutions. The first one is to maintain a **Propagation** data structure for each implication recording the abducibles which have been propagated to that implication. But we want to avoid it. The second one is very simple: give a higher priority to the **Factoring** rule than to the **Propagation** rule. Due to the behavior of the **Factoring**, after that **Factoring** has been applied exhaustively to a node, it is ensured that no abducible CIFF conjunct can occur twice. Then, simply moving up the `sat` clause for **Factoring** all potential loops for **Propagation** are avoided.

5.1.5 The CIFF$^\neg$ proof procedure

Until now, we have not taken into account the NAF extension apart from a short overview in Section 5.1. The implementation of the NAF extension, which can be activated through a CIFF flag, has been done trying to be as modular as possible. Nevertheless, it requires additional routines and data structures both in the preprocessing phase and in the abductive phase. We "passed over" them in the previous sections to clarify the presentation of the system as the extra machinery does not affect the guidelines of what was discussed before.

Implementing the CIFF$^\neg$ proof procedure requires handling the following three main issues:

- marked integrity constraints in the preprocessing phase
- new CIFF$^\neg$ proof rules
- labeling of a CIFF$^\neg$ node after the application of **NAF Switch**

The addition of new CIFF$^\neg$ proof rules is reflected in adding new `sat` clauses: we do not focus on this point as their implementation is the same as any other CIFF proof rule.

More interesting is the preprocessing phase where some changes in the representation of integrity constraints are needed, which in turn change the representation of the implications in the current

`State`. To address the CIFF$^\neg$ specifications, each integrity constraint (and thus each implication) needs a further `Mark` field indicating if it is *marked* (`Mark = 1`) or it is *unmarked* (`Mark = 0`). Moreover, **Negation rewriting** is never applied to a marked implication, hence negative literals in their bodies are never moved into the head, requiring a further field `Negs` for their storage. The final representation of an implication in the CIFF System is the following:

```
body(BCons,BEqs,BAts,BAbds,BNegs) implies
      head(HCons,HEqs,HDiseqs,HAts,HAbds,Mark).
```

Note that if the NAF extension is not activated, the `Mark` field is set to `0` and `BNegs` is empty for each implication throughout the computation.

Whenever no other proof rules can be applied to a node the **NAF Switch** rule is applied and the node is marked with the current set of `Abducibles` in the state. Roughly speaking the current `Abducibles` have to be "frozen" in order to be retrieved by the next CIFF$^\neg$ step, in particular in the application of **NAF factoring** rules.
Rather than adding a new fixed component in the current `State`, the **NAF Switch** rule starts a new preprocessing phase, building an iff-definition for each abducible predicate considering each abducible CIFF conjunct as a clause for the corresponding predicate. For example, if $a(1), a(2)$ are all the abducible CIFF conjunct for $a/1$, then the iff-definition for a would be:

$$a(X) \leftrightarrow X = 1 \vee X = 2$$

.
Instead, if no atom has been abduced so far for $a/1$, then we would obtain:

$$a(X) \leftrightarrow \bot$$

.
In order to distinguish between the original iff-definitions and these ones, the new iff-definitions are asserted through the predicate `naf_def/3` as follows:

```
naf_def(+PredName, +Arguments, +Disjuncts).
```

Whenever the system starts the **NAF Switch** rule, all the previous `naf_def` assertions (if any) are retracted.
Asserting new iff-definitions for abducibles, however, requires a big care due to the presence of existential variables. Consider again the abducible predicate $a/1$ and suppose that $a(1), a(Y)$ (with Y existentially quantified) have been abduced so far. The `naf_def` atom to be asserted would be:

```
naf_def(a, [X], [[1],[Y]]).
```

The problem is that when the `naf_def` atom is asserted, the binding to `Y` is lost, but we need exactly that variable `Y` because it can be shared to other components of the current CIFF node.
The solution adopted is to *skolemise* the existential variables occurring in the abduced atoms and then to assert the skolemised abducibles maintaining in the current `State` the *skolem bindings*, i.e. the bindings between existential variables and skolem constants for performing a *deskolemisation* when needed. In the above example, the system generates a skolem constant `sk_1` for `Y` and the `naf_def` atom becomes:

```
naf_def(a, [X], [[1],[sk_1]]).
```

As `sk_1` is a ground term, nothing is lost. To restore the right existential variable when needed, the association `Y = sk_1` is maintained as an argument of the current `State`.

Recall that, as said in Section 5.1, the current `State` is of the form:

```
state(Diseqs,Constraints,Implications,DefinedAtoms,Abducibles,Disjunctions):
 naf_state(DeltaNAF,NAFSwitch)
```

and that the `naf_state(DeltaNAF,NAFSwitch)` component is initialized (also if the NAF extension is activated) to:

```
 naf_state([],0).
```

Summarizing, the application of the **NAF Switch** rule changes `naf_state(DeltaNAF,NAFSwitch)` by populating `DeltaNAF` with all the *skolem bindings* and by setting `NAFSwitch` to `1`.
In this way in the next CIFF$^{\neg}$ steps, the system is aware of the fact that **NAF Switch** has been applied and all the information about the frozen abducibles can be easily retrieved.
Indeed the cost of retrieving a `naf_def` atom (plus, eventually, deskolemising non-ground terms) is much cheaper than maintaining a flat data-structure containing all the frozen abducibles throughout the computation and then traversing it to check whether it contains a certain abducible atom or not.
In particular, maintaining in the global Prolog state the set of frozen abducibles simplifies the application of the **NAF factoring** rules which are implemented in a similar way to **Unfolding atoms**. We explain this by a simple example. Suppose we have the set of frozen abducibles $a(1), a(Y)$ for the predicate $a/1$. In this case, the NAF Switch rule asserts the following `naf_def` atom:

```
naf_def(a, [X], [[1],[sk_1]])
```

and the skolem binding `Y = sk_1` is maintained in the `State`. Suppose now that the new abducible $a(X)$ occurs in the current node with X existentially quantified. The **NAF Factoring** rules would produce a disjunction of the form

$$X = 1 \vee X = Y$$

But the same disjunction is produced by unfolding $a(X)$ with the new iff-definition plus a deskolemisation step through the equality `Y = sk_1`. This is exactly what the *NAF factoring* rules do.

5.1.6 Ground Integrity Constraints

Much work has been done in order to reach an acceptable level of efficiency. However, the main source of inefficiency in a CIFF computation is probably represented by the set of implications in a CIFF node. As each implication has to be checked against each instance of each universally quantified variable occurring in it, increasing the number of implications in a node pulls down performance dramatically.

Example 5.8. *Let us consider the following abductive framework $\langle P, A, IC\rangle$ which simply represents a graph specification:*

$$\begin{array}{ll} P: & vertex(v1) \\ & vertex(v2) \\ & vertex(v3) \\ & edge(v1,v2) \\ & edge(v2,v3) \\ A: & \oslash \\ IC: & edge(X,Y) \rightarrow vertex(X) \\ & edge(X,Y) \rightarrow vertex(Y) \end{array}$$

*It is straightforward to see that the instances of both integrity constraints will be ground. More precisely, through the exhaustive application of **Unfolding in implication** rule, we will obtain the following set of implications:*

$$(X = v1, Y = v2) \rightarrow vertex(X)$$
$$(X = v2, Y = v3) \rightarrow vertex(X)$$
$$(X = v1, Y = v2) \rightarrow vertex(Y)$$
$$(X = v2, Y = v3) \rightarrow vertex(Y).$$

Then the application of other proof rules managing equalities will eliminate variables making the next implications totally ground:

$$\top \rightarrow vertex(v1)$$
$$\top \rightarrow vertex(v2)$$
$$\top \rightarrow vertex(v2)$$
$$\top \rightarrow vertex(v3).$$

In this case there are only four implications, but what if the graph was composed of hundreds of edges? We would get thousands of implications in a node, heavily pulling down performance. Even worse, the above example is really simple because the implications will be eliminated from a CIFF node. But what if an abducible occurred in the body of the original integrity constraint? Potentially, the derived set of instantiated implications is maintained all along a CIFF branch due to the presence of the abducible. □

To (partially) deal with this problem, we introduce a dedicated algorithm for managing efficiently some classes of integrity constraints (and, in turn, the instantiated implications).
The idea is that in the above example and in many other cases the integrity constraints are such that:

- the defined atoms in their bodies are not recursive, i.e. their iff-definitions do not generate a loop and
- the implications (or at least the most part of them) deriving from the integrity constraints via unfolding will be ground.

The first observation allows for thinking about a pre-compilation of the bodies of the integrity constraints because all the possible unfolding operations are finite and they could be determined statically. The second observation, i.e. that the implications become ground implications allows for thinking about an algorithm which asserts and checks them in the Prolog global state instead of maintaining them in a CIFF node. If a ground body is satisfied, the relative ground instance of the head will be put into the CIFF node as a CIFF conjunct.

We define formally the class of predicates which can occur in the body of an integrity constraint to be a ground one, and then we sketch the algorithm on simple examples.
We start defining the class of *ground extensional* predicates.

Definition 5.1. *Let $\langle P, A, IC\rangle_{\Re}$ be an abductive logic program with constraints. We say that a non-equality and non-constraint predicate p with a certain arity n in the language of $\langle P, A, IC\rangle_{\Re}$ is a* ground extensional *predicate if and only if it is*

- *a non-abducible predicate and*
- *defined in P by a (possibly empty) set of ground facts.*

□

The second definition concerns predicates whose definitions do not depend on other predicates.

Definition 5.2. *Let $\langle P, A, IC\rangle_{\Re}$ be an abductive logic program with constraints. We say that a non-equality and non-constraint predicate p with a certain arity n in the language of $\langle P, A, IC\rangle_{\Re}$ is a* final safe *predicate if and only if*

- *it is an abducible predicate or*
- *it is a ground extensional predicate.*

□

The next class of predicates is the *1-step safe* class. The main idea is that no variable could cause floundering and that a predicate is neither a recursive predicate nor depends on another recursive predicate.

Definition 5.3. *Let $\langle P, A, IC\rangle_{\Re}$ be an abductive logic program with constraints and let GD_P be the dependency graph of P. We say that a non-equality and non-constraint predicate p with a certain arity n in the language of $\langle P, A, IC\rangle_{\Re}$ is a* 1-step safe *predicate if and only if*

- *it is a final safe predicate or*
- *each clause $C = p(t_1, \ldots, t_n) \leftarrow B$ in P is such that*
    - *for each defined predicate q in B, q does not belong to a loop in GD_P, and*
    - *for each variable X occurring in C,*
        * *X occurs also in an equality atom $X = c$ with c ground or*
        * *X occurs also in a non-equality, non-constraint atom in B*

□

We are ready to introduce the class of *safe* predicates.

Definition 5.4. *Let $\langle P, A, IC\rangle_{\Re}$ be an abductive logic program with constraints. We say that a non-equality and non-constraint predicate p with a certain arity n in the language of $\langle P, A, IC\rangle_{\Re}$ is a* safe *predicate if and only if*

- *it is a* 1-step safe *predicate and*
- *each clause $C = p(t_1, \ldots, t_n) \leftarrow B$ in P is such that each defined predicate q in B is a* 1-step safe *predicate.*

We say that an atom is a safe *atom if its predicate is a safe predicate.* □

Finally, we define the syntactical conditions for which an integrity constraint is a ground one.

Definition 5.5. *Let $\langle P, A, IC\rangle_{\Re}$ be an abductive logic program with constraints. An integrity constraint I in IC is a* ground integrity constraints *if and only if*

- *each atom occurring in its body is a* safe *atom and*
- *each variable X occurring in I occurs also in a* safe *atom in its body.*

□

The above syntactical conditions are automatically checked by the system which decides whether an integrity constraint is a *ground* one or not.

Now we briefly sketch the algorithm itself. The basic idea of the algorithm is to assert incrementally in the Prolog global state all the partial instances of the implications derived from ground integrity constraints.
At preprocessing-time, all the partial instances which can be built through the grounding of the ground extensional atoms are asserted. If for some such instances the body is satisfied, the relative head is added to the query.
In Example 5.8, the two ground integrity constraints can be instantiated at preprocessing time through the grounding of $edge(X, Y)$, adding to the query the set

$$vertex(v1), vertex(v2), vertex(v3).$$

However, the most interesting part of the algorithm is in the presence of an abducible atom in the body of a ground integrity constraint. The idea is to check, after each application of a CIFF proof rule, if a new ground abducible *Abd* has been added to the current CIFF node. If it is the case, *Abd* is matched against all the partial instances asserted so far in the current CIFF branch, and then all the new (partial) instances obtained through *Abd* are asserted.

Example 5.9. *Let us consider the following ground integrity constraint:*

```
[ext(X),abd_1(X,Y),abd_2(Y)] implies [false]
```

where `abd_1/2`*,* `abd_2/1` *are abducibles and* `ext/1` *is defined as follows:*

```
ext(1)
ext(2).
```

At preprocessing time the system asserts the following partial instances of the ground integrity constraint:

```
[abd_1(1,Y),abd_2(Y)] implies [false]
[abd_1(2,Y),abd_2(Y)] implies [false]
```

taking into account the iff-definition of `ext/1`*. If during the computation,* `abd_1(1,4)` *is abduced then it is matched with all the instances obtaining the new set of instances:*

```
[abd_1(1,Y),abd_2(Y)] implies [false]
[abd_1(2,Y),abd_2(Y)] implies [false]
[abd_2(4)] implies [false]
```

Now if `abd_2(3)` *is abduced, the new set of instances becomes:*

```
[abd_1(1,Y),abd_2(Y)] implies [false]
[abd_1(2,Y),abd_2(Y)] implies [false]
[abd_2(4)] implies [false]
[abd_1(1,3)] implies [false]
[abd_1(2,3)] implies [false]
```

Finally if `abd_2(4)` *is abduced, then we obtain*

```
[abd_1(1,Y),abd_2(Y)] implies [false]
[abd_1(2,Y),abd_2(Y)] implies [false]
[abd_2(4)] implies [false]
[abd_1(1,3)] implies [false]
[abd_1(2,3)] implies [false]
[abd_1(1,4)] implies [false]
[abd_1(2,4)] implies [false]
[] implies [false]
```

At this stage, `false` *is added to the current CIFF node, thus failing this branch.* □

The only problem is that, while for the extensional ground atoms we are sure that all the instances will be ground (due to their iff-definition), we cannot say the same for abducibles.
To address this point, the solution we adopted is the following: when no other CIFF proof rules can be applied to the node, if there are some non-ground abducibles as CIFF conjuncts, they are matched against all the partial instances asserted in the branch. The instances so obtained are inserted in the CIFF node and then handled by ordinary CIFF proof rules.

Example 5.10. *Consider again Example 5.9 and let us assume that no CIFF proof rule can be applied to a node, an abducible* `abd_2(Z)` *(*`Z` *existentially quantified) is in the node and the partial instances asserted are:*

```
[abd_1(1,Y),abd_2(Y)] implies [false]
[abd_1(2,Y),abd_2(Y)] implies [false]
[abd_2(4)] implies [false]
```

In this case, the following implications will be inserted in the CIFF node:

```
[abd_1(1,Y),Z=Y] implies [false]
[abd_1(2,Y),Z=Y] implies [false]
[Z=4] implies [false]
```

Roughly speaking, to obtain the instances to be added to the node, we apply exhaustively a propagation *step.* □

In order to check efficiently at every CIFF proof rule application if an abducible has been grounded, a further element `GroundAbds` is added to the `state` term of the current `State`, containing the current set of non-ground abducibles in the node. If an abducible becomes ground it is matched to the asserted partial instances of the ground integrity constraints and then it is removed from `GroundAbds`. When no other CIFF proof rules can be applied, the elements in `GroundAbds` are the remaining non-ground abducibles.

Handling the ground integrity constraints in the Prolog global state allows for a huge boost in performance. The decision whether to use this algorithm or not is left to the user through a CIFF flag, but however, the only case in which it should not be used is for debugging. I.e. when the user want to keep track step by step of the CIFF proof rules over the implications.

5.2 Related work and comparison

As seen in Chapter 3, there is a huge literature about abductive logic programming (with constraints), see for example [99, 97, 96, 102, 100, 109, 143, 75, 60, 61, 139, 111, 82, 157, 9, 121, 46, 33], and an exhaustive comparison of CIFF with the other procedures/systems would not be feasible. So here we focus on the procedure/system which is the closest to CIFF: the $\mathcal{A}$-System. At first glance this choice could be seen as very restrictive, but the $\mathcal{A}$-System, as remarked in [139], is a combination of three older abductive proof procedures, namely the IFF proof procedure [82], the ACLP proof procedure [109] and, most importantly, the SLDNFA proof procedure [61] of which the $\mathcal{A}$-System is a direct descendant. The $\mathcal{A}$-System could be considered the state-of-the-art of abductive logic programming with constraints, borrowing the most interesting features from the above cited proof procedures. Moreover it is one of the few proof procedures which have been effectively implemented, and, probably, it is the only one which puts efficiency as one of its main features (the other notable exception is ACLP, but it was outperformed by the $\mathcal{A}$-System). Hence, we argue that a comparison with the $\mathcal{A}$-System implicitly covers the abductive logic programming field. This comparison is shown in Section 5.2.1.
However, the $\mathcal{A}$-System (like CIFF) relies upon the three-valued completion semantics. So all the literature referring to the generalized stable models semantics [102, 88], including for example [100, 109, 121], also needs to be addressed. To cater for this, we chose to compare CIFF with Answer Sets Programming (ASP) and, in particular, with the two dominant answer sets solvers: DLV [68] and SMODELS [136]. Doing this we believe will also cover, as a side effect, the abductive logic programming literature relying upon stable models, with which answer sets programming shares many common points. This comparison is shown in Section 5.2.2.

In Section 5.2.3, the experimental results for some concrete examples are shown. To evidence the fact that the CIFF System is a concrete tool for declarative problem solving, an interesting

comparison has been made in [44], where several systems are tested on various problems: overall, the results of the CIFF System are very good computational results proving the robustness and the versatility of the system.

5.2.1 Comparison with $\mathcal{A}$-System

The $\mathcal{A}$-System [111, 139] is a proof procedure for abductive logic programming with constraints, and it is a direct descendant of the SLDNFA proof procedure [61]. The $\mathcal{A}$-System integrates the main features of other approaches: the constraint solver as in the ACLP proof procedure [109], the elegant presentation through rewrite rules (proof rules) of the IFF proof procedure [82], and the theoretical results of both SLDNFA and IFF proof procedures (indeed the theoretical results of IFF and SLDNFA are substantially the same, as noted e.g. in [81]).

The $\mathcal{A}$-System and CIFF share many common points. They both rely upon the three-valued completion semantics and their computational schemas are both based on rewrite (proof) rules. Moreover, both systems are implemented under SICStus Prolog, the syntax of the input programs is almost the same and, overall, much effort has been done in both systems, even if using completely different solutions, for obtaining considerable efficiency, exploiting the data structures and the services available in a modern Prolog platform such as SICStus.
However there are also some important differences.

Expressiveness of Integrity Constraints - The $\mathcal{A}$-System framework allows only integrity constraints in denial form. Despite the fact that, using classical negation, it holds that

$$(B \rightarrow H) \equiv ((B, \neg H) \rightarrow \bot),$$

we argue that the CIFF (left-hand side) representation is more expressive. Assume that $a1$ and $a2$ are two abducible predicates and consider both the CIFF implication and the corresponding $\mathcal{A}$-System denial:

$$a1, a2 \rightarrow H$$

$$a1, a2, \neg H \rightarrow \bot.$$

CIFF tries to prove H only when both $a1$ and $a2$ are satisfied while the $\mathcal{A}$-System treats "at the same level" the model in which H holds and thus the denial is satisfied. In practice the $\mathcal{A}$-System considers two alternatives at the same time: or an atom ($a1$ or $a2$ in the example) fails or a negative literal ($\neg H$ in the example) fails. Thus the $\mathcal{A}$-System gives no priority to one solution with respect to another [7].

Negation in implications/denials - The presence of a negative literal (*not* A) in the body of an implication is handled by CIFF through a **Negation rewriting** rule (we do not consider the NAF extension here) which moves A to the head of the implication. The $\mathcal{A}$-System, instead, manages such negations with a rule similar to a **Case Analysis** rule. That is, it creates a disjunction with a disjunct containing A and the other disjunct containing (*not* A) in conjunction with the rest of the original implication. This is exactly what CIFF does in the **Case Analysis for equalities (R12)** and **Case Analysis for constraints (R6)** rules. However, as noted also in [81], applying a **Case Analysis** rule to a defined/abducible atom A seems not to be completely *sound* with respect to three-valued completion semantics because it implies a two-valued truth assignment for A. Note that the two **Case Analysis** rules for CIFF, instead, are sound because they involve equalities and constraints whose semantics is not a three-valued semantics.
But, even if we assume that the $\mathcal{A}$-System behavior is sound with respect to three-valued completion semantics, it results in some important differences in the returned abductive answers. Let us consider the following abductive logic program with constraints $\langle P,\ A,\ IC\rangle_{\Re}$:

[7] This difference is accentuated even more if we take into account the NAF extension which is based on the same philosophy.

$$\begin{array}{ll} P: & \oslash \\ A: & \{a, b\} \\ IC: & a, b \rightarrow \bot \\ & not\ b, a \rightarrow \bot \\ & not\ a, b \rightarrow \bot \end{array}$$

Let Q be the empty query. The $\mathcal{A}$-System ends the computation returning no answers whereas the CIFF System returns, correctly, the empty abductive answer $\langle \oslash, \langle \oslash, \oslash, \oslash \rangle \rangle$. If we drop the latter integrity constraint, the $\mathcal{A}$-System returns only one abductive answer, corresponding to the CIFF answer $\langle \{b\}, \langle \oslash, \oslash, \oslash \rangle \rangle$ whereas the CIFF System returns, correctly, the empty answer. Note that in the first case we have a completeness problem for the $\mathcal{A}$-System and, in the second case, there is also a problem of minimality of the answer.

Managing Herbrand equalities/inequalities - Both CIFF and the $\mathcal{A}$-System have rewrite (proof) rules managing Herbrand equalities and inequalities. However, the $\mathcal{A}$-System implements those rules as another black-box constraint solver on that domain. The result is that, in this respect, the $\mathcal{A}$-System implementation is more modular and a bit more efficient than the CIFF implementation. Nevertheless, it also seems that the $\mathcal{A}$-System returns less interesting answers concerning equalities and inequalities. Consider the following "lamp" example, borrowed from [82] (a simplified version has been shown in Example 3.8). In this setting, we have to do an "abductive diagnosis" for the observation (the query Q) that there is a $faulty_lamp$. We recall here its specification:

$$\begin{array}{lll} P: & faulty_lamp & \leftarrow power_failure(X), not\ backup(X) \\ & faulty_lamp & \leftarrow lamp(X), broken(X) \\ & backup(X) & \leftarrow battery(X, Y), not\ empty(Y) \\ & lamp(l) & \\ & battery(b, c) & \\ A: & \{power_failure, broken, empty\} & \\ IC: & \oslash & \end{array}$$

Both systems return three abductive answers to the query Q, representing all the possible abductive answers. In particular the two answers

$$Ans_1 = \langle \{broken(l)\}, \langle \oslash, \oslash, \oslash \rangle \rangle$$

$$Ans_2 = \langle \{power_failure(b), empty(c)\}, \langle \oslash, \oslash, \oslash \rangle \rangle$$

are returned by both systems. The third answer, instead, requires some attention because while CIFF (and the CIFF System) returns the following answer $\langle \Delta, C \rangle$ (with a single CIFF inequality in C):

$$Ans_3^{CIFF} = \langle \{power_failure(X)\}, \langle \oslash, \oslash, \{X = b \rightarrow \bot\} \rangle \rangle,$$

the $\mathcal{A}$-System returns:

$$Ans_3^{\mathcal{A}-System} = \langle \{power_failure(X)\}, \langle \oslash, \oslash, \oslash \rangle \rangle.$$

As we can see, in the $\mathcal{A}$-System answer, the binding between X and b is lost. This limitation of the $\mathcal{A}$-System can be found in many examples. We do not know whether this is a problem of the specification or of the implementation, also because we were not able to trace an $\mathcal{A}$-System computation step by step.

