THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Présentée et soutenue par :
Antoun Yaacoub

le mercredi 28 novembre 2012

Titre :

Flux de l'Information en Programmation Logique

École doctorale et discipline ou spécialité :

ED MITT : Domaine STIC : Intelligence Artificielle

Unité de recherche :

Institut de Recherche en Informatique de Toulouse

Directeur(s) de Thèse :

Mr Philippe BALBIANI	Directeur de recherche CNRS - Université Paul Sabatier	Directeur de thèse
Mr Ali AWADA	Professeur - Université Libanaise	Codirecteur de thèse

Jury (noms, prénoms et qualité des membres) :

Mme Olga KOUCHNARENKO	Professeur - Université Franche-Comté	Rapportrice
Mr Nicola OLIVETTI	Professeur - Université Paul Cézanne	Rapporteur
Mr Olivier GASQUET	Professeur - Université Paul Sabatier	Président

INFORMATION FLOW IN LOGIC PROGRAMMING

Thesis presented and defended by

Antoun YAACOUB

On the 28th of November 2012

To obtain the degree of

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Delivered by: Université Toulouse III Paul Sabatier (UPS)

Speciality: Computer Science

Advisors

Philippe Balbiani
Directeur de Recherche CNRS
Université Paul Sabatier

Ali Awada
Professeur
Université Libanaise

Reviewers

Olga Kouchnarenko
Professeur
Université Franche-Comté

Nicola Olivetti
Professeur
Université Paul Cézanne

President

Olivier Gasquet
Professeur
Université Paul Sabatier

École doctorale:

Mathématiques Informatique Télécommunications de Toulouse (MITT)

Unité de recherche:

Institut de recherche en Informatique de Toulouse (IRIT)

To my father - À la mémoire de mon père

Contents

Acknowledgements

I would like, first and foremost, to express my deepest gratitude to, my thesis advisor, Philippe Balbiani because he allowed me to fulfill one of my most cherished dreams. Thanks to him, I was able to do my PhD in a very pleasant environment. I thank him for his patience, advices and especially for sharing his passion for research. His high competence combined with his human qualities provided me a very valuable supervision.

I cordially thank Ali Awada for his help and his constant support during my thesis by encouraging me to pursue my PhD thesis adventure.

I would like to thank too both Olga Kouchnarenko and Nicola Olivetti for accepting to review my thesis, for their relevant remarks and detailed reviews and for their positive and encouraging reports. I also thank Olivier Gasquet for the honor and for his interest in my work by agreeing to chair my thesis committee.

I would like to acknowledge the "Association Libanaise pour le Développement Scientifique" (ALDS) for the financial support of my thesis.

I firmly believe that the work environment makes the greater part of the learning experience and for this I would like to thank my colleagues in the Laboratory of Logic, Interaction, Language and Computation (LILaC). Thank you especially to my 3 years office mate, Frédéric Moisan.

Out with the work setting, I would like to offer my fondest regards to my friends: Soha Zeitouny, Abir Awada, Tannous Gemayel, Abir Slim, Khodor El Koujok, Maha Zeitouni, Sasy Hanna. You are my second family. A special thanks to General Tannous Mouawad for his support, his numerous phone calls and his encouragement.

I would like to thank too my small family for their love, for their daily support, and for their encouragement for me to come to France in order to achieve my dream. To them all, I dedicate this thesis.

Finally, let all those who have contributed, directly or indirectly, to the development of this work found here the expression of my full gratitude.

Abstract

This thesis is developed in order to tackle the issue of information flow in logic programming. The contributions of this thesis can be split into three mains parts:

- **Information flow in logic programming**: we propose a theoretical foundation of what could be an information flow in logic programming. Several information flow definitions (based on success/failure, substitution answers, bisimulation between resolution trees of goals) are stated and compared. Decision procedures are given for each definition and complexity is studied for specific classes of logic programs.
- **Bisimulation of logic goals**: We introduce the concept of bisimulation between Datalog goals: two Datalog goals are bisimilar with respect to a given Datalog program when their SLD-trees, considered as relational structures, are bisimilar. We address the problem of deciding whether two given goals are bisimilar with respect to given programs. When the given programs are hierarchical or restricted, this problem is decidable in 2EXPTIME.
- **Preventive inference control for deductive databases**: We propose a secure and a precise security mechanism for deductive databases based on the notion of information flow in logic programming.

Keywords: Logic programming, Information flow, Datalog, Equivalence of goals, Bisimulation, Decision method, Computational complexity, Security mechanisms, Security policy, Inference control, Information system, Deductive database.

Résumé

Cette thèse est développée dans le but d'aborder la question du flux de l'information en programmation logique. Les contributions de cette thèse peuvent être divisées en trois parties:

- **Flux de l'information en programmation logique**: Nous proposons une base théorique de ce que pourrait être un flux de l'information en programmation logique. Plusieurs définitions de flux d'information (basées sur la réussite / échec, les substitutions réponses, bisimulation entre les arbres de résolution des buts logiques) sont évaluées et comparées. Des problèmes de décision sont donnés pour chaque définition et la complexité est étudiée pour certaines catégories de programmes logiques.

- **Bisimulation de buts logiques**: Nous introduisons la notion de bisimulation entre les buts Datalog: deux buts Datalog sont bisimilaires par rapport à un programme Datalog donné lorsque leurs SLD-arbres, considérés comme des structures relationnelles, sont bisimilaires. Nous abordons le problème de décider si deux buts donnés sont bisimilaires à l'égard d'un programme donné. Lorsque les programmes sont hiérarchiques ou *restricted*, ce problème est décidable en 2EXPTIME.

- **Contrôle préventif de l'inférence dans les bases de données déductives**: Nous proposons un mécanisme de sécurité sûr et précis pour les bases de données déductives basé sur la notion de flux de l'information dans la programmation logique.

Mots-clés: Programmation logique, Flux de l'information, Datalog, Equivalence de buts, Bisimulation, Méthodes de décision, Complexité, Mécanisme de sécurité, Politique de sécurité, Contrôle d'inférence, Système d'information, Bases de données déductives.

Chapter 1

Aims and Results

In this chapter, we position our work and outline the contents of this thesis. We explain our methodology, which is to apply the notion of information flow for security systems in a logic programming setting, and we point out the main influences. We also list the main results of this thesis.

1.1 Motivation

The theory of **deductive Databases** is well studied especially by seeing the emergence of efficient and easy to use systems that support **queries**, **reasoning**, and **application development** on databases through **declarative logic-based languages**.

Building on solid theoretical foundations, the field has benefited in the recent years of important advances.

The earliest work in the 70s focused on establishing the theoretical foundations for the field (a review on the impacts that this work had on other disciplines of computing and database area can be found in [32]).

In the 80s, significant system-oriented developments took place in two fields very close to deductive databases, causing a limit on the system implementations of this latter idea. The first field was relational databases, where systems featuring logic-based query languages of good performance, but limited expressive power, were becoming very successful in the commercial world. The second field is Logic Programming, where successive generations of Prolog systems were demonstrating performance and effectiveness in a number of symbolic applications, ranging from compiler writing to expert systems.

Suddenly, a renewed interest in deductive database systems came about by realizing that the rule based reasoning of logic, combined with the capability of database systems of managing and efficiently storing and retrieving large amounts of information could provide the basis on which to build the next-generation of knowledge base systems. Consequently, several works focused on coupling for example Prolog systems with relational databases [15, 42, 47], leading to a multitude of commercial systems that support this scheme.

This progress is demonstrated by the completion of prototype systems offering such levels of generality, performance and robustness that they support well complex application development. Thus, it became easy to learn about algorithms and architectures for building powerful deductive database systems, and to understand the programming environments and paradigms they are conducive to.

Thus, several application areas have been identified where these systems are particularly effective, including areas well beyond the domain of traditional database applications. One of these application is the need to maintain security in these databases. In multi-user databases, where different users are allowed to access different pieces of data, mechanisms must be in place where all users can access data that they are allowed to access, unless doing so would violate security. Several works focused on this issue. For example, Bonatti *et al.* [10] developed two models of interaction between the user and the database called "yes-no" dialogs, and "yes-no-don't know" dialogs. Briefly, both dialog frameworks allow the database to lie to the user.

Thus, our aim in this thesis, is to provide a model of interaction in deductive databases that preserves the database security while minimizing the number of denials answers.

1.2 Methodology

In order to fulfill our aim, we used logic programming setting to represent deductive databases. We adapted the notions of information flow for imperative programming to first-order logic programming. The main difficulty in adapting these definitions is that variables in logic programs behave differently from variables in conventional programming languages. They stand for an unspecified but single entity rather than for a store location in memory. For this, we were forced to propose new definitions that fit with logic programming scheme. Once done, a natural question arose whether it is decidable to decide of the existence of such a flow in logic programs. Unfortunately, this question is undecidable in the general setting. This led us to consider several types of logic programs for which the existence of the flow became decidable. Even by considering these types of logic programs, we were not able to answer another question that concerns if two logic goals are bisimilar. For this, we were forced to deal with SLD-trees with infinite branches. We were rescued by using loop checking techniques. These techniques detect some repetition in the SLD-derivations. We used these techniques jointly with our adapted definition of information flow to decide if two goals are bisimilar. We studies this for some type of logic programs, as this question is undecidable in the general setting. Lastly, in order to present a secure and precise mechanism for protecting deductive databases, we extended our definitions of flow and presented the definitions of confidentiality policies, secure and precise mechanisms in a logic programming context.

1.3 Results

Below, we list the main contributions of the thesis:

- **Information flow in logic programming**: we propose three definitions of information flows in logic programs. These definitions correspond to what can be observed by the user when a query $\leftarrow G(x, y)$ is run on a logic program P. We consider first that the user only sees whether her queries succeed or fail. In this respect, we say information flows from x to y in G when there exists constants a, b such that $P \cup \{\leftarrow G(a, y)\}$ succeeds whereas $P \cup \{\leftarrow G(b, y)\}$ fails. Then, we assume that the user has also access to the sets of substitution answers computed by the interpreter with respect to her queries. As a result, in this case, there is a flow of information from x to y in G if there are constants a, b such that the set of substitution answers of $P \cup \{\leftarrow G(a, y)\}$ and $P \cup \{\leftarrow G(b, y)\}$ are different. Lastly, we suppose that the user, in addition to the substitution answers, also observes the SLD-refutation trees produced by the interpreter. If the SLD- trees of the queries $P \cup \{\leftarrow G(a, y)\}$ and $P \cup \{\leftarrow G(a, y)\}$ can be distinguished (i.e. non bisimilar in our context) in one way or another by the user, then we will say that information flows from x to y in G.

- **Undecidability / decidability of the flow in logic programming**: For a logic program P and a two variables goal $\leftarrow G(x, y)$, determining whether there is a flow of information from x to y relatively to the three definitions of the flow is undecidable in the general setting. The problem became decidable in EXPTIME for Datalog programs relatively to the definitions of flow based on success/failure and substitution answers, and the same problem is in $\Delta 2P$ for binary hierarchical Datalog programs relatively to the definition of flow based on success/failure.

- **Bisimulation of logic goals**: A natural question to ask was to decide if two goals in logic programs are bisimilar. We showed that this decision problem is undecidable for Prolog programs and it became decidable in 2EXPTIME for hierarchical and restricted Datalog logic programs.

- **Preventive inference control for deductive databases**: We propose a secure and a precise security mechanism for deductive databases based on the notion of information flow in logic programming and its extension to the notion of levels. A formal proof is given, showing that the security mechanism proposed is secure.

Some material from this thesis has been published in referred conferences and journals. The notion of information flow in logic programming has been described

in [JIAF 2011], and the result about bisimulation have been published in [JeLIA 2012].

1.4 Thesis Outline

The rest of this thesis is organized as follows:

In chapter 2, we present the basic components for data security. We explore security policies by talking about confidentiality, noninterference, nondeducibility and integrity policies. We describe some security mechanism implementations, namely, access control, information flow and inference control.

In chapter 3, we describe the relevant background material, including an overview of first-order logic programming. We present also loop checking concept and show soundness and completeness results for some loop checking techniques. We end with an overview of control flow and dependence analysis in logic programming.

In chapter 4, we give three definitions of information flow in logic programming based respectively, on success/failure, substitution answers and bisimulation. We explore the links between these definitions and prove the non-transitivity of the flow. We show the undecidability of the existence of a flow relatively for a general logic program and a goal. We give complexity and decidability results for other restricted types of logic programs.

In chapter 5, we prove first undecidability of logic goals bisimulation for Prolog programs, and later decidability of bisimulation of goals for hierarchical and restricted Datalog logic programs.

In chapter 6, we go further beyond the definitions of information flow and we present the notion of level of indistinguishability of the flow. We apply this notion to prevent illicit inferences for information systems by presenting a secure and precise security mechanism.

In chapter 7, we conclude with a summary of our thesis and present some future works.

Chapter 2

Data Security

Data security is the science and study of methods of protecting data in computer and communication systems from unauthorized disclosure and modification. One of the aspects of data security is the control of information flow in the system. In some sense, an information flow should describe controls that regulate the dissemination of information. These controls are needed to prevent programs from leaking confidential data, or from disseminating classified data to users with lower security clearances.

We begin with basic security-related aspects in section 2.1, namely the **confidentiality**, **integrity**, and **availability** conditions. While confidentiality focuses on preventing disclosure of information to unauthorized users, integrity means that data cannot be modified undetectably. Availability addresses the fact that the information must be available when it is needed. Section 2.2 discusses **security policies** that identify the threats and define the requirements for ensuring a secure system. We review the major security policies, namely, **confidentiality** and **integrity** policies respectively in subsections 2.2.2 and 2.2.3.

Section 2.2.2 presents **confidentiality policies**. These policies, as their noun suggests, emphasize the protection of confidentiality, as, the **Bell-LaPadula Model**, **noninterference** and **nondeducibility**. **Integrity policies**, which focuses on the incorruptibility of the data, are mentioned in section 2.2.3.

Another aspect of confidentiality is to ensure that no high-level information is visible, or can be deduced, by a low-level process. Implementation of security is discussed in section 2.3, which can be done through **cryptography** and/or by **sharing rights and information**. We will focus in section 2.3 on this latter mechanism by presenting briefly in subsections 2.3.1 and 2.3.3 **access control**

mechanism and **inference control mechanism** respectively. Section 2.3.2 will be devoted to **information flow checking mechanisms for imperative programming**. For this, we begin by exposing the theory of information flow in imperative programming. We discuss compiler and execution based mechanisms that determine if an information flow in an imperative program could violate a given information flow policy. We will end the section with a concrete example on how to control information flows in practice.

2.1 Basic Components

Data Security is based on three major aspects: **confidentiality**, **integrity**, and **availability**. The interpretations of these aspects vary according to the contexts in which they arise.

- **Confidentiality** represents the dissimulation of information (**data / resources** and its **existence**). The importance of this aspect arises from the need to keep information secret in sensitive fields such as government and industry. For example, all types of institutions keep personnel records secret.

  The concealment of the existence of data, which is sometimes more revealing than the data itself, is considered too as an aspect of confidentiality. For example, knowing the exact squandering amount in a government is less important than knowing that such practice occurred.

  Another important aspect of confidentiality is resource hiding. Some organizations, for example, using equipment without authorization, may not wish to conceal their configuration publicly, as they will be faced legal charges.

  Different mechanisms support confidentiality:

  - Access control mechanisms: One of the access control mechanisms is cryptography. **Cryptography** scrambles data to make it incomprehensible. Nevertheless, as a *cryptographic key* is needed to retrieve the original data from the scrambled one, the *cryptographic key* itself becomes another datum to be protected.
  - System-dependent mechanisms: These mechanisms prevent processes from illicitly accessing information. However, they can protect the secrecy of data more completely than cryptography, but when the controls fail or are bypassed, the protected data can be read.

- **Integrity** prevents unauthorized changes of data or resources. It includes data integrity (the content of the information) and origin integrity (the source of the data, often called authentication). For example, a local newspaper may print information obtained from Wikileaks but attribute it to the wrong source. The information is printed as received (preserving data integrity), but its source is incorrect (corrupting origin integrity).

  Mainly, two mechanisms support integrity:

  - **Prevention mechanisms**: it seek to maintain the integrity of the data by:

    - ∗ **blocking any unauthorized attempts to change the data** (the case where a user tries to change data which she has no authority to change; circumvented by the use of adequate authentication and access controls) or
    - ∗ **any attempts to change the data in unauthorized ways** (the case where a user authorized to make certain changes in the data tries to change the data in other ways. Preventing this attempt requires very different controls).
  - **Detection mechanisms**: it simply report that the data's integrity is no longer trustworthy, by analyzing for example: the system events, or if the constraints on data are still holding.