Finally, the above differences do not take into account the NAF extension which is totally absent in the $\mathcal{A}$-System but which grants a higher level of expressiveness for CIFF as remarked in Section 4.3. The Hamiltonian example that we will see in Section 5.2.3 is such an example of the added expressiveness.

5.2.2 Comparison with Answer Sets Programming

Answer Sets Programming (ASP) (see, e.g. [128, 24, 25]) is a very suitable framework, like Abductive Logic Programming with Constraints (ALPC), for representing knowledge and reasoning about it. The two frameworks are strongly interconnected. This interconnection arises at first glance, just by noticing that ASP is based on the Answer Sets Semantics [90], an evolution of the stable models semantics [88] (which in turn is used as the core semantics for many abductive proof procedures, e.g. [109, 100, 121]) and that abduction can be modeled in ASP, as shown e.g. in [28].

Nevertheless, ASP and ALPC also show important differences which we briefly discuss here. Then, we conclude this section trying to find the role of CIFF in this respect. We assume the reader has some familiarity with ASP.

The ASP framework is based upon some concrete assumptions. In particular ASP relies upon *function-free* programs and the *finiteness* of the grounding of a program. These assumptions have a high impact on the computational model and, hence, on the implemented answer sets solvers.
Despite the fact that, at a theoretical level, the semantics of ASP and ALPC are close, the computational model of ASP, relying upon programs with a finite grounding, shares many common points with typical constraint solving algorithms and it is very distinct from the classic computational model of logic programming (mostly used in ALPC and also in CIFF). For an excellent comparison of the two computational models, see [128].
Directly from the above observations, the implemented answer sets solvers benefit from a number of features which have made them popular tools for knowledge representation and reasoning: *completeness*, *termination* and *efficiency*.
Completeness and termination follows directly from the assumption that the Herbrand universe of a program is finite.
The idea of applying constraint solving techniques in the computational model, together with hardware improvements, also makes it possible to have efficient answer sets solvers, and, indeed, state-of-the-art solvers are able to handle hundreds of thousands of ground Herbrand terms in acceptable times. This is enough for many medium to large size applications.

However, the ASP assumptions also introduce some important limitations on the expressiveness of the framework. Even if many application domains can be modeled through ASP, there are some applications which need the possibility of introducing non-ground terms. The web sites repairing example described in Chapter 6 is one such application. Moreover, there are applications which can be effectively modeled in ASP, but for which non-ground answers could be more suitable. Consider, for example, a planning application where we search for a plan to solve a goal G by time $T = 5$. Assume that a certain action A solves the goal. In a plan obtained from an answer sets solver the action A will be bound to a ground time, for example 4 or 3. However, it might be preferable to have a more general plan with A associated with a non-ground time TA together with the constraint $TA < 5$. Obviously, this is just a hint of a planning framework which is outside the scope of this thesis. Work focused on these topics include, for example, [124], and part of the SOCS European Project [167].

Summarizing, we argue that there is not a sure winner between ALPC and ASP because both show interesting (and exclusive) features and drawbacks. The ideal would be a framework combining the strong points of both, and we argue that CIFF, and more precisely, CIFF$^\neg$, could be seen as a step in this direction.

In [157], Sadri and Toni argued that there is a particular case in which the IFF proof procedure together with the NAF extension accomplish the answer sets semantics. The precondition for this is an abductive logic program $\langle P, A, IC\rangle$ such that P is empty and all the predicates occurring in the integrity constraints IC are abducibles. We do not enter into the formal details of this property, however we strongly believe that:

- it is easily extendible to CIFF$^\neg$, and
- its conditions on P could be relaxed allowing for a limited form of logic programs.

In this sense, CIFF$^\neg$ could be seen as a concrete step towards the combination of ASP and ALPC because it can be used as a particular answer sets solver with the added benefits of non-ground terms expressiveness and an interface to a finite-domain constraint solver. Here we show a simple example to demonstrate this feature which is currently being investigated.

Let us consider the well-known N-queens domain, where N queens have to be placed on an N*N board in such a way that for no pair of queens Q_i and Q_j, Q_i and Q_j are in the same row or in the same column or in the same diagonal.
We represent the problem in CIFF as follows (N is a placeholder for a natural number).

$$\begin{array}{ll} P: & exists_q(R) \leftarrow q_domain(R) \wedge q_domain(C) \wedge q_pos(R,C) \\ & q_domain(R) \leftarrow R \geq 1 \wedge R \leq N \\ & safe(R1,C1,R2,C2) \leftarrow C1 \neq C2 \wedge \ (R1+C1 \neq R2+C2) \wedge \\ & \qquad\qquad\qquad\qquad\qquad (C1-R1 \neq C2-R2) \\ A: & \{q_pos\} \\ IC: & q_pos(R1,C1) \wedge q_pos(R2,C2) \wedge R1 \neq R2 \rightarrow safe(R1,C1,R2,C2) \\ Q: & exists_q(1) \wedge \ldots \wedge exists_q(N) \end{array}$$

The CIFF specification of the problem is very compact. A CIFF computation for the query Q proceeds as follows (we abstract away from the concrete CIFF selection function). Each $exists_q(R)$ atom in the query (where R is one of the N integer values between 1 and N) is unfolded giving rise to three atoms: $q_domain(R)$, $q_domain(C)$ and the abducible $q_pos(R,C)$. The first two atoms are in turn unfolded populating the CIFF node with the finite-domain constraints:

$$R \geq 1, R \leq N, \qquad\qquad C \geq 1, C \leq N$$

which will be evaluated by the constraint solver. Note that the constraints concerning R are obviously *ground*, while the constraints concerning C are not *ground* due to the presence of C.
The third atom $q_pos(R,C)$ is instead an abducible non-ground atom (due to the presence of the constraint variable C).
Assuming that all the unfolding, the equality rewriting and the substitutions have been done, we will obtain a node with the following abducible atoms:

$$q_pos(1,C_1),\ldots,q_pos(N,C_N)$$

Each pair of these has to be propagated to the integrity constraint firing N^2 non-ground instances of the $safe$ atom. The condition $R1 \neq R2$ in the body of the integrity constraint in IC avoids propagating twice the same abducible, i.e. it avoids having an instance like $safe(R_1,C_1,R_1,C_1)$.
At this point the $safe$ atoms are unfolded, resulting in the whole set of non-ground finite-domain constraints needed to ensure correct positioning of the queens. Finally, this set, once the solver checks its satisfiability, is returned as part of the extracted answer. The extracted answer contains all the possible solutions: the corresponding ground answers identifying the concrete positions of the queens can be obtained performing a *labeling* on the constraint variables (the CIFF System automatically performs the final labeling if the user wishes it).

Consider now the following ASP representation[8]:

[8] We choose the DLV representation, borrowed from
`http://www.dbai.tuwien.ac.at/proj/dlv/tutorial/`,
because it is the closest representation to our representation and we can easily highlight the differences. For the same reason we present the DLV specification as a set of ALPC integrity constraints: DLV syntax is a little different. However other representations including negative literals, and thus requiring the CIFF$^\neg$ proof procedure to operate correctly, could be used safely as well. We return to this example in Section 5.2.3 and we also show a CIFF$^\neg$ specification.

$row(1)$
$\ldots$
$row(N)$
$row(R) \rightarrow q_pos(R,1) \vee \ldots \vee q_pos(R,N)$
$q_pos(R1,C) \wedge q_pos(R2,C) \wedge R1 \neq R2 \rightarrow \bot$
$q_pos(R1,C1) \wedge q_pos(R2,C2) \wedge row(R) \wedge R2 = R1 + R \wedge C1 = C2 + R \rightarrow \bot$
$q_pos(R1,C1) \wedge q_pos(R2,C2) \wedge row(R) \wedge R2 = R1 + R \wedge C2 = C1 + R \rightarrow \bot$

In this case all the possible solutions are also returned by the answer sets solvers, even if enumerating them in a ground form.
In this example, it is obvious that CIFF accomplishes the answer sets semantics because the set of returned solutions is the same. Nevertheless, abstracting from syntactical differences there is an important difference between the two specifications. The CIFF specification takes advantage of the constraint solver because it delegates the constraints on the variables inside the clause concerning the $safe$ predicate as informally described above. Conversely, in an ASP computation, the conditions on the queen positions are checked locally, resulting in a huge set of *ground* integrity constraints, each one containing a ground pair of queen positions.
As expected (and as shown in Section 5.2.3 below), delegating the checks to a finite-domain constraint solver results in performance an order of magnitude faster than any answer sets solver.
Much theoretical work is to be done yet, but, if the CIFF$^{\neg}$ proof procedure could be used as an answer sets solver (for abductive frameworks of appropriate form, corresponding to ASP programs) then we argue that many answer sets applications could be reformulated to exploit the finite-domain constraint solving interface and also extended in order to accommodate non-ground terms.

5.2.3 Experimental results

In this section, we show some experimental results obtained running three of the most typical benchmark examples, namely the *N-Queens* problem, the *Hamiltonian cycles* problem and the *graph coloring* problem. We also present a simple instance of a web sites repairing framework which could be used with CIFF.
In this performance comparison we restricted our attention to three systems: the A-System [139] and two state-of-the-art answer set solvers, namely the DLV system [68] and SMODELS [136].
All the tests have been run on a Fedora Core 5 Linux machine equipped with a 2.4 Ghz PENTIUM 4 - 1Gb DDR Ram. The SICStus Prolog version used throughout the tests is the 3.12.2 version. All execution times are expressed in seconds of CPU time ("—" means that the system was still running after 10 minutes). In all examples, unless otherwise specified, the CIFF System query is the empty list `[]` representing *true* and the algorithm ground integrity constraint is activated.

The N-Queens problem

We recall the N-Queens, already seen in Section 5.2.2: N queens have to be placed on an N*N board in such a way that for no pair of queens Q_i and Q_j, Q_i and Q_j are in the same row or in the same column or in the same diagonal.

The CIFF System formalization (**CIFF (1)**) of this problem is very simple (the query is a conjunction of N `exists_q(R)` where each R is a natural number, distinct from each other, in $[1, N]$):

```
%%% CIFF (1)
%%% ABDUCIBLES
abducible(q_pos(_,_)).

%%% CLAUSES
q_domain(R)  :- R #>= 1, R #=< N.
```

```
    %%% N must be an integer in real code!

  exists_q(R) :- q_domain(R),q_pos(R,C),q_domain(C).

  safe(R1,C1,R2,C2) :- C1#\=C2, R1+C1#\=R2+C2, C1-R1#\=C2-R2.

  %%% INTEGRITY CONSTRAINTS
  [q_pos(R1,C1),q_pos(R2,C2),R1#\=R2] implies [safe(R1,C1,R2,C2)].
```

We also show two other CIFF formalizations which are direct translations of the DLV and SMODELS formalizations, respectively. In these formalizations, the checks on the queen position conditions, are made locally in each ground integrity constraint instance and they are not delegated to the constraint solver. In these programs, `abs` is the absolute value function.

The DLV translation (**CIFF (2)**) is very similar to the (**CIFF (1)**) formalization and the query is the same. But in this case the conditions on the queen positions is done locally in the body of the integrity constraints[9].

```
%%% CIFF (2)
%%% DLV translation
%%% ABDUCIBLES
abducible(q_pos(_,_)).

%%% CLAUSES
row(1).
...
row(N).

%%% INTEGRITY CONSTRAINTS
[row(R)] implies [q_pos(R,1), ..., q_pos(R,N)].
  %%% N must be an integer in real code!

[q_pos(R1,C),q_pos(R2,C),R1\==R2] implies [false].

[q_pos(R1,C1),q_pos(R2,C2),R1\==R2,(abs(R1-R2)#=abs(C1-C2))]
  implies [false].
```

The SMODELS translation [10] (**CIFF (3)**) needs the CIFF$^\neg$ proof procedure in order to operate correctly. This example illustrates how the CIFF$^\neg$ proof procedure could be used as an answer sets solver.

```
%%% CIFF (3)
%%% Smodels translation
%%% ABDUCIBLES
abducible(q_pos(_,_)).
abducible(neg_q_pos(_,_)).
abducible(has_q(_)).

%%% CLAUSES
row(1).
```

[9]The concrete CIFF syntax differs a bit from that of the program shown in Section 5.2.2. The conditions which avoid placing two queens in the same diagonal are integrated in a single integrity constraint, taking advantage of the `-` and `abs` functions of the constraint solver: the DLV system does not allow for such functions to be expressed. The straight DLV translation with two integrity constraints runs a bit slower in CIFF, as expected.

[10]The SMODELS formalization is borrowed from
`http://www.public.asu.edu/ cbaral/bahi/code.html`.

```
...        %%% N facts
row(N).    %%% N must be an integer in real code!

col(1).
...        %%% N facts
col(N).    %%% N must be an integer in real code!

%%% INTEGRITY CONSTRAINTS
[C\==C2,q_pos(R,C),q_pos(R,C2)] implies [false].
[R\==R2,q_pos(R,C),q_pos(R2,C)] implies [false].

[row(R),col(C),not(neg_q_pos(R,C))] implies [q_pos(R,C)].
[row(R),col(C),not(q_pos(R,C))] implies [neg_q_pos(R,C)].

[q_pos(R,C)] implies [has_q(R)].
[row(R),not(has_q(R))] implies [false].

[q_pos(R,C),q_pos(R2,C2),R\==R2,C\==C2,(abs(R-R2)#=abs(C-C2))]
      implies [false].

[q_pos(R,C),neg_q_pos(R,C)] implies [false].
```

Arguably, this formalization is the least intuitive. It is based on the fact that each row of the board must have a queen placed on it (represented by the abducible `has_q`) and on the fact that each position `(R, C)` on the board either has a queen placed on it or it does not (represented by the two abducible predicates `q_pos` and `neg_q_pos`). In this case the query is the empty query and the computation starts unfolding the bodies of the integrity constraints.

Finally, we show the results for the first solution found. In the tables, we denote the $\mathcal{A}$-System as **ASYS** and the SMODELS as **SM**.

Table 5.1: `N-Queens` results (first solution)

Queens	CIFF 3.0 (1)	CIFF (1)	CIFF (2)	CIFF (3)	ASYS	SM	DLV
n = 4	0.11	0.01	0.02	0.79	0.01	0.01	0.01
n = 6	1.53	0.01	0.21	43.40	0.01	0.01	0.01
n = 8	14.75	0.03	1.29	527.44	0.03	0.01	0.01
n = 12	75.37	0.05	5.98	—	0.05	0.01	0.01
n = 16	—	0.09	410.33	—	0.07	0.36	0.61
n = 24	—	0.20	—	—	0.17	4.88	5.44
n = 28	—	0.29	—	—	0.27	55.32	35.17
n = 32	—	0.37	—	—	0.32	—	—
n = 64	—	1.62	—	—	1.52	—	—
n = 100	—	4.55	—	—	4.24	—	—

All systems return all the correct solutions, but we do not show the times for all solutions because the number of possible solutions is huge when N grows. In the above table, we also include CIFF System 3.0 to underline the performance improvements of CIFF System 4.0.

Only the CIFF System and the $\mathcal{A}$-System, through the use of the finite domain constraint solver, can solve the problem, in a reasonable time, for a high number of queens. Note also that the CIFF System performance in the other two "answer sets" variants of the specification, i.e. **CIFF (2)** and **CIFF (3)**, are, as expected, worse in comparison with the first one, i.e. **CIFF (1)**. In particular the **CIFF (3)** (corresponding to the SMODELS representation) shows very poor performance:

this is due to the high number of integrity constraints and their negative conditions. However, we argue that, on the whole, the results show that the system is able to handle a reasonable number of ground instances.

The Hamiltonian cycles problem

A Hamiltonian cycle in a graph can be defined as follows: given a graph specification and a starting node, a hamiltonian cycle is a cycle that includes *each node* of the graph *only once*.

The CIFF System encoding makes use of the CIFF$^\neg$ proof procedure in order to avoid loops and to collect all possible answers. The abductive answers are in terms of the `ham_final_cycle` predicate whose argument is a list containing the Hamiltonian cycle. In this formalization we also illustrate the use of a function, in this case for building the lists. The `checked` abducible is used to avoid looping.

```
%%%ABDUCIBLES
abducible(ham_final_cycle(_)).
abducible(checked(_,_)).

%%%CLAUSES
ham_cycle(X) :- ham_cycle(X,X,[X,[]],0).

ham_cycle(X,Y,L,N) :- edge(X,Y),checked(X,N),
                      ham_final_cycle([Y,L]).
ham_cycle(X,Y,L,N) :- edge(X,Z),ham_cycle(Z,Y,[Z,L],M),
                      M#=N+1,Z\==Y,checked(X,N).
is_checked(V2) :- checked(V2,M).

%%%INTEGRITY CONSTRAINTS
[checked(X,N),checked(X,M),M#\=N] implies [false].
[vertex(V2),not(is_checked(V2))] implies [false].
```

The predicates `edge` and `vertex` represent any given graph and they are given as (domain-dependent) additional clauses. The query is `[ham_cycle(V)]` where `V` is any vertex of the graph. Note that, in the second integrity constraint, if we had used `checked(V2,N)` directly instead of `is_checked(V2)`, the CIFF System would have returned an `undefined` answer, because of `N` appearing only in a negative literal (thus violating an allowedness condition).

Table 5.2: `Hamiltonian cycles` results (all solutions)

Nodes	CIFF	CIFF (G)	SM	DLV
4	0.04	0.03	0.03	0.02
20	0.45	0.15	0.16	0.02
40	1.93	0.41	1.53	0.03
80	10.95	1.20	11.41	0.04
120	27.62	2.39	43.43	0.07

In the first column we consider CIFF$^\neg$ without the algorithm for *ground* integrity constraints, while in the second column we use CIFF$^\neg$ with that algorithm. We omitted here a comparison with the A-system as we have been unable to specify the problem in such a way that all answers are returned without looping. The same problem arises with CIFF without the NAF extension.
As we can see from the results, CIFF performance are comparable with answer sets solver performance.

The Graph Coloring problem

The graph coloring problem can be defined as follows: given a connected graph we want to color its nodes in a way that each node does not have the color of any of its neighbors.
The CIFF System formalization is as follows (again, we omit the domain-dependent definitions of any specific graph):

```
%%% ABDUCIBLES
abducible(abd_color(_,_)).

%%% CLAUSES
coloring(X) :- color(C),abd_color(X,C).

%%% INTEGRITY CONSTRAINTS
[vertex(X)] implies [coloring(X)].
[edge(X,Y),abd_color(X,C),abd_color(Y,C)] implies [false].
```

The results are the following, where `Jean` and `Games` are two graph instances (up to a 120-nodes graph)[11]:

Table 5.3: `Graph coloring` results (first solution).

Nodes	CIFF	CIFF (G)	ASYS	SM	DLV
4	0.09	0.01	0.01	0.01	0.01
Jean	—	0.68	0.60	0.19	0.48
Games	—	2.39	3.61	0.28	1.14

As for the N-Queens problem all the systems return all the solutions. Here answer sets solvers have the best performance as the constraint solver is not involved in the computation. However, it is worth noticing that performance of both the $\mathcal{A}$-System and the CIFF System, when the algorithm for *ground* integrity constraints is activated (second column), are encouraging, even if the domain is a typical ASP application.

CIFF Scalability

We present a last set of experiments for testing the robustness and the scalability of the CIFF System. For this kind of experiments we do not take into account other related systems. The idea recurring in these experiments is to use a very simple abductive logic program, in particular the grass example seen in Example 3.1, and to augment it with a certain quantity of "noise", i.e. clauses, abducibles and integrity constraints which are not related to the problem. In each test we do, each integrity constraint is a *ground integrity constraints*, thus we show the results obtained by switching either on or off the ground integrity constraints algorithm for all the tests in the bag.

Note that in our experiments on real benchmark problems in previous sections, we test the scalability of CIFF because there are problem instances of medium/big sizes. However all the abductive logic programs with constraints seen until now, have in common a compact representation of the problem with a very limited number of distinct predicates, clauses and integrity constraints and they "grow" at run-time through the abductive reasoning. In this section we build the abductive logic programs using the opposite philosophy: big programs and low reasoning. It is very unlikely to encounter abductive logic programs of that type in practice, but doing so we stress the CIFF System from another point of view. In all the results tables the number between parenthesis indicates the quantity of the "noise" elements in the program.

[11] They are borrowed from http://mat.gsia.cmu.edu/COLOR/instances.html.

The programs in the first two tests **Test1** and **Test2**, whose results are in tables 5.4 and 5.5, contain a number of distinct abducibles in the query and distinct facts in the program respectively.

Table 5.4: `Scalability` results (**Test1**).

Nodes	CIFF	CIFF (G)
Test1(100)	0.01	0.01
Test1(500)	0.04	0.04
Test1(1000)	0.10	0.12
Test1(2000)	0.38	0.40
Test1(5000)	2.09	2.21

Table 5.5: `Scalability` results (**Test2**).

Nodes	CIFF	CIFF (G)
Test2(100)	0.01	0.03
Test2(500)	0.08	0.12
Test2(1000)	0.28	0.48
Test2(2000)	1.04	1.83
Test2(5000)	6.47	12.07

We obtain the best results by switching off the algorithm of ground integrity constraints because both tests are not focused on integrity constraints, and by switching on the ground integrity constraint algorithm we suffer of an initial overhead for calculating the dependency graph of the program. This overhead is bigger in **Test2** due to the presence of many distinct defined predicates: the algorithm for building the dependency graph takes advantage of the information about abducible predicates in order to avoid many dependency checks.

Test3 (table 5.5) adds to the grass example a number of distinct clauses of the form $p_i \leftarrow q_i$.

Table 5.6: `Scalability` results (**Test3**).

Nodes	CIFF	CIFF (G)
Test3(100)	0.01	0.06
Test3(500)	0.12	1.40
Test3(1000)	0.43	5.25
Test3(2000)	1.68	21.370
Test3(5000)	10.03	169.65

Results of **Test3** show dramatically the overhead introduced for building the dependency graph: in the last benchmark, about 10000 distinct predicates must be checked in order to build the dependency graph. As for previous tests, all the computational time, basically, is spent in the preprocessing phase. With respect to **Test1** and **Test2**, the preprocessing phase is heavier in **Test3** due to the presence of clauses rather than facts, also if the algorithm for ground integrity constraints is switched off.

Test4 and **Test5** (tables 5.7 and 5.8) add to the grass example a number of distinct integrity constraints in denial form. The body of each integrity constraint is composed of a single abducible atom (**Test4**) or a single defined atom p_i with no clause in the program (**Test5**).

Table 5.7: `Scalability` results (**Test4**).

Nodes	**CIFF**	**CIFF (G)**
Test4(100)	0.01	0.01
Test4(500)	0.15	0.06
Test4(1000)	0.50	0.17
Test4(2000)	1.74	0.48
Test4(5000)	10.63	2.32

Table 5.8: `Scalability` results (**Test5**).