- **Availability** refers to the ability to use the information or resource desired. This means that the computing systems used to store and process the information, the security controls used to protect it, and the communication channels used to access it must be functioning correctly. The aspect of availability that is relevant to security is that someone may deliberately arrange to deny access to data or to a service by making it unavailable. It is worth noting that attempts to block availability (called denial of service attacks) can be the most difficult to detect, because the analyst must determine if the unusual access patterns are attributable to deliberate manipulation of resources or of environment.

As we have stated, confidentiality checks if the data is compromised or not, while integrity, except that it checks the data correctness and trustworthiness, it deals also with the origin of the data, and how it was protected. Thus, evaluating integrity is often very difficult because it relies on assumptions about the source of the data. Formally, to state what is and what is not allowed in data security, security policies were introduced.

2.2 Security Policies

A security policy defines what *secure* means for a system or a set of systems. Security policies can be informal or highly mathematical in nature. In fact, policies may be presented mathematically, as a list of allowed (secure) and disallowed (nonsecure) states. In practice, policies are rarely so precise; they normally describe in natural language what users and staff are allowed to do. The ambiguity inherent in such a description leads to states that are not classified as *allowed* or *disallowed*.

2.2.1 Foundations of Security Policies

We consider a computer system to be a **finite-state automaton** with a set of transition functions that change state. Then:

Definition 2.2.1 (**Security policy**). *A* ***security policy*** *is a statement (usually written in natural language) that partitions the states of the system into a set of* ***secure/authorized states*** *and a set of* ***nonsecure/unauthorized states****.*

A security policy sets the context in which we can define a secure system. What is secure under one policy may not be secure under a different policy. More precisely:

Definition 2.2.2 (**Secure system**). *A* ***secure system*** *is a system that, initially, is in an authorized state and cannot be in any unauthorized state after one or several transitions.*

Consider the finite-state machine in Figure 2.1. It consists of four states and five transitions. The security policy partitions the states into a set of authorized states $A = \{s_1, s_2\}$ and a set of unauthorized states $UA = \{s_3, s_4\}$. This system is not secure, because regardless of which authorized state it starts in, it can enter an unauthorized state. However, if the edge from s_1 to s_3 were not present, the system would be secure, because it could not enter an unauthorized state from an authorized state.

Definition 2.2.3 (**Breach of security**). *A* ***breach of security*** *occurs when a system enters an unauthorized state.*

A security policy considers all relevant aspects of confidentiality, integrity, and availability. With respect to **confidentiality**, it identifies those states in which information leaks to those not authorized to receive it. This includes the illicit transmission of information, called *information flow*. Also, the policy must handle dynamic changes of authorization, so it includes a temporal element. For example, a contractor working for a company may be authorized to access proprietary information during the lifetime of a nondisclosure agreement, but

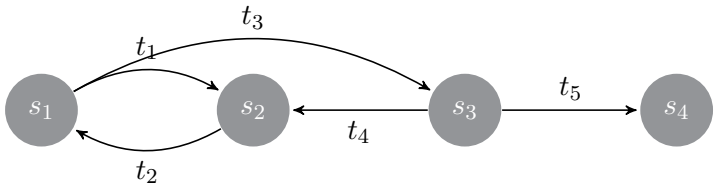


Figure 2.1: A finite-state machine. In this example, the authorized states are s_1 and s_2.

when that nondisclosure agreement expires, the contractor can no longer access that information. This aspect of the security policy is often called a *confidentiality policy.*

With respect to **integrity**, a security policy identifies authorized ways in which information may be altered and entities authorized to alter it. Authorization may derive from a variety of relationships, and external influences may constrain it; for example, in many transactions, a principle called *separation of duties* forbids an entity from completing the transaction on its own. Those parts of the security policy that describe the conditions and manner in which data can be altered are called the *integrity policy.*

With respect to **availability**, a security policy describes what services must be provided. It may present parameters within which the services will be accessible (for example, that a browser may download Web pages but not Java applets). It may require a level of service (for example, that a server will provide authentication data within 1 minute of the request being made). This relates directly to issues of quality of service.

The statement of a security policy may formally state the desired properties of the system. If the system is to be provably secure, the formal statement will allow the designers and implementers to prove that those desired properties hold. If a formal proof is unnecessary or infeasible, analysts can test that the desired properties hold for some set of inputs.

In practice, a less formal type of security policy defines the set of authorized states. Typically, the security policy assumes that the reader understands the context in which the policy is issued - in particular, the laws, organizational policies, and other environmental factors. The security policy then describes conduct, actions, and authorizations defining "authorized users" and "authorized use."

A security policy defines what information is to be protected; it has a non-procedural form. For example, a security policy might state that a user is not to obtain "top secret" information. In contrast, a protection mechanism defines how information is to be protected; it has a procedural form. For example, a protection mechanism might check each operation performed by a user.

As a second example, consider that a university has a non-cheating policy stating that no student may copy another student's homework. One mechanism is the file access controls; if the second student had set permissions to prevent the first student from reading the file containing her homework, the first student could not have copied that file.

In computer security, a distinction is made between policy and mechanism. Recall that a security policy is a statement of what is, and what is not, allowed. Mechanisms can be nontechnical, such as requiring proof of identity before changing a password; in fact, policies often require some procedural mechanisms that technology cannot enforce.

In the remainder of this section, we will present briefly **confidentiality**, **integrity**, **noninterference** and **nondeducibility** policies, but before this, we need to recall some basic notions concerning lattices.

Lattices

A lattice is a mathematical construction built on the notion of a group. First, we review some basic terms. Then we discuss lattices.

Basics

For a set S, a relation R is any subset of $S \times S$. For convenience, if $(a, b) \in R$, we write aRb.
For example, let $S = \{1, 2, 3\}$. Then the relation less than or equal to (written $\leq$) is defined by the set $R = \{(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)\}$. We write $1 \leq 1$ and $2 \leq 3$ for convenience, because $(1, 2) \in R$ and $(2, 3) \in R$, but not $3 \leq 2$, because $(3, 2) \notin R$.

The following definitions describe properties of relations.

Definition 2.2.4. *A relation R defined over a set S is reflexive if aRa for all $a \in S$.*

Definition 2.2.5. *A relation R defined over a set S is antisymmetric if aRb and bRa imply $a = b$ for all $a, b \in S$.*

Definition 2.2.6. *A relation R defined over a set S is transitive if aRb and bRc imply aRc for all $a, b, c \in S$.*

For example, consider the set of complex numbers C. For any $a \in C$, define a_R as the real component and a_I as the imaginary component (that is, $a = a_R + a_I i$). Let $a \leq b$ if and only if $a_R \leq b_R$ and $a_I \leq b_I$. This relation is reflexive, antisymmetric, and transitive.

A **partial ordering** occurs when a relation orders some, but not all, elements of a set. Such a set and relation are often called a **poset**. If the relation imposes an ordering among all elements, it is a **total ordering**.
For example, the relation **less than or equal to**, as defined in the usual sense, imposes a total ordering on the set of integers, because, given any two integers, one will be less than or equal to the other. However, the relation in the preceding example imposes a partial ordering on the set C. Specifically, the numbers $1 + 4i$ and $2 + 3i$ are not related under that relation (because $1 \leq 2$ but $4 \not\leq 3$).

Under a partial ordering (and a total ordering), we define the **upper bound** of two elements to be any element that follows both in the relation.

Definition 2.2.7. *For two elements $a, b \in S$, if there exists a $u \in S$ such that aRu and bRu, then u is an **upper bound** of a and b.*

A pair of elements may have many upper bounds. The one *closest* to the two elements is the **least upper bound**.

Definition 2.2.8. *Let U be the set of upper bounds of a and b. Let $u \in U$ be an element such that there is no $t \in U$ for which tRu. Then u is the **least upper bound** of a and b (written $lub(a, b)$ or $a \bigotimes b$).*

Lower bounds, and greatest lower bounds, are defined similarly.

Definition 2.2.9. *For two elements $a, b \in S$, if there exists an $l \in S$ such that lRa and lRb, then l is a **lower bound** of a and b.*

Definition 2.2.10. *Let L be the set of lower bounds of a and b. Let $l \in L$ be an element such that there is no $m \in L$ for which lRm. Then l is the **greatest lower bound** of a and b (written $glb(a, b)$ or $a \bigoplus b$).*

For example, consider the subset of the set of complex numbers for which the real and imaginary parts are integers from 0 to 10, inclusive, and the relation defined in the second example in this appendix. The set of upper bounds for $1 + 9i$ and $9 + 3i$ is $\{9 + 9i, 9 + 10i, 10 + 9i, 10 + 10i\}$. The least upper bound of $1 + 9i$ and $9 + 3i$ is $9 + 9i$. The set of lower bounds is $\{1 + 1i, 1 + 0i, 0 + 0i\}$. The greatest lower bound is $1 + 1i$.

Lattices

A **lattice** is the combination of a set of elements S and a relation R meeting the following criteria.

1. R is reflexive, antisymmetric, and transitive on the elements of S.
2. For every $s, t \in S$, there exists a greatest lower bound.
3. For every $s, t \in S$, there exists a least upper bound.

For example, the set $\{0, 1, 2\}$ forms a lattice under the relation "less than or equal to" ($\leq$). By the laws of arithmetic, the relation is reflexive, antisymmetric, and transitive. The greatest lower bound of any two integers is the smaller, and the least upper bound is the larger. Back to the example of the set of complex numbers. Let $a \leq b$ if and only if $a_R \leq b_R$ and $a_I \leq b_I$. This set and relation define a lattice, because, as it shown before, the relation is reflexive, antisymmetric, and transitive and any pair of elements a, b have a least upper bound and a greatest lower bound.

2.2.2 Confidentiality Policies

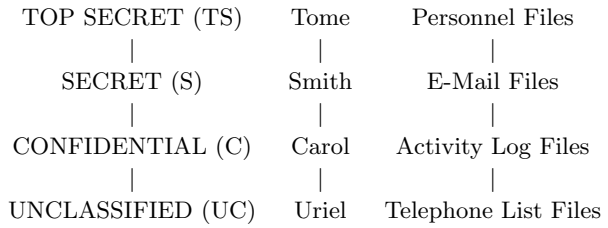
The main aim of a confidentiality policy is to prevent the unauthorized disclosure of confidential information. Confidentiality policies are also called **information flow policies**. For example, the navy must keep confidential the date on which a troop ship will sail. If the date is changed, the redundancy in the systems and paperwork should catch that change. But if the enemy knows the date of sailing, the ship could be sunk. Because of extensive redundancy in military communications channels, availability is also less of a problem.

The Bell-LaPadula Model

The simplest type of confidentiality classification is a set of **security clearances/security classes** arranged in a linear (total) ordering (see example 2.1). These clearances represent sensitivity levels. The higher the security clearance, the more sensitive the information (and the greater the need to keep it confidential). A subject has a security clearance. The goal of the Bell-LaPadula security model [4, 46] is to prevent read access to objects at a security classification higher than the subject's clearance.

Example 2.1 Security Clearances - security classification

We consider the following confidentiality classification, security levels and clearances. At the left is the basic confidentiality classification system. The four security levels are arranged with the most sensitive at the top and the least sensitive at the bottom. In the middle are individuals grouped by their security clearance. At the right is a set of documents grouped by their security levels.



From this figure, one can read and deduce the following:

- Carol's security clearance is C (CONFIDENTIAL).
- Tome's security clearance is TS (TOP SECRET).
- The security classification of the object "E-mail files" is S (SECRET).
- The security classification of the object "Telephone list files" is UC (UNCLASSIFIED).
- Carol cannot read personnel files.
- Tome can read the activity log files.
- Tome (with a security clearance of TS) cannot write to the activity log files (with a security classification of C) and thus she cannot copy the content of the personnel files into the activity log files and thus neither Carol can read the personnel files.

The Bell-LaPadula Model forbids reading of higher-level objects (called *the simple security condition*) and writing to lower-level objects (called *the *-property*). However, writing can take many forms. For example, suppose two users are sharing a single system. The users are separated, each one having a virtual machine, and they cannot communicate directly with one another. However, the CPU is shared on the basis of load. If user Abby (cleared for SECRET) runs a CPU-intensive program, and user Bob (cleared for CONFIDENTIAL) does not, Abby's program will dominate the CPU. This provides a covert channel through which Abby and Bob can communicate. They agree on a time interval and a starting time (say, beginning at noon, with intervals of 1 minute). To transmit a 1-bit, Abby runs his program in the interval; to transmit a 0-bit, Abby does not. Every minute, Bob tries to execute a program, and if the program runs, then Abby's program does not have the CPU and the bit is 0; if the program does not run in that interval, Bob's program has the CPU and the transmitted bit is 1. Although not "writing" in the traditional sense, information is flowing from Abby to Bob in violation of the Bell-LaPadula Model's constraints.

This example demonstrates the difficulty of separating policy from mechanism. In the abstract, the CPU is transmitting information from one user to another. This violates the *-property, but it is not writing in any traditional sense of the word, because no operation that alters bits on the disk has occurred. So, either the model is insufficient for preventing Bob and Abby from communicating, or the system is improperly abstracted and a more comprehensive definition of "write" is needed. This is one problem, and in what follows, exploring it will lead to the notions of **noninterference** and **nondeducibility**.

Noninterference

The previous example suggests an alternative view of security phrased in terms of *interference*. A system is secure if groups of subjects cannot interfere with one another. In the same example, the "interference" would be Abby's interfering with Bob's acquiring the CPU for her process. Goguen and Meseguer [35] used this approach to define security policies. A security policy, based on noninterference, would describe states in which forbidden interferences do not occur.

Nondeducibility

Goguen and Meseguer [35] characterize security in terms of state transitions. If state transitions caused by high-level commands interfere with a sequence of transitions caused by low-level commands, then the system is not noninterference-secure. But their definition skirts the intent of what a secure system is to provide. The point of security, in the Bell-LaPadula sense, is to restrict the flow of information from a high-level entity to a low-level entity. That is, given a set of low-level outputs, no low-level subject should be able to deduce anything about the high-level outputs. Sutherland [68] reconsidered this issue in these terms.

Consider a system as a "black box" with two sets of inputs, one classified High and the other Low. It also has two outputs, again, one High and the other Low.

If an observer cleared only for Low can take a sequence of Low inputs and Low outputs, and from them deduce information about the High inputs or outputs, then information has leaked from High to Low.

2.2.3 Integrity Policies

As much of the companies and firms are more concerned with accuracy than with disclosure, in the sense that a company can function correctly even if its confidential data is released, integrity policies, as their name suggests, focus on

integrity rather than on confidentiality. Three major integrity security policies were proposed in the literature, namely, the Biba [6], Lipner [49] and Clark-Wilson [19] integrity models.

Integrity policies take into account concepts like the separation of duty, the separation of function and auditing, concepts that are beyond the scope of confidentiality security policies.

2.3 Implementation

Implementation of data security can be achieved through several means: **cryptography** mechanism by the principle of cryptosystems, key management, ciphering techniques and authentication; and mechanisms based on the **sharing of rights and information**. In the remainder of this section, we will be concerned only by the second type of mechanisms. We will talk briefly about access control mechanisms, information flow and inference control mechanisms.

2.3.1 Access control

In this section, we present the notion of **protection system**. In fact, a protection system describes the conditions under which a system is secure. Recall that the **state** of a system is the collection of the current values of all memory locations, all secondary storage, and all registers and other components of the system. The subset of this collection that deals with protection is the protection state of the system.

For example, the access control matrix [26, 36, 45] is the most precise tool that can describe the current protection state. It can express any expressible security policy. It characterizes the rights of each subject (active entity, such as a process) with respect to every other entity.

Briefly, the set of all protected entities is called the set of objects O. The set of subjects S is the set of active objects, such as processes and users. In the access control matrix model, the relationship between these entities is captured by a matrix A with rights drawn from a set of rights R in each entry $a[s, o]$, where $s \in S$, $o \in O$, and $a[s, o] \subseteq R$. The subject s has the set of rights $a[s, o]$ over the object o. The set of protection states of the system is represented by the triple (S, O, A). For example, the figure 2.2 shows the protection state of a system. The system has two processes and two files and the rights are *read*, *write* or *append*. Process 1 can read or write file 1 and can read file 2; process 2 can append to file 1 and read file 2.

	file 1	file 2
Process 1	read, write	read
Process 2	append	read

Figure 2.2: An access control matrix

We should note that the access control matrix mechanism in computer security is not used in practice because of space requirements (most systems have thousands of objects and could have thousands of subjects too).

2.3.2 Information Flow

A security policy, as it was discussed earlier in this thesis, regulates information flows in a system. It is one of the system jobs to ensure that these informations flows do not violate the constraints of the security policy. In this section, we will review the major mechanisms proposed in the literature for **imperative programming** to check information flows, namely, the compile-time mechanisms and the runtime mechanisms. We will end this section by giving an example on how to control information flows in practice.