Nodes	**CIFF**	**CIFF (G)**
Test5(100)	0.02	0.06
Test5(500)	0.17	0.21
Test5(1000)	0.64	0.78
Test5(2000)	2.21	2.98
Test5(5000)	13.64	15.86

In the case of **Test4** we a have a number of integrity constraints to which no CIFF proof rule can be applied because their body is composed of a single abducible and there are no abduced atoms in the program. Switching off the ground integrity constraints algorithm, the integrity constraints remain in the node extending the computational times. This overhead is again balanced in **Test5** by the construction of the dependency graph because each defined atom p_i has an implicit clause $p_i \leftarrow false$.

The last test is **Test6** (table 5.9) which adds again to the grass example a number of distinct integrity constraints in denial form, but now each body is composed of two distinct abducibles. One abducible of each body is also in the program as an initial observation, thus a **Propagation** proof rule can be applied to each integrity constraint.

Table 5.9: `Scalability` results (**Test6**).

Nodes	**CIFF**	**CIFF (G)**
Test6(100)	0.04	0.06
Test6(500)	0.71	0.29
Test6(1000)	2.65	0.92
Test6(2000)	10.69	3.20
Test6(5000)	70.73	22.22

The application of a **Propagation** rule to each integrity constraints shows the power of the ground integrity constraints algorithm: the initial overhead due to the construction of the dependency graph is largely compensated if the number of integrity constraints grows.

All the scalability tests conducted in this section show that:

- CIFF scales up quite well also with big abductive logic programs with constraints even if the preprocessing phase becomes an important computational bottleneck;
- the ground integrity constraint algorithm has an important overhead for the construction of the dependency graph, but if the integrity constraints are quite involved in the abductive process, that overhead is largely rewarded.

5.3 Conclusions

The CIFF and the CIFF$^\neg$ proof procedures represent, to our knowledge, a step forward at both theoretical and implementative levels in the field of abductive logic programming (with constraints). We have proved that CIFF is sound with respect to the three-valued completion semantics. CIFF is also able to handle variables in a non-straightforward way, and it is equipped with a useful interface to a constraint solver. In addition, its implementation, namely the CIFF System, reaches good levels of efficiency, flexibility and scalability, and is directly comparable to other state-of-the-art tools for knowledge representation and reasoning.
The CIFF$^\neg$ proof procedure, which has been integrated in the CIFF System, takes advantage of its NAF treatment of negation in integrity constraints to further extend the expressiveness of the framework. In this way we can naturally represent application domains which are outside the capabilities of other existing tools. In particular, it seems that, putting some limitations on the input programs, the CIFF$^\neg$ proof procedure is able to compute the answer sets semantics without renouncing either the constraint solver interface or the handling of non-ground terms. As pointed out in Section 5.2.2, much work needs to be done in this respect but this is outside the scope of this thesis. However, the result in [157] for the IFF proof procedure and the running of concrete examples with the CIFF System give evidence for this intuition.

Finally, we briefly discuss ongoing and future work on CIFF.
The interconnection between ASP and ALPC are, obviously, a very interesting line of ongoing and future work as discussed in Section 5.2.2.
At a theoretical level there are many interesting lines of work. A main issue is to show formally a completeness result for the CIFF proof procedure (and, possibly, the CIFF$^\neg$ proof procedure). In principle, it should be possible to extend the results in [81] given for the IFF proof procedure. However, for CIFF there is the added difficulty represented by the **Dynamic Allowedness** rule as discussed in Section 4.2. Concerning the CIFF$^\neg$ proof procedure, it seems more difficult to obtain a completeness result due to the nature of the procedure itself which is aimed to discard sound (though counter-intuitive) abductive answers.
Another interesting line of work at the theoretical level is to further extend the expressiveness of the CIFF proof procedure. A concrete idea is *aggregative integrity constraints*. It seems possible, even if we are at a very early stage in this concern, to allow for integrity constraints containing aggregative functions in their body. Just to show the idea consider an integrity constraint of the form:

$$abd(X), sum(X, Y), Y > 10 \rightarrow H$$

where, intuitively, Y is represents the sum of the X values in all the propagated abducibles *abd*, and *head* is fired if $Y > 10$. We think that this would be a very interesting feature because many application domains require aggregates to be easily modeled.

At the implementation level, a big deficiency in CIFF is the lack of a Graphical User Interface (GUI) which would hugely improve its usability: we hope to add it in the CIFF System 5 release.

Other interesting features which are planned to be added to the CIFF System 5 release, are the following.

- Full compatibility to the SICStus Prolog 4 release (which is claimed to be much faster: a porting of the system will benefit at once of this boost in performance) and to SWI-Prolog [181], an interesting open-source Prolog platform.

- The possibility of invoking Prolog built-in functions and predicates directly. E.g. the use of the Prolog `append` predicate for lists directly in CIFF programs. We think that this would enhance performance and ease-of-programming in CIFF. However, some work has to be done in order to understand how to integrate them safely.

- Further improvements in the management of ground integrity constraints both in the preprocessing phase and in the processing phase.
- Adding the possibility to save a preprocessed abductive logic program in order to avoid further preprocessing phases on the same problem.
- Aggregative integrity constraint (if the theory will be ready).

Chapter 6

Web Site Verification and Repair

The exponential growth of the WWW raises the question of maintaining and repairing automatically web sites, in particular when the designers of these sites require them to exhibit certain properties at both structural and data level. The capability of maintaining and repairing web sites is also important to ensure the success of the Semantic Web [175, 13] vision. As the Semantic Web relies upon the definition and the maintenance of consistent data schemas (XML/XMLSchema [177], RDF/RDFSchema [174, 92, 35], OWL [176, 131] and many other formal languages [13]), tools for reasoning over such schemas (and possibly extending the reasoning to multiple web pages) show great promise.

We strongly believe that declarative languages such as Prolog [54], if they are well integrated with the web, will play a crucial role as the computational paradigms in the Semantic Web vision, as noted, e.g., in [182]. Also abduction, as it is a very suitable form of reasoning for diagnosis and repairing, could play a prominent role in that context, as noted, e.g., in [39].

We argue that the CIFF (and CIFF$^\neg$) proof procedure has very useful features for applications in a web reasoning scenario, namely implicative integrity constraints which could act as condition-action rules, handling of unbound variables which could directly represent missing data, and arithmetical constraint solving capabilities.

In this Chapter, we describe the CIFFWEB (prototype) tool, which, roughly speaking, uses CIFF (and CIFF$^\neg$) as the computational core for verifying and, possibly, repairing web sites against sets of requirements which have to be fulfilled by a web site instance.

We define an expressive characterization of rules for checking web sites' errors by using (a fragment of) the well-known semi-structured data query language Xcerpt [37, 38]. In contrast to the other semi-structured query languages (like XQuery [27], XPath [47]) which all propose a path-oriented approach for querying semi-structured data, Xcerpt is a rule-based language which relies upon a (partial) pattern matching mechanisms allowing complex queries to be easily expressed in a natural and human-tailored syntax. Xcerpt shares many features of logic programming, for example its use of variables as place-holders and unification. However, to the best of our knowledge, it lacks (1) a clear semantics for negation constructs and (2) an implemented tool for running Xcerpt programs/evaluating Xcerpt queries. A by-product of this Chapter is the provision of both (1) and (2) for a fraction of Xcerpt, namely the subset of this language that we adopt for expressing *web checking rules*.
We map formally the chosen fragment of Xcerpt for expressing checking rules into *programs for checking*, i.e. abductive logic programs with constraints that can be fed as input to the general-purpose CIFF proof procedure. By mapping web checking rules onto abductive logic programs with constraints and deploying CIFF for determining fulfillment (or identify violation) of the rules, we inherit the soundness properties of CIFF thus obtaining a sound concrete tool for web checking.
CIFF is a general-purpose logic programming procedure and it is not able to handle directly semi-structured data of the kind Xcerpt does. Thus, we also define, as part of the CIFFWEB tool, a

simple XML/XHTML translation into a representation of the web pages suitable for CIFF, and we rely upon that representation in defining the translation function for the web checking rules. At the end of the translation process the CIFF System can be successfully used to reason upon the (translation of the) web checking rules finding those XML/XHTML instances not fulfilling the rules, and representing errors as abducibles in abductive logic programs.

However, abductive reasoning seems to be very suitable not only for identifying errors in a web site instance but also for *suggesting* possible repairing actions for them. In this respect, abducibles may represent not only an error instance fired by an XML/XHTML instance violating a rule r (for checking) but also possible modifications (repairs) to that XML/XHTML data such that both r is fulfilled and no other rules are violated.

Following these observations, we have identified some types of errors, arising from the violation of web checking rules, which are suitable to be abductively repaired, and we have done a further mapping from web checking rules to another type of abductive logic programs with constraints: *programs for repairing*. Again, through the use of programs for repairing with CIFF, for determining fulfillment of the rules, or suggesting appropriate repairing actions, we inherit the soundness properties of CIFF thus obtaining a sound concrete tool for web repairing.

The Chapter is organized as follows. In the next section, we introduce a motivating example for our CIFFWEB tool which will be a piece of the main example used throughout the chapter. In Sections 6.2 and 6.3 we introduce the Xcerpt fragment used for defining our web checking rules together with examples of rules and a formal grammar. In Sections 6.4 and 6.5 we define the translation process from web checking rules to programs for checking and in Sections 6.6 and 6.7 we outline a brief analysis of the checking framework and we describe a CIFFWEB run for checking. Similarly to what is done for checking, in Section 6.8 we describe the repairing framework together with the methodology for building programs for repairing. In Sections 6.9, 6.9.1 and 6.10 we show two CIFFWEB runs for repairing and we discuss the issue of possible errors introduced by repairing actions and we propose a solution. In Section 6.11 we outline a brief analysis of the repairing framework. Finally in Section 6.12 we conclude the chapter placing our work in the context of the existing literature and outlining the ongoing and the future work.
The full example used throughout the chapter, concerning a possible (and simple) web site instance of a theater company, is described in Section 6.13.

6.1 A Motivating Example

Searching through the web, it is easy to encounter web pages containing errors in their structure and/or their data. The main goal of this section is to categorize some typical errors and define a rule pattern for checking these errors.
We argue that, in most cases, considering an XML/XHTML web site instance, the errors can be divided into two main categories: structural errors and content-related (data) errors. Structural errors are those errors concerning the presence and/or absence of tag elements and relations amongst tag elements in the pages. For example, if a tag $tag1$ is intended to be a child of a tag $tag2$, the occurrence in the web site of a $tag1$ instance outside the scope of a $tag2$ instance is a structural error. Data errors, instead, are about the in-tag data contents of tag elements. For example a $tag3$ could be imposed to hold a number greater than 100.
To better exemplify the types of error we consider, we present here a very simple XML web site instance of a theater company. The site is composed of two pages representing a list of shows produced by the company and the list of directors of the company, given below[1]

```
%%%directorindex.xml                 %%%showindex.xml

<directorlist>                       <showlist>
```

[1]Throughout the chapter, we use the convention that each code-line starting with `%` is a line of comment.

```
    <director>John</director>
    <director>Mary</director>
    <director>Paul</director>
    <director>Mary</director>
</directorlist>
```

```
    <show>
        <showname>Mela</showname>
        <dir_by>John</dir_by>
        <year>2000</year>
    </show>
    <show>
        <showname>Epiloghi</showname>
        <year>2001</year>
    </show>
    <show>
        <showname>Toccata e fuga</showname>
        <dir_by>Paul</dir_by>
        <year>2002</year>
    </show>
</showlist>
```

We could specify a number of rules which any web site instance should fulfill. For example, we could specify that the right *structure* of a `show` tag in the second page must contain both a `showname` tag element and a `dir_by` tag element as its child. In the example, we would have a *structural error*, due to the lack of a `dir_by` tag element in the second show of the list. Moreover we could specify that two `director` tags in the first page must not contain the same data. In this case we have a *data error* due to the double occurrence of the `Mary` data.
Requirements (and thus errors) can involve more than one web page. For example, a possible requirement for the theater company specification may be that

each director must direct at least one show.

The above requirement can lead to content-related errors which involve both pages. There is a simple way to check this requirement: for each `director` tag data in the `directorindex` page, if there does not exist a matching data of a `dir_by` tag in the `showindex` page then a data-content error is detected. This is the case for `Mary` in the example.
In all examples in this section, we have assumed that an error instance is fired by a piece of XML/XHTML data which does not fulfill a certain requirement (or specification). We will first formalize any such requirement as a *web checking rule*. Each such rule is composed of a *condition part* and an *error part*. For each instance of the considered data such that the condition part is matched, an error is detected.

6.2 A Formal Language for Expressing Web Checking Rules

In order to formalize and characterize web site requirements as web checking rules, such as the requirements expressed in natural language in the earlier section, we first need a formal language. Our choice is to use the Xcerpt [37] language: a deductive, rule-based query language for semi-structured data which allows for direct access to XML data in a very natural way.
Our characterization of web checking rules can be accommodated straightforwardly in the Xcerpt language. It is worth noticing that the Xcerpt language is much more expressive than the fragment we adopt here for expressing web checking rules. Here, we give some background notions about the Xcerpt fragment we use (for further information about Xcerpt see [38, 37]).
An Xcerpt program is composed of a `GOAL` part and a `FROM` part. The `FROM` part provides access to the sources (XML files or other sources) via (partial) pattern matching among terms, while the `GOAL` part reassembles the results of the query into new terms. Variables can be used within either parts and act as placeholders (as in logic programming).
As an example, the requirement, in the context of our earlier example pages, that *each director must appear at most once in the director list [Rule1]* can be expressed in Xcerpt as follows:

```
%%%%%%%Rule 1 - Double occurrence of a director in the director list
%%%%%%%rule_director_twice_inlist.xce
```

```
GOAL
        all err [ var Dir1,
                "director twice in the director list" ]
FROM
        in {
         resource {"file:directorindex.xml"},
          directorlist {{
                director {{ var Dir1 }},
                director {{ var Dir1 }}
          }}
        }
END
```

The main Xcerpt statement we use in the `GOAL` part is the `all t` statement (where `t` is a term), indicating that each possible instance of `t` satisfying the `FROM` part gives rise to a new instance of `t` returned by the `GOAL` part. In our methodology for writing web checking rules, `all t` will always be `all err`, where `err` stands for "error".
In the `FROM` part, an access to a `resource` is wrapped within a `in` statement. Multiple accesses must be connected by `and` indicating that all subqueries have to succeed in order to make the whole query succeed. The main Xcerpt query terms `t` which we use in our work are: (1) double curly brackets, i.e. `t{{ }}`, denoting *partial term specification*; the order of the subterms of `t` within the curly brackets is irrelevant; (2) variables, expressed by `var` followed by an identifier (variable name); variable values can be bound to strings and numbers; (3) `where` statements for expressing constraints through standard operators like `=, \=, <, >, <=`; (4) subterms of the form `without t` denoting *subterm negation*.
The following rule *[Rule2]* expressing the requirement *the showlist must contain shows produced since year 2000* illustrates the use of constraints in the `where` statement.

```
%%%%%%Rule 2 - Show produced before year 2000
%%%%%%rule_year_before_2000.xce
GOAL
        all err [ var Year, "show produced before 2000" ]
FROM
        in {
         resource {"file:showindex.xml"},
           year {{
             var Year
           }}
        }
        where ( var Year < 2000)
END
```

The following rule *[Rule3]* expressing the requirement *each director directs at least one show* illustrates the use of *subterm negation*:

```
%%%%%%Rule 3 - Director in the director list without a show associated with
%%%%%%rule_director_without_show.xce
GOAL
        all err [ var DirName,
                "director without a show" ]
FROM
   and (
        in {
         resource {"file:directorindex.xml"},
           director {{ var DirName }}
```

```
        },
        in {
         resource {"file:showindex.xml"},
           showlist {{
             without show {{
                                 dir_by {{ var DirName }}
                          }}
           }}
        }
   )
END
```

The `without` statement is only applicable to subterms `t` that do not occur at *root level* in the underlying web pages (in our example, `showlist` and `directorlist` occur at root level); `without` subterms cannot occur nested and finally all variables that occur within a `without` have to appear elsewhere outside a `without` in the `FROM` part of the rule. We will refer to this condition over variables as to the *web checking rules allowedness*.

6.3 A Xcerpt-like grammar for positive web checking rules

To simplify the presentation, we first focus our attention on *positive* web checking rules, i.e. rules in which negation (the `without` statement in Xcerpt syntax) does not occur.
In defining the syntax, we distinguish between *basic* syntactic categories and *structural* syntactic categories. The basic categories are used to define the basic components of the rules, namely variables, constants (strings and numbers), XML tags and constraints (e.g. $X > Y$). The structural categories, instead, are the main parts of the Xcerpt rules, as we have seen in the previous examples: the query part, the error part, the `in` construct and so on.
For each syntactic category, we avoid stating explicitly the syntactic rules for sequences of elements derived from it. We use instead the following notational convention. Given a syntactic category C, $C^\star$ denotes the syntactic category for sequences of elements derived from C. The metarule for $C^\star$ is the following:

$C^\star$ ::= C, $C^\star \mid \epsilon$ (ϵ denotes the empty string)

The grammar for the basic categories is the following:

Const	::=	*String* | *Number*	*VarOrConst*	::=	*Var* | *Const*
Tag	::=	*String*	*Rel*	::=	$< \mid > \mid \leq \mid \geq \mid = \mid \backslash=$
VarName	::=	*String*	*Constraint*	::=	*Var Rel VarOrConst*
Var	::=	`var` *VarName*			

Notice that the *Tag* and the *VarName* categories generate simply strings. However we keep them distinct in order to improve readability of the syntactic rules for the structural categories.
In the second part of the grammar we define effectively the interesting constructs of the web checking rules. Recall that a web checking rule is composed of two main parts: a query part and an error part. The error part gives the error specification, while the query part is a conjunction of queries represented by a `and` wrapping a list of `in` constructs. Each `in` construct expresses a query involving a specific resource. At the end of the whole conjunction a set of constraints can be expressed by the `where` construct.

CheckRule	::=	`GOAL` *Error* `FROM` *Query* `END`
Error	::=	`all err [` $VarOrConst^\star$, *String*]
Query	::=	*InPart Where*
InPart	::=	*In* | `and` $In^\star$
In	::=	`in` { { *Resource* } *Term* }
Resource	::=	`resource` { `file:` *String* }
Term	::=	*Tag* {{ $VarOrConst^\star$ $Term^\star$ }}
Where	::=	`where` { $Constraint^\star$ } | ϵ

We denote by $\alpha \in C$ that α is a string derived from the syntactic category C.

6.4 The translation process for positive rules

In this section we show formally how to translate the positive web checking rules into abductive logic programs suitable for the CIFF proof procedure. This section is organized as follows: first we show how we represent the XML/XHTML pages in a useful way for CIFF and then we show the formal translation function for the rules. Finally we give some examples of translations.

6.4.1 XML representation

To pave the way to writing, in a format suitable for the CIFF system, specification rules for properties of web sites, we first provide a suitable representation of web site data. Obviously CIFF is not able to handle XML data and XML structures directly, hence a translation into an abductive logic program suitable for CIFF is needed. We propose a translation which maps each XML page into a set of (ground) facts, each one uniquely identified by a numeric `ID` in its arguments. In doing that we make the following assumptions:

- each XML page has no XML attributes, and
- each XML tag name occurs in at most one XML page.

The above assumptions simplify the framework, but the methodology described below could be easily extended in order to remove those restrictions. Moreover, our methodology can also be applied to web pages dynamically generated from a server-side script like PHP pages, ASP pages and so on. Indeed, is very easy to find web sites created through a combination of a server-side script and a database, but in general the client-side output of those web pages is a XML/XHTML file like the ones used here. Thus our methodology, in principle, is fully valid.

For each XML page, we store a fact of the form

```
xml_page(FileName,ID)
```

where the arguments are the name of the file and the unique identifier of the page.
For each tag element of the original XML file, an atom `pg_el` is generated having the following form:

```
pg_el(ID,TagName,IDFather)
```

where `TagName` is the name of the tag element, and `ID` and `IDFather` represent the unique identifiers for the tag element and its father. In particular, we define the father of root level elements of the XML pages as the XML page itself, represented by the `xml_page` atom above. The father of non-root level elements of the XML pages are the tag elements containing them. The identifiers of the father elements are needed for keeping information about the structure of the XML page.
Furthermore a data item inside a tag element is represented by an atom of the form:

```
data_el(ID,Data,IDFatherTag)
```

where `Data` is the raw data, and `ID` and `IDFatherTag` represent the unique identifiers for the tag element and its father (evidencing that the father of a data element can only be a tag element in the corresponding XML page).
The following is the translation of the XML pages seen in section 6.1

```
xml_page('directorindex.xml',1).

pg_el(2,directorlist,1).
  pg_el(3,director,2).
    data_el(4,'John',3).
  pg_el(5,director,2).
    data_el(6,'Mary',5).
  pg_el(7,director,2).
    data_el(8,'Paul',7).
  pg_el(9,director,2).
    data_el(10,'Mary',9).
```

```
xml_page('showindex.xml',11).

pg_el(12,showlist,11).
pg_el(13,show,12).
    pg_el(14,showname,13).
    data_el(15,'Mela',14).
    pg_el(16,dir_by,13).
    data_el(17,'John',16).
    pg_el(18,year,13).
    data_el(19,2000,18).

pg_el(20,show,12).
    pg_el(21,showname,20).
    data_el(22,'Epiloghi',21).
    pg_el(23,year,20).
    data_el(24,2001,23).

pg_el(25,show,12).
    pg_el(26,showname,25).
    data_el(27,'Toccata e fuga',26).
    pg_el(28,dir_by,25).
    data_el(29,'Paul',28).
    pg_el(30,year,25).
    data_el(31,2002,30).
```

For each page, we store also a fact of the form `xml_page(FileName,ID)`, holding the name of the file and the unique identifier of the page

6.4.2 The translation function

We are now ready to show the formal translation function for positive web checking rules mapping them from Xcerpt-like syntax to abductive logic programs. We define inductively two functions, namely $\mathcal{T}[\![\alpha]\!]$ and $\mathcal{T}'[\![\alpha]\!](id)$ which, given a string α derivable from one of the syntactic categories defined in the previous section, both return a string, representing a part of an abductive logic program[2]. The abductive logic program corresponding to a whole web checking rule is obtained by applying the function $\mathcal{T}[\![\mathtt{R}]\!]$ to a rule $\mathtt{R} \in$ *CheckRule*. This abductive logic program with constraints is then suitable as input for CIFF. In particular the translation of a *CheckRule* gives rise to a single CIFF integrity constraint.

The two translation functions differ in that the $\mathcal{T}'[\![\alpha]\!](id)$ has an extra argument, id, representing a number. Intuitively, a call of the form $\mathcal{T}'[\![\alpha]\!](id)$ amounts to translating the element α which is a component of another element, uniquely identified by id. This identifier id is needed in the translation of α to keep the correspondence between α and the element it is part of. In what follows we assume that the auxiliary function

newID()

generates a fresh variable whenever called. This function will be used when a new identifier id is needed to uniquely identify the element being translated.

For brevity we omit the specification of the translation of the basic syntactic elements, since they basically remain unchanged (e.g. the translation of a string α representing a variable name is α itself).

In Xcerpt, variables are used as in logic programming, that is they can be shared between distinct parts of a rule by using the same name. In the translation, variable names are simply left unchanged, so that sharing is maintained.

[2]As usual, we use $[\![\cdot]\!]$ to highlight the syntactic arguments of the translation functions

We use a top-down approach in defining the translation functions. Hence, the first rule is for the *CheckRule* category.