Let us recall that confidentiality policies (i.e. information flow policies) define the authorized paths along which information can flow. Previously in Sections 2.2.2, we reviewed several models of information flow, namely the Bell-LaPadula Model, nondeducibility and noninterference models. Each model associates a label, representing a security class with information and with entities containing that information. Each model has rules about the conditions under which information can move throughout the system.

In the following, for an object in the system (an object may be a logical structure such as a file, record, or program variable, or it may be a physical structure such as a memory location, or a user), we shall write x for both the **name** and the **value** of x, and $\underline{x}$ for its **security class**. We will use the notation $\underline{x} \rightsquigarrow \underline{y}$ to mean that information can flow from an element of class x to an element of class y. A **policy** is represented by a set of such statements.

Compiler-Based Mechanisms

The mechanisms described in this section follow those developed in [24, 25]. The aim of compiler-based mechanisms is to check that information flows throughout a program are authorized, by determining if flows could violate a given information flow policy. Note that this determination is not precise (because secure paths of information flow could be marked as violating the policy), but it is secure (no unauthorized path along which information may flow will be undetected).

Definition 2.3.1 (Certification of statements)**.** *A set of statements is **certified** with respect to an information flow policy if the information flow within that set of statements does not violate the policy.*

Example 2.2 Certification of statements

Consider the following *if then else* statement, where x, y, a and b are variables:
if $x = 1$ *then*

$y := a;$
else
$y := b;$

Obviously, information flows from: x and a to y or from x and b to y.
Suppose now that the policy states that: $\underline{a} \rightsquigarrow \underline{y}$, $\underline{b} \rightsquigarrow \underline{y}$, and $\underline{x} \rightsquigarrow \underline{y}$, then the information flow is secure.

In the following:

- x : *integer class*$\{A, B\}$
  states that x is an integer variable and that data from security classes A and B may flow into x. Note that security classes are viewed as a lattice, and thus x's class must be at least the least upper bound of classes A and B (i.e. $lub\{A, B\} \rightsquigarrow \underline{x}$).
- *Low* and $High$ are two special classes representing the greatest lower bound and least upper bound, respectively, of the lattice.
- All constants of the programming language are of class *Low*.
- Information can be passed into or out of a procedure (i.e. a set of instructions that performs a specific task) through parameters (input i_s, output o_s and input/output io_s parameters).
- i_s : *type class*$\{i_s\}$ (the class of an input parameter is the class of the actual argument)
- o_s : *type class*$\{i_1, \cdots, i_k, io_1, \cdots, io_k\}$ (the class of an output parameter is the class of the input and output arguments because information can flow from any input parameter to any output parameter)
- io_s : *type class*$\{i_1, \cdots, i_k, io_1, \cdots, io_k\}$
- For an element $a[i]$ in an array, information flows from $a[i]$ and from i. Thus, the class involved is $lub\{a[i], i\}$. As for information flowing into $a[i]$, it affects only the value in $a[i]$, and the class involved is $a[i]$.

Next, we will review several types of program statements and their certification mechanism in imperative programming.

Assignment Statements
Consider an assignment statement of the following form: $y := f(x_1, \cdots, x_n)$ where y and $x_1, \cdots, x_n$ are variables and f is some function of those variables. Information flows from each of the x_i's to y. Hence, the requirement for the information flow to be secure is $lub\{\underline{x_1}, \cdots, \underline{x_n}\} \rightsquigarrow \underline{y}$.

Compound Statements
Consider a compound statement of the form:
begin
S_1;
...
S_n;
end;
where each of the S_i's is a statement. If the information flow in each of the statements is secure, then the information flow in the compound statement is secure.

Conditional Statements
Consider a conditional statement of the form:
$if\ f(x_1, \cdots, x_n)\ then$
S_1;
else
S_2;
end;
where $x_1, \cdots, x_n$ are variables and f is some (Boolean) function of those variables. One of the two statements S_1 or S_2 may be executed. This depends on the value of f (consequently, on the values of the variables $x_1, \cdots, x_n$). Thus, information must be able to flow from those variables to any targets of assignments in S_1 and S_2. This is possible if and only if the lowest class of the targets dominates the highest class of the variables $x_1, \cdots, x_n$. Thus, in order for the information flow to be secure: S_1 and S_2 must be secure and $lub\{\underline{x_1}, \cdots, \underline{x_n}\} \rightsquigarrow glb\{\underline{y} |$ y is the target of an assignment in S_1 and $S_2\}$.

Iterative Statements
Consider an iterative statement of the form:
$while\ f(x_1, \cdots, x_n)\ do$
S;
where $x_1, \cdots, x_n$ are variables and f is some (boolean) function of those variables. Note that an iterative statement contains implicitly a conditional statement, so the requirements for information flow to be secure for a conditional statement apply here. Nevertheless, secure information flow also requires that the loop terminate. In fact, if the program never leaves the iterative statement, statements after the loop will never be executed. In this case, information has flowed from the variables $x_1, \cdots, x_n$ by the absence of execution. Hence, secure information flow also requires that the loop terminate. For the information flow to be secure: the iterative statement must terminates, S must be secure and $lub\{\underline{x_1}, \cdots, \underline{x_n}\} \rightsquigarrow$

$glb\{\underline{y}|$ y is the target of an assignment in $S\}$.

Procedure Calls

Consider a procedure call of the form:
proc procname$(i_1, \cdots, i_m : int;\ var\ o_1, \cdots, o_n : int)$;
begin
    S;
end;
where each of the i_j's is an input parameter and each of the o_j's is an input/output parameter. Obviously, S must be secure, but in order to achieve this, relationships between input and output parameters must be captured too. Let $x_1, \cdots, x_m$ and $y_1, \cdots, y_n$ be the actual input and input/output parameters, respectively. For the information flow to be secure: S must be secure and for $j = 1, \cdots, m$ and $k = 1, \cdots, n$, if $\underline{i_j} \rightsquigarrow \underline{o_k}$ then $\underline{x_j} \rightsquigarrow \underline{y_k}$ and for $j = 1, \cdots, n$ and $k = 1, \cdots, m$, if $\underline{o_j} \rightsquigarrow \underline{o_k}$ then $\underline{y_j} \rightsquigarrow \underline{y_k}$

Example 2.3 Examples of flow certification in imperative programming

We will give examples of program certification for the different program statements previously presented:

- The requirement for the statement $x := y+z$; to be secure is that $lub\{\underline{y}, \underline{z}\} \rightsquigarrow \underline{x}$.
- The requirement for the compound statement
  S_1 $x := y + z$;
  S_2 $y := x * z$;
  to be secure is that $lub\{\underline{y}, \underline{z}\} \rightsquigarrow \underline{x}$ for S_1 and $lub\{\underline{x}, \underline{z}\} \rightsquigarrow \underline{y}$ for S_2.
- The requirement for the conditional statement
  if $x + y < z$ *then*
      S_1 $a := b$;
  else
      S_2 $y := x * z$;
  to be secure is that $\underline{b}, \rightsquigarrow \underline{a}$ for S_1 and $lub\{\underline{x}, \underline{z}\} \rightsquigarrow \underline{y}$ for S_2, and $lub\{\underline{x}, \underline{y}, \underline{z}\} \rightsquigarrow glb\{\underline{a}, \underline{y}\}$.

Exceptions and infinite loops can cause information to flow as shown in the following two cases:

- Consider the following procedure, which copies the (approximate) value of x to y.[1][1] From Denning [269], p. 306.
  proc copy$(x :\ int\ class\{x\};\ var\ y :\ int\ class\ Low)$;
  var sum $:\ int\ class\{x\};\ z :\ int\ class\ Low$;
  begin
      $z := 0$;
      $sum := 0$;
      $y := 0$;
      while $z = 0$ *do begin*
          $sum := sum + x$;

$y := y + 1;$
end
end
When *sum* overflows, a trap occurs. If the trap is not handled, the procedure exits. The value of x is $MAXINT/y$, where $MAXINT$ is the largest integer representable as an int on the system. At no point, however, is the flow relationship $\underline{x} \rightsquigarrow \underline{y}$ checked.

- The following procedure copies data from x to y. It assumes that x and y are either 0 or 1.
proc copy(x : *int* 0..1 *class* $\{x\}$; *var* y : *int* 0..1 *class Low*);
begin
$y := 0;$
while $x = 0$ *do*
$(*nothing*);$
$y := 1;$
end

If x is 0 initially, the procedure does not terminate. If x is 1, it does terminate, with y being 1. At no time is there an explicit flow from x to y.

Execution-Based Mechanisms

Before talking about execution-based mechanisms, it is time to define what we mean by implicit information flow.

Definition 2.3.2 (**Implicit information flows**). *An **implicit flow** of information occurs when information flows from x to y without an explicit assignment of the form $y := f(x)$, where $f(x)$ is an arithmetic expression with the variable x.*

As for execution-based mechanism, its aim is to prevent information flows that violate the policy. For explicit flows, and before the execution of the assignment $y = f(x_1, \cdots, x_n)$, the execution-based mechanism verifies that $lub(\underline{x_1}, \cdots, \underline{x_n}) \rightsquigarrow \underline{y}$. Thus, in the case where the condition is true, the assignment proceeds; otherwise, it fails. As for checking implicit flows, it is a bit more complicated because sometimes, statements may be incorrectly certified as complying with the confidentiality policy. For example, let x and y be two variables. The requirement for certification for a particular statement y *op* x is that $\underline{x} \rightsquigarrow \underline{y}$. The conditional statement:
if $x = 1$ *then*
$y := a;$
causes a flow from x to y. For the case where, $x \neq 1$, $\underline{x} = High$ and $\underline{y} = Low$, the implicit flow would not be checked.

In order to deal with implicit flows, Fenton [30] proposed an abstract machine call the Data Mark Machine.

Fenton's Data Mark Machine

Fenton's Data Mark Machine [30] handles, in fact, implicit flows at execution time. To each variable in his machine, he associates a security class (or as he calls it a **tag**). In order to treat implicit flows as explicit flows, Fenton included a tag for the program counter (PC - is a processor register that indicates where a computer is in its program sequence. PC is incremented after fetching an instruction, and holds the memory address of the next instruction that would be executed.) since branches are considered as assignments to the PC. Fenton defined the semantics of the Data Mark Machine. In the following:

- *skip* means that the instruction is not executed,
- $push(x, \underline{x})$ means to push the variable x and its security class $\underline{x}$ onto the program stack (the program stack is the memory set aside as scratch space for a thread of execution),
- $pop(x, \underline{x})$ means to pop the top value and security class off the program stack and assign them to x and $\underline{x}$, respectively.

Fenton defined five instructions that a sufficient computer needs only. The relationships between execution of the instructions and the classes of the variables are depicted in table 2.1.

	Instruction	Execution
1	$x := x + 1$	$if\ \underline{PC} \rightsquigarrow \underline{x}\ then\ x := x + 1;\ else\ skip$
2	$if\ x = 0$ $\ then\ goto\ n$ $\ else\ x := x - 1$	$if\ x = 0$ $\ then\ \{push(PC, \underline{PC});\ \underline{PC} = lub(\underline{PC}, x);\ PC := n;\}$ $\ else\ \{if\ \underline{PC} \rightsquigarrow \underline{x}\ then\{x := x - 1;\}\ else\ skip\}$
3	$if'\ x = 0$ $\ then\ goto\ n$ $\ else\ x := x - 1$	$if\ x = 0$ $\ then\ \{if\ \underline{x} \rightsquigarrow \underline{PC}\ then\{PC := n;\}\ else\ skip\}$ $\ else\ \{if\ \underline{PC} \rightsquigarrow \underline{x}\ then\{x := x - 1;\}\ else\ skip\}$
4	$return$	$pop(PC, \underline{PC});$
5	$halt$	if program stack empty $then$ halt execution $else\ skip$

Table 2.1: Fenton's Data Mark Machine

In the second instruction for example, Fenton pushed the PC because it captures information about the variable x, whereas in the third instruction, the PC

is in a higher security class than the conditional variable x, so adding information from x to the PC does not change the PC's security class. As for the *halt* instruction, the program stack should be empty to ensure that the user cannot obtain information by looking at the program stack after the program has halted.

Example 2.4 Fenton's Data Mark Machine

Consider the following program, in which x initially contains 0 or 1 (borrowed from [24], Figure 5.7, p. 290)
1. $if\ x = 0\ then\ goto\ 4\ else\ x := x - 1$
2. $if\ z = 0\ then\ goto\ 6\ else\ z := z - 1$
3. *halt*
4. $z := z + 1$
5. *return*
6. $y := y + 1$
7. *return*

This program copies the value of x to y. Suppose that initially $x = 1$. The following table shows the contents of memory, the security class of the PC at each step, and the corresponding certification check.

x	y	z	PC	$\underline{PC}$	stack	certification check
1	0	0	1	*Low*	-	
0	0	0	2	*Low*	-	$Low \rightsquigarrow \underline{x}$
0	0	0	6	$\underline{x}$	(3, *Low*)	
0	1	0	7	$\underline{x}$	(3, *Low*)	$\underline{PC} \rightsquigarrow \underline{y}$
0	1	0	3	*Low*	-	

In addition, Fenton's machine handles errors (i.e. error messages informing the user of the failure of the certification check at some step) by ignoring them. The problem with reporting of errors is that a user with lower clearance than the information causing the error can deduce the information from knowing that there has been an error.

In fact, in the example 2.4, the classes of the variables are fixed. But Fenton's machine alters the class of the PC as the program runs. This suggests a notion of dynamic classes. For example, after executing the assignment $y := f(x_1, \cdots, x_n)$, y's class is changed to $lub(\underline{x_1}, \cdots, \underline{x_n})$. The next example shows that implicit flows complicate matters.

Example 2.5 Implicit flows handling in Fenton's Data Mark Machine

Consider the following program (borrowed from Denning [269], Figure 5.5, p. 285). *proc copy*(x : *integer class*$\{x\}$; *var* y : *integer class*$\{y\}$);
var z : *integer class variable*$\{Low\}$;
begin
1. $y := 0$;
2. $z := 0$;

3. $if\ x = 0\ then\ z := 1;$
4. $if\ z = 0\ then\ y := 1;$

end;

Let the class of z, $\underline{z}$ be variable and initialized to Low, and let flows be certified whenever anything is assigned to y. Suppose also that $\underline{y} < \underline{x}$.

- Statement 1 sets y to 0 and checks that $Low \rightsquigarrow \underline{y}$.
- Statement 2 sets z to 0 and $\underline{z}$ to Low.
- If $x = 0$; statement 3 changes z to 1 and $\underline{z}$ to $lub(Low, \underline{x}) = \underline{x}$, and the fourth statement is skipped. Hence, y is set to 0 on exit.
- If $x = 1$; statement 3 is skipped and the fourth statement assigns 1 to y and checks that $lub(Low, \underline{z}) = Low \rightsquigarrow \underline{z}$. Hence, y is set to 1 on exit.

Thus, an information flow occurred from x to y even though $\underline{y} < \underline{x}$. The program violates the policy but is nevertheless certified.

However, using Fenton's Data Mark Machine, one could detect this violation as it shown in the following:

x	y	z	PC	$\underline{PC}$	stack	certification check
1	0	0	1	Low	-	
1	0	0	2	Low	-	$Low \rightsquigarrow \underline{y}$
1	0	0	3	Low	-	$Low \rightsquigarrow \underline{z}$
1	0	0	4	$\underline{x}$	(4, Low)	
1	0	0	-	$\underline{x}$	(4, Low)	$\boxed{\underline{x} \rightsquigarrow \underline{y}}$

Example of Information Flow Controls

We present in this subsection a practical example of information flow control: a mail guard application for electronic mail moving between a classified network and an unclassified one follows. The goal is to prevent the illicit flow of information from one system unit to another.

Secure Network Server Mail Guard

The Secure Network Server Mail Guard (SNSMG) [66] is a process that analyzes (by blocking or deleting messages when necessary) the traffic between two networks. Consider two networks, N_1 and N_2. Let N_1 has data that are classified SECRET only, and N_2 only public ones. The authorities controlling the SECRET network need to allow electronic mail to go to the public network, as they do not want SECRET information to be dissimulated. The SNSMG works as follows: first, it receives a message from one network; then, it applies several filters (that depend on the source address, destination address, sender, recipient, and/or contents of the message) to the message.

In fact, the SNSMG deals with two kind of different message transfer agents (MTA). One is reserved for the SECRET network, and the other one for the public

one. Suppose that the messages output from the SECRET network's MTA have type a, and messages output from the filters have a different type, type b; then the public network MTA will accept inputs messages of type b only and reject any messages of any other type.

Finally, we recall that **confidentiality**, one of the security interests, demand not only to **control the messages** in the system but also to control of **flow of information** and thus of **inferences**. We will address this aspect in the following section.

2.3.3 Inference control

We say that an **observer** can achieve an **information gain** if the observer, by observing a message, can convert his **a priori knowledge** into strictly *increased* **a posteriori knowledge**. The computational capabilities and the available computational resources of the observer decide whether this gain can be achieved or not.