CheckRule
$\mathcal{T}[\![\texttt{GOAL}\ Err\ \texttt{FROM}\ Query\ \texttt{END}]\!] =$
let Head $= \mathcal{T}[\![Err]\!]$
and Body $= \mathcal{T}[\![Query]\!]$
in
Body `implies` Head.

As we can see the result is a string representing a CIFF integrity constraint: the body is the translation of the query part and the head is the translation of the error part. Intuitively, each instance of the XML data matching the body will fire an instance of the head representing the corresponding error. The predicate used in Head of such constraint is abducible, hence firing the constraint will amount to abducing its head, constructed as follows:

Error
$\mathcal{T}[\![\texttt{all err}[\texttt{Vars},\texttt{Msg}]]\!] =$ `[abd_err([Vars],Msg)]`

The predicate `abd_err/2` is defined as abducible. The two arguments are (1) the variable list occurring in the error part of the rule and (2) the error message.
For the query component, the translation produces the conjunction of the translations of the *InPart* component and of the *Where* component of the query.

Query
$\mathcal{T}[\![InPart\ Where]\!] =$
let Conjunction $= \mathcal{T}[\![InPart]\!]$
and Constraints $= \mathcal{T}[\![Where]\!]$
in
[Conjunction`,` Constraints]

The constraints in the *Where* part of a query are simply translated into the same conjunction of CIFF constraints as follows.

Where
$\mathcal{T}[\![\texttt{where Constr}]\!] =$ `Constr`

The translation of the *InPart* of a query results in the conjunction of the translations of the single components.

In-1
$\mathcal{T}[\![\texttt{in}\{\{\texttt{resource}\{\texttt{file}:\texttt{Res}\}\ \ Term\}\}]\!] = \mathcal{T}[\![Term]\!]$

In-2
$\mathcal{T}[\![\texttt{In},\texttt{InRest}]\!] =$
let Conjunct $= \mathcal{T}[\![In]\!]$
and OtherConjuncts $= \mathcal{T}[\![InRest]\!]$
in
Conjunct, OtherConjuncts

The following rules translate terms in *Term*, and produce the single conjuncts in the conjunction constituting the body of the integrity constraint corresponding to a checking rule. Each term specifies a tag and hence the translation amounts to producing an instance of the `pg_el` predicate, possibly in conjunction with other conjuncts corresponding to the translation of the subtags. A fresh variable is generated through the *newID()* function, to uniquely represent the tag being translated. Note the usage of the function $\mathcal{T}'[\![\cdot]\!](\cdot)$ for the translation of the data elements (variables and constants) and of the subtags of the tag being translated.

Term-1
$\mathcal{T}[\![$`tagName {{VarConstSeq, TermSeq}}`$]\!] =$
let Id = *newID()*
and DataElements = $\mathcal{T}'[\![$`VarConstSeq`$]\!](Id)$
and SubTerms = $\mathcal{T}'[\![$`TermSeq`$]\!](Id)$
and Ids = $IdVars$(DataElements) $\cup$ $IdVars$(SubTerms)
and Constraints = *all_distinct*(Ids)
in
`pg_el(`Id`, tagName,_)`, DataElements, SubTerms, Constraints

In the above rule, we have introduced two new auxiliary functions:

- $IdVars(X)$ which is assumed to return the set of all Id's generated by the *newID()* function occurring in its argument X;
- *all_distinct*($Vars$) which is assumed to generate the constraint

$$\bigwedge_{\substack{x,y \in Vars \\ x \neq y}} x \neq y$$

Term-2.1
$\mathcal{T}'[\![$`var VarName`$]\!](Id) =$
let $Id' = newID()$
in
`data_el(`Id'`, VarName, `Id`)`

Term-2.2
$\mathcal{T}'[\![$`Const`$]\!](Id) =$
let $Id' = newID()$
in
`data_el(`Id'`, Const, `Id`)`

Term-3
$\mathcal{T}'[\![$`subTagName {{VarConstSeq, TermSeq}}`$]\!](Id) =$
let $Id' = newID()$
and DataElements = $\mathcal{T}'[\![$`VarConstSeq`$]\!](Id')$
and SubTerms = $\mathcal{T}'[\![$`TermSeq`$]\!](Id')$
in
`pg_el(`Id'`, subTagName, `Id`)`, DataElements, SubTerms

Term-4
$\mathcal{T}'[\![$`X, Xs`$]\!](Id) =$
let Conjunct = $\mathcal{T}'[\![$`X`$]\!](Id)$
and OtherConjuncts = $\mathcal{T}'[\![$`Xs`$]\!](Id)$
in
Conjunct, OtherConjuncts

Cases **Term-2.1** and **Term-2.2** produce instances of the `data_el/3` predicate. Notice that the translation amounts at producing an instance `data_el(`Id'`, El, `Id`)`, where `El` is the actual data element, Id' is the unique identifier assigned to it and Id is the unique identifier of the term the data element is part of. In the case **Term-3** we have a translation similar to case **Term-1**, the only difference being that, in the case **Term-1** the third argument of `pg_el/3` is the anonymous variable, whereas in case **Term-3** is the unique identifier of the term containing the subterm being translated. Finally, the case **Term-4** corresponds to the translation of a sequence `X, Xs` in *VarOrConst** or in *Term**.

Example 6.1. *Let us consider the web checking rule* [Rule1] *for the theater company web site, seen in section 6.2:*

```
%%%%%%%Rule 1 - Double occurrence of a director in the director list
%%%%%%%rule_director_twice_inlist.xce
GOAL
        all err [var Dir1,
                "director twice in the director list" ]
FROM
        in {
         resource {"file:directorindex.xml"},
          directorlist {{
                director {{ var Dir1 }},
                director {{ var Dir1 }}
          }}
        }
END
```

To simplify the presentation of the translation results, let us identify the main syntactic components of the rule:

err_1 = `all error [var Dir1, "director twice in the director list"]`
$query_1$ = in_1
in_1 = `in {resource {"file:directorindex.xml"}` $root_1$ `}`
$root_1$ = `directorlist {{`
`director {{var Dir1}}, director {{var Dir1}} }}`

Let us apply the translation function $\mathcal{T}$ to rule_1.

$\mathcal{T}[\![Rule1]\!]$ = $\mathcal{T}[\![query_1]\!]$ `implies` $\mathcal{T}[\![err_1]\!]$.
$\mathcal{T}[\![err_1]\!]$ = `abd_err([Dir1],"director twice in the director list")`
$\mathcal{T}[\![query_1]\!]$ = `[`$\mathcal{T}[\![in_1]\!]$`]`
$\mathcal{T}[\![in_1]\!]$ = $\mathcal{T}[\![root_1]\!]$
$\mathcal{T}[\![root_1]\!]$ = `pg_el(ID1,directorlist,_), ID2 #\= ID4,`
`pg_el(ID2,director,ID1), data_el(ID3,Dir1,ID2),`
`pg_el(ID4,director,ID1), data_el(ID5,Dir1,ID4),`

Summarizing we have the following translation:

```
[pg_el(ID1,directorlist,_), ID2 #\= ID4
pg_el(ID2,director,ID1), data_el(ID3,Dir1,ID2)
pg_el(ID4,director,ID1), data_el(ID5,Dir1,ID4)]
  implies
[abd_err([Dir1], "director twice in the director list")].
```

□

Example 6.2. *Let us consider the web checking rule* [Rule1] *for the theater company web site, seen in section 6.2:*

```
%%%%%%%Rule 2 - Show produced before year 2000
%%%%%%%rule_year_before_2000.xce
GOAL
        all err [ var Year, "show produced before 2000" ]
FROM
        in {
         resource {"file:showindex.xml"},
           year {{
             var Year
           }}
        }
        where ( var Year < 2000)
END
```

To simplify the presentation of the translation results, let us identify the main syntactic components of the rule:

err_2	=	`all err [var Year, "show produced before 2000"]`
$query_2$	=	in_2 , $where_2$
in_2	=	`in {resource {"file:showindex.xml"}` $root_2$ `}`
$root_2$	=	`year {{var Year}}`
$where_2$	=	`where { Year < 2000}`

Let us apply the translation function $\mathcal{T}$ to rule_1.

$\mathcal{T}[\![Rule2]\!]$	=	$\mathcal{T}[\![query_2]\!]$ `implies` $\mathcal{T}[\![err_2]\!]$.
$\mathcal{T}[\![err_2]\!]$	=	`abd_err([Year], "show produced before 2000")`
$\mathcal{T}[\![query_2]\!]$	=	`[`$\mathcal{T}[\![in_2]\!]$ $\mathcal{T}[\![where_2]\!]$`]`
$\mathcal{T}[\![in_2]\!]$	=	$\mathcal{T}[\![root_2]\!]$
$\mathcal{T}[\![where_2]\!]$	=	`Year #< 2000`
$\mathcal{T}[\![root_2]\!]$	=	`pg_el(ID1,year,_), data_el(ID2,Year,ID1),`

Summarizing we have the following translation:

```
[pg_el(ID1,year,_), data_el(ID2,Year,ID1), Year #< 2000]
  implies
[abd_err([Year], "show produced before 2000").
```

□

6.5 Adding negation to the translation process

The mapping of web checking rules considering also negative parts, i.e. including `without` statements, is slightly more complicated. The resulting abductive logic program is no longer a single integrity constraint but includes also a set of clauses which define new (fresh) predicates needed for handling negation correctly. Also the grammar must be extended for covering the `without` construct on which we impose the following restrictions: (1) a `without` can not appear at the top level of a `in` statement, (2) `without` statements can not be nested, and (3) variables appearing in the scope of a `without` appear elsewhere outside a `without` in the `FROM` part of the rule.
As a whole, the structure of a rule is essentially the same as before. All the grammar rules regarding basic constructs remain unchanged as do all other grammar rules which do not regard negation directly. In the following we show only the grammar rules regarding the newly introduced syntactical categories and the grammar rules regarding the *Term* syntactical category which need to be changed.

Term	::=	*Tag* {{ *VarOrConst** *Term** *Without** }}
Without	::=	`without` *PosTerm* | `without` *VarOrConst* | *WithoutData*
PosTerm	::=	*Tag* {{ *VarOrConst** *PosTerm** }}
WithoutData	::=	`without_data`

In the above grammar rules, there is also the *WithoutData* syntactical category which, as suggested by its name, indicates the absence of raw data inside the father tag. This is the only modification we did to the Xcerpt syntax, but this statement is very useful because one of the most common errors in a web site instance is the absence of data inside a tag.
Removing the *WithoutData* category, the `without` statement is only able to express the absence of a *particular* raw data, i.e. either a constant or a variable which has to be bound to a particular instance elsewhere outside the `without` due to the web checking rules allowedness conditions. The *WithoutData* category, instead, as we will see in the next section, is able to express the absence of *any* data, not violating the web checking rules allowedness conditions.

6.5.1 Translation function

As anticipated, the abductive logic program with constraint obtained by translating a rule containing a `without` statement, is composed by an integrity constraint (as for positive rules) and a

set of clauses.
Hence, the translation function $\mathcal{T}$ must be updated accordingly, returning a pair $\langle\ IC\ ;\ Clauses\ \rangle$ where IC is a string, representing an integrity constraint exactly as for positive rules, and $Clauses$ is a set of definitions for fresh predicates introduced to address negation. Intuitively only those parts of a rule within the scope of a `without` statement augment the $Clauses$ component while the other rules leave it unchanged. Hence, apart from having a different codomain, the translation function $\mathcal{T}$ operates as the old one over most syntactical categories.
As before, we need a further translation function $\mathcal{T}'$ which gets as input a further numeric argument for handling the identifiers of the XML elements. To simplify the presentation we abuse notation in the sequel denoting again as `c` the translation of a basic construct $c \in$ C. I.e., we avoid representing formally the translation of basic constructs which would be $\langle$ `c` ; $\oslash$ $\rangle$, and we treat them as if they were strings as in positive rules.

We follow again a top-down approach in presenting the function, hence the first rule is for the *Checkrule* syntactic category.

CheckRule
$\mathcal{T}[\![$`GOAL` Err `FROM` $Query$ `END`$]\!] =$
  let $\langle$ Head ; $\oslash$ $\rangle = \mathcal{T}[\![Err]\!]$
    and $\langle$ Body ; Clauses $\rangle = \mathcal{T}[\![Query]\!]$
  in
    $\langle$ Body `implies` Head`.` ; Clauses $\rangle$

As we can see, the output of this rule represents a whole abductive logic program with constraints consisting of an integrity constraint and a set of clauses. The latter is only populated by the translation of the `without` statements.

In the following list of rules, the main difference between a rule and the corresponding rule of the positive translation function is only represented by its codomain. We insert further comments only if there are other differences.

Error
$\mathcal{T}[\![$`all err`$[$`Vars,Msg`$]]\!] = \langle$ `[abd_err([Vars],Msg)]` ; $\oslash$ $\rangle$

Query
$\mathcal{T}[\![$`InPart Where`$]\!] =$
  let $\langle$ Conjunction ; Clauses $\rangle = \mathcal{T}[\![$`InPart`$]\!]$
    and $\langle$ Constraints ; $\oslash$ $\rangle = \mathcal{T}[\![$`Where`$]\!]$
  in
    $\langle$ `[`Conjunction`,` Constraints`]` ; Clauses $\rangle$

Where
$\mathcal{T}[\![$`where Constr`$]\!] = \langle$ `Constr` ; $\oslash$ $\rangle$

In-1
$\mathcal{T}[\![$`in{{resource{file:Res} Term}}`$]\!] = \mathcal{T}[\![$`Term`$]\!]$

In-2
$\mathcal{T}[\![$`In,InRest`$]\!] =$
  let $\langle$ Conjunct ; Clauses $\rangle = \mathcal{T}[\![In]\!]$
    and $\langle$ OtherConjuncts ; OtherClauses $\rangle = \mathcal{T}[\![InRest]\!]$
  in
    $\langle$ Conjunct, OtherConjuncts ; Clauses $\cup$ OtherClauses $\rangle$

Term-1

$\mathcal{T}[\![\texttt{tagName}\ \{\{\texttt{VarConstSeq},\ \texttt{TermSeq},\ \texttt{WithoutSeq}\}\}]\!] =$
let $Id = newID()$
and ⟨ DataElements ; ∅ ⟩ = $\mathcal{T}'[\![\texttt{VarConstSeq}]\!](Id)$
and ⟨ SubTerms_Conjs ; SubTerms_Clauses ⟩ = $\mathcal{T}'[\![\texttt{TermSeq}]\!](Id)$
and Ids = $IdVars$(DataElements) ∪ $IdVars$(SubTerms_Conjs)
and Constraints = *all_distinct*(Ids)
and ⟨ Without_Conjs ; Without_Clauses ⟩ = $\mathcal{T}'[\![\texttt{WithoutSeq}]\!](Id)$
in
⟨ `pg_el(`Id`, tagName,_)`, DataElements, SubTerms, Constraints ;
Without_Clauses ∪ SubTerms_Clauses ⟩

Term-2.1

$\mathcal{T}'[\![\texttt{var VarName}]\!](Id) =$
let $Id' = newID()$
in
⟨ `data_el(`Id'`, VarName, `Id`)` ; ∅ ⟩

Term-2.2

$\mathcal{T}'[\![\texttt{Const}]\!](Id) =$
let $Id' = newID()$
in
⟨ `data_el(`Id'`, Const, `Id`)` ; ∅ ⟩

Term-3

$\mathcal{T}'[\![\texttt{tagName}\ \{\{\texttt{VarConstSeq},\ \texttt{TermSeq},\ \texttt{WithoutSeq}\}\}]\!](Id) =$
let $Id' = newID()$
and ⟨ DataElements ; ∅ ⟩ = $\mathcal{T}'[\![\texttt{VarConstSeq}]\!](Id')$
and ⟨ SubTerms_Conjs ; SubTerms_Clauses ⟩ = $\mathcal{T}'[\![\texttt{TermSeq}]\!](Id')$
and Ids = $IdVars$(DataElements) ∪ $IdVars$(SubTerms_Conjs)
and Constraints = *all_distinct*(Ids)
and ⟨ Without_Conjs ; Without_Clauses ⟩ = $\mathcal{T}'[\![\texttt{WithoutSeq}]\!](Id')$
in
⟨ `pg_el(`Id'`, tagName,`Id`)`, DataElements, SubTerms, Constraints ;
Without_Clauses ∪ SubTerms_Clauses ⟩

Term-4

$\mathcal{T}'[\![\texttt{X},\texttt{Xs}]\!](Id) =$
let ⟨ Conjunct ; Clause ⟩ = $\mathcal{T}'[\![\texttt{X}]\!](Id)$
and ⟨ OtherConjuncts ; OtherClauses ⟩ = $\mathcal{T}'[\![\texttt{Xs}]\!](Id)$
in
⟨ Conjunct, OtherConjuncts ; Clauses ∪ OtherClauses ⟩

Case **Term-1** takes into account the new syntax of the modified *Term* syntactic category, in particular the *Without* statements which could be added at the end of the subterm specification. Those statements are the statements which will augment the *Clauses* part of the output. The other cases for the *Term* category must take into account *Without* statements similarly.

The next translation rules are for the new syntactic categories, namely the *Without* and *PosTerm* categories.

Without-1

$\mathcal{T}'[\![\texttt{without PosTerm}]\!](Id) =$
let $Pred = newPred()$
and $Vs = vars(\texttt{PosTerm})$
and ⟨ PosTerm_Conjs ; ∅ ⟩ = $\mathcal{T}'[\![\texttt{PosTerm}]\!](Id)$

in
$\langle$ `not(`$Pred$`(`Vs`,`Id`))` ; {$Pred$`(`Vs`,`Id`) :-` PosTerm_Conjs`.`} $\rangle$

Without-2
$\mathcal{T}'$⟦`without VarConst`⟧(Id) =
let $Pred = newPred()$
and $V = vars($`VarConst`$)$
and $\langle$ DataElement ; $\oslash$ $\rangle$ = $\mathcal{T}'$⟦`VarConst`⟧(Id)
in
$\langle$ `not(`$Pred$`(`V`,`Id`))` ; {$Pred$`(`V`,`Id`) :-` DataElement`.`} $\rangle$

In the above rules, we have introduced two new auxiliary functions:

- $newPred()$ which returns a fresh predicate name of unspecified arity;
- $vars(t)$ which returns the sequence of all the distinct variables appearing in t.

Cases **Without-1** and **Without-2** both create a new (fresh) predicate *Pred* for handling negation. The definition of *Pred* is represented by *PosTerm_Conjs* which is, roughly speaking, the evaluation of the *PosTerm* inside the `without`. Note that the *PosTerm* category will not augment the *Clauses* part of the output because `without` statements can not be nested. A singleton containing the definition of *Pred* obtained in this way, augments the *Clauses* part of the output.
The *Conjs* part, instead, is augmented by a single conjunct of the following form:
`not(`$Pred$`(`$Args$`))`
where the set of arguments *Args*, is represented by the `ID` of the father of the `without` statement and all the variables which appear inside it. In this way, we keep the variable sharing outside the integrity constraint. I.e. when the clause defining *Pred* are evaluated, the arguments of *Pred* ensure that the variable references into the integrity constraint are kept.

The next case **WithoutData**, handles the `without_data` statement. It is worth noticing that there is a fundamental difference between the **Without-2** case and the **WithoutData** case. At first glance, the **Without-2** case seems to be suitable for representing the absence of data inside a tag. But due to the web checking rules allowedness restrictions, the **Without-2** case is only able to express the absence of *particular* data, i.e. some data (variables or constants) which appear also in the positive part of the web checking rule. The **WithoutData** case, instead, is able to express the absence of any data. As we will see, this particular case is handled straightforwardly by the CIFF proof procedure.

WithoutData
$\mathcal{T}'$⟦`without_data`⟧(Id) =
$\langle$ `not(some_data(`Id`))` ; {`some_data(`Id`) :- data_el(_,_,`Id`).`} $\rangle$

The case **Without-3** which follows is a standard rule for handling sequences.

Without-3
$\mathcal{T}'$⟦`W,Ws`⟧(Id) =
let $\langle$ Conjunct ; Clause $\rangle$ = $\mathcal{T}'$⟦`W`⟧(Id)
and $\langle$ OtherConjuncts ; OtherClauses $\rangle$ = $\mathcal{T}'$⟦`Ws`⟧(Id)
in
$\langle$ Conjunct, OtherConjuncts ; Clauses $\cup$ OtherClauses $\rangle$

Finally, we have the rules for handling the *PosTerm* category.

PosTerm-1

$\mathcal{T}'[\![\texttt{tagName}\ \{\{\texttt{VarConstSeq},\ \texttt{PosTermSeq}\}\}]\!](Id) =$
  let $Id' = newID()$
    and $\langle$ DataElements ; $\oslash$ $\rangle$ = $\mathcal{T}'[\![\texttt{VarConstSeq}]\!](Id')$
    and $\langle$ SubPosTerms_Conjs ; $\oslash$ $\rangle$ = $\mathcal{T}'[\![\texttt{PosTermSeq}]\!](Id')$
    and Ids = $IdVars$(DataElements) $\cup$ $IdVars$(SubPosTerms_Conjs)
    and Constraints = *all_distinct*(Ids)
  in
    $\langle$ `pg_el(`Id'`, tagName,`Id`)`, DataElements, SubPosTerms,
      Constraints, Without_Conjs ; $\oslash$ $\rangle$

PosTerm-2

$\mathcal{T}'[\![\texttt{X},\texttt{Xs}]\!](Id) =$
  let $\langle$ Conjunct ; $\oslash$ $\rangle$ = $\mathcal{T}'[\![\texttt{X}]\!](Id)$
    and $\langle$ OtherConjuncts ; $\oslash$ $\rangle$ = $\mathcal{T}'[\![\texttt{Xs}]\!](Id)$
  in
    $\langle$ Conjunct, OtherConjuncts ; $\oslash$ $\rangle$

Case **PosTerm-1** is very similar to **Term-1**. The only difference is that no `without` statements can appear in the subterm specification and thus the *Clauses* part of the output is not affected by this syntactic category. Case **PosTerm-2** is again a standard translation rule for handling sequences.