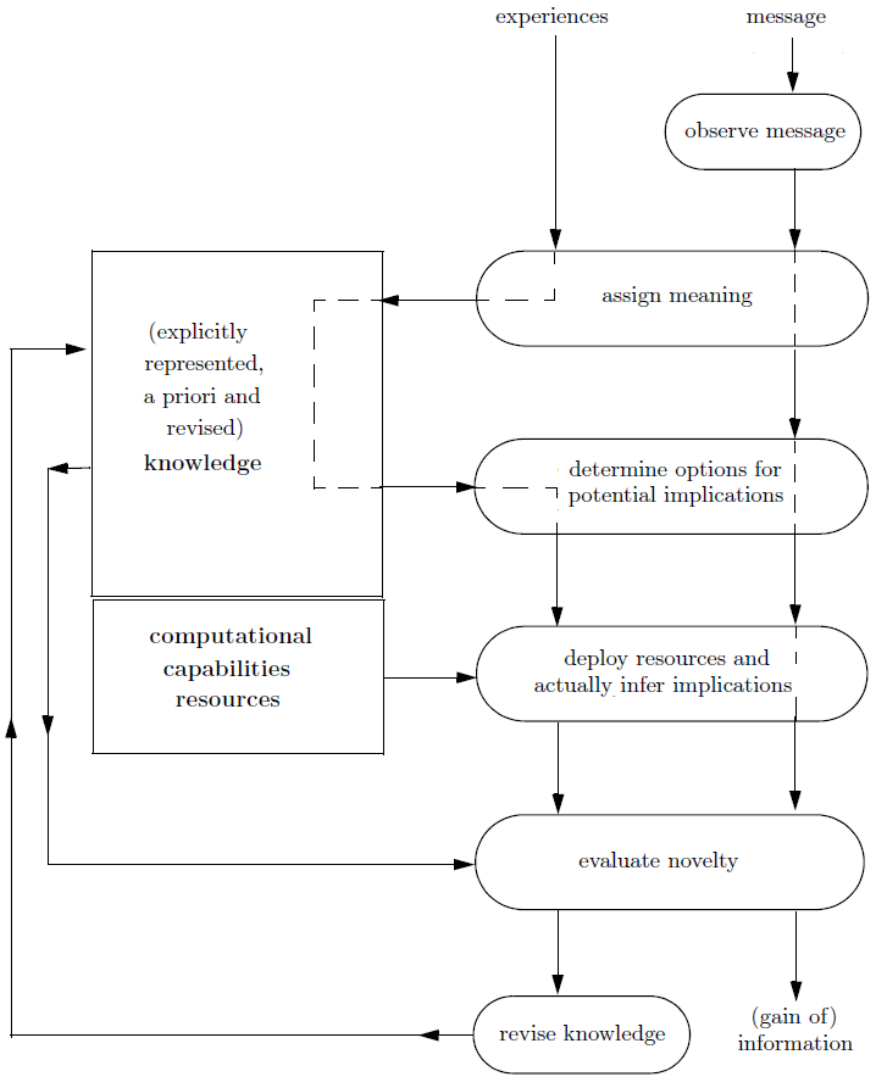


Figure 2.3: A general perspective of messages, inferences, information and knowledge, and the impact of an observers computational capabilities and resources.

Figure 2.3 (borrowed from Biskup [7] p. 68) resumes the previous idea. It also distinguishes between **observing a message** and **gaining information**.

As noted by Biskup [7], the difference between observing a message and gaining information might depend on various circumstances:

- The observer selects a **framework for reasoning** as the pertinent **communicative context**.
- The observer interprets an observation within the selected framework, and thereby assigns a **meaning** to the observation.
- The observer has some **a priori knowledge** related to the meaning of the observation.
- The observer employs a declarative notion of **logical implication** that is applicable to the selected framework, and thus he can potentially reason about the **implicational closure** of his a priori knowledge and the added meaning of the observation.
- The observer **computationally infers** implications, and evaluates actual inferences concerning **novelty**.
- The observer treats the newly inferred implications as the **information gained** from his observation.
- The observer **revises** his previous knowledge in order to reflect the recent gain, thereby getting **a posteriori knowledge**.

We should note here that, the knowledge of the observer determines those worlds that he sees as possible, but all of them remain **indistinguishable** to him. Furthermore, it is possible that, after an observation, the **a priori knowledge** and the **a posteriori knowledge** might be identical, or it is possible that the **a posteriori knowledge** determines exactly one possible world (the observer has learned a property expressible in the selected framework). In figure 2.3 on page 37, an observed message is seen as a **syntactic object**, while its assigned meaning is treated as a **semantic object**. This semantic object combined with the current knowledge of the observer might contain some information about possibly other semantic objects. To enforce confidentiality, the **designated receiver** should be provided with appropriate (a priori) knowledge beforehand in order to **enable** him to actually gain the pertinent information, whereas other **unauthorized** observers should be **prevented** from achieving this goal, by a lack of appropriate knowledge or sufficient computational capabilities and resources.

Furthermore, in some applications, the interest of an observer might be directed toward some specific properties of the semantic items rather than on their identity. In this case, all objects that share the same properties can be seen as equivalent and, accordingly, the observers goal might only be a gain of information about the equivalence class of the hidden semantic object.

As stated in the previous section, inference control:

- enforces a specific **security interest** (mostly confidentiality)
- targets a specific **participant** of a computing system
- targets suspected **threats**
- controls the **information gain** options of the participant.

In the next sections, we will focus on the preventive aspects of inference control.

In general, inference control cannot force threatening participants to ignore observations or to not infer logic implications and thus to achieve some **information gain**. Thus, it is the duty of the inference control to ensure that the accessible observations (even to a malicious omnipotent observer) are not harmful. This can be achieved by **dynamic monitoring** or **static verification**.

- **Dynamic monitoring** inspects each event and checks if any harmful inferences can be observed by the participant. Dynamic monitoring keeps track of the actual history of previous observations in order to **block** any critical observation.
- **Static verification** analyzes globally, and **in advance**, all the possible observable events by the participant. Static verification inspects thus all the possible events to check whether it exists any harmful inferences and thus **blocking** critical observations right from the beginning, without further monitoring at runtime.

Note that both dynamic monitoring and static verification need a specification of the security requirements, i.e., a **security policy** that captures the pertinent notion of **harmfulness**.

Furthermore, computationally, both approaches necessitates an algorithmic treatment of implication problems that might be, in general, computationally **unsolvable** and thus can only be **approximated** at best. An approximation that might cause more **blockings** than it is needed. A **blocking** can be deployed in the form of making harmful observations **invisible**, or to **substitute** them by harmless ones, or even more to totally **suppress** them.

Note that when the observer is **notified** of a blocking, either explicitly or implicitly by some reasoning, the recognition of a blocking might constitute also harmful information too.

For explicitly notified **refusal**, the observer can, in some cases, determine the reason for the refusal, and consequently, find the hidden event.

2.4 Summary

In this chapter, we begun by making a review of the basic components of a secure system, namely confidentiality, integrity and availability. Formally, these components are represented by security policies, which define "security" for a system or site. A policy may be formal, in which case ambiguities arise either from the use of natural languages or from the failure to cover specific areas.
Formal mathematical models of policies enable analysts to deduce a rigorous definition of "security" but do little to improve the average user's understanding of what "security" means for a site. The average user is not mathematically sophisticated enough to read and interpret the mathematics.
Policies themselves make assumptions about the way systems, software, hardware, and people behave. At a lower level, security mechanisms and procedures also make such assumptions.

The influence of the Bell-LaPadula Model permeates all policy modeling in computer security. It was the first mathematical model to capture attributes of a real system in its rules. It formed the basis for several standards, including the Department of Defense's Trusted Computer System Evaluation Criteria . Even in controversy, the model spurred further studies in the foundations of computer security. Confidentiality models may be viewed as models constraining the way information moves about a system. The notions of noninterference and nondeducibility provide an alternative view that in some ways matches reality better than the Bell-LaPadula Model. It asserts that a strict separation of subjects requires that all channels, not merely those designed to transmit information, must be closed.

Implementing security policies can be achieved through cryptography or by regulating the dissemination of information. Several mechanisms exist in the literature to control this dissemination, namely, access mechanisms and inference mechanisms. In this thesis, we will be interested in this theory of information flow and how to detect and / or prevent (i.e. to control) inferences and we will try to adapt it to logic programming.

Chapter 3

Logic Programming

In this chapter, we examine **first-order logic programming** in section 3.1 and review the important processes of **substitutions** in section 3.1.1 and **unification** in section 3.1.2 and **SLD-resolution, SLD-trees** in section 3.1.3, since they provide the foundations upon which later chapters are built. Meanwhile, **problems** of SLD-resolutions is then exposed in section 3.1.4, revealing the fact that, sometimes, the search for an SLD-refutation can result in non-termination. A non-termination due to the presence of **loops**.

Section 3.2 is devoted to **loop checking techniques** in logic programming. After presenting basic concepts on loop checking in section 3.2.1, several loop checking techniques are exposed in sections 3.2.2 – 3.2.7. **Soundness** and **completeness** of the presented loop checking techniques is also discussed. **Notes on efficient loop checks** are given in section 3.2.9.

After exposing the notion of information flow in security systems for imperative programs in section 2.3.2 of the chapter 2, it was tempting to check whether this kind of flows exists in logic programming. In section 3.3, we will expose briefly, **control flow analysis** and **data/dependence analysis**, highlighting the fact that the information flow notion presented in chapter 2 is not covered by those analysis in logic programming.

3.1 First-Order Logic Programming Language

In this section, we review the aspects of first-order logic programming that will be used pervasively in subsequent chapters: the structure of terms, substitutions, unification, and SLD-resolution. We restrict our attention to definite programs only, (i.e. those free of negated literals, the extension to include negative literals in definite programs can be found in [50]). The material contained in this section is standard from the literature and further details can be found in [40, 50], for example.

3.1.1 Terms, Programs, Goals, and Substitutions

Let us begin by defining the structure of terms and programs in first-order logic languages.

Definition 3.1.1 (**Terms and literals**). *Let X be a set of variables and Ω a set of constant/function symbols. The terms (or literals) of the programming language are defined inductively by the grammar $\mathbb{T}_\Omega(X) = x|f(t_1, \cdots, t_n)$ where $x \in X$, $f \in \Omega$, and $t_1, \cdots, t_n \in \mathbb{T}_\Omega(X)$ for $n \geq 0$. In the case where $n = 0$, we write f rather than $f()$. A **ground term** is a term not containing a variable.*

Definition 3.1.2 (**Herbrand universe**). *Let L be a first order language. The **Herbrand universe** U_L for L is the set of all ground terms, which can be formed out of the constant symbols and function symbols appearing in L.*

Definition 3.1.3 (**Definite clauses and programs**). *A **definite clause**, or just clause, is a formula $A \leftarrow B_1, \cdots, B_n$, for $n \geq 0$. The literal A is called the **head** of the clause and the outermost constant in A is called a **predicate symbol**. When the literal A is equal for example to $p(t_1, \cdots, t_n)$, where p is a predicate symbol and $t_1, \cdots, t_n$ are terms, A is referred to as **p-literal**. A clause with a p-literal as its head is said to be a **p-clause**. The subset of all p-clauses of a program is called the **procedure/predicate definition** for p. The literals $B_1, \cdots, B_n$ form the **body** of the clause. When $n = 0$, we write the clause as $A \leftarrow$ and refer to it as a **fact**. Finally, a **program** is a set of clauses.*

Definition 3.1.4 (**Goal clauses**). *A **goal clause**, also known as a **query** or just a **goal**, is a formula $\leftarrow B_1, \cdots, B_n$ where $B_1, \cdots, B_n$ are literals, for $n \geq 0$. When $n = 0$, we write the goal as $\square$ and refer to it as the **empty clause**.*

The operational framework of a first-order logic language requires a procedure to determine whether or not a query succeeds with respect to a given program. The resulting *proof procedure* must match literals in a goal with clauses in a program and apply the technique of choosing only those values for the variables of a literal when an appropriate value becomes apparent. In other words, logic

variables are 'place holders' until the proof procedure can determine suitable values for them. *Substitutions* bind variables to terms and can be considered as functions, as defined next.

Definition 3.1.5 (**Substitutions**). *Let X and Y be sets of variables. A substitution $\phi : X \to \mathbb{T}_\Omega(Y)$ is a total function mapping variables to terms. For $X = \{x_1, \cdots, x_n\}$ and $n \geq 0$, we represent ϕ using the notation $\{x_1 \mapsto s_1, \cdots, x_n \mapsto s_n\}$ where variables x_i map to terms s_i, for $1 \leq i \leq n$. (For brevity, we will ignore pairs of the form $x_i \mapsto x_i$, simply assuming their existence.) The set Y is given by $\cup_{i=1}^{n}$ vars s_i, where the function vars $: \mathbb{T}_\Omega(Y) \to \mathrm{P}Y$ takes a term and returns the set of variables contained in that term.*

The result of a computation in a logic language is a substitution generated by the matching process mentioned above. The proof procedure uses these substitutions to systematically replace variables in a term by other terms. The *instance* of a term under a substitution is obtained according to the definition below.

Definition 3.1.6 (**Application of substitutions**). *Given $\phi = \{x_1 \mapsto s_1, \cdots, x_n \mapsto s_n\} : X \to \mathbb{T}_\Omega(Y)$, we define a function $[\phi] : \mathbb{T}_\Omega(X) \to \mathbb{T}_\Omega(Y)$ (pronounced "apply ϕ") that determines the instance of a term t under ϕ" as follows:*

$$t[\phi] \begin{cases} \phi x & if\ t = x \in X \\ f(t_1[\phi], \cdots, t_n[\phi]) & if\ t = f(t_1, \cdots, t_n) \end{cases}$$

That is, when ϕ is applied to a variable, that variable gets replaced by whatever ϕ maps it to and when ϕ is applied to a composite term, ϕ is applied to each subterm.

Substitutions obey several important algebraic properties that aid the construction of the desired proof procedure. In particular, we can define the composition of two substitutions; given two substitutions $\sigma : X \to \mathbb{T}_\Omega(Y)$ and $\phi : Y \to \mathbb{T}_\Omega(Z)$, there exists another substitution $\phi \circ \sigma : X \to \mathbb{T}_\Omega(Z)$, called the *composition* of σ and ϕ, such that the identity $(\phi \circ \sigma)x = (\sigma x)[\phi])$ holds. Furthermore, it is easy to prove that $t[\phi \circ \sigma] = t[\sigma][\phi]$ for all terms t and that the composition of substitutions is associative.
The identity substitution $\iota : X \to \mathbb{T}_\Omega(X)$ maps each variable to itself and is defined simply as $\iota x = x$. The substitution ι is the unit of composition since $\iota \circ \phi = \phi = \phi \circ \iota$.

3.1.2 Unification

The matching of literals in a goal with clauses in a program is the key component of the proof procedure used to determine whether a goal is a logical consequence of a program. The matching process is known as *unification* [58] and

unifying two terms produces a substitution such that the terms become identical after applying this substitution over them. Moreover, the substitution created by the unification algorithm is the 'most general' one possible in the sense that all other unifiers are an instance of it. Most general unifiers prevent the generation of useless instances of literals in the proof procedure and help keep the search space that a computer must examine as small as possible.

Definition 3.1.7 (**Most general unifiers**). *Let $D = \{(s_1, t_1), \cdots, (s_n, t_n)\}$ be a set of pairs of terms from $\mathbb{T}_\Omega(X)$, for $n \geq 1$, and $\phi : X \to \mathbb{T}_\Omega(Y)$ be a substitution. We say that ϕ **unifies**, or **is a unifier of**, D if $s_i[\phi] = t_i[\phi]$, for all $1 \leq i \leq n$. Moreover, we call ϕ a **most general unifier** if, for every other unifier $\psi : X \to \mathbb{T}_\Omega(Y)$ of D, there exists a substitution $\sigma : Y \to \mathbb{T}_\Omega(Y)$, such that $\psi = \sigma \circ \phi$. The most general unifier ϕ of two terms is unique up to the renaming of the variables in Y.*

An algorithm that determines the most general unifier of two terms is given in Figure 3.1 on page 45. By a slight abuse of notation, we understand the application of a substitution over a set D of pairs of terms by defining $D[\phi] = \{(s[\phi], t[\phi])|(s, t) \in D\}$.

The gist of the algorithm is that *ok*=false if the terms u and v are not unifiable, otherwise *ok*=true. In the latter case, we let ϕ be a unifier of u and v; moreover, we write $\psi = \sigma \circ \phi$, for ψ some other unifier of the disagreement set D and σ some substitution. Therefore, ϕ accumulates the most general unifier of u and v as the computation progresses whilst D represents the disagreement set containing those parts of u and v pending unification. The invariant maintained by the algorithm, which is used in a proof of the correctness of *unify*, is as follows.