Example 6.3. *Let us consider again the theater company web site. We want to express that* each show has at least a director (dir_by) tag [Rule4]. *The Xcerpt representation is as follows:*

```
%%%%%%%Rule 4 - No dir_by tag inside a show tag
%%%%%%%rule_no_dir_by.xce
GOAL
        all err [ "show without a dir_by tag" ]
FROM
        in {
         resource {"file:showindex.xml"},
           show {{
             without dir_by {{ }}
           }}
        }
END
```

Let us identify the main syntactic components of the rule:

err_4 = `all err [ "show without a dir_by tag"]`
$query_4$ = `in {resource {"file:showindex.xml"}` $root_4$ `}`
$root_4$ = `show {{` $wout_4$ `}}`
$wout_4$ = `without dir_by {{ }}`

The $root_4$ *element is translated as follows (note that now the output is composed of two parts: the conjuncts part and the clauses part resulting from* $wout_4$*).*

$\mathcal{T}[\![Rule_4]\!] =$
  let $\langle$ *query_conjs ; query_clauses* $\rangle$ = $\mathcal{T}[\![query_4]\!]$
    and $\langle$ *err_conjs ;* $\oslash$ $\rangle$ = $\mathcal{T}[\![err_4]\!]$
  in $\langle$ `[`*query_conjs* `implies` *err_conjs*`].` *; query_clauses* $\rangle$

$\mathcal{T}[\![err_4]\!]$ = $\langle$ `[abd_err([],"show without a dir_by")]` *;* $\oslash$ $\rangle$

$\mathcal{T}[\![query_4]\!] = \mathcal{T}[\![root_4]\!]$

$\mathcal{T}[\![root_4]\!] =$
  let $\langle$ *wout_conjs ; wout_clauses* $\rangle = \mathcal{T}'[\![wout_4]\!]$(`ID1`)
  in $\langle$ `pg_el(ID1,show,_)`, *wout_conjs ; wout_clauses* $\rangle$

Finally, we translate the $wout_4$ part adding a negated atom (of a fresh predicate `pred_1`*) into the body of the integrity constraint, and we define* `pred_1` *with the conjuncts derived from the term specification inside the* `without` *statement. The variables used as arguments for* `pred_1` *are for keeping the right bindings outside the integrity constraint.*

$\mathcal{T}'[\![wout_4]\!]$(`ID1`) $= \langle$ `not(pred_1(ID1))` ; `pred_1(ID1) :- pg_el(ID2,dir_by,ID1)` $\rangle$

Summarizing we have that the output is the following ALPC:

```
[pg_el(ID1,show,_),
 not(pred_1(ID1))]
 implies
[abd_err([],"show without a dir_by tag")].

pred_1(ID1) :- pg_el(ID2,dir_by,ID1).
```

□

Example 6.4. *We continue the Example 6.4, with the requirement that* each `dir_by` tag in the `showlist` must contain some data [Rule5]. *The Xcerpt representation is as follows:*

```
%%%%%%%Rule 5 - No data inside a dir_by tag
%%%%%%%rule_no_dir_by_data.xce
GOAL
        all err [ "dir_by tag without data" ]
FROM
        in {
         resource {"file:showindex.xml"},
           dir_by {{
             without_data
           }}
        }
END
```

Let us identify the main syntactic components of the rule:

err_5	=	`all err [ "dir_by tag without data"]`
$query_5$	=	`in {resource {"file:showindex.xml"}` $root_5$ `}`
$root_5$	=	`dir_by {{` $wout_5$ `}}`
$wout_5$	=	`without_data`

The $root_5$ element is translated as follows (note that now the output is composed of two parts: the conjuncts part and the clauses part resulting from $wout_5$).

$\mathcal{T}[\![Rule_5]\!] =$
  let $\langle$ *query_conjs ; query_clauses* $\rangle = \mathcal{T}[\![query_5]\!]$
    and $\langle$ *err_conjs* ; $\oslash$ $\rangle = \mathcal{T}[\![err_5]\!]$
  in $\langle$ `[`*query_conjs* `implies` *err_conjs*`].` ; *query_clauses* $\rangle$

$\mathcal{T}[\![err_5]\!] = \langle$ `[abd_err([],"dir_by tag without data")]` ; $\oslash$ $\rangle$

$\mathcal{T}[\![query_5]\!] = \mathcal{T}[\![root_5]\!]$

$\mathcal{T}[\![root_5]\!] =$
$\quad$ ***let*** $\langle$ *wout_conjs ; wout_clauses* $\rangle = \mathcal{T}'[\![wout_5]\!]$(`ID1`)
$\quad$ ***in*** $\langle$ `pg_el(ID1,dir_by,_)`, *wout_conjs ; wout_clauses* $\rangle$

Finally, we translate the $wout_5$ *part adding the negative* `not(some_data(ID1))` *literal into the body of the integrity constraint.* `some_data` *is defined as described in the translation function.*

$\mathcal{T}'[\![wout_5]\!]$(`ID1`) $= \langle$ `not(some_data(ID1))` ; `some_data(ID1) :- data_el(_,_,ID1)` $\rangle$

Summarizing we have that the output is the following ALPC:

```
[pg_el(ID1,dir_by,_),
 not(some_data(ID1))]
 implies
[abd_err([],'dir_by tag without data')].

 some_data(ID1) :- data_el(_,_,ID1).
```

□

Abstracting away from the specific name of the identifier as its argument, the `some_data` predicate has a fixed definition. Thus, if there is more than one web checking rule containing the `without_data` statement, in the final abductive logic program with constraints there is only one clause defining the `some_data` predicate, i.e.:

```
some_data(ID) :- data_el(_,_,ID).
```

Section 6.13, describes the full theater example we use throughout the chapter. It is composed of seventeen web checking rules which are quite self-explanatory. In Section 6.13.4 there is the whole abductive logic program with constraints containing the translation of all rules. It also includes the translation of *[Rule3]*, seen in Section 6.2.
In the remainder of this chapter, we will refer to that theater example, unless otherwise indicated.

6.6 Analysis for checking

In this section we outline how our translation provides a semantics for the fragment of Xcerpt we have adopted for defining web checking rules, and how CIFFWEB provides a sound mechanism for performing the checking, by virtue of the soundness of CIFF$^{\neg}$ for abductive logic programming. Given a web site W, let $\mathcal{X}(W)$ be the result of applying the translation given in section 6.4.1 to W. $\mathcal{X}(W)$ is a set of ground unit clauses. Given a set of web checking rules R, let $\mathcal{T}[\![R]\!] = \{\mathcal{T}[\![r]\!] | r \in R\}$. $\mathcal{T}[\![R]\!]$ is an abductive logic program with constraints $\langle P, A, IC\rangle_{\Re}$ such that A consists of all atoms in the predicate *abd_err* and P and IC are given as in section 6.5. [3] Trivially, $\langle P \cup \mathcal{X}(W), A, I\rangle_{\Re}$ is an ALPC. Note that the abducible predicate *abd_err* only occurs in the conclusions of integrity constraints in I.
We say that the rules in R are fulfilled in W if and only if $P \cup \mathcal{X}(W) \models_{3(\Re)} IC$. Namely, there exists no $i \in IC$ such that $P \cup \mathcal{X}(W) \models_{3(\Re)} b$, where b is the conjunction of premises of a ground instance of i. We also say that the rules in R are violated in W if and only if $P \cup \mathcal{X}(W) \not\models_{3(\Re)} IC$ and there exists $\Delta \subseteq A$ such that $P \cup \mathcal{X}(W) \cup \Delta \models IC$.
Intuitively, due to the fact that the abducible atoms can appear only in the head of a rule, the need of abducible atoms (i.e. a non-empty Δ) for satisfying the integrity constraints means that some web checking rule has been violated. Furthermore the abducible atoms in Δ represent

[3] $\mathcal{T}$ actually returns the representation of the ALPC in the format required by CIFF System. We assume here the logical - rather than system - version of this ALPC.

those violations. By soundness of CIFF$^{\neg}$ (see Chapter 4) we obtain that, with an empty query, if CIFFWEB succeeds returning an empty answer then all rules in R are satisfied in W; if CIFFWEB succeeds returning a non-empty answer then some rule in R is violated.
It is worth noticing that the checking process can also be seen as a deductive task. We can obtain a deductive account of the checking process by viewing integrity constraints as deductive clauses and then querying a deductive reasoner for returning *all* the instances of *all* the web site errors. This would require to query explicitly such clauses, which is not needed in our approach since integrity constraints are implicitly taken into account. More importantly, we obtain a coherent framework for repairing web sites, as we will see in the next sections: abductive logic programs for repairing are built upon programs for checking and the repairing process is a genuine abductive task which is difficult to be accounted in a deductive way.

6.7 Running the System

The CIFFWEB system is a prototype system which includes:

- a Java translator from web checking rules to abductive logic programs with constraints,
- and the CIFF System.

Each web checking rule has to be written in a `.xce` file and all such files have to be placed in the same directory together with the XML source files. The Java translator will translate the selected rules together with the related XML sources, creating an `.alp` file representing the abductive logic program with constraints obtained by the application of the translation function seen in the previous sections. Figure 6.1 is a screenshot of the very simple graphical user interface of the Java translator.

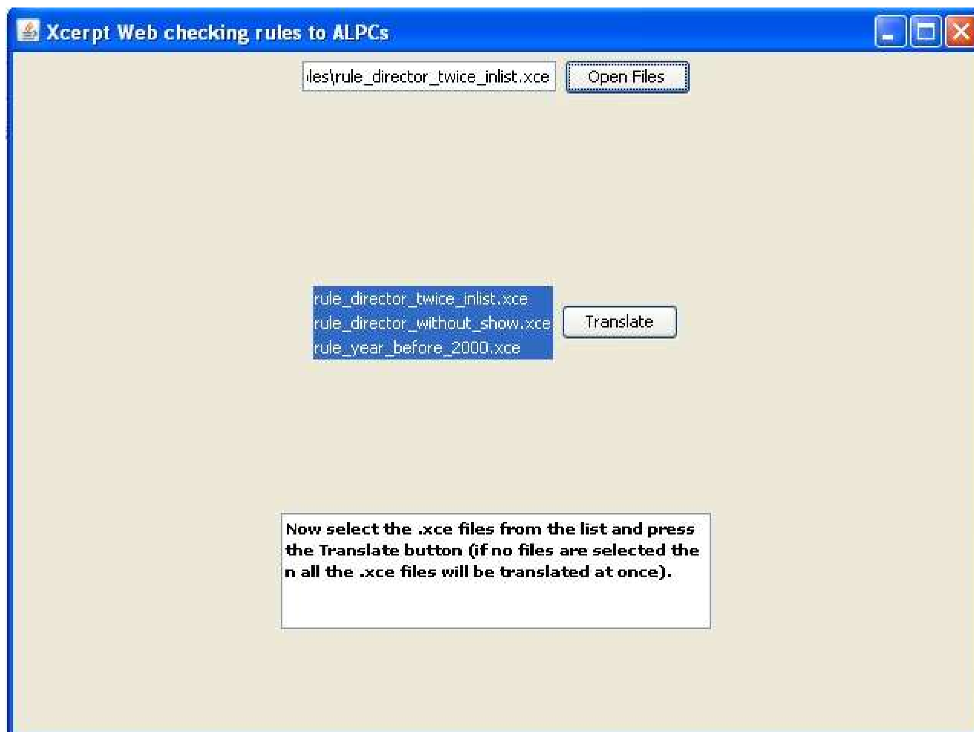


Figure 6.1: The CIFFWEB System: JAVA translator GUI

To date, there is no automatic link between the Java translator and the CIFF System, hence the abductive logic program has to be manually used as input for CIFF together with the empty query.

Furthermore, even if the program could be used directly as input for the CIFF System, we included in the CIFFWEB system a slightly optimized version of the CIFF System. The modifications are just a modified presentation of the abductive answers (below there is one such example) and some machinery useful to boost performance in the process of repairing web sites: they will be described in the next sections. In order to run the CIFFWEB system the command to launch is:

```
ciffweb([File],check).
```

where `File.alp` is the name of the file containing the program and `check` is a flag which specifies to the system that only the check of the web site has to be performed. As we will see in the next sections, the `repair` option could be used as well.

Example 6.5. *Running the CIFFWEB system with the full theater example in Section 6.13, the following abductive answer is produced:*

```
*************** Web site errors ***************
 abd_err(['Mary'],"director twice in the director list")
 abd_err([],"show without a dir_by tag")
 abd_err(['Mary'],"director without a show").
```

representing correctly (1) the fact that `'Mary'` *appears twice in the director list, (2) that* `'Mary'` *does not direct any show and finally (3) that there is a* `show` *tag without a* `dir_by` *tag (in particular the* `show` *identified by* `20` *in the translated data). As we can see, the abducibles in the answer correspond to all the error instances of all the web checking rules.* □

6.7.1 A preliminary experimentation

In [23], there is an interesting experimentation about the scalability of the GVERDI-R system for verifying web sites. The GVERDI-R system is a closely related system, further discussed in Section 6.12, and a state-of-the-art system for its efficiency of verification of web sites.
In those tests a set of XML pages (created by using using the XMLGen tool [4], a tool for creating well-formed, valid, meaningful and very large XML pages with a randomly generated content regarding an auction website) have been verified against three sets of web specification rules. The tests are just quantitative tests of the verification process against very large XML data, from a minimum size of about 1Mb and 30.000 XML tags to a maximum of about 10Mb and 300.000 XML tags. The three web specification are not given by the authors as they are quite meaningless in this setting, even if they are said to be in ascending order of complexity with *completeness* rules occurring only in the third specification. In the GVERDI-R framework, *completeness* rules are similar to our *negative* web checking rules which are computationally harder than *positive* ones. Being the specifications not directly comparable with our specifications we do not report their computational times, but we only notice that

- the GVERDI-R system is very efficient in the first two tests and it scales up to 10Mb in linear time;
- *completeness* rules pull down performance dramatically.

Analogously, we built four web specifications in ascending order of complexity: the first two specifications consist of only positive web checking rules and the last two specifications include also negative web checking rules. Each web specification is a superset of the previous one. CIFFWEB tests are limited to a maximum size of about 3Mb and 90.000 XML tags. This is a SICStus Prolog 3 limitation because the number of atoms required to manage larger XML data exceeds the admissible number of run-time atoms of the Prolog platform. SICStus Prolog 4 should also not suffer from this memory limitation. Results reported in Table 6.1 are expressed in seconds of

Table 6.1: CIFFWEB scalability results for checking

	1Mb (30,000 XML Tags)	**3Mb (90,000 XML Tags)**
WebSpec1	2.92 (0.41 + 2.31 + 0.20)	10.19 (1.18 + 7.23 + 1.78)
WebSpec2	5.92 (0.41 + 2.32 + 3.19)	39.57 (1.18 + 7.28 + 31.10)
WebSpec3	18.24 (0.41 + 2.37 + 15.46)	167.35 (1.18 + 7.36 + 158.81)
WebSpec4	251.58 (0.41 + 2.41 + 248.76)	2359.78 (1.18 + 7.48 + 2351.12)

CPU time and each result is the sum of (1) the time spent by the JAVA translator, (2) the CIFF preprocessing time, and (3) CIFF processing time.
As seen in Section 5.2.3, CIFF preprocessing time can be a heavy component of the total computational time. For obtaining the above results, we have taken advantage of the a-priori structure of the abductive logic programs and here we used a slightly modified version of CIFFWEB in which the JAVA module translates directly the abductive logic programs into iff-definitions. I.e. we perform a large part of preprocessing phase directly in the JAVA module.

WebSpec1 consists of positive web checking rules which give rise to few errors, while the rules in **WebSpec2** give rise to thousands of errors. The same happens for **WebSpec3** and **WebSpec4** respectively. It is worth noticing that the computational time strictly depends on the number of errors and even more on the presence of negative web checking rules, similarly to the GVERDI-R for completeness rules. Overall, we claim that the system shows acceptable computational times on huge XML pages. In particular we stress that the CIFFWEB system is a prototype system which can be largely optimized. Just to say an example, an interface from JAVA to Prolog would allow to pass abductive logic programs directly to CIFF without passing through `.alp` files. This is a clear point of ongoing and future work.

6.8 A Web Repairing Framework

Abductive reasoning can be exploited not only for detecting errors of a web site, but also for repairing such errors, translating web checking rules into more complex abductive logic programs. These abductive logic programs can be used with CIFF in order to *suggest*, through abductive answers, how to repair the errors.

There can be several error types in a web site instance which could be repaired in more than one way as also noted in [11]. For example there can be duplicated data and so a repairing action can be identified as a *deletion* action. Another error can be a wrong tag name or a wrong data item, e.g. a wrong result of a sum. In that case a good action may be to *change* the tag name or the data item which fired the error. A third error type is a missing data item: in that case an *inserting* action is arguably the best repairing action. All of these error types bring a number of issues to be taken into account when a repairing action is performed. A simple example is the wrong sum result. Suppose we have two data tags $A1 = 500$ and $A2 = 400$ together with the third tag $A3$ which is supposed to contain the sum of the values $A1$ and $A2$. Suppose that the value of $A3$ is 1000. It is clear that a changing action should be performed, but on which data? The sum or one of the addends? It is clear that it will be very difficult for an automatic tool to decide which data has to be changed. However, an automatic tool could perform a reasoning process to *suggest* to human experts some repairing actions. In our proposal, we take into account the errors which could be repaired by *inserting* actions, i.e. those errors which arise, arguably, from some missing XML data. This is because abductive reasoning, which, roughly speaking, *adds* elements to a given "world" in order to satisfy a certain "observation", is particularly suitable for proposing *inserting* actions fixing the missing data in a web site instance rather than to *change* or *delete* some piece of information.

[4]The XMLGen tool is available at `http://www.xml-benchmark.org/`

In our framework for web checking, there is a straightforward relation between web checking rules and errors arising from missing data: each web checking rule containing a `without` statement produces errors of that type. This is very intuitive because the `without` statement expresses the absence of information. Consider, for example, the web site requirement that *each show has a dir_by tag [Rule4]*, whose Xcerpt counterpart is:

```
%%%%%%%Rule 4 - No dir_by tag inside a show tag
%%%%%%%rule_no_dir_by.xce
GOAL
        all err [ "show without a dir_by tag" ]
FROM
        in {
         resource {"file:showindex.xml"},
           show {{
             without dir_by {{ }}
           }}
        }
END
```

Each error arising from this web checking rule represents the absence of a `dir_by` tag as a child of a `show`. The idea is that an abductive answer could suggest to *insert* such a `dir_by` tag in the XML data.

In the next sections we describe how our framework can be used for repairing missing data errors, proposing a tool which tries to repair such errors. As we will see, the syntax of web checking rules remains unchanged: what changes is how the rules are translated into abductive logic programs with constraints and how the CIFFWEB system uses the new programs.

6.8.1 Abductive logic programs for repairing

In our approach for repairing framework, we use a framework which is adapted from the framework for checking seen in the previous sections. The difference lies in the way web checking rules are translated into abductive logic programs with constraints.

As said before, the web checking rules we consider for repairing are those rules containing a `without` statement. For simplicity, we call these rules *negative rules* and we call an error arising from one such rule a *negative error*. Conversely, we call all the others web checking rules *positive rules* and we call an error arising from one such rule a *positive error*.
The main idea in building a *program for repairing* is that the negative parts of the negative web checking rules (i.e. the conjunction of the `without` statements) indicate the missing information. Thus, the abductive reasoner should be able to *abduce* such a missing information for repairing the lack of data. In particular, in a program for repairing we need:

1. a way to express new (XML) elements to be added, i.e expressing these elements as abducibles,
2. a way to make the program aware of abduced (XML) elements, and
3. a way to abduce the appropriate (XML) elements to fix the missing data errors.

Recall what we have seen for the checking framework. A *negative rule* is mapped into a program for checking of the form:

$$
\begin{array}{l}
el_1, \ldots, el_m, not\ \ pred_1, \ldots, not\ \ pred_n \rightarrow err \\
pred_1 \leftarrow el_1^1 \wedge \ldots \wedge el_1^{t1} \\
\ldots \\
pred_n \leftarrow el_n^1 \wedge \ldots \wedge el_n^{tn}
\end{array}
$$

where:

- each *pred* is an atom of the form `pred_i(Args)` for handling the `without` statements;
- each *el* is either a `pg_el(ID,X,ID_F)` atom or a `data_el(ID,X,ID_F)`; and
- *err* is the `abd_err` abducible representing the error.

Given an instance of an integrity constraint of the above form such that the XML specification satisfies $el_1, \ldots, el_m$, an error is detected if the XML specification satisfies *none* of the correspondent $pred_1, \ldots, pred_n$ instances.
In the case of repair, the idea is that these errors could be *repaired* by *abducing* the XML elements which satisfy at least one of the corresponding `pred` instances in the body of the integrity constraint itself. Intuitively, from the above piece of program for checking, our goal is to generate a corresponding piece of program for repairing where (1) the integrity constraint is of the form

$$el_1, \ldots, el_m \rightarrow pred_1 \vee \ldots \vee pred_n$$

and (2) the definition of each $pred_i$ is such that XML elements could also be abduced and not only matched against the given XML data. In this way, once the body of an instance of an integrity constraint is satisfied, appropriate XML elements could be abduced in order to satisfy at least a $pred_i$ instance.

Hence, the first thing a program for repairing needs is a way to abduce XML elements. Thus, we declare two new abducible predicates: i.e. the `abd_pg_el` and the `abd_data_el` predicates representing the "abducible versions" of `pg_el` and `data_el` respectively. These relationships are represented by the following set of clauses $Repair_A$ added to a program for repairing:

```
all_pg_el(ID,TagName,IDFather) :- pg_el(ID,TagName,IDFather)
all_pg_el(ID,TagName,IDFather) :- abd_pg_el(ID,TagName,IDFather)

all_data_el(ID,TagName,IDFatherTag) :- data_el(ID,Data,IDFatherTag)
all_data_el(ID,TagName,IDFatherTag) :- abd_data_el(ID,Data,IDFatherTag)
```

An `all_pg_el(X,Y,Z)` atom (resp. an `all_data_el(X,Y,Z)` atom) is satisfied either by an XML element represented by `pg_el(X,Y,Z)` (resp. `data_el(X,Y,Z)`) or by an *abduced* XML element represented by `abd_pg_el(X,Y,Z)` (resp. `abd_data_el(X,Y,Z)`).

The new `all_pg_el` and `all_data_el` predicates (represented by all_i in the remainder) could be used in the $pred_i$ definitions, replacing the el_i predicates:

$$pred_i \leftarrow all_i^1 \wedge \ldots \wedge all_i^{t1}$$

In this way a $pred_i$ instance is satisfied either abductively or not.

However, a program for repairing should not abduce indiscriminately XML elements, but only those elements for fixing a "real" lack of data. Considering again the theater example, no `dir_by` tag should be abduced within the first and the third `show` tags because they already are associated with a director in the original XML data. The above clauses do not prevent this behavior. Thus we introduce new rep_pred_i predicates defined as follows:

$$\begin{array}{ll} rep_pred_i \leftarrow & pred_i \\ rep_pred_i \leftarrow & not\ \ pred_i \wedge all_i^1 \wedge \ldots \wedge all_i^{ti} \wedge \\ & \\ pred_i \leftarrow & el_i^1 \wedge \ldots \wedge el_i^{ti} \end{array}$$

Intuitively a rep_pred_i predicate is satisfied if either the correspondent $pred_i$ is satisfied as well (i.e. without abductions) or at least a XML element is abduced (but only if $pred_i$ is not satisfied). Note that the presence of *not* in the second clause makes the two clauses mutually exclusive. As one can expect those rep_pred_i are put in the head of an integrity constraint in a *program for repairing*:

$$el_1, \ldots, el_m \rightarrow rep_pred_1 \vee \ldots \vee rep_pred_n$$

The abduction of new XML elements leads to another issue to be taken into account: new abducibles introduced to repair an error could violate another web checking rule. A program for repairing must be aware and should be able to repair or, at least, detect these new errors. In order to make a program for repairing aware of the new abducibles, we simply replace the el_i elements by their all_i counterparts in the body of the integrity constraints, thus obtaining:

$$all_1, \ldots, all_m \rightarrow rep_pred_1 \vee \ldots \vee rep_pred_n$$

In this way each abduced XML element can satisfy the body of an integrity constraint leading to new abductions for repairing a *chain* of errors. This modification must be applied also to the bodies of the integrity constraints drawn from positive rules: in that case *positive errors* due to abduced XML elements are detected in the same way positive errors are detected in the original XML data.