Proposition 3.1.1. *The invariant of the while loop in the unification algorithm, illustrated in Figure 3.1 on page 45, is as follows:*

1. *If u and v have a unifier then ok = true.*
2. *If ok = true then some substitution ψ unifies u and v if and only if there exists a substitution σ such that $\psi = \sigma \circ \phi$ and ϕ unifies D.*

The function *unify*, therefore, returns either 'Nothing', signaling that the disagreement pair is not unifiable, or 'Just ϕ' where ϕ is the most general unifier of the terms u and v as established by the following lemma.

Lemma 3.1.1. *If two terms u and v have a unifier, then they have a most general unifier. Moreover, the function unify u v determines this fact.*

Example 3.1 Unification

```
unify(u, v)
begin
    D ← {(u, v)};
    φ ← ι;
    ok ←true;
    while ok and D ≠ {} do
        let D′ ∪ {(s, t)} = D
        if s = x = t then
            D ← D′;
        else
            if s = x ∉ t then
                D ← D′[x ↦ t];
                φ ← {x ↦ t} ∘ φ;
            else
                if t = x ∉ s then
                    D ← D′[x ↦ s];
                    φ ← {x ↦ s} ∘ φ;
                else
                    if s = f(t1, ··· , tn) and t = f(s1, ··· , sn) then
                        D ← D′ ∪ {(ti, si)|1 ≤ i ≤ n};
                    else
                        ok ←false;
    if ok then
        return Just φ;
    else
        ;
```

Figure 3.1: The unification algorithm for two terms u and v.

Let us now step through an example unification of two terms. Consider the computation of $unify\ f(a, g(X), Y)\ f(Y, g(h(Z)), V)$. After the first iteration of the while loop in the algorithm of Figure 3.1 on page 45, we obtain the disagreement set $D_1 = \{(a, Y), (g(X), g(h(Z))), (Y, V)\}$.
On the next iteration of the loop, we can select any disagreement pair from D_1; suppose we select (Y, V). The unification of this pair produces the single substitution $\phi_1 = \{Y \mapsto V\}$ and we apply this over the remaining pairs in D_1 to form the new disagreement set $D_2 = \{(a, V), (g(X), g(h(Z)))\}$
Suppose we select (a, V) at the subsequent unification step. The resulting substitution is $\phi_2 = \{V \mapsto a\}$ which we apply over D_2 to obtain the new set $D_3 = \{(g(X), g(h(Z)))\}$.
There is only one pair to choose at the next unification step which results in the formation of the set $D_4 = \{(X, h(Z))\}$.
The final unification step produces the substitution $\phi_3 = \{X \mapsto h(Z)\}$ and the empty disagreement set. Therefore, the most general unifier of the two terms

$f(a, g(X), Y)$ and $f(Y, g(h(Z)), V)$ is $\phi = \phi_3 \circ \phi_2 \circ \phi_1$. The application of ϕ over both terms produces the unified term $f(a, g(h(Z)), a)$.

In this section, we have reviewed an algorithm to calculate the most general unifier for two terms if it exists. What we now require is a proof procedure to check whether a goal is a logical consequence of a program. The systematic search for the proof of a goal, with respect to a given program, is centered around the process of *resolution* and we discuss this in the following subsection.

3.1.3 SLD-Resolution

In this section, the computational strategy of resolution is reviewed. Logic programming languages, like Prolog, use a particularly simple form of resolution, called **SLD-resolution** (Selected literal using Linear resolution for Definite clauses), that allows a programmer to write efficient programs. The drawback of using SLD-resolution, as we shall see shortly, is that the search for a proof of a goal can often fail to terminate even for logically simple queries. Consequently, the declarative nature of Prolog programs is often severely restricted.

Resolution is called an 'inference rule' because it derives information about a goal and a program. A choice exists of how to perform resolution; for example, individual clauses in the program could be selected to match with literals in a goal or, alternatively, program clauses could be matched together to form new clauses to resolve a goal with.

The former of these methods, called **linear resolution**, involves resolving a literal in the goal with a clause taken from the program, producing a new goal which we continue to resolve in the same manner, and forms the basis of resolution for languages like Prolog. The soundness and completeness of resolution are proved, for example, by Apt & van Emden [2] where 'soundness' means that each derived resolvent is a logical consequence of the program and 'completeness' means that all such consequences of the program may be derived by resolution.

Linear resolution itself allows various choices at each resolution step during the proof of a goal: if a goal has more than one literal, any literal from the goal may be selected for resolution and, moreover, any program clause may be chosen to resolve with the selected literal. The method of selecting program clauses is called the **search strategy** and the method of selecting the goal literal is called the **computation rule** (or selection strategy). It is, however, well known that any applicable program clause or goal literal may be selected and still retain the aforementioned soundness and completeness properties of resolution [2, 40, 50,

67]. With this in mind, both the search strategy and the computation rule can be fixed in advance to allow the systematic search for proofs of a goal. The proof of a goal can be visualized in the form of a tree structure, with each node labeled with a goal and each branch representing the result of a resolution step. Given the myriad of choices possible in a resolution, it is not difficult to imagine that this tree can quickly become enormous. From the point of view of systematically searching for a proof of a goal, a priority is to reduce the size of search space that a machine must examine before successfully finding an answer. One resolution-based method of searching for proofs that restricts the size of a search tree is called **SLD-resolution**, adopting linear resolution and a predetermined computation rule and search strategy. We now give a few definitions to formalize the use of SLD-resolution for searching for a proof of a query with respect to some program.

Definition 3.1.8 (**SLD-resolution**). *Let $C = A \leftarrow B_1, \cdots, B_m$, for $m \geq 0$, be a definite clause, and $G = \leftarrow A_1, \cdots, A_n$, for $n \geq 1$, a goal such that A_1 and A are unifiable with most general unifier ϕ. Then the* ***SLD-resolvent*** *of C and G is the goal $G_0 = (\leftarrow B_1, \cdots, B_m, A_2, \cdots, A_n)[\phi]$. The substitution ϕ is called the substitution of the SLD-resolution.*

A sequence of SLD-resolutions forms an **SLD-derivation** where the selected literal, the substitution of the SLD-resolution, and the program clause used to resolve with, are recorded for each resolution step.

Definition 3.1.9 (**SLD-derivation**). *For a program P and a goal G, an SLD-derivation of $P \cup \{G\}$ is a (possibly infinite) sequence of triples $(G_1, C_1, \phi_1), \cdots, (G_n, C_n, \phi_n)$, for $n \geq 1$ (denoted also $G_1 \Rightarrow_{C_1,\phi_1} \Rightarrow \cdots \Rightarrow_{C_n,\phi_n} G_n$), where C_n denotes a 'variant' of a definite clause in P, G_n is a goal (we distinguish $G_1 = G$), and ϕ_n is a substitution. Moreover, for $1 \leq i < n$, G_{i+1} is derived from G_i and C_i via substitution ϕ_i by an SLD-resolution.*

A **variant clause** C is identical to its corresponding program clause except that all variables in C are renamed to avoid clashes with variables in the current derivation. The basic method of searching for a proof of a goal $G = \leftarrow A_1, \cdots, A_n$ with respect to a program P involves repeatedly resolving the literals in G with relevant clauses from P until the process either fails or derives the empty clause $\square$. This process, called **refutation**, utilizes SLD-resolution and the unification algorithm presented earlier to determine the most general values of any free variables in G. The definition of this process, called **SLD-refutation**, is as follows.

Definition 3.1.10 (**SLD-refutation**). *For a program P and a goal G, an* ***SLD-refutation*** *of G is an SLD-derivation for $P \cup \{G\}$ starting at G and ending with $G_n = \square$, for $n \geq 1$. If the substitutions of the SLD-resolutions in the derivation are $\phi_1, \cdots, \phi_n$ then the* ***substitution answer of the refutation*** *is $\phi = \phi_n \circ \cdots \circ \phi_1$.*

The search tree formed from an SLD-resolution is commonly known as an **SLD-tree**, and we define them as follows.

Definition 3.1.11 (**SLD-trees**). *Let P be a program and G a goal. An **SLD-tree** for $P \cup \{G\}$ is a (possibly infinite) tree with each node labeled with a goal and each branch labeled with a substitution so that the following conditions are satisfied:*

1. *The root node of the tree is G.*

2. *Let $G' = \leftarrow A_1, \cdots, A_n$, for $n \geq 1$, be a node in the tree. For each program clause $A \leftarrow B_1, \cdots, B_m$, for $m \geq 0$, such that A_1 and A are unifiable with most general unifier ϕ, the node has a child $\leftarrow (B_1, \cdots, B_m, A_2, \cdots, A_n)[\phi]$. We label the arc connecting G' to a child with the substitution of the SLD-resolution and we call A_1 the **selected literal**.*

3. *Nodes labeled with the empty goal $\square$ have no children.*

Each sequence of nodes in an SLD-tree is clearly either an SLD-refutation or an SLD-derivation.

The shape of the SLD-tree generated for a query depends on the particular computation rule employed in the SLD-resolution; indeed, the choice of rule can have a tremendous influence on the size of the corresponding SLD-tree. Nevertheless, every SLD-tree is essentially the same with respect to the refutations it contains. In other words, every SLD-tree contains the same number of refutations, irrespective of the computation rule used in the resolution steps. This result is a reformulation of the completeness of SLD-resolution.

In this thesis, we have only considered the resolution of definite programs, i.e., those free of negation. The essential alteration is the use of a safe computation rule [50] which selects a negative literal in a goal only when it is ground (variable free) to ensure its resolution cannot result in an unsound answer substitution. In addition, we will be interested in the following types of logic programs: **Datalog**, **hierarchical**, **restricted**, *nvi* and *svo* logic programs.

In many respects, **Datalog** is a simplified version of Prolog. Any Datalog program must satisfy the following condition: each variable which occurs in the head of a clause must also occur in the body of the same clause.

Clark [20] and Shepherdson [64] introduced the notion of hierarchical programs which is motivated by database theory. In fact, they were looking for logic programs that satisfies certain syntactic conditions with respect to the occurrence of positive / negative literals. When considering negative literals, such programs are called **stratified** and when considering only positive literals, these programs

are called hierarchical. Formally, a Datalog program P is said to be **hierarchical** iff there exists a mapping l associating a nonnegative integer $l(p)$ to every predicate symbol p occurring in P and such that for all clauses $A_0 \leftarrow A_1, \cdots, A_n$ in P, if each A_i is a literal of the form $p_i(t_1, \cdots, t_{k_i})$ then $l(p_0) > l(p_1), \cdots, l(p_n)$.

The dependency graph of a Datalog program P is the graph (N, E) where N is the set of all predicate symbols occurring in P and E is the adjacency relation defined on N as follows: pEq iff P contains a clause $A_0 \leftarrow A_1, \cdots, A_n$ such that A_0 is a literal of the form $p(\cdots)$ and for some $1 \leq i \leq n$, A_i is a literal of the form $q(\cdots)$. Let E^* be the reflexive transitive closure of E.

A Datalog program P is said to be **restricted** iff for all clauses $A_0 \leftarrow A_1, \cdots, A_n$ in P and for all $1 \leq i \leq n-1$, if A_0 is of the form $p(\cdots)$ and A_i is of the form $q(\cdots)$, then not qE^*p.

A program P is **nonvariable introducing** (in short *nvi*) if for every clause $A \leftarrow B_1, \cdots, B_n$ in P, every variable that occurs in $B_1, \cdots, B_n$ occurs also in A.

A program P is **single variable occurrence** (in short *svo*) if for every clause $A \leftarrow B_1, \cdots, B_n$ in P, no variable in $B_1, \cdots, B_n$ occurs more than once.

Restricted, **nonvariable introducing** and **single variable occurrence** logic programs were introduced by Bol *et al.* [9]. These programs generate correlated SLD-resolvents, in such a way, that some SLD-resolvents can be equal or that, an SLD-resolvent can be a variant of some other SLD-resolvent.

Now that we have reviewed the important aspects of SLD-resolution, we can expose its problems.

3.1.4 The Problem of SLD-Resolution

The resolution strategy adopted by Prolog exhibits unfortunately intrinsic deficiencies when executing relational programs and we discuss them in the remainder of this section.

Most implementations of logic programming languages, whose computational model is based on SLD-resolution, employ a left-to-right (or depth-first) computation rule. So, for the goal $\leftarrow A_1, \cdots, A_n$, the leftmost literal A_1 is always selected at each resolution step and any introduced literals take its place. However, the search for an SLD-refutation can sometimes be substantially more difficult, often resulting in non-termination, when using a left-to-right computation rule than some other one. The following example demonstrates this phenomenon.

Example 3.2 SLD-resolution, SLD-tree

Consider the following program P:
$C_1 : p(x, z) \leftarrow q(x, y), p(y, z);$

$C_2 : p(x, y) \leftarrow r(x, y);$
$C_3 : r(b, b) \leftarrow;$
$C_4 : q(a, a) \leftarrow;$
$C_5 : p(a, b) \leftarrow$

Furthermore, consider the following goal G: $\leftarrow p(a, u)$.

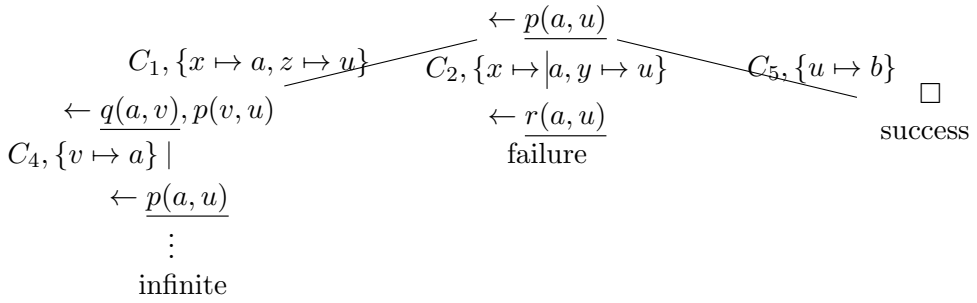


Figure 3.2: SLD-tree which illustrates the problem with left-to-right computation rule and depth-first search

In the SLD-tree resulting from the resolution of the goal G using the left-to-right computation rule of Prolog (i.e. selection rule that selects the leftmost atom in the clause and tries the clauses given by their order in the program), depicted in figure 3.2, the leftmost branch is infinite. Thus, a depth-first search will never find the success branch. In figure 3.2, The selected atoms are underlined and the success, failure and infinite branches are shown.

In fact, the use of a left-to-right computation rule means that SLD-resolution can be incomplete, in a practical sense, when searching for a proof of a goal. Despite the limitations of the left-to-right computation rule, it remains predominant in logic programming owing almost entirely to the ease of its implementation and the fact that programmers can write more efficient programs once educated about the order of evaluation.

Thus, one can try to detect any infinite branch in an SLD-tree. Unfortunately, this problem is obviously undecidable, as the logic programming has the full power of recursion theory. However, several heuristics were proposed in the literature to avoid this nontermination:

- Poole & Goebel [59] proposed to apply some reformulation techniques on the original specification of the program. However, the resulting program may be different from the original one, in the sense that it works for some subsets of the possible queries covered by the original one; also, it can reintroduce recursion. Smith *et al.* [65] pointed the difficulty to do such reformulations.

- Apt *et al.* [1], Bol *et al.* [9], Bol [8], Smith *et al.* [65], Van Gelder [33], Vieille [72], Besnard [5], Convington [21], Sahlin [60], Brough & Walker [12], Shen [62] proposed to modify the computation mechanism by adding a capability of pruning. Thus, at some point, the interpreter is forced to stop its search through a certain part of the SLD-tree. These mechanisms are called loop check mechanisms, as they are based on excluding some kinds of repetition in the SLD-derivations.

3.2 Loop Check

The main purpose of a loop check is to reduce the search space for top-down interpreters in order to obtain a finite search-space. Mainly, two different forms of loop check were considered in the literature:

- Vieille [72] proposed that the decision on whether a node in a tree should be pruned must depend on the whole portion of the considered SLD-tree previously traversed (the technique proposed is based on reusing answers for previously solved subgoals).

- Bol *et al.* [9], Shen [62], Shen *et al.* [63] proposed that the pruning of a node in an SLD-tree must depend only on its ancestors (from the root node of the SLD-derivation up to this node). In what follows, we will be concerned only with this type of pruning techniques.

In this section, we review the basic concepts concerning loop checking and the proposals in this approach.

3.2.1 Basic Concepts

The aim of a loop check is to prune every infinite SLD-tree to a finite subtree of it containing the root. The following definitions were introduced by Bol *et al.* For further motivations, examples and proofs, refer to [9].

Definition 3.2.1. *For two substitutions σ and τ, we write $\sigma \leqslant \tau$ when σ is **more general** than τ and for two expressions E and F, we write $E \leqslant F$ when F is an instance of E. We then say that F is **less general** than E.*

Definition 3.2.2 (**Variant of an SLD-derivation**)**.** *A **variant** of an SLD-derivation D is an SLD-derivation D' that is the same as D up to variable renaming.*

In the previous definition, a *variant* D' of an SLD-derivation D means that in every derivation step in D', atoms in the same positions are selected and same program clauses are used. Thus, D' may differ from D in the renaming that is applied to the program clauses.

Definition 3.2.3 (**Subderivation - subderivation free**)**.** *Let P be a logic program, G a goal, and L be a set of SLD-derivations of $P \cup \{G\}$. Define: $RemSub(L) = \{D \in L | L$ does not contain a proper subderivation of $D\}$.*
*L is **subderivation free** if $L = RemSub(L)$.*

Definition 3.2.4 (**Simple loop check**). *A **(simple) loop check** is a computable set L of finite SLD-derivations such that L is closed under variants and is subderivation free.*

Definition 3.2.5 (**Pruned SLD-tree**). *Let P be a program, G a goal, T the SLD-tree of $P \cup \{G\}$, and L a loop check. By applying L to T, we obtain a new SLD-tree $f_L(P \cup \{G\})$ which consists of T with all the nodes in $\{G' \mid$ the SLD-derivation from the goal G to G' is in $L\}$ pruned. By pruning a node from an SLD-tree, we mean removing all its descendants.*

One of the most important property when using loop checks is not to loose successful results or any individual solution. Furthermore, as the purpose of a loop check is to reduce an "infinite" search space into a finite one, the second important property is thus to prune every infinite derivation. In order to justify this, Bol *et al.* [9] introduced the following notions:

Definition 3.2.6 (**Soundness and completeness**).

1. *A loop check L is **weakly sound** if the following condition holds: for every program P, goal G, and SLD-tree T of $P \cup \{G\}$, if T contains a successful branch, then $f_L(P \cup \{G\})$ contains a successful branch.*

2. *A loop check L is **sound** if for every program P and goal G, and SLD-tree T of $P \cup \{G\}$: if T contains a successful branch with a computed answer substitution σ, then $f_L(P \cup \{G\})$ contains a successful branch with a computed answer substitution σ' such that $G[\sigma'] \leqslant G[\sigma]$.*

3. *A loop check L is **complete** if every infinite SLD-derivation is pruned by L. (Put another way, a loop check L is complete if for any program P and goal G, $f_L(P \cup \{G\})$ is finite.)*

An ideal loop check would be both weakly sound and complete. Unfortunately, since logic programs have the full power of the recursive theory, there is no loop check that is both weakly sound and complete even for function-free logic programs [9].

In the remainder of this section, several simple loop checks are recalled from [5, 9, 12, 21] . Soundness and completeness of these loop checks are then studied and this for several types of logic programs.

In chapter 5, and in order to show if two goals are bisimilar, we will be interested exclusively in loop checks that are only sound and complete. For a better understanding of loop checking techniques, we will begin by exposing some basic loop checks; i.e. basic in the sense that the reduction of the search space is based on a simple condition. We will show that these basic loop checks

are not sound and complete. We will continue by investigating some complex but sound and complete loop checks, for which the conditions imposed are more difficult to fulfill, necessitating sometimes the imposition of some restrictions on the syntax of the clauses in the logic program.

3.2.2 Loop Check Based on the Repeated Application of Clauses

As a first approach/attempt, Brough & Walker [12] proposed to prune SLD-derivations whenever a clause is used more than once to resolve a goal.

Definition 3.2.7 (**Loop check based on the repeated application of clauses**). *This loop check prunes a derivation as soon as it detects that a clause was already used to resolve an earlier subgoal.*
Loop check 1=RemSub$(\{D|D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$ *such that for some* $i, 0 \leq i < k$, $C_i = C_k\})$.

Example 3.3 Application of *Loop check* 1 **over an infinite derivation**

Consider the following logic program P:
$C_1 : p \leftarrow p$
and let G be the goal $\leftarrow p$.
Obviously (as shown in figure 3.3), the SLD-tree corresponding to the goal $\leftarrow p$ is infinite. By applying *Loop check* 1, the SLD-tree is pruned and thus one can get a finite SLD-tree.

SLD-tree of $P \cup \{G\}$	Pruned SLD-tree of $P \cup \{G\}$
$\leftarrow p$	$\leftarrow p$
$\vert\ C_1, \iota$	$\vert\ C_1, \iota$
$\leftarrow p$	$\underline{\leftarrow p}$
$\vert\ C_1, \iota$	
$\leftarrow p$	
$\vert\ C_1, \iota$	
$\leftarrow p$	
$\vdots$	

Figure 3.3: SLD-tree and its associated pruned SLD-tree using *Loop check* 1 for $P \cup \{G\}$

Loop check 1, based on a very simple condition, is too restrictive because it can discards all success paths of a search space. In this sense:

Theorem 3.2.1 (**Soundness and completeness of** *Loop check* 1)**.** *1. Loop check 1 is not sound w.r.t. the leftmost selection rule for Datalog, restricted, nvi, svo and general logic programs.*

2. Loop check 1 is complete w.r.t. the leftmost selection rule for Datalog, restricted, nvi, svo and general logic programs.

Proof. (*Soundness*) To prove that *Loop check* 1 is not sound, it suffices to consider this example.
Let P be the following Datalog / restricted / *nvi* / *svo* logic program:
$C_1 : p(x, y) \leftarrow q(x, y)$;
$C_2 : q(x, y) \leftarrow p(y, x)$;
$C_3 : q(b, a) \leftarrow$;
and let G be the following goal $\leftarrow p(a, b)$.
The corresponding SLD-tree and its associated pruned SLD-tree using *Loop check* 1 are depicted in figure 3.4.

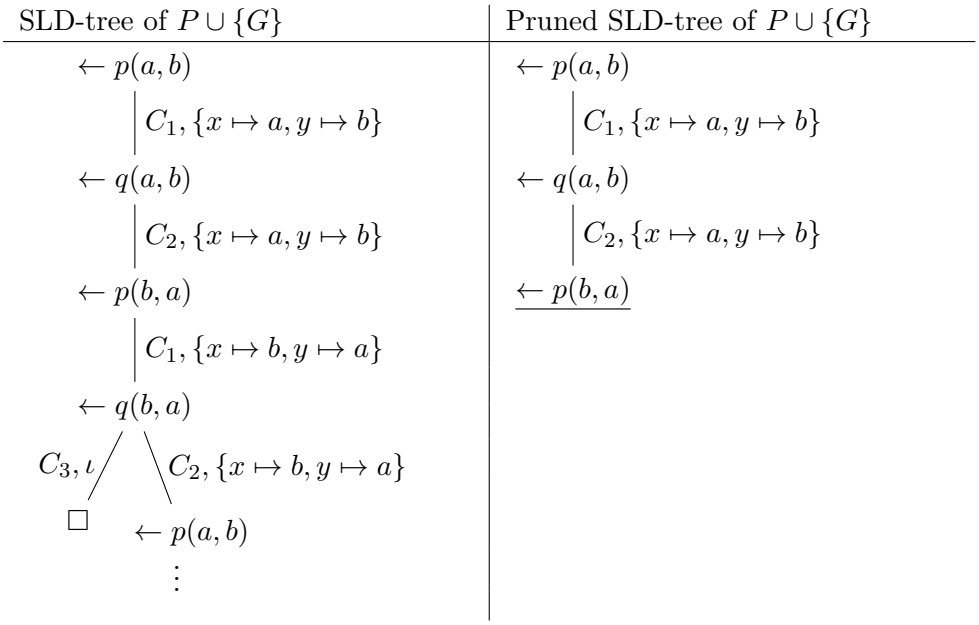


Figure 3.4: SLD-tree and its associated pruned SLD-tree using *Loop check* 1 for $P \cup \{G\}$

As one can see, *Loop check* 1 pruned successful derivations (here all the successful ones) and thus it is not sound.

(*Completeness*) As to prove that *Loop check* 1 is complete, one can rely on the definition of logic programs, stating that a logic program is composed of a finite set of clauses. Thus, there is a predefined number n of different clauses in a logic program. Consequently, the depth of each branch in $f_L(P \cup \{G\})$ would

not exceeds n. This proves that $f_L(P \cup \{G\})$ is finite and thus *Loop check* 1 is complete. □

3.2.3 Loop Check Based on Repeated Goals

As a second approach/attempt, Brough & Walker [12] proposed to prune SLD-derivations whenever a subgoal is equal to one of its ancestors goals.

Definition 3.2.8 (**Loop check based on the repeated goals**). *This loop check prunes a derivation as soon as it detects that a goal is its own subgoal. Loop check 2=RemSub*$(\{D | D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$ *such that for some* $i, 0 \leq i < k$, $G_k = G_i\})$.

Example 3.4 Application of *Loop check* 2 over an infinite derivation

Consider the following logic program P:
$C_1 : p(x) \leftarrow p(a)$
and let G be the goal $\leftarrow p(a)$.
As shown in figure 3.5 on page 56, the SLD-tree of the goal $\leftarrow p(a)$ is infinite. But, by applying *Loop check* 2, the SLD-tree is pruned due to the recursive occurrences of the subgoal $\leftarrow p(a)$.

SLD-tree of $P \cup \{G\}$	Pruned SLD-tree of $P \cup \{G\}$
$\leftarrow p(a)$	$\leftarrow p(a)$
$\quad \Big\vert\, C_1, \{x \mapsto a\}$	$\quad \Big\vert\, C_1, \{x \mapsto a\}$
$\leftarrow p(a)$	$\underline{\leftarrow p(a)}$
$\quad \Big\vert\, C_1, \{x \mapsto a\}$	
$\leftarrow p(a)$	
$\quad \Big\vert\, C_1, \{x \mapsto a\}$	
$\leftarrow p(a)$	
$\quad \vdots$	

Figure 3.5: SLD-tree and its associated pruned SLD-tree using *Loop check* 2 for $P \cup \{G\}$

Loop check 2 is not very satisfactory because it does not prune all infinite paths as shown next.

Theorem 3.2.2 (**Soundness and completeness of** *Loop check* 2)**.** *1. Loop check 2 is sound w.r.t. the leftmost selection rule for Datalog, restricted, nvi, svo and general logic programs.*

2. Loop check 2 is not complete w.r.t. the leftmost selection rule for Datalog, restricted, nvi, svo and general logic programs.

Proof. (*Soundness*) Let P be a program, G_0 a goal and T an SLD-tree of $P \cup \{G_0\}$. Suppose that T contains a successful branch $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_{i-1},\theta_{i-1}} G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \cdots \Rightarrow_{C_m,\theta_m} \square)$ and suppose that D is pruned at G_k (i.e. there exists an i such that $G_k = G_i$). Knowing that $G_k = G_i$, the SLD-derivation $D_1 = (G_i \Rightarrow \cdots \Rightarrow_{C_m,\theta_m} \square)$ exists. In order to show the soundness of this loop check, we use induction on m, by showing that $f_L(T)$ contains a successful branch D' shorter than D. Let D_2 be the derivation formed by the concatenation of the initial part of D (till the i^{th} derivation) and D_1: $D_2 = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_{i-1},\theta_{i-1}} G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots \Rightarrow_{C_m,\theta_m} \square)$. By the independence of the selection rule, T contains a branch D_3 such that $|D_3| = |D_2| < |D|$. By the induction hypothesis, $f_L(T)$ contains a successful branch D_3 shorter than D.

(*Completeness*) To prove that *Loop check* 2 is not complete for Datalog, restricted, *svo* and general logic programs, let us consider the following program:
$C_1 : p(x) \leftarrow q(x), p(z)$;
$C_2 : q(a) \leftarrow$;
and let G be the following goal $\leftarrow p(a)$.
The corresponding SLD-tree, depicted in figure 3.6, is the same as its associated pruned SLD-tree using *Loop check* 2, since in the derivation, there is no subgoals that are equal to one of its ancestors goals.

As *Loop check* 2 can not prune all infinite derivations, *Loop check* 2 is not complete for Datalog, restricted, *svo* and general logic programs.

For *nvi* logic programs, one can consider the following program:
$C_1 : p(x) \leftarrow p(x), p(x)$;
and the following goal $G = \leftarrow p(x)$.

Figure 3.7 shows the SLD-tree. As the size of the resolvent goals increases with each derivation step, no equal subgoals will be generated and thus *Loop check* 2 will not prune this SLD-tree.

□

$$
\begin{array}{l}
\leftarrow p(a) \\
\quad \Big| \, C_1, \{x \mapsto a\} \\
\leftarrow q(a), p(z) \\
\quad \Big| \, C_2, \iota \\
\leftarrow p(z) \\
\quad \Big| \, C_1, \{x \mapsto z\} \\
\leftarrow q(z), p(z') \\
\quad \Big| \, C_2, \{z \mapsto a\} \\
\leftarrow p(z') \\
\quad \vdots
\end{array}
$$

Figure 3.6: SLD-tree for $P \cup \{G\}$

$$
\begin{array}{l}
\leftarrow p(u) \\
\quad \Big| \, C_1, \{x \mapsto u\} \\
\leftarrow p(u), p(u) \\
\quad \Big| \, C_1, \{x \mapsto u\} \\
\leftarrow p(u), p(u), p(u) \\
\quad \vdots
\end{array}
$$

Figure 3.7: SLD-tree for $P \cup \{G\}$

3.2.4 Loop Check Based on Repeated Goals and Repeated Applications of Clauses

As a third approach/attempt, Brough & Walker [12] proposed to prune SLD-derivations whenever a subgoal is equal to one of its ancestors goals and the clauses used to resolve the subgoals are the same as the one used to resolve the ancestors goals.

Definition 3.2.9 (**Loop check based on the repeated goals and repeated applications of clauses**). *This loop check prunes a derivation as soon as it detects that a goal is its own subgoal and that the clause used to resolve this goal was already used to resolve an earlier subgoal.*
Loop check 3=$RemSub(\{D | D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$ *such that for some* $i, 0 \leq i < k$,

- $G_k = G_i$
- $C_k = C_i$

}).

Example 3.5 Application of *Loop check* 3 over an infinite derivation

As an example, one can reconsider the same logic program P and goal G as in example 3.4. The corresponding pruned SLD-tree of the goal G remains the same, because, the subgoal $\leftarrow p(a)$ is recursively resolved by the same clause C_1.

Seeing that *Loop check* 3 is less restrictive than *Loop check* 2, the major problem of its incompleteness remains unsolvable.

Theorem 3.2.3 (**Soundness and completeness of** *Loop check* 3)**.** *1. Loop check 3 is sound w.r.t. the leftmost selection rule for Datalog, restricted, nvi, svo and general logic programs.*

2. Loop check 3 is not complete w.r.t. the leftmost selection rule for Datalog, restricted, nvi, svo and general logic programs.

Proof. (*Soundness*) Seeing that *Loop check* 3 prunes less derivations than *Loop check* 2, *Loop check* 3 is sound for Datalog, restricted, *nvi*, *svo* and general logic programs.

(*Completeness*) To prove that *Loop check* 3 is not complete, it suffices to reproduce the same incompleteness proof of *Loop check* 2. □

3.2.5 Loop Check Based on Repeated Atoms through Syntactic Variants

According to Convington [21], infinite loops arise mainly from three relations:

- biconditional relations of the form:
  $p(x) \leftarrow q(x)$;
  $q(x) \leftarrow p(x)$
- symmetrical relations of the form:
  $p(x, y) \leftarrow p(y, x)$
- transitive relations of the form:
  $p(x, z) \leftarrow p(x, y), p(y, z)$

Convington's proposed to prune SLD-derivations whenever an atom in a subgoal is a variant of an atom in one of its ancestors goals.

Definition 3.2.10 (**Loop check based on the repeated atoms through syntactic variants**). *This loop check prunes a derivation as soon as it detects that an atom in a goal is a variant of an atom in some ancestors subgoals.*
Loop check 4=RemSub($\{D | D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$ *such that for some* $i, 0 \leq i < k$, *where* $G_i = A_{i_1}, \cdots, A_{i_n}$ *and* $G_k = A_{k_1}, \cdots, A_{k_m}$, *there exists* $1 \leq j \leq n$, $1 \leq l \leq m$ *and a substitution* τ, *such that:* $A_{k_l} = A_{i_j}[\tau]\}$).

Example 3.6 Application of *Loop check* 4 **over an infinite derivation**

Consider the following logic program P:
$C_1 : p(x, y) \leftarrow p(y, x)$
and let G be the goal $\leftarrow p(a, z)$.
As shown in figure 3.8 on page 60, the SLD-tree of the goal $\leftarrow p(a, z)$ is infinite. But, by applying *Loop check* 4, the SLD-tree is pruned because one can see that the atom $p(a, z)$ (i.e. for the atom $p(z, a)$) is a variant of some ancestors atoms.