Summarizing, we denote as $\mathcal{TR}$ a function which, given a program for checking $\mathcal{C}$:

1. replaces each piece of $\mathcal{C}$ of the form

$$el_1, \ldots, el_m, not\ \ pred_1, \ldots, not\ \ pred_n \rightarrow err$$

$$\begin{array}{l} pred_1 \leftarrow el_1^1 \wedge \ldots \wedge el_1^{t1} \\ \ldots \\ pred_n \leftarrow el_n^1 \wedge \ldots \wedge el_n^{tn} \end{array}$$

   and obtained from the translation of a *negative rule*, by

$$all_1, \ldots, all_m \rightarrow rep_pred_1 \vee \ldots \vee rep_pred_n$$

$$\begin{array}{l} rep_pred_1 \leftarrow pred_1 \\ rep_pred_1 \leftarrow not\ \ pred_1 \wedge all_1^1 \wedge \ldots \wedge all_1^{t1} \\ pred_1 \leftarrow el_1^1 \wedge \ldots \wedge el_1^{t1} \\ \\ \ldots \\ \\ rep_pred_n \leftarrow pred_1 \\ rep_pred_n \leftarrow not\ \ pred_n \wedge all_n^1 \wedge \ldots \wedge all_n^{tn} \\ pred_n \leftarrow el_n^1 \wedge \ldots \wedge el_n^{tn} \end{array}$$

2. replaces each piece of $\mathcal{C}$ of the form

$$el_1, \ldots, el_m \rightarrow err$$

   and obtained from the translation of a *positive rule*, by

$$all_1, \ldots, all_m \rightarrow err.$$

In the sequel, given a program for checking $\mathcal{C}$, we will refer to $Repair_{\mathcal{C}}$ as the results of $\mathcal{TR}(\mathcal{C})$.

Example 6.6. *Let us consider the program for checking $\mathcal{C}$ obtained from the translation of $Rule1$ and $Rule4$ seen in previous sections, i.e.:*

```
%%% Rule 1
[pg_el(ID1,year,_), data_el(ID2,Year,ID1),
 Year #< 2000]
 implies
[abd_err([Year], "show produced before 2000")].

%%% Rule 4
[pg_el(ID1,show,_), not(pred_1(ID1))]
 implies
[abd_err([],'show without a dir_by tag')].

pred_1(ID1) :- pg_el(ID2,dir_by,ID1).
```

Applying the function $\mathcal{TR}$ to $\mathcal{C}$ we obtain the following program for repairing $Repair_{\mathcal{C}}$:

```
%%% Rule 1
[all_pg_el(ID1,year,_), all_data_el(ID2,Year,ID1),
 Year #< 2000]
 implies
[abd_err([Year], "show produced before 2000")].

%%% Rule 4
[all_pg_el(ID1,show,_)]
 implies
[rep_pred_1(ID1)].

rep_pred_1(ID1) :- pred_1(ID1).

rep_pred_1(ID1) :- not(pred_1(ID1)),all_pg_el(ID2,dir_by,ID1).

pred_1(ID1) :- pg_el(ID2,dir_by,ID1).
```

□

We need to take into account also the `without_data` statement. In a program for checking a `without_data` statement was mapped to a fixed clause of the form:

```
some_data(ID) :- data_el(_,_,ID)
```

In a program for repairing, the `some_data` predicate is treated similarly to the others $pred_i$ predicates. Thus, the `some_data` predicate is replaced by the `rep_some_data` defined as:

```
rep_some_data(ID) :- data_el(_,_,ID)

rep_some_data(ID) :- not(data_el(_,_,ID)), abd_data_el(_,_,ID)
```

In the following, given a program for checking $\mathcal{C}$, we assume that $Repair_C = \mathcal{TR}(\mathcal{C})$ contains the above clauses and that each `not(some_data(Args))` in the body of each integrity constraint in $\mathcal{C}$ is replaced by a corresponding `rep_some_data(Args)` in the head of the corresponding integrity constraint in $Repair_{\mathcal{C}}$.

Example 6.7. *Let us consider the program for checking $\mathcal{C}$ obtained from the translation of Rule3, i.e.:*

```
%%%Rule 5
[pg_el(ID1,dir_by,_),
```

```
  not(some_data(ID1))]
  implies
[abd_err([],'dir_by tag without data')].

  some_data(ID1) :- data_el(_,_,ID1).
```

Applying the function $\mathcal{TR}$ to $\mathcal{C}$, we obtain the following piece of program for repairing $Repair_{\mathcal{C}}$:

```
%%% Rule 5
[all_pg_el(ID1,dir_by,_)]
  implies
[rep_some_data(ID1)].

rep_some_data(ID) :- data_el(_,_,ID)

rep_some_data(ID) :- abd_data_el(_,_,ID), not(data_el(_,_,ID))
```

□

In a *program for repairing* there is a last issue to cope with: the identifiers of the abduced XML elements. As seen in Section 6.4.1, each element in the original XML data is associated to a unique numerical identifier and these identifiers maintain the original XML tree structure. When a XML element is abduced a unique identifier should be assigned to it in a similar way. Obviously, for each pair of abduced XML elements, their identifiers should be distinct and each newly generated identifier should be distinct from each identifier of the original XML elements. This can be done by adding, in a *program for repairing*, a set of integrity constraints $Repair_{\mathcal{ID}}$ of the form:

```
[pg_el(X,Y1,Z1),abd_pg_el(X,Y2,Z2)] implies [false]
[data_el(X,Y1,Z1),abd_pg_el(X,Y2,Z2)] implies [false]
[pg_el(X,Y1,Z1),abd_data_el(X,Y2,Z2)] implies [false]
[data_el(X,Y1,Z1),abd_data_el(X,Y2,Z2)] implies [false]

[abd_pg_el(X1,Y1,Z1),abd_pg_el(X2,Y2,Z2),Y1 #\= Y2] implies [X1 #\= X2]
[abd_pg_el(X1,Y1,Z1),abd_pg_el(X2,Y2,Z2),Z1 #\= Z2] implies [X1 #\= X2]
[abd_data_el(X1,Y1,Z1),abd_pg_el(X2,Y2,Z2),Y1 #\= Y2] implies [X1 #\= X2]
[abd_data_el(X1,Y1,Z1),abd_pg_el(X2,Y2,Z2),Z1 #\= Z2] implies [X1 #\= X2]
[abd_data_el(X1,Y1,Z1),abd_data_el(X2,Y2,Z2),Y1 #\= Y2] implies [X1 #\= X2]
[abd_data_el(X1,Y1,Z1),abd_data_el(X2,Y2,Z2),Z1 #\= Z2] implies [X1 #\= X2]
```

The newly generated identifiers can be either skolemized or instantiated to appropriate numerical values by a constraint solver.

The CIFFWEB system relies upon the integrated constraint solver to address this issue. However the above machinery would lead to a huge computational overhead due to the presence of unbound variables in the body of a lot of implications during the abductive process. Thus we slightly modified the CIFF$^{\neg}$ engine in that when either a new `abd_pg_el` atom or a new `abd_data_el` atom is abduced and factorized (i.e. when we are sure that it is distinct from each other abduced atom), its identifier is automatically bound to the least natural number not yet used for identifying an (abduced or not) XML element.

6.9 Running the CIFFWEB System for repairing

The Java translator of web checking rules seen in Section 6.7 creates, automatically, both the program for checking and the program for repairing placing them in two distinct `.alp` files. Let `Repair.alp` be the file containing the program for repairing: in order to run the CIFFWEB system, it is enough to launch

```
ciffweb(['Repair'],repair).
```

If we run the example in Section 6.13, we obtain the abductive answer seen in the previous examples, i.e.:

```
********** Repairing abducibles: *******************************
abd_err(['Mary'],'director twice in the director list')
abd_pg_el(32,dir_by,20)
abd_data_el(33,Mary,32).
```

which correctly associates the second `show` with `Mary` for repairing all the errors in the XML pages. Note that the fact that `Mary` appears twice in the list of directors is not repaired by the system but it is simply detected like in a program for checking. This is because this error arises from a *positive rule*.

6.9.1 Abductively Generated Errors

The presence of `all_pg_el` atoms and `all_data_el` atoms in the body of the integrity constraints in the abductive logic programs for repairing, make the integrity constraints aware about new abduced elements, i.e. their bodies can also be satisfied from abduced elements and not only from the original XML data. This process could lead to further errors.

The abductive answer seen in the previous section is not the unique abductive answer returned by the CIFFWEB system for the example in Section 6.13. A further abductive answer is the following:

```
*************** Repairing abducibles: ***************
abd_err([Mary],director twice in the director list)
abd_err([Mary,John],double data in dir_by tag)
abd_pg_el(32,dir_by,20)
abd_data_el(33,X1,32)
abd_data_el(34,Mary,16)
```

In this case the data item inside the abduced `dir_by` tag is bound to an unspecified data `X1`, while `Mary` has been associated with the first `show` tag. As the first `show` tag was already associated with another director, namely `John`, a further abducible error has been detected because there are two data items inside the `dir_by` tag of the first show.
Note that this solution, even if less intuitive than the previous solution, is a *sound* abductive solution with respect to our repairing framework.
This is a general issue and we argue, that, usually, it might be preferable to avoid that abduced XML elements introduce new errors satisfying the conditions of instances of positive rules. A way to avoid this is by imposing that an instance of an integrity constraint obtained from a *positive rule* leads to a failure in the abductive process if its body is satisfied through (at least) an abduced atom. I.e. we could replace the abductive errors in the head of the integrity constraint by $false$. In this way, if an abduced atom leads to a new error, then the abductive process fails in searching for an alternative abductive answer.

Consider an integrity constraint I, in a program for repairing, which represents a positive web checking rule, i.e.:

$$[I] \quad all_1 \wedge \ldots \wedge all_m \rightarrow err$$

Suppose now we "unfold" all the all_i atoms. What we obtain is a set of 2^m integrity constraints of the form:

$$\begin{array}{ll} [I^1] & noabd_1 \wedge noabd_2 \wedge \ldots \wedge noabd_m \rightarrow err \\ [I^2] & abd_1 \wedge noabd_2 \wedge \ldots \wedge noabd_m \rightarrow err \\ & \ldots \\ [I^{2^m}] & abd_1 \wedge abd_2 \wedge \ldots \wedge abd_m \rightarrow err \end{array}$$

where each $noabd_i$ represents either the `pg_el` or the `data_el` atom and each abd_i represents either the `abd_pg_el` or the `abd_data_el` atom, respectively, obtained by unfolding the corresponding all_i atom. It is worth noticing that there is only one integrity constraint among the 2^m integrity constraints which does not contain abducibles in its body. For simplicity, we say that this integrity constraint is I^1. If we want to avoid that the abduced atoms could not generate new errors, we simply need to replace err by $\bot$ in each integrity constraint other than I^1. We denote as $\mathcal{TR}_\bot$ the translation function which behaves like the translation function $\mathcal{TR}$ plus the addition of the above rewriting step.
Adopting this solution, the latter abductive answer would not be returned by the system, because associating `Mary` with a `show` which is already associated with a director would lead to a failure in the abductive process.

These two levels of *error generation* are both available in the CIFFWEB system by setting the `error_level` flag to `negative` (if no *positive errors* can be generated through abduced XML elements, the default value) or `any` otherwise. If needed, the system automatically preprocesses the integrity constraints as described above. The default value is `negative` and this is assumed in the next section.

6.10 A more complex running example

Two important features of the CIFF proof procedure are the handling of existentially and universally quantified variables and the possibility of returning non-ground answers. Taking advantage of these features, the CIFFWEB system is able to perform a non-straightforward abductive reasoning for repairing web sites which makes it a unique tool in that field. In particular CIFFWEB is able to build chains of repairing actions bringing with itself non-ground elements during the abductive process and constraining them appropriately.

We exemplify this important feature through a slightly modified version of the theater example in Section 6.13. In particular we modify the original XML data by removing the `'Mary'` data item inside the last `director` tag in the `directorindex.xml` page. Accordingly, we delete the atom `data_el(10,'Mary',9)` from the output of the XML translation. Running the CIFFWEB system we obtain the following abductive answer for repairing the web site:

```
 ********** Repairing abducibles: ******************************
 abd_data_el(32,X2,9)
 abd_pg_el(33,dir_by,20)
 abd_data_el(34,Mary,33)
 abd_pg_el(37,show,12)
 abd_pg_el(38,dir_by,37)
 abd_data_el(39,X2,38)
 abd_pg_el(40,showname,37)
 abd_data_el(41,X1,40)
 abd_pg_el(42,year,37)
 abd_data_el(43,X3,42)

********** Disequalities on repairing variables: ****************
 X2\==Paul
 X2\==Mary
 X2\==John

********** Finite domain constraints on repairing variables: ****
 X3#>=2000
```

Indeed, removing `'Mary'` from the original XML files, the error

```
abd_err(['Mary'],'director twice in the director list')
```

is correctly no longer detected by the system. However, the absence of a data item inside the last `director` tag requires the addition of such a data item, represented, in the answer, by the atom:

```
abd_data_el(32,X2,9).
```

This non-ground data `X2` represents the name of the last `director` in the list, and it should be distinct from the other names in that list. This is represented, in the answer, by the set of disequalities on `X2` listed above.
The presence of `X2`, in turn ,requires a `show` associated with it. Thus, the system has to repair two errors: the absence of `show` associated with both `X2` and `'Mary'` (recall that the first instance of `'Mary'` is in the XML data and there is no `show` associated with it). The abducibles

```
abd_pg_el(33,dir_by,20)
abd_data_el(34,Mary,33)
```

repairs the absence of a `show` for `'Mary'`. But now, to associate a `show` with `X2`, a completely new `show` has to be abduced, which in turn fires a set of new abductions in order to get the right XML structure inside it. The abducibles

```
abd_pg_el(37,show,12)
abd_pg_el(38,dir_by,37)
abd_data_el(39,X2,38)
abd_pg_el(40,showname,37)
abd_data_el(41,X1,40)
abd_pg_el(42,year,37)
abd_data_el(43,X3,42)
```

represent the XML data to be added in order to have a further `show` in the list. Note that inside the `dir_by` tag identified with `38` there is exactly the `X2` data, identified with `39`. Note also that the variable `X3`, representing the raw data inside the `year` tag, is associated, correctly, with the finite domain constraint

```
X3#>=2000.
```

in order to avoid further errors.

6.11 Analysis for repairing

An analysis similar to the analysis done for the checking framework (see Section 6.6) could also be outlined for the repairing framework.
Given a web site W, let $\mathcal{X}(W)$ be the result of applying the translation given in section 6.4.1 to W. $\mathcal{X}(W)$ is a set of ground atoms. Given a set of web checking rules R, let $Check_R = \mathcal{T}[\![R]\!] = \{\mathcal{T}[\![r]\!] | r \in R\}$ be the abductive logic program with constraint for checking obtained from the rules R.
Let *Repair* be the program for repairing such that

$$Repair = Repair_A \cup Repair_{\mathcal{ID}} \cup \mathcal{TR}(Check_R)$$

and let $Repair_\perp$ be the program for repairing such that

$$Repair_\perp = Repair_A \cup Repair_{\mathcal{ID}} \cup \mathcal{TR}_\perp(Check_R).$$

The choice of either $\mathcal{TR}$ or $\mathcal{TR}_\perp$ depends on the chosen *error level*, as described in Section 6.9.1.

Both $\mathcal{R}$ and $\mathcal{R}_\bot$ are abductive logic programs with constraints of the form $\langle P,\ A,\ IC\rangle_\Re$ and $\langle P, A, IC_\bot\rangle_\Re$ where

$$A = \{abd_err, abd_pg_el, abd_data_el\}$$

Trivially, also $\langle P \cup \mathcal{X}(W), A, IC\rangle_\Re$ and $\langle P \cup \mathcal{X}(W), A, IC_\bot\rangle_\Re$ are abductive logic programs with constraints.
By construction $IC = IC^- \cup IC^+$ where IC^- is the set of integrity constraints drawn from *negative rules* and IC^+ is the set of integrity constraints drawn from *positive rules.* Each $I^- \in IC^-$ is of the form:

$$all_1, \dots, all_m \rightarrow rep_pred_1 \vee \dots \vee rep_pred_n$$

while each integrity constraint in IC^+ is an integrity constraint I^+ of the form:

$$atom_1, \dots, atom_m \rightarrow err$$

Similarly, $IC_\bot = IC^- \cup IC^+_\bot$ such that each integrity constraint $I^+_\bot \in IC^+_\bot$ is of the form:

$$atom_1, \dots, atom_m \rightarrow \bot$$

where at least an $atom_i$ is abducible.

Consider an answer $\langle \Delta, \sigma, \Gamma\rangle$ for the empty query Q with respect to $\langle P \cup \mathcal{X}(W), A, IC\rangle_\Re$. By definition, we have that there exists a ground $\sigma'' = \sigma\sigma'$ such that $P \cup \Delta\sigma'' \models_{3(\Re)} Q\sigma'' \wedge IC$.
Being Q empty, we only need to consider $P \cup \Delta\sigma'' \models_{3(\Re)} IC$.
By construction, we have that

$$\Delta = \Delta^{err} \cup \Delta^{el}$$

where Δ^{err} contains all the abduced `abd_err` atoms and Δ^{el} contains all the abduced `abd_pg_el` and `abd_data_el` atoms.

Let r^- be a negative rule and let I^- be the corresponding integrity constraint in IC^-. We say that r^- is fulfilled if for each ground instance b^- of the body of I^- such that

$$P \cup \mathcal{X}(W) \cup \Delta^{el}\sigma'' \models_{3(\Re)} b^- \qquad \text{then} \qquad P \cup \mathcal{X}(W) \cup \Delta^{el}\sigma'' \models_{3(\Re)} h^-$$

where h^- is the corresponding instance of the head of I^-.

Let r^+ be a positive rule and let I^+ the corresponding integrity constraint in IC^+. We say that r^+ is fulfilled either if for each ground instance b^+ of the body of I^+ such that

$$P \cup \mathcal{X}(W) \cup \Delta^{el}\sigma'' \models_{3(\Re)} b^+ \qquad \text{then} \qquad P \cup \mathcal{X}(W) \cup \Delta^{err}\sigma'' \models_{3(\Re)} h^+$$

where h^+ is the corresponding instance of the head of I^+.

The same reasoning can be done for the program $\langle P \cup \mathcal{X}(W), A, IC_\bot\rangle_\Re$, but let $I^+_\bot$ be one of the corresponding integrity constraints in $IC^+_\bot$ of a positive rule r^+. We say that r^+ is fulfilled if there is no ground instance of the body $b^+_\bot$ of $I^+_\bot$ such that

$$P \cup \mathcal{X}(W) \cup \Delta^{el}\sigma'' \models_{3(\Re)} b^+_\bot.$$

Summarizing, by soundness of CIFF we obtain that, if CIFF for the empty query

- succeeds returning an empty answer, then all rules in R are satisfied in W;
- succeeds returning an answer with a non-empty Δ, then:
  - if Δ^{err} is non-empty then some positive rule in R is violated and
  - if Δ^{el} is non-empty then some error due to negative rules in R is repaired;
- returns no answer, then some integrity constraints of the form $I^\bot_+$ are violated.

6.12 Related Work, Future Work and Conclusions

In this Chapter, we have presented the CIFFWEB system for verifying and repairing XML/XHTML web sites by using abductive logic programming and in particular the CIFF proof procedure as computational counterpart. Despite the exponential WWW growth and the success of the Semantic Web, there is limited work on specifying, verifying and repairing web sites at a semantic level. Notable exceptions for verifying web sites, are represented by works such as [173] (which mainly inspired our work) [62] and [78]; the XLINKIT framework [40]; the Veriflog tool [51, 50] and the GVERDI-R system [10, 23, 12]. The GVERDI-R system and the Veriflog tool also address (in a preliminary stage) the repairing problem.
The Veriflog is an application of the Xcentric language [52], a XML processing language built on top of Prolog which uses unification for matching and transforming XML data against Xcentric statements. The Veriflog tool uses the Xcentric language as the computational core for verifying and repairing web site instances against web requirements. The repairing problem is addressed through two types of repair actions: the *delete* action and the *replace* action. Despite the capabilities of Veriflog are similar to those of our approach, the syntax for expressing web requirements seems more verbose than our, the implemented tool is at a very early stage and, to our knowledge, the Veriflog tool does not have a formal semantics. Furthermore the repairing process can produce modifications to the XML data which can violate other web requirements: in the Veriflog tool these violations, automatically addressed in our framework for the abduction of missing data through the CIFF abductive reasoning, can only be checked at a later stage.

The work most closely related to ours is the GVERDI-R system which verifies web sites against rules written in an ad-hoc language which allows the specification of both correctness and completeness rules to be fulfilled. The GVERDI-R web rule language is an ad-hoc language which relies upon a (partial) pattern-matching mechanism very similar to the Xcerpt one and its expressiveness is comparable to our Xcerpt fragment. However we argue that our use of negation constructs in the query part of the rules allows for a bit more expressiveness. Furthermore, it seems that queries like the *[Rule1]* seen in section 6.2 are difficult to express in the GVERDI-R language due to the ordering imposed by its language specification in a subterm specification of a query.
Another key difference is that the GVERDI-R system relies upon an ad-hoc computational counterpart for its framework, while our system relies upon the general purpose CIFF abductive proof procedure. This is an important issue because while the GVERDI-R system had to define both the language and its formal properties, the CIFFWEB systems inherits all from the CIFF proof procedure which has a well-defined syntax and has been proved sound with respect to the three-valued completion semantics.
Conversely, the GVERDI-R system allows for the use of functions for managing strings and data in a non-straightforward way, e.g. by matching strings to regular expressions or using arithmetic functions on numbers. While the CIFFWEB system deals with arithmetic functions thanks to the underlying integrated constraint solver, it lacks the use of other types of functions. As pointed out in [10], this is an important feature for a web verification tool. However, a more sophisticated use of functions as in [10, 23] is work in progress, as pointed out in Chapter 5.
Another clear drawback of our system is the absence of a GUI, apart from the simple GUI for the Java translator, which allows for a better usability as for systems like GVERDI-R and XLINKIT.

In section 6.8 we have shown a methodology for switching from programs for checking to programs for repairing. Then we have illustrated, by means of some examples, how the CIFFWEB system is able, through the abductive computational core, to suggest non-trivial and non-ground repairing actions in terms of abduced XML elements which, added to the original data, could repair certain types of errors, in particular those errors concerning missing pieces of information. To our knowledge, this is a novel feature which is absent in the other existing tools for web verification.
We argue that the capability of repairing XML data, both at structural and data level is a very interesting feature for web tools under the Semantic Web [13] vision and that abductive reasoning plus the unique features of CIFF (and $\text{CIFF}^{\neg}$) are very suitable for exploring this field.

The GVERDI-R system also seems to take some steps in that direction as pointed out in [23], but a repairing specification and a concrete tool are still work in progress. In turn, the XLINKIT system seems to be capable of some form of repairing actions but they are limited to broken links in web sites.

Another interesting approach to modifying web data instances is represented by the XChange framework [36] which is proposed by the same Xcerpt authors and from which it derives. It proposes a framework to make the web data aware of events which should lead to changes in the web data. We are currently studying possible interrelations with our repair approach even though the XChange framework is not focused on verification and repair of web site instances but rather seems to propose new data paradigms which can accommodate event-driven changes.

A last mention is deserved to the A^{l}LoWS framework [6] where abductive logic programming is used for web verification similarly to our work. But there the attention is focused on the verification of web services and in particular on the verification of interaction protocols among the agents involved in the web services.

The choice of Xcerpt [37] as our specification XML query language is due to both its similarities to logic programming and its features. As in LP, Xcerpt variables acts as placeholders and the language has a declarative and operational semantics based on the concept of unification. With respect to other well-known query languages such as XPath [47] and XQuery [27], Xcerpt allows complex queries to be expressed in a simpler way and it allows for negation through the `without` construct. Finally, we remark that there are a number of interesting features of Xcerpt which could be integrated in our framework. One of them is the possibility of expressing *aggregates* as aggregated data which is a very typical content of a web page. This last feature is planned to be integrated in CIFF, as pointed out in Chapter 5.