SLD-tree of $P \cup \{G\}$	Pruned SLD-tree of $P \cup \{G\}$
$\leftarrow p(a, z)$	$\leftarrow p(a, z)$
$\quad \Big\vert\ C_1, \{x \mapsto a, y \mapsto z\}$	$\quad \Big\vert\ C_1, \{x \mapsto a, y \mapsto z\}$
$\leftarrow p(z, a)$	$\leftarrow p(z, a)$
$\quad \Big\vert\ C_1, \{x \mapsto z, y \mapsto a\}$	$\quad \Big\vert\ C_1, \{x \mapsto z, y \mapsto a\}$
$\leftarrow p(a, z)$	$\underline{\leftarrow p(a, z)}$
$\quad \Big\vert\ C_1, \{x \mapsto a, y \mapsto z\}$	
$\leftarrow p(z, a)$	
$\quad \vdots$	

Figure 3.8: SLD-tree and its associated pruned SLD-tree using *Loop check* 4 for $P \cup \{G\}$

The scope of *Loop check* 4 is rather limited because the three relations of biconditionality, symmetry and transitivity can be circumvented as shown in the next Theorem.

Theorem 3.2.4 (**Soundness and completeness of** *Loop check* 4). *1. Loop check 4 is not sound w.r.t. the leftmost selection rule for Datalog, restricted, nvi, svo and general logic programs.*

2. Loop check 4 is complete w.r.t. the leftmost selection rule for Datalog, restricted, nvi and svo logic programs.

3. *Loop check 4 is not complete w.r.t. the leftmost selection rule for general logic programs.*

Proof. (*Soundness*) To prove that *Loop check* 4 is not sound for Datalog, restricted, *nvi*, *svo* and general logic programs; let us consider the following propositional logic program:
$C_1 : p \leftarrow q, r$;
$C_2 : r \leftarrow q, s$;
$C_3 : q \leftarrow$;
$C_4 : s \leftarrow$
and let G be the following goal $\leftarrow p$.
The corresponding SLD-tree and its associated pruned SLD-tree using *Loop check* 4 are depicted in figure 3.9.

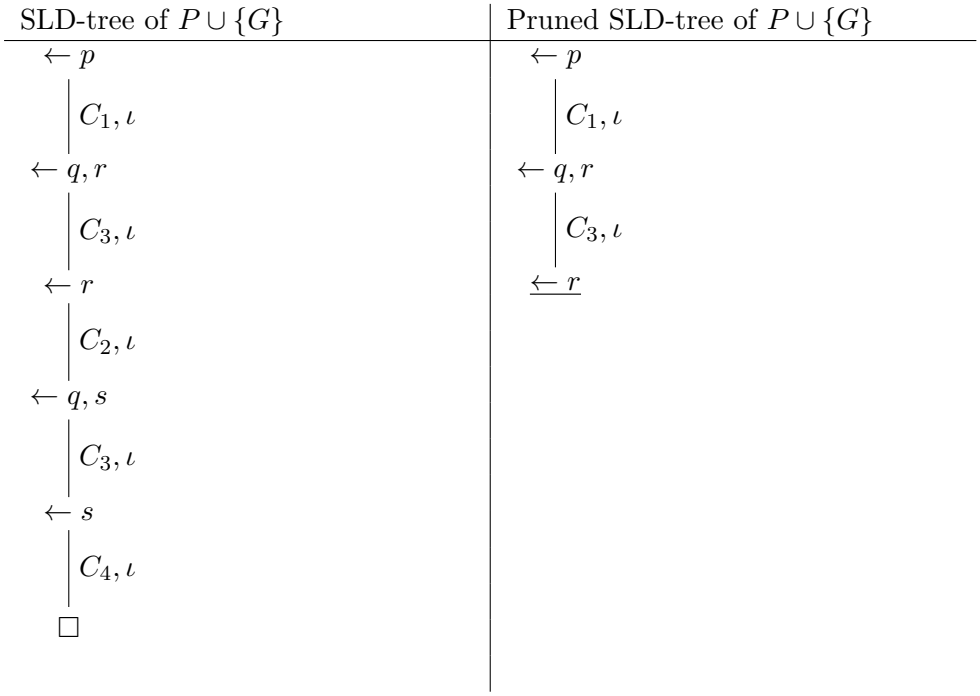


Figure 3.9: SLD-tree and its associated pruned SLD-tree using *Loop check* 4 for $P \cup \{G\}$

As one can see, *Loop check* 4 pruned successful derivations (here the successful one) and thus it is not sound. In fact, the atom r in the subgoal $\leftarrow r$ is a variant of the atom r in the subgoal $\leftarrow q, r$ since $r = r[\iota]$.

(*Completeness*) Let P be a function-free program, G a top goal and D an infinite SLD-derivation of $P \cup \{G\}$. As P contains finite program clauses, finite predicate symbols, and finite set of constants; there must be an infinite set of

atoms, and two goals G_i and G_j in which an atom in G_j is a variant of an atom in G_i.

(*Completeness*) To prove that *Loop check* 4 is not complete for general logic programs, it suffices to consider the following program:
$C_1 : p(x) \leftarrow p(f(x))$
and the following goal $G = \leftarrow p(f(a))$.

Figure 3.10 shows the SLD-tree (resp. the pruned SLD-tree) using *Loop check* 4. All the generated goals in the derivation are not variant of any of the ancestors generated goals, *Loop check* 4 will not prune this SLD-tree.

$$\begin{array}{l} \leftarrow p(f(a)) \\ \qquad \Big| \, C_1, \{x \mapsto f(a)\} \\ \leftarrow p(f(f(a))) \\ \qquad \Big| \, C_1, \{x \mapsto f(f(a))\} \\ \leftarrow p(f(f(f(a)))) \\ \qquad \vdots \end{array}$$

Figure 3.10: SLD-tree for $P \cup \{G\}$

□

We have exposed four basic loop checking techniques that are sometimes sound but not complete or not sound but complete. We will move now and show some complex loop checks which are sound and complete but for some specific types of logic programs.

3.2.6 Loop Check Based on Equal Goals

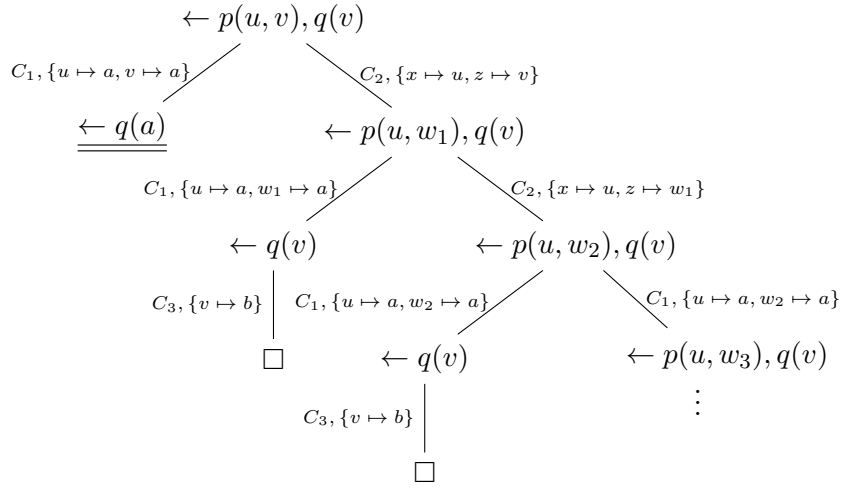
Bol *et al*. [9] proposed to prune infinite SLD-derivations of the form $(G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots)$ whenever there occur two goals G_i and G_j $(0 \leq i < j)$ such that G_j is a variant of G_i.

Definition 3.2.11 (**Loop check based on equal goals**). *This loop check prunes a derivation as soon as it detects that a goal is a variant of an ancestor goal.*
Loop check 5=RemSub$(\{D | D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$*, such that for some* $i, 0 \leq i < k$*, there exists a substitution* τ*, such that:* $G_k = G_i[\tau]\})$.

Example 3.7 Application of *Loop check* 5 **over an infinite derivation**

Consider the following logic program P:
$C_1 : p(a, a) \leftarrow$;
$C_2 : p(x, z) \leftarrow p(x, y)$;
$C_3 : q(b) \leftarrow$
and let G be the goal $\leftarrow p(u, v), q(v)$.

SLD-tree of $P \cup \{G\}$



Pruned SLD-tree of $P \cup \{G\}$

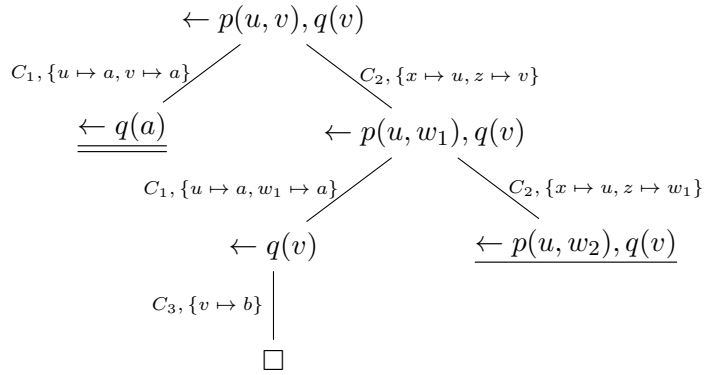


Figure 3.11: SLD-tree and its associated pruned SLD-tree using *Loop check* 5 for $P \cup \{G\}$

As shown in figure 3.11 on page 63, the SLD-tree of the goal $\leftarrow p(a, z)$ is infinite. By applying *Loop check* 5 to the SLD-tree of $P \cup \{G\}$, one can consider the subgoal $\leftarrow p(u, w_2), q(v)$ which is a variant of the goal $\leftarrow p(u, w_1), q(v)$ and thus prunes the infinite branch at $\leftarrow p(u, w_2), q(v)$.

Even with a loop check based on equal goals, soundness and completeness were compromised even for Datalog program. Nevertheless, for restricted Datalog

programs, i.e. a type of logic programs proposed by Bol *et al.* [9] allowing some restricted type of recursiveness, soundness and completeness is guaranteed.

Theorem 3.2.5 (**Soundness and completeness of** *Loop check* 5)**.** *1. Loop check 5 is not sound w.r.t. the leftmost selection rule for Datalog, nvi, svo and general logic programs.*

2. Loop check 5 is sound w.r.t. the leftmost selection rule for restricted logic programs.

3. Loop check 5 is complete w.r.t. the leftmost selection rule for restricted logic programs.

4. Loop check 5 is not complete w.r.t. the leftmost selection rule for Datalog, nvi, svo and general logic programs.

Proof. (*Soundness*) To check that *Loop check* 5 is not sound w.r.t. the leftmost selection rule for Datalog, *svo* and general logic programs, it suffices to refer to the **Example 5.4** given by Bol *et al.* in [9].

(*Soundness*) To check that *Loop check* 5 is not sound w.r.t. the leftmost selection rule for *nvi* logic programs, consider the following program:
$C_1 : p(u, v) \leftarrow p(0, v), r(u, v)$;
$C_2 : p(0, u) \leftarrow$;
$C_3 : q(1) \leftarrow$;
$C_4 : r(u, v) \leftarrow q(u), p(v, v)$
and the following goal $G = \leftarrow p(x, y)$.

Figure 3.12 shows the SLD-tree using *Loop check* 5. The corresponding pruned SLD-tree is the same except for the leftmost branch where a pruning occurs at the resolvent goal $\leftarrow p(y, y)$. In fact, the goal $\leftarrow p(y, y)$ is a variant of the top goal $\leftarrow p(x, y)$ (see that $p(y, y) = p(x, y)[\{x \mapsto y\}]$). Thus, the only successful branch will be pruned and consequently, *Loop check* 5 is not sound for *nvi* logic programs.

(*Completeness*) To check that *Loop check* 5 is complete w.r.t. the leftmost selection rule for restricted logic programs, consult the **Corollary 4.19** presented in Bol *et al.* [9].

(*Completeness*) To prove that *Loop check* 5 is not complete w.r.t. the leftmost selection rule for Datalog *nvi*, *svo* and general logic programs, it suffices to consider the following program:
$C_1 : p(x) \leftarrow p(x), p(x)$

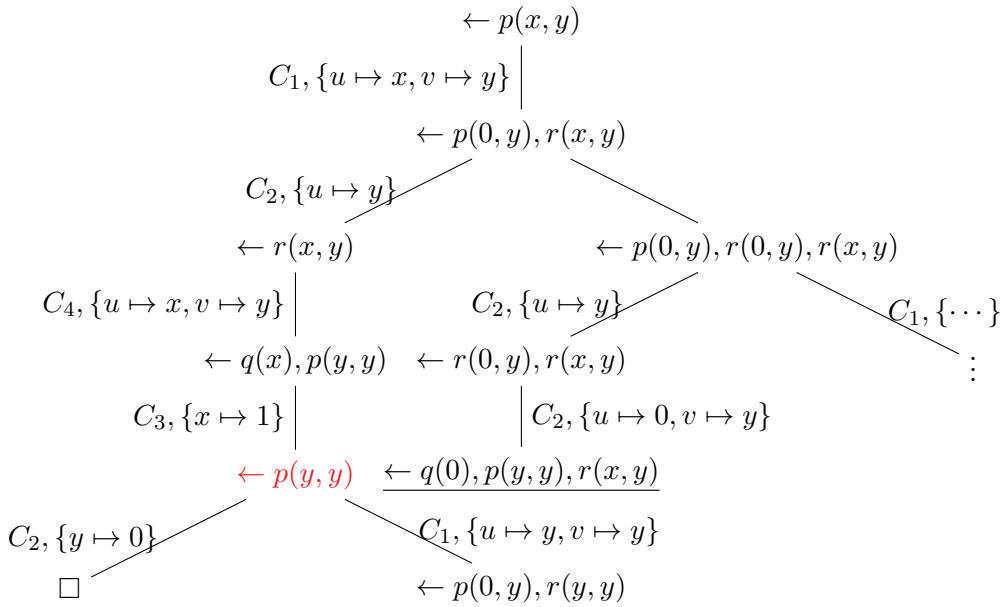


Figure 3.12: SLD-tree - pruned SLD-tree using *Loop check* 5 for $P \cup \{G\}$

and the following goal $G = \leftarrow p(x)$.

Figure 3.7 shows the SLD-tree of $P \cup \{G\}$. The size of the generated goals in the single branch increases with each derivation step and thus a subgoal will never be equal (or variant equal) to any of its ancestor goals. Consequently, *Loop check* 5 is not complete.

□

3.2.7 Loop Check Based on Subsumed Goals

Subsumption loop check proposed by Bol *et al.* [9] prunes infinite SLD-derivations of the form $(G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots)$ whenever there occur two goals G_i and G_j $(0 \leq i < j)$ such that G_j is a variant of some ancestor goal G_i.

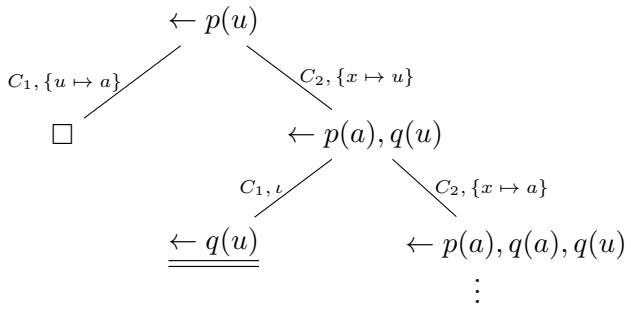
Definition 3.2.12 (Loop check based on equal goals). *This loop check prunes a derivation as soon as it detects that a goal is a general variant of an ancestor goal.*
Loop check 6=RemSub($\{D | D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$, such that for some $i, 0 \leq i < k$, there exists a substitution τ, such that: $G_k \supseteq G_i[\tau]\}$).

Example 3.8 Application of *Loop check* 6 over an infinite derivation

Consider the following logic program P:
$C_1 : p(a) \leftarrow;$

$C_2 : p(x) \leftarrow p(a), q(x)$
and let G be the goal $\leftarrow p(x)$.

SLD-tree of $P \cup \{G\}$



Pruned SLD-tree of $P \cup \{G\}$

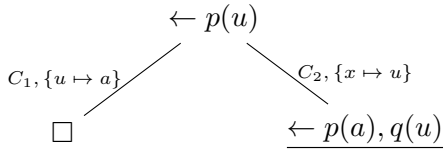


Figure 3.13: SLD-tree and its associated pruned SLD-tree using *Loop check* 7 for $P \cup \{G\}$

Figure 3.13 on page 66 shows the SLD-tree and the pruned SLD-tree of the goal $\leftarrow p(x)$. By applying *Loop check* 6 to the SLD-tree of $P \cup \{G\}$, one can consider the subgoal $\leftarrow p(a), q(u)$ which contains the atom $p(a)$, an instance of $p(u)$ and thus prunes the infinite branch at $\leftarrow p(a), q(u)$. Not that $\underline{\underline{\leftarrow q(u)}}$ is a failure goal.