6.13 Full theater example

6.13.1 XML data

```
%%%directorindex.xml

<directorlist>
    <director>John</director>
    <director>Mary</director>
    <director>Paul</director>
    <director>Mary</director>
</directorlist>
```

```
%%%showindex.xml

<showlist>
    <show>
        <showname>Mela</showname>
        <dir_by>John</dir_by>
        <year>2000</year>
    </show>
    <show>
        <showname>Epiloghi</showname>
        <year>2001</year>
    </show>
    <show>
        <showname>Toccata e fuga</showname>
        <dir_by>Paul</dir_by>
        <year>2002</year>
    </show>
</showlist>
```

6.13.2 XML translation

```
xml_page('directorindex.xml',1).

pg_el(2,directorlist,1).
  pg_el(3,director,2).
    data_el(4,'John',3).
  pg_el(5,director,2).
    data_el(6,'Mary',5).
  pg_el(7,director,2).
    data_el(8,'Paul',7).
  pg_el(9,director,2).
    data_el(10,'Mary',9).
```

```
xml_page('showindex.xml',11).

pg_el(12,showlist,11).
pg_el(13,show,12).
    pg_el(14,showname,13).
    data_el(15,'Mela',14).
    pg_el(16,dir_by,13).
    data_el(17,'John',16).
    pg_el(18,year,13).
    data_el(19,2000,18).

pg_el(20,show,12).
    pg_el(21,showname,20).
    data_el(22,'Epiloghi',21).
    pg_el(23,year,20).
    data_el(24,2001,23).

pg_el(25,show,12).
    pg_el(26,showname,25).
    data_el(27,'Toccata e fuga',26).
    pg_el(28,dir_by,25).
    data_el(29,'Paul',28).
    pg_el(30,year,25).
    data_el(31,2002,30).
```

6.13.3 Web checking rules

```
%%%%%%%Rule 1 - Double occurrence of a director in the director list
%%%%%%%rule_director_twice_inlist.xce
GOAL
        all err [ var Dir1,
                "director twice in the director list" ]
FROM
        in {
         resource {"file:directorindex.xml"},
          directorlist {{
                director {{ var Dir1 }},
                director {{ var Dir1 }}
          }}
        }
END

%%%%%%%Rule 2 - Show produced before year 2000
%%%%%%%rule_year_before_2000.xce
GOAL
        all err [ var Year, "show produced before 2000" ]
FROM
        in {
         resource {"file:showindex.xml"},
           year {{
             var Year
           }}
        }
        where ( var Year < 2000)
END

%%%%%%%Rule 3 - Director in the director list without a show associated with
%%%%%%%rule_director_without_show.xce
GOAL
        all err [ var DirName,
                "director without a show" ]
FROM
   and (
        in {
         resource {"file:directorindex.xml"},
           director {{ var DirName }}
        },
        in {
         resource {"file:showindex.xml"},
           showlist {{
             without show {{
                               dir_by {{ var DirName }}
                        }}
           }}
        }
   )
END
```

```
%%%%%%%Rule 4 - No dir_by tag inside a show tag
%%%%%%%rule_no_dir_by.xce
GOAL
        all err [ "show without a dir_by tag" ]
FROM
        in {
         resource {"file:showindex.xml"},
           show {{
             without dir_by {{ }}
           }}
        }
END

%%%%%%%Rule 5 - No data inside a dir_by tag
%%%%%%%rule_no_dir_by_data.xce
GOAL
        all err [ "dir_by tag without data" ]
FROM
        in {
         resource {"file:showindex.xml"},
           dir_by {{
             without_data
           }}
        }
END

%%%%%%%Rule 6 - Double data inside a director tag %%%%%%%
%%%%%%%rule_double_director_data.xce
GOAL
        all err [ var Dir1, var Dir2,
                "double data in director tag" ]
FROM
        in {
         resource {"file:directorindex.xml"},
                director {{
      var Dir1,
      var Dir2 }}
        }
END

%%%%%%%Rule 7 - No data inside a director tag
%%%%%%%rule_no_director_data.xce
GOAL
        all err [ "director tag without data" ]
FROM
        in {
         resource {"file:directorindex.xml"},
           director {{
             without_data
           }}
        }
END
```

```
%%%%%%%Rule 8 - Double showname tag inside a show tag
%%%%%%%rule_double_showname.xce
GOAL
        all err [ "double showname" ]
FROM
        in {
         resource {"file:showindex.xml"},
                show {{
      showname {{ }},
      showname {{ }}
    }}
        }
END

%%%%%%%Rule 9 - No showname tag inside a show tag
%%%%%%%rule_no_showname.xce
GOAL
        all err [ "show without a showname tag" ]
FROM
        in {
         resource {"file:showindex.xml"},
           show {{
             without showname {{ }}
           }}
        }
END

%%%%%%%Rule 10 - Double dir_by tag inside a show tag
%%%%%%%rule_double_dir_by.xce
GOAL
        all err [ "double dir_by" ]
FROM
        in {
         resource {"file:showindex.xml"},
                show {{
      dir_by {{ }},
      dir_by {{ }}
    }}
        }
END
```

```
%%%%%%%Rule 11 - Double year tag inside a show tag
%%%%%%%rule_double_year.xce
GOAL
        all err [ "show with double year" ]
FROM
        in {
         resource {"file:showindex.xml"},
                show {{
      year {{ }},
      year {{ }}
    }}
        }
END

%%%%%%%Rule 12 - No year tag inside a show tag
%%%%%%%rule_no_year.xce
GOAL
        all err [ "show without a year tag" ]
FROM
        in {
         resource {"file:showindex.xml"},
           show {{
             without year {{ }}
           }}
        }
END

%%%%%%%Rule 13 - Double data inside a showname tag
%%%%%%%rule_double_showname_data.xce
GOAL
        all err [ var Showname1, var Showname2,
                "double data in showname tag" ]
FROM
        in {
         resource {"file:showindex.xml"},
                showname {{
      var Showname1,
      var Showname2
    }}
        }
END
```

```
%%%%%%%Rule 14 - No data inside a showname tag
%%%%%%%rule_no_showname_data.xce
GOAL
        all err [ "showname tag without data" ]
FROM
        in {
         resource {"file:showindex.xml"},
           showname {{
             without_data
           }}
        }
END

%%%%%%%Rule 15 - Double data inside a dir_by tag
%%%%%%%rule_double_dir_by_data.xce
GOAL
        all err [ var Dir1, var Dir2, "double data in dir_by tag" ]
FROM
        in {
         resource {"file:showindex.xml"},
                dir_by {{
      var Dir1,
      var Dir2
    }}
        }
END


%%%%%%%Rule 16 - No data inside a year tag
%%%%%%%rule_no_year_data.xce
GOAL
        all err [ "year tag without data" ]
FROM
        in {
         resource {"file:showindex.xml"},
           year {{
             without_data
           }}
        }
END

%%%%%%%Rule 17 - Double data inside a year tag
%%%%%%%rule_double_year_data.xce
GOAL
        all err [ var Year1, var Year2,
                "double data in year tag" ]
FROM
    in {
         resource {"file:showindex.xml"},
            year {{
                var Year1, var Year2
            }}
        }
END
```

6.13.4 Abductive logic program for checking

```
 abducible(abd_err(_,_)).

 some_data(ID) :- data_el(_,_,ID).

%%%% rule_director_twice_inlist.xce %%%
[pg_el(ID1,directorlist,_),
 pg_el(ID2,director,ID1),
 data_el(ID3,Dir1,ID2),
 pg_el(ID4,director,ID1),
 data_el(ID5,Dir1,ID4),ID2#\=ID4]
 implies
[abd_err([Dir1],'director twice in the director list')].

%%%% rule_year_before_2000.xce %%%
[pg_el(ID1,year,_),
 data_el(ID2,Year,ID1),
 Year#<2000]
 implies
[abd_err([Year],'show produced before 2000')].

%%%% rule_director_without_show.xce %%%
pred_0(ID3,DirName) :-
    pg_el(ID4,show,ID3),
    pg_el(ID5,dir_by,ID4),
    data_el(ID6,DirName,ID5).

[pg_el(ID1,director,_),
 data_el(ID2,DirName,ID1),
 pg_el(ID3,showlist,_),
 not(pred_0(ID3,DirName))]
 implies
[abd_err([DirName],'director without a show')].

%%%% rule_no_dir_by.xce %%%
pred_1(ID1) :-
    pg_el(ID2,dir_by,ID1).

[pg_el(ID1,show,_),
 not(pred_1(ID1))]
 implies
[abd_err([],'show without a dir_by tag')].

%%%% rule_no_dir_by_data.xce %%%
[pg_el(ID1,dir_by,_),
 not(some_data(ID1))]
 implies
[abd_err([],'dir_by tag without data')].

%%%% rule_double_dir_by.xce %%%
[pg_el(ID1,show,_),
 pg_el(ID2,dir_by,ID1),
 pg_el(ID3,dir_by,ID1),ID2#\=ID3]
```

```
 implies
[abd_err([],'double dir_by')].

%%%% rule_double_dir_by_data.xce %%%
[pg_el(ID1,dir_by,_),
 data_el(ID2,Dir1,ID1),
 data_el(ID3,Dir2,ID1),ID2#\=ID3]
 implies
[abd_err([Dir1,Dir2],'double data in dir_by tag')].

%%%% rule_double_director_data.xce %%%
[pg_el(ID1,director,_),
 data_el(ID2,Dir1,ID1),
 data_el(ID3,Dir2,ID1),ID2#\=ID3]
 implies
[abd_err([Dir1,Dir2],'double data in director tag')].

%%%% rule_double_showname.xce %%%
[pg_el(ID1,show,_),
 pg_el(ID2,showname,ID1),
 pg_el(ID3,showname,ID1),ID2#\=ID3]
 implies
[abd_err([],'double showname')].

%%%% rule_double_showname_data.xce %%%
[pg_el(ID1,showname,_),
 data_el(ID2,Showname1,ID1),
 data_el(ID3,Showname2,ID1),ID2#\=ID3]
 implies
[abd_err([Showname1,Showname2],'double data in showname tag')].

%%%% rule_double_year.xce %%%
[pg_el(ID1,show,_),
 pg_el(ID2,year,ID1),
 pg_el(ID3,year,ID1),ID2#\=ID3]
 implies
[abd_err([],'show with double year')].

%%%% rule_double_year_data.xce %%%
[pg_el(ID1,year,_),
 data_el(ID2,Year1,ID1),
 data_el(ID3,Year2,ID1),ID2#\=ID3]
 implies
[abd_err([Year1,Year2],'double data in year tag')].

%%%% rule_no_director_data.xce %%%
[pg_el(ID1,director,_),
 not(some_data(ID1))]
 implies
[abd_err([],'director tag without data')].

%%%% rule_no_showname.xce %%%
pred_2(ID1) :-
    pg_el(ID2,showname,ID1).
```

```
[pg_el(ID1,show,_),
 not(pred_2(ID1))]
 implies
[abd_err([],'show without a showname tag')].

%%%% rule_no_showname_data.xce %%%
[pg_el(ID1,showname,_),
 not(some_data(ID1))]
 implies
[abd_err([],'showname tag without data')].

%%%% rule_no_year.xce %%%
pred_3(ID1) :-
    pg_el(ID2,year,ID1).

[pg_el(ID1,show,_),
 not(pred_3(ID1))]
 implies
[abd_err([],'show without a year tag')].

%%%% rule_no_year_data.xce %%%
[pg_el(ID1,year,_),
 not(some_data(ID1))]
 implies
[abd_err([],'year tag without data')].
```

6.13.5 Abductive logic program for repairing

```
 abducible(abd_err(_,_)).
 abducible(abd_pg_el(_,_,_)).
 abducible(abd_data_el(_,_,_)).

 all_pg_el(X,Y,Z) :- pg_el(X,Y,Z).
 all_pg_el(X,Y,Z) :- abd_pg_el(X,Y,Z).

 all_data_el(X,Y,Z) :- data_el(X,Y,Z).
 all_data_el(X,Y,Z) :- abd_data_el(X,Y,Z).

 rep_some_data(F) :- data_el(_,_,F).
 rep_some_data(F) :- abd_data_el(_,_,F), not(data_el(_,_,F)).

%%%% rule_director_twice_inlist.xce %%%
[all_pg_el(ID1,directorlist,_),
 all_pg_el(ID2,director,ID1),
 all_data_el(ID3,Dir1,ID2),
 all_pg_el(ID4,director,ID1),
 all_data_el(ID5,Dir1,ID4),ID2#\=ID4]
 implies
[abd_err([Dir1],'director twice in the director list')].

%%%% rule_year_before_2000.xce %%%
[all_pg_el(ID1,year,_),
 all_data_el(ID2,Year,ID1),
 Year#<2000]
 implies
[abd_err([Year],'show produced before 2000')].

%%%% rule_director_without_show.xce %%%
pred_0(ID3,DirName) :-
    pg_el(ID4,show,ID3),
    pg_el(ID5,dir_by,ID4),
    data_el(ID6,DirName,ID5).
rep_pred_0(ID3,DirName) :-
    pred_0(ID3,DirName).
rep_pred_0(ID3,DirName) :-
    all_pg_el(ID4,show,ID3),
    all_pg_el(ID5,dir_by,ID4),
    all_data_el(ID6,DirName,ID5),
    not(pred_0(ID3,DirName)).
head_pred_1(ID3,DirName) :-
    rep_pred_0(ID3,DirName).

[all_pg_el(ID1,director,_),
 all_data_el(ID2,DirName,ID1),
 all_pg_el(ID3,showlist,_)]
 implies
[head_pred_1(ID3,DirName)].

%%%% rule_no_dir_by.xce %%%
pred_1(ID1) :-
```

```
    pg_el(ID2,dir_by,ID1).
rep_pred_1(ID1) :-
    pred_1(ID1).
rep_pred_1(ID1) :-
    all_pg_el(ID2,dir_by,ID1),
    not(pred_1(ID1)).
head_pred_9(ID1) :-
    rep_pred_1(ID1).

[all_pg_el(ID1,show,_)]
 implies
[head_pred_9(ID1)].

%%%% rule_no_dir_by_data.xce %%%
[all_pg_el(ID1,dir_by,_)]
 implies
[rep_some_data(ID1)].

%%%% rule_double_dir_by.xce %%%
[all_pg_el(ID1,show,_),
 all_pg_el(ID2,dir_by,ID1),
 all_pg_el(ID3,dir_by,ID1),ID2#\=ID3]
 implies
[abd_err([],'double dir_by')].

%%%% rule_double_dir_by_data.xce %%%
[all_pg_el(ID1,dir_by,_),
 all_data_el(ID2,Dir1,ID1),
 all_data_el(ID3,Dir2,ID1),ID2#\=ID3]
 implies
[abd_err([Dir1,Dir2],'double data in dir_by tag')].

%%%% rule_double_director_data.xce %%%
[all_pg_el(ID1,director,_),
 all_data_el(ID2,Dir1,ID1),
 all_data_el(ID3,Dir2,ID1),ID2#\=ID3]
 implies
[abd_err([Dir1,Dir2],'double data in director tag')].

%%%% rule_double_showname.xce %%%
[all_pg_el(ID1,show,_),
 all_pg_el(ID2,showname,ID1),
 all_pg_el(ID3,showname,ID1),ID2#\=ID3]
 implies
[abd_err([],'double showname')].

%%%% rule_double_showname_data.xce %%%
[all_pg_el(ID1,showname,_),
 all_data_el(ID2,Showname1,ID1),
 all_data_el(ID3,Showname2,ID1),ID2#\=ID3]
 implies
[abd_err([Showname1,Showname2],'double data in showname tag')].

%%%% rule_double_year.xce %%%
```

```
[all_pg_el(ID1,show,_),
 all_pg_el(ID2,year,ID1),
 all_pg_el(ID3,year,ID1),ID2#\=ID3]
 implies
[abd_err([],'show with double year')].

%%%% rule_double_year_data.xce %%%
[all_pg_el(ID1,year,_),
 all_data_el(ID2,Year1,ID1),
 all_data_el(ID3,Year2,ID1),ID2#\=ID3]
 implies
[abd_err([Year1,Year2],'double data in year tag')].

%%%% rule_no_director_data.xce %%%
[all_pg_el(ID1,director,_)]
 implies
[rep_some_data(ID1)].

%%%% rule_no_showname.xce %%%
pred_2(ID1) :-
    pg_el(ID2,showname,ID1).
rep_pred_2(ID1) :-
    pred_2(ID1).
rep_pred_2(ID1) :-
    all_pg_el(ID2,showname,ID1),
    not(pred_2(ID1)).
head_pred_12(ID1) :-
    rep_pred_2(ID1).

[all_pg_el(ID1,show,_)]
 implies
[head_pred_12(ID1)].

%%%% rule_no_showname_data.xce %%%
[all_pg_el(ID1,showname,_)]
 implies
[rep_some_data(ID1)].

%%%% rule_no_year.xce %%%
pred_3(ID1) :-
    pg_el(ID2,year,ID1).
rep_pred_3(ID1) :-
    pred_3(ID1).
rep_pred_3(ID1) :-
    all_pg_el(ID2,year,ID1),
    not(pred_3(ID1)).
head_pred_14(ID1) :-
    rep_pred_3(ID1).

[all_pg_el(ID1,show,_)]
 implies
[head_pred_14(ID1)].

%%%% rule_no_year_data.xce %%%
```

```
[all_pg_el(ID1,year,_)]
 implies
[rep_some_data(ID1)].
```

Chapter 7

Conclusions

In this thesis we have presented the CIFF (and CIFF$^\neg$) proof procedure for abductive logic programming with constraints together with its Prolog implementation, the CIFF System. We have also presented a CIFF application for checking and (possibly) repairing XML web sites instances against sets of domain-dependent high level requirements, thus evidencing even more how declarative approaches and abduction could be very useful tools supporting the (Semantic) Web vision.

The CIFF proof procedure, based on the IFF proof procedure [82], represents an advance in the expressiveness of the abductive framework including a number of interesting features:

- the non-straightforward handling of existentially quantified and universally quantified variables (managing at run-time the presence of "dangerous" quantification patterns, i.e. quantification patterns leading to *floundering*),
- the integration with a finite domain constraint solver,
- the use of implicative integrity constraints which are very expressive form of integrity contraints and
- a Negation As Failure (NAF) treatment for negation in the body of integrity constraints (in the CIFF$^\neg$ proof procedure).

The above set of features raises the bar of the expressiveness of the abductive framework seen in state-of-the-art abductive proof procedures (with constraints), e.g. the IFF proof procedure [82, 157, 81], the SLDNFA proof procedure [60, 61], the ACLP proof procedure [109, 108] and the $\mathcal{A}$-System [111, 139].
Another main contribution of this thesis is the proof of soundness of both CIFF and CIFF$^\neg$ proof procedures with respect to the three-valued completion semantics.

CIFF and CIFF$^\neg$ are both implemented in the CIFF System, in Prolog. The CIFF System required about two years of work to be fully developed, and most of the time has been spent in order to make the system both efficient and reflecting as much as possible the computational schema of the specification. A run of the CIFF System can be traced by the user following the application of CIFF proof rules as in the specification. Experimental results, over a number of benchmarks and applications, show that the CIFF System is comparable in performance to the other existing state-of-the-art abductive systems (the $\mathcal{A}$-System) and other related tools, namely the answer sets solvers for Answer Sets Programming (in particular the DLV [68, 119] solver and the SMODELS [136, 166] solver).
Comparing CIFF with the Answer Sets Programming paradigm, which is nowadays a very popular paradigm in the nonmonotonic knowledge representation field, we also outline how the CIFF$^\neg$ proof procedure can behave as an answer sets solver, thus representing a step towards an unifying framework including all the benefits of Abductive Logic Programming with Constraints (constraint

solving, goal-orientedness, handling of non-ground terms and so on), with the benefits of Answer Sets Programming (termination, completeness with respect to the answers sets semantics and good computational results).
The CIFF proof procedure, however, leaves open many possibilities of future work. At a theoretical level, apart from the formal work about the relation of CIFF with the ASP paradigm, it would be interesting to prove some completeness results with respect to the three-valued completion semantics, at least characterizing a certain class of abductive logic programming with constraints as the IFF procedure does, ensuring its completeness with respect to the class of IFF allowed frameworks [81]. Concerning the CIFF System, some lines of future work could be:

- the development of a friendly GUI,
- the possibility of invoking Prolog built-in predicates and functions directly, and
- further improvements in the management of integrity constraints.

The last contribution of this thesis is the development of the CIFFWEB System for checking and (possibly) repairing XML web sites instances against sets of domain-dependent high level requirements.
The CIFFWEB system is a tool which integrates:

- the CIFF System, and
- a JAVA translator from *web checking rules* to both abductive logic programs with constraints for checking the web sites instances and abductive logic programs with constraints for repairing the web sites instances.

Web checking rules are human-tailored rules for expressing the web site requirements which the XML web site instances have to fulfill. We have defined a formal syntax for web checking rules by using a fragment of the well-known XML query language Xcerpt [37]. This choice is due to both its similarities to logic programming, its expressiveness and its very intuitive syntax. Then we have defined a formal translation function from web checking rules to both abductive logic programs with constraints for checking and abductive logic programs with constraints for repairing the web sites instances. Those programs are then used as input for CIFF which either determines the fulfillment of the web checking rules or detects (and possibly suggests appropriate repairing actions for) the errors in the XML instances. The errors captured by this methodology are both *structural* errors (i.e. errors concerning the compliance of the XML tag structure with respect to the web checking rules) and *data* errors (i.e. errors concerning the raw values inside the XML tags).
We have outlined, by means of examples, how the CIFFWEB tool is able to suggest non-trivial and non-ground repairing actions in terms of abduced XML elements taking advantage of the unique features of CIFF as a tool for knowledge representation. These abduced XML elements, added to the original data, could repair certain types of errors, in particular those errors concerning the lack of some piece of information. To our knowledge, this is a novel feature which is absent in the other existing tools for web verification.
We argue that the capability of repairing XML data, both at structural and data level, is a very interesting feature for web tools under the Semantic Web [13] vision and the abductive reasoning plus the unique features of CIFF (and CIFF$^{\neg}$) are very suitable for exploring this field.

Bibliography

[1] SICStus Prolog. `http://www.sics.se/isl/sicstuswww/site/index.html`.

[2] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[3] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. A rule-based approach for reasoning about collaboration between smart web services. In *Web Reasoning and Rule Systems*, pages 279–288, 2007.

[4] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Web service contracting: Specification and reasoning with SCIFF. In *4th European Semantic Web Conference*, pages 68–83, 2007.

[5] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logic (ToCL)*, 2007.

[6] M. Alberti, M. Gavanelli, E. Lamma, F. Chesani, P. Mello, and M. Montali. An abductive framework for a-priori verification of web services. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 39–50, Venice, Italy, 2006.

[7] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. The SCIFF abductive proof-procedure. In *Congress of the Italian Association for Artificial Intelligence (AI*IA)*, pages 135–147, 2005.

[8] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *J. Log. Program.*, 45(1-3):43–70, 2000.

[9] J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, 2004.

[10] M. Alpuente, D. Ballis, and M. Falaschi. A rewriting-based framework for web sites verification. *Electronic Notes in Theoretical Computer Science*, 124:41–61, 2005.

[11] M. Alpuente, D. Ballis, M. Falaschi, and D. Romero. A semi-automatic methodology for repairing faulty web sites. *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, 0:31–40, 2006.

[12] M. Alpuente, D. Ballis, M. Falaschi, and D. Romero. A semi-automatic methodology for repairing faultyweb sites. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 31–40, Pune, India, 2006.