Loop check 6 is not as restrictive as *Loop check* 5 in the sense that equality is substituted by inclusion. So soundness and completeness results for *Loop check* 5 hold for *Loop check* 6. It is shown also that these results hold for *nvi* and *svo* logic programs.

Theorem 3.2.6 (**Soundness and completeness of** *Loop check* 6)**.** *1. Loop check 6 is not sound w.r.t. the leftmost selection rule for Datalog and general logic programs.*

2. Loop check 6 is sound w.r.t. the leftmost selection rule for restricted, nvi and svo logic programs.

3. Loop check 6 is complete w.r.t. the leftmost selection rule for Datalog, restricted, nvi and svo logic programs.

4. *Loop check 6 is not complete w.r.t. the leftmost selection rule for general logic programs.*

Proof. (*Soundness*) To check that *Loop check* 6 is not sound w.r.t. the leftmost selection rule for Datalog and general logic programs, one can check **Example 5.4** given by Bol *et al.* in [9].

(*Soundness*) To check that *Loop check* 6 is sound w.r.t. the leftmost selection rule for restricted, *nvi* and *svo* logic programs, interested readers can refer to the **Corollary 5.8** presented in Bol *et al.* [9].

(*Completeness*) Let P be a Datalog function-free program, G a top goal and D an infinite SLD-derivation of $P \cup \{G\}$. As P contains finite program clauses, finite predicate symbols, and finite set of constants; there must be an infinite set of atoms, and two goals G_i and G_j such that G_j subsumes G_i. This proves that *Loop check* 6 is complete w.r.t. the leftmost selection rule for Datalog logic programs.

(*Completeness*) To check that *Loop check* 6 is complete w.r.t. the leftmost selection rule for restricted, *nvi* and *svo* logic programs, you can refer to the corollary 5.9, 5.18 and 5.20 presented in Bol *et al.* [9].

(*Completeness*) To prove that *Loop check* 6 is not complete w.r.t. the leftmost selection rule for general logic programs, it suffices to reproduce the incompleteness proof of *Loop check* 4 for general logic programs. □

3.2.8 Loop Check Based on Contextual Approach

Besnard [5] mentioned that the existence of a refutation of $\rightarrow p(x), q(y)$ does not imply the existence of a refutation of $\rightarrow p(x), q(x)$. He suggested to keep track of the links between the variables in atoms and those in the rest of the goal. Thus, Besnard proposed to prune SLD-derivations whenever a goal G_k occurs that contains an instance $A[\tau]$ of an atom A that occurred in an earlier goal G_i. But when a variable occurs both inside and outside of A in G_i, the derivation is not pruned if this link has been altered. Such a variable x in G_i is substituted by $x\theta_{i+1} \cdots \theta_k$ when G_k is reached. Therefore τ and $\theta_{i+1} \cdots \theta_k$ should agree on x.

Definition 3.2.13 (**Loop check based on contextual approach**). *Loop check* $7{=}RemSub(\{D|D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$, *such that for some* i *and* $j, 0 \leq i \leq j < k$, *there is a substitution* τ, *such that for some atom* A *in* G_i*:* $A[\tau]$ *appears in* G_k *as the result of resolving* $A[\theta_{i+1}] \cdots [\theta_j]$ *in* G_j *and for every variable* x *that occurs both inside and outside of* A *in* G_i, $x\theta_{i+1} \cdots \theta_k = x\tau\})$.

Example 3.9 Application of *Loop check* 7 over an infinite derivation

Reconsider the logic program of the example 3.7 on page 63.But, by applying *Loop check* 7, the SLD-tree is pruned (at the same place as *Loop check* 5 pruned the SLD-tree) because firstly, one can see that the atom $p(u, w_2)$ in the subgoal $\leftarrow p(u, w_2), q(v)$ is a variant of $p(u, w_1)$ in the goal $\leftarrow p(u, w_1), q(v)$, and secondly, that by replacing $p(u, w_1)$ by $p(u, w_2)$ in $\leftarrow p(u, w_1), q(v)$, yields $\leftarrow p(u, w_2), q(v)$ which is an instance of $\leftarrow p(u, w_1), q(v)$.

In fact, Besnard, seeing that the existence of a refutation of $\leftarrow A(y), B(x)$ does not imply the existence of a refutation of $\leftarrow A(x), B(x)$, wanted to keep track of the links between the variables in the atom and those of the rest of the goal. By tracking the same variable and thus the same atom, the condition of detecting some repeated subgoals is now stronger, in such a way that soundness is now guaranteed for restricted, *nvi* and *svo* logic programs.

Theorem 3.2.7 (**Soundness and completeness of** *Loop check* 7)**.** *1. Loop check 7 is sound w.r.t. the leftmost selection rule for restricted, nvi and svo logic programs.*

2. Loop check 7 is complete w.r.t. the leftmost selection rule for Datalog, restricted, nvi and svo logic programs.

3. Loop check 7 is not complete w.r.t. the leftmost selection rule for general logic programs.

Proof. (*Soundness*) To check that *Loop check* 7 is sound w.r.t. the leftmost selection rule for restricted, *nvi* and *svo* logic programs, refer to the **Corollary 6.7** presented in Bol *et al.* [9].

(*Completeness*) Similar proof as for Datalog programs for *Loop check* 6.

(*Completeness*) To check that *Loop check* 7 is complete w.r.t. the leftmost selection rule for restricted, *nvi* and *svo* logic programs, refer to the **Corollary 6.12** and **Corollary 6.14** presented in Bol *et al.* [9].

(*Completeness*) To prove that *Loop check* 7 is not complete w.r.t. the leftmost selection rule for general logic programs, it suffices to reproduce the incompleteness proof of *Loop check* 4 for general logic programs. □

3.2.9 Notes on Efficient Loop Checks

All the loop checks presented in the previous subsections compare the derived goals in the derivation and prune it until a sufficiently similar goal is encountered.

Theoretically, a subgoal is usually compared with every previous goal in the derivation. In practice, such loop checks are too expensive since a loop check can perform $\frac{1}{2}|D||D-1|$ comparisons for a finite SLD-derivation D.

Two attempts were proposed in the literature to reduce the number of comparisons by a loop check:

1. **The tortoise-and-hare technique**: Van Gelder [33] proposed to compare every goal G_k to one previous goal, namely the goal halfway the derivation $G_{k/2}$. It is shown that the tortoise-and-hare technique preserves the soundness of the loop check but not its completeness.

2. **The selected technique**: Bol [8] proposed not just to select the goal halfway the derivation to compare it to G_k but to make an infinite selection of goals and compare each of the goals to G_k. It is shown that the selected technique preserves the soundness of the loop check and its completeness (for specific kind of loop checks and logic programs).

3.3 "Flows" In Logic Programming

Generally, in logic programs, the concept of **input** and **output** arguments does not exist. We also say that logic programs are not directed, in the sense that a logic program may be run in either a **forward** or a **backward** direction. Moreover, the unifications of subgoals (i.e. the control flow) in logic programs can proceed in two directions: continuing after a success and backtracking after a failure. Thus, in order to generate efficient executable code of logic programs, researchers studied intensively various dependencies in logic programs like **mode information** [23, 53, 70, 73], **mode inference** [13, 22, 54] and **data dependencies** [17, 18, 27]. While mode information is used primarily for the good understanding of the logic program, data dependency information can be used for various source-level optimizing transformations and to improve backtracking behavior. In the next, we will present data flow and control flow analysis briefly for logic programming, highlighting the fact that the notion of information flow in security systems previously presented in chapter 2 is not covered by those analysis for logic programs.

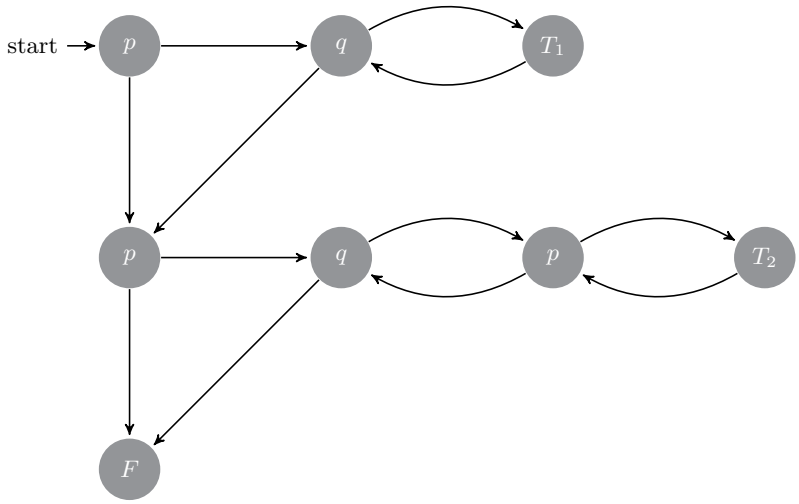
3.3.1 Control Flow Analysis

Control flow in logic programs refers to the order in which a goal is evaluated. These control flows are not so obvious as in imperative programs. This is due to the fact that the control flows are hidden in logic programs, which is in turn a result from the declarative nature of these programs. The semantics of Prolog, for example, implies several types of backtracking (caused by the failure of subgoal matching and by the requirement for multi-answers for one single goal), and searching for a subgoal to which control is transferred after backtracking. To describe these control flows explicitly, **control flow graph**s were proposed. A control flow graph (in short CFG) is a quadruple $< s, e, V, E >$, where (V, E) is a directed graph, $S \in V$ is the unique start node and $e \in V$ the unique end node, such that there is a path from s to every node in V and there is a path from any node in V to e. Constructing the CFG is done incrementally by adding nodes and arcs to the graph. The following example shows a logic program along with its corresponding control flow graph representation (interested reader can refer to [51] for algorithms on how to generate control flow graph representations). Note that, in this representation, the node s is p (the upper right one) and the node e is F.

Example 3.10 Flowgraph representation of a logic program

Let P the following logic program:
$C_1 : p(a, y) \leftarrow q(a, y);$
$C_2 : p(x, y) \leftarrow q(x, z), p(z, y);$

$C_3 : q(x, y) \leftarrow$
The corresponding flowgraph representation of the predicate definition p is as follows:



This flowgraph represents how a goal $\leftarrow p(x, y)$ could be evaluated by showing the different predicates invocation and all the possible paths between them.

3.3.2 Data Flow Analysis / Dependence analysis

A data dependency in logic programs exists when a clause is referring to a variable/argument in the same clause definition or in the head of some other clause definition.

To refer to an argument in a program P, Boye *et al.* [11] give each argument of P a unique label called **argument position**.

Definition 3.3.1 (Argument position)**.** *if c is a clause of the form $a_0 \leftarrow a_1, \cdots, a_n$, the position of the k^{th} argument of the j^{th} literal is uniquely defined in the program P by the tuple $< c, j, p, k >$, where p is the predicate symbol of the j^{th} literal of c. By convention, the fictive position $< c, i, p, 0 >$ denotes the i^{th} literal of the clause c of which the predicate symbol is p.*

Example 3.11 Examples of argument position in logic program

Let c be the following clause:
$c : p(x, y) \leftarrow r(x, z), q(z, y)$
The position of the argument z in the body literal $q(z, y)$ is $< c, 2, q, 1 >$.
The position of the argument y in the head literal $p(x, y)$ is $< c, 0, p, 2 >$.
The position of the body literal $r(x, y)$ is $< c, 1, r, 0 >$.

Unification of positions is defined as follows:

Definition 3.3.2 (Unification of positions)**.** *Let α and β be the positions of two arguments a and b such that $\alpha =< c_1, j, p, k >$ and $\beta =< c_2, 0, p, k >$ where $j \neq 0$.*
unify(α, β) iff there exists a renaming substitution θ such that $a[\theta]$ has no variable in common with b and there exists a substitution σ such that $a[\theta][\sigma] = b[\sigma]$

Many works have been carried on data dependency analysis. The purpose of this section is neither to expose in details any analysis nor to compare existing algorithms. We only recall the definitions of Boye *et al.* [11] concerning internal and external data dependencies and program dependency graph.

Predicate arguments in a logic program, can be annotated by the user either as **inherited**$(\downarrow)$ or **synthesized**$(\uparrow)$ or **unannotated**$(\square)$ (Boye *et al.* [11] was inspired by attribute grammar theory). Intuitively, data is *brought in* to a clause through the input positions, and *sent out* through the output positions.

Definition 3.3.3 (Input and Output positions)**.** *$< c, j, q, k >$ is called an **input position** if:*
** the annotation of the k^{th} argument of q is $\downarrow$ and a_j is the head atom of c, or*
** the annotation of the k^{th} argument of q is $\uparrow$ and a_j is a body atom of c.*
Let $I(c)$ denote the input positions of the clause c.

*$< c, j, q, k >$ is called an **output position** if:*
** the annotation of the k^{th} argument of q is $\uparrow$ and a_j is the head atom of c, or*
** the annotation of the k^{th} argument of q is $\downarrow$ and a_j is a body atom of c.*
Let $O(c)$ denote the output positions of the clause c.

In logic program, information is passed in two ways: either **within** a clause (between two positions sharing a variable), or **between** two clauses (through unification). If an annotation for the program is known, the **direction**, in which the information is passed, became also known: from input positions to output positions within a clause, and from output positions to input positions at unification.

Definition 3.3.4 (Internal Data Dependency)**.** *For a clause c, and two positions α and β such that $\alpha \in I(c)$ and $\beta \in O(c)$, the internal data dependence relation $\rightarrow_i$ is defined as follows:*
$\alpha \rightarrow_i \beta$ iff α and β have at least one common variable.

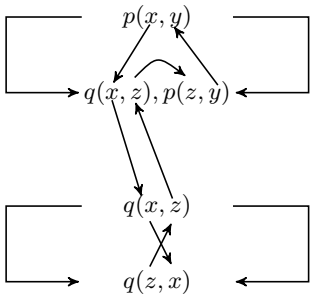
Definition 3.3.5 (External Data Dependency)**.** *For two clauses c, d, and two positions $\alpha =< c, j, q, k >$ and $\beta =< d, 0, q, k >$ such that $unify(\alpha, \beta)$ holds, the external data dependence relation $\rightarrow_e$ is defined as follows:*

$\alpha \rightarrow_e \beta$ *iff* $\alpha \in O(c)$ *and* $\beta \in I(c)$, *and*
$\beta \rightarrow_e \alpha$ *iff* $\alpha \in I(c)$ *and* $\beta \in O(c)$.

The data dependency relation $\rightarrow$ is equal to $\rightarrow_i \cup \rightarrow_e$. The transitive closure is denoted by $\rightarrow^*$. The **program dependency graph** $\rightarrow_p$ is equal to the union of all the internal and external data dependencies in the program, denoted by $\bigcup \rightarrow_i \cup \bigcup \rightarrow_e$.

Example 3.12 Program dependency graph

Let P the following logic program:
$C_1 : p(x, y) \leftarrow q(x, z), p(z, y)$;
$C_2 : q(x, z) \leftarrow q(z, x)$;
$C_3 : q(b, a) \leftarrow$;
$C_4 : p(a, b) \leftarrow$
The corresponding dependency graph is as follows:



Dependence analysis in logic programs determines, for instance, whether or not it is safe to reorder program clauses or furthermore to parallelize the program [17, 18, 73].

3.4 Summary

In this chapter, we examined first-order logic programming by reviewing the notions of substitutions, unification, SLD-resolution, SLD-trees and the problems of SLD-resolutions. Limitation due to the presence of loops that can causes the search for an SLD-refutation result in non-termination.

Thus, we have exposed several loop checking techniques and discussed its soundness and completeness. Table 3.1 on page 75 summarizes this results, where ✗ in the sound column means that the loop check is *not sound* and ✗ in the complete column means that the loop check is *not complete*. ✓ in the sound column means that the loop check is *sound* and ✓ in the complete column means that the loop check is *complete*.

Flows in logic programming, like control flow analysis and data/dependence analysis are then exposed, highlighting the fact that the information flow notion presented in chapter 2 is not covered by those analysis in logic programming.

Thus, in the next chapter, we will try to adapt the notion of information flow in security systems to logic programming. We will propose three definitions of information flows in logic programs. Definitions that correspond to what can be observed by a user when a query $\leftarrow G(x, y)$ is run on a logic program P. Implications between these definitions are discussed and complexity results are also given for selected decision problems.

	Datalog		Restricted		*Nvi*, *Svo*		General logic program	
	Sound	Complete	Sound	Complete	Sound	Complete	Sound	Complete
Loop check 1	✗	✓	✗	✓	✗	✓	✗	✓
Loop check 2	✓	✗	✓	✗	✓	✗	✓	✗
Loop check 3	✓	✗	✓	✗