[13] G. Antoniou and F. van Harmelen. *A Semantic Web Primer (Cooperative Information Systems).* The MIT Press, April 2004.

[14] K. R. Apt. Introduction to logic programming. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 493–574. Elsevier and MIT Press, 1990.

[15] K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[16] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, pages 89–148. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[17] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[18] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982.

[19] C. Aravindan and P. M. Dung. Knowledge base dynamics, abduction, and database updates. *Journal of Applied Non-Classical Logics*, 5(1):51–76, 1995.

[20] O. Arieli, M. Denecker, B. V. Nuffelen, and M. Bruynooghe. Coherent integration of databases by abductive logic programming. *Journal of Artificial Intelligence Research*, 21:245–286, 2004.

[21] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[22] M. Balduccini, M. Gelfond, and M. Nogueira. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, 47(1-2):183–219, 2006.

[23] D. Ballis and D. Romero. Fixing web sites using correction stategies. In *Proc. of the 2nd International Workshop on Automated Specification and Verification of Web Systems (WWV'06)*, 2006.

[24] C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press, New York, NY, USA, 2003.

[25] C. Baral and M. Gelfond. Logic programming and knowledge representation. *J. Log. Program.*, 19/20:73–148, 1994.

[26] K. V. Belleghem, M. Denecker, and D. D. Schreye. A strong correspondence between description logics and open logic programming. In *Proc. of the 14th International Conference on Logic Programming*, pages 346–360, 1997.

[27] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML query language, 2007. `http://www.w3.org/TR/xquery/` seen on April 30th, 2008.

[28] P. A. Bonatti. Abduction, ASP and open logic programs. *The Computing Research Repository (CoRR)*, cs.AI/0207021, 2002.

[29] P. A. Bonatti. Abduction over unbounded domains via ASP. In *Proc. of the 16th European Conference on Artificial Intelligence*, pages 288–292, 2004.

[30] A. Bondarenko, P. M. Dung, R. A. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intellegence*, 93(1-2):63–101, 1997.

[31] A. Bracciali, N. Demetriou, U. Endriss, A. C. Kakas, W. Lu, P. Mancarella, F. Sadri, K. Stathis, G. Terreni, and F. Toni. The KGP model of agency for global computing: Computational model and prototype implementation. In *Global Computing*, pages 340–367, 2004.

[32] S. Bressan, C. H. Goh, K. Fynn, M. J. Jakobisiak, K. Hussein, H. B. Kon, T. Lee, S. E. Madnick, T. Pena, J. Qu, A. W. Shum, and M. Siegel. The context interchange mediator prototype. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 525–527, 1997.

[33] S. Bressan, C. H. Goh, T. Lee, S. E. Madnick, and M. Siegel. A procedure for mediation of queries to sources in disparate contexts. In *Proc. of the International Logic Programming Symposium*, pages 213–227, 1997.

[34] G. Brewka. Preferred subtheories: An extended logical framework for default reasoning. In *Proc. of the 11st International Joint Conference on Artificial Intelligence*, pages 1043–1048, 1989.

[35] D. Brickley and R. V. Guha. RDF vocabulary description langauge 1.0: RDF Schema, 2003. `http://www.w3.org/TR/rdf-schema/` seen on April 30th, 2008.

[36] F. Bry and M. Eckert. A high-level query language for events. In *Proc. of the Electrical Design of Advanced Packaging and Systems 2006*, pages 31–38, 2006.

[37] F. Bry, T. Furche, and B. Linse. Data model and query constructs for versatile web query languages: State-of-the-art and challenges for Xcerpt. In *Proc. of the 4th Workshop on Principles and Practice of Semantic Web Reasoning*, pages 90–104, 2006.

[38] F. Bry and S. Schaffert. The XML query language Xcerpt: Design principles, examples, and semantics, 2002.

[39] O. K. C. Elsenbroich and U. Sattler. A case for abductive reasoning over ontologies. In *Proc. of OWL: Experiences and Directions*, 2006.

[40] L. Capra, W. Emmerich, A. Finkelstein, and C. Nentwich. XLINKIT: a consistency checking and smart link generation service. *ACM Transac. on IT*, 2:151–185, 2002.

[41] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[42] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[43] P. Cholewinski, V. W. Marek, and M. Truszczynski. Default reasoning system DeReS. In *Proc. of the 5th Principles of Knowledge Representation and Reasoning Conference*, pages 518–528, Cambridge, MA, USA, 1996.

[44] H. Christiansen. Executable specifications for hypotheses-based reasoning with prolog and constraint handling rules. *Journal of Applied Logic*, 2008. Accepted for publication.

[45] A. Ciampolini, E. Lamma, P. Mello, and C. Stefanelli. Abductive coordination for logic agents. In *Proc.of the 14th ACM Symposium on Applied Computing*, pages 134–140, 1999.

[46] A. Ciampolini, E. Lamma, P. Mello, F. Toni, and P. Torroni. Cooperation and competition in alias: A logic framework for agents that negotiate. *Annals of Mathematics and Artificial Intelligence*, 37(1-2):65–91, 2003.

[47] J. Clark and S. DeRose. XML Path language (XPath) version 1.0, 1999.

[48] K. L. Clark. Negation as failure. In *Logic and Data Bases.* Plenum Press, 1978.

[49] C. Codognet and P. Codognet. Abduction and concurrent logic languages. In *Proc. of the 11th European Conference on Artificial Intelligence*, pages 75–79, 1994.

[50] J. Coelho and M. Florido. Veriflog: A constraint logic programming approach to verification of website content. In *Advanced Web and Network Technologies, and Applications, APWeb 2006 International Workshops: XRA, IWSN, MEGA, and ICSE*, pages 148–156, Harbin, China, 2006.

[51] J. Coelho and M. Florido. Type-based static and dynamic website verification. In *International Conference on Internet and Web Applications and Services (ICIW)*, page 32, Le Morne, Mauritius, 2007.

[52] J. Coelho and M. Florido. Xcentric: logic programming for xml processing. In *9th ACM International Workshop on Web Information and Data Management (WIDM)*, pages 1–8, Lisbon, Portugal, 2007.

[53] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

[54] A. Colmerauer and P. Roussel. The birth of Prolog. In *History of Programming Languages Preprints*, pages 37–52, 1993.

[55] L. Console, D. T. Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[56] L. Console, L. Portinale, and D. T. Dupré;. Using compiled knowledge to guide and focus abductive diagnosis. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):690–706, 1996.

[57] P. T. Cox and T. Pietrzykowski. Causes for events: their computation and applications. In *Proc. of the 8th International Conference on Automated Deduction*, pages 608–621, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[58] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. In *Distributed Artificial Intelligence*, pages 333–356. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[59] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *Proc. of the 10th European Conference on Artificial Intelligence*, pages 384–388, 1992.

[60] M. Denecker and D. D. Schreye. SLDNFA: An abductive procedure for normal abductive programs. In *Joint International Conference and Symposium on Logic Programming*, pages 686–700, 1992.

[61] M. Denecker and D. D. Schreye. SLDNFA: An abductive procedure for abductive logic programs. *J. Log. Program.*, 34(2):111–167, 1998.

[62] T. Despeyroux and B. Trousse. Semantic verification of web sites using natural semantics. In *Proc. of the 6th Conference on Content-Based Multimedia Information Access*, Paris, France, 2000.

[63] Y. Dimopoulos and A. C. Kakas. Information integration and computational logic. *The Computing Research Repository (CoRR)*, cs.AI/0106025, 2001.

[64] P. M. Dung. Negations as hypotheses: An abductive foundation for logic programming. In *Proc. of the 8th International Conference on Logic Programming*, pages 3–17, 1991.

[65] P. M. Dung. On the relations between stable and well-founded semantics of logic programs. *Theor. Comput. Sci.*, 105(1):7–25, 1992.

[66] T. Eiter and G. Gottlob. The complexity of logic-based abduction. In *Proc. of the 10th Symposium on Theoretical Aspects of Computer Science*, pages 70–79, 1993.

[67] T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: Semantics and complexity. *Theor. Comput. Sci.*, 189(1-2):129–177, 1997.

[68] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for nonmonotonic reasoning. In *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 364–375, London, UK, 1997. Springer-Verlag.

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

[70] U. Endriss, M. Hatzitaskos, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Refinements of the CIFF procedure. In *Proc. of the 12th Workshop on Automated Reasoning*, Edinburgh, UK, 2005.

[71] U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Abductive logic programming with CIFF: implementation and applications, 2004. Convegno Italiano di Logica Computazionale (CILC).

[72] U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Abductive logic programming with CIFF: system description. In *Proc. of the 9th European Conference on Logics in Artificial Intelligence (JELIA)*, Lisbon, Portugal, 2004.

[73] U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure for abductive logic programming with constraints. In *Proc. of the 9th European Conference on Logics in Artificial Intelligence (JELIA)*, Lisbon, Portugal, 2004.

[74] K. Eshghi. Abductive planning with event calculus. In *Proc. of the 5th International Conference on Logic Programming/Symposium on Logic Programming*, pages 562–579, 1988.

[75] K. Eshghi and R. A. Kowalski. Abduction compared with negation by failure. In *Proc. of the 6th International Conference on Logic Programming*, pages 234–254, 1989.

[76] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using database optimization techniques for nonmonotonic reasoning. In *Proc. of the 7th International Workshop on Deductive Databases and Logic Programming*, pages 135–139, 1999.

[77] A. J. Fernández and P. M. Hill. A comparative study of eight constraint programming languages over the boolean and finite domains. *Constraints*, 5(3):275–301, 2000.

[78] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Verifying integrity constraints on web site. In *Proc. of the 16th International Joint Conference on Artificial Intelligence*, 1999.

[79] M. Fitting. A deterministic Prolog fixpoint semantics. *J. Log. Program.*, 2(2):111–118, 1985.

[80] T. W. Frühwirth. Theory and practice of constraint handling rules. *J. Log. Program.*, 37(1-3):95–138, 1998.

[81] T. H. Fung. *Abduction by deduction.* PhD thesis, Imperial College, University of London, 1996.

[82] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *J. Log. Program.*, 33(2):151–165, 1997.

[83] D. Gabbay, R. Kempson, and J. Pitt. Labelled abduction and relevance reasoning. In *Nonstandard queries and nonstandard answers: studies in logic and computation*, pages 155–185. Oxford University Press, Oxford, UK, 1994.

[84] J. Gartner, T. Swift, A. Tien, C. V. Damásio, and L. M. Pereira. Psychiatric diagnosis from the viewpoint of computational logic. In *Proc. of Computational Logic, First International Conference*, pages 1362–1376, London, UK, 2000.

[85] A. V. Gelder. The alternating fixpoint of logic programs with negation. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–10, New York, NY, USA, 1989. ACM Press.

[86] A. V. Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.

[87] M. Gelfond and N. Leone. Logic programming and knowledge representation — the A-Prolog perspective. *Artificial Intelligence*, 138(1–2):3–38, 2002.

[88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th International Conference on Logic Programming/Symposium on Logic Programming*, pages 1070–1080, 1988.

[89] M. Gelfond and V. Lifschitz. Logic programs with classical negation. *Proc. of the 7th International Conference on Logic Programming*, pages 579–597, 1990.

[90] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[91] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *J. Autom. Reason.*, 36(4):345–377, 2006.

[92] P. Hayes. RDF semantics, 2003. `http://www.w3.org/TR/rdf-mt/` seen on April 30th, 2008.

[93] C. Holzbaur. Metastructures versus attributed variables in the context of extensible unification. In *Proc. of the 4th Symposium on Programming Language Implementation and Logic Programming*, pages 260–268, Leuven, Belgium, 1992.

[94] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[95] J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

[96] A. Kakas and M. Denecker. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond. Part I*, pages 402–436. Springer Verlag, 2002.

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

[98] A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. A logic-based approach to model computees., 2003. Deliverable D4. SOCS Consortium.

[99] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.

[100] A. C. Kakas and P. Mancarella. Abductive logic programming. In *Proc. of the 1st Logic Programming and Non-Monotonic Reasoning (LPNMR)*, pages 49–61, 1990.

[101] A. C. Kakas and P. Mancarella. Database updates through abduction. In *Proc. of the 16th Very Large Data Bases (VLDB) Conference*, pages 650–661, Brisbane, Queensland, Australia, 1990. Morgan Kaufmann.

[102] A. C. Kakas and P. Mancarella. Generalized stable models: A semantics for abduction. In *Proc. of the 9th European Conference on Artificial Intelligence*, pages 385–391, 1990.

[103] A. C. Kakas and P. Mancarella. Knowledge assimilation and abduction. In *Proc. of the Truth Maintenance Systems (European Conference on Artificial Intelligence Workshop)*, pages 54–70, Stockholm, Sweden, 1990.

[104] A. C. Kakas and P. Mancarella. On the relation of truth maintenance and abduction. In *Proc. 1st Pacific Rim International Conference on Artificial Intelligence, PRICAI90*, Nagoya, Japan, 1990.

[105] A. C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In *Proc. of the 16th European Conference on Artificial Intelligence*, pages 33–37, 2004.

[106] A. C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In *Proc. of the 12th International Conference on Logic Programming*, pages 399–413, 1995.

[107] A. C. Kakas and A. Michael. Air-crew scheduling through abduction. In *Proc.of the 12th Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE)*, pages 600–611, Cairo, Egypt, 1999.

[108] A. C. Kakas and A. Michael. An abductive-based scheduler for air-crew assignment. *Applied Artificial Intelligence*, 15(3):333–360, 2001.

[109] A. C. Kakas, A. Michael, and C. Mourlas. ACLP: Abductive constraint logic programming. *Journal of Logic Programming*, 44:129–177, 2000.

[110] A. C. Kakas and C. Mourlas. ACLP: Flexible solutions to complex problems. In *Proc. of the 4th Logic Programming and Non-Monotonic Reasoning (LPNMR)*, pages 388–399, 1997.

[111] A. C. Kakas, B. V. Nuffelen, and M. Denecker. A-System: Problem solving through abduction. In *Proc. of the 17th International Joint Conference on Artificial Intelligence*, pages 591–596, 2001.

[112] S. C. Kleene. *Introduction to Metamathematics.* Bibl. Matematica. North-Holland, Amsterdam, 1952.

[113] R. Kowalski. *Logic for Problem Solving.* North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1986.

[114] R. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):391–419, 1999.

[115] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, 1986.

[116] R. A. Kowalski. Predicate logic as programming language. In *Proc. of the 6th International Federation for Information Processing (IFIP)) World Computer Congress*, pages 569–574, Stockholm, Sweden, 1974.

[117] R. A. Kowalski, F. Toni, and G. Wetzel. Executing suspended logic programs. *Fundam. Inform.*, 34(3):203–224, 1998.

[118] K. Kunen. Negation in logic programming. *J. Log. Program.*, 4(4):289–308, 1987.

[119] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.

[120] V. Lifschitz. Answer set planning. In *International Conference on Logic Programming*, pages 23–37, 1999.

[121] F. Lin and J.-H. You. Abduction in logic programming: A new definition and an abductive procedure based on rewriting. *Artificial Intelligence*, 140(1/2):175–205, 2002.

[122] I. W. Lloyd and R. W. Topor. A basis for deductive database systems - II. *J. Log. Program.*, 30(1):55–67, 1986.

[123] J. W. Lloyd. *Foundations of Logic Programming; (2nd extended ed.).* Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[124] P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Planning partially for situated agents. In *Proc. of the 5th Computational Logic in Multi-Agent Systems*, pages 230–248, Lisbon, Portugal, 2004.

[125] P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Programming application in CIFF. In *Proc. of the 9th Logic Programming and Non-Monotonic Reasoning (LPNMR)*, Tempe, AZ, USA, 2007.

[126] P. Mancarella and G. Terreni. An abductive proof procedure handling active rules. In *Congress of the Italian Association for Artificial Intelligence (AI*IA)*, pages 105–117, 2003.

[127] P. Mancarella, G. Terreni, and F. Toni. Web sites verification: An abductive logic programming tool. In *Proc. of the 23rd International Conference on Logic Programming*, pages 434–435, 2007.

[128] W. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[129] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[130] J. McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationary Office.

[131] D. McGuinness and F. V. Harmelen. OWL web ontology language, 2003 . `http://www.w3.org/TR/owl-features/` seen on April 30th, 2008.

[132] P. Mello, E. Lamma, P. Torroni, M. Gavanelli, M. Alberti, M. Milano, and F. Chesani. A logic-based approach to model interaction amongst computees., 2003. Deliverable D5. SOCS Consortium.

[133] R. Miller and M. Shanahan. Some alternative formulations of the event calculus. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 452–490, London, UK, 2002. Springer-Verlag.

[134] J. Minker and C. Ruiz. On extended disjunctive logic programs. In *International Syposium on Methodologies for Intelligent Systems*, pages 1–18, 1993.

[135] L. Missiaen, M. Bruynooghe, and M. Denecker. CHICA, an abductive planning system based on event calculus. *J. Log. Comput.*, 5(5):579–602, 1995.

[136] I. Niemela and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 421–430, London, UK, 1997. Springer-Verlag.

[137] U. Nilsson and J. Maluszynski. *Logic, Programming, and Prolog.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[138] B. V. Nuffelen. SLDNFA-system. *The Computing Research Repository (CoRR)*, cs.AI/0003027, 2000.

[139] B. V. Nuffelen. *Abductive Constraint Logic Programming: Implementation and Applications.* PhD thesis, K. U. Leuven, 2004.

[140] B. V. Nuffelen and M. Denecker. Problem solving in ID-logic with aggregates: some experiments. *The Computing Research Repository (CoRR)*, cs.AI/0003030, 2000.

[141] J. Pearl. Embracing causality in formal reasoning. In *Proc. of the 6th National Conference on Artificial Intelligence (AAAI)*, pages 369–373, Seattle, WA, 1987.

[142] N. Pelov, E. D. Mot, and M. Denecker. Logic programming approaches for representing and solving constraint satisfaction problems: A comparison. In *Proc. of the 7th Logic Programming and Automated Reasoning*, pages 225–239, Reunion Island, France, 2000.

[143] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Nonmonotonic reasoning with well founded semantics. In *Proc. of the 8th International Conference on Logic Programming*, pages 475–489, 1991.

[144] C. S. Pierce. *The Collected Papers of Charles Sanders Peirce.* Harvard University Press, 1935.

[145] D. Poole. On the comparison of theories: Preferring the most specific explanation. In *Proc. of the 9th International Joint Conference on Artificial Intelligence*, pages 144–147, 1985.

[146] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1988.

[147] C. Preist and K. Eshghi. Consistency-based and abductive diagnoses as generalised stable models. In *Proc. of the 3rd Fifth Generation Computer Systems*, pages 514–521, Tokyo, Japan, 1992.

[148] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *Proc. of the 3rd Principles of Knowledge Representation and Reasoning (KR) International Conference*, pages 439–449, Cambridge, Massachusetts, USA, 1992.

[149] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proc. of the 1st International Conference on Multiagent Systems (ICMAS)*, pages 312–319, San Francisco, CA, USA, 1995.

[150] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A system for efficiently computing WFS. In *Proc. of the 4th Logic Programming and Non-Monotonic Reasoning (LPNMR)*, pages 431–441, 1997.

[151] R. Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76. Plenum Press, 1977.

[152] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[153] R. Reiter. On asking what a database knows. In J. W. Lloyd, editor, *Proceedings of the Symposium on Computational Logic*, pages 96–113. Springer-Verlag, Berlin, Heidelberg, 1990.

[154] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[155] D. Saccà and C. Zaniolo. Deterministic and non-deterministic stable models. *J. Log. Comput.*, 7(5):555–579, 1997.

[156] F. Sadri and R. A. Kowalski. A theorem-proving approach to database integrity. In *Foundations of Deductive Databases and Logic Programming.*, pages 313–362. Morgan Kaufmann, 1988.

[157] F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In *Congress of the Italian Association for Artificial Intelligence (AI*IA)*, pages 49–60, 1999.

[158] F. Sadri, F. Toni, and P. Torroni. An abductive logic programming architecture for negotiating agents. In *Proc. of the 8th European Conference on Logics in Artificial Intelligence (JELIA)*, pages 419–431, Cosenza, Italy, 2002.

[159] T. Sato. Completed logic programs and their consistency. *J. Log. Program.*, 9(1):33–44, 1990.

[160] K. Satoh. Using two level abduction to decide similarity of cases. In *Proc. of the 13th European Conference on Artificial Intelligence*, pages 398–402, 1998.

[161] R. Schindlauer. Nonmonotonic logic programs for the Semantic Web. In *Proc. of the 21st International Conference on Logic Programming*, pages 446–447, 2005.

[162] B. Selman and H. J. Levesque. Abductive and default reasoning: A computational core. In *Proc. of the 8th National Conference on Artificial Intelligence (AAAI)*, pages 343–348, Boston, Massachusetts, 1990.

[163] M. Sergot. A query-the-user facility for logic programming. In *New horizons in educational computing*, pages 145–163. Halsted Press, New York, NY, USA, 1984.

[164] M. Shanahan. An abductive event calculus planner. *J. Log. Program.*, 44(1-3):207–240, 2000.

[165] M. Shanahan. Perception as abduction: Turning sensor data into meaningful representation. *Cognitive Science*, 29(1):103–134, 2005.

[166] P. Simons. Extending and implementing the stable model semantics. Research Report 58, Helsinki University of Technology, Helsinki, Finland, 2000.

[167] SOCS-consortium. Societies of computees (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. IST200132530. `http://lia.deis.unibo.it/Research/SOCS/`.

[168] T. Swift. A new formulation of tabled resolution with delay. In *Proc. of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, pages 163–177, Évora, Portugal, 1999.

[169] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[170] E. Teniente and T. Urp. On the abductive or deductive nature of database schema validation and update processing problems. *Theory Pract. Log. Program.*, 3(3):287–327, 2003.

[171] G. Terreni. The CIFF System 4.0: User manual. . `http://www.di.unipi.it/$\sim$terreni/research.php`.

[172] F. Toni. A semantics for the Kakas-Mancarella procedure for abductive logic programming. In *Joint Conference on Declarative Programming (GULP-PRODE)*, pages 231–244, Marina di Vietri, Italy, 1995.

[173] F. Toni. Automated information management via abductive logic agents. *Telematics and Informatics*, 18(1):89–104, 2001.

[174] W3C CConsortium. The RDF official web site. `http://www.w3.org/RDF/` seen on April 30th, 2008.

[175] W3C CConsortium. The Semantic Web official web site. `http://www.w3.org/2001/sw/` seen on April 30th, 2008.

[176] W3C Consortium. The OWL official web site. `http://www.w3.org/2004/OWL/` seen on April 30th, 2008.

[177] W3C Consortium. The XML Schema official web site. `http://www.w3.org/XML/Schema` seen on April 30th, 2008.

[178] M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming. Technical report, Imperial College, London, UK, 1997.

[179] D. H. D. Warren. An abstract Prolog instruction set., 1983. Technical Note 309. SRI International, Menlo Park, CA.

[180] G. Wetzel, R. A. Kowalski, and F. Toni. PROCALOG - programming with constraints and abducibles in logic (poster abstract). In *Joint International Conference and Symposium on Logic Programming*, page 535, Bonn, Germany, 1996.

[181] J. Wielemaker. An overview of the SWI-Prolog programming environment. In *Proc. of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Mumbai, India, 2003.

[182] J. Wielemaker, Z. Huang, and L. van der Meij. Swi-prolog and the web. *Theory and Practice of Logic Programming*, 2008. Accepted for publication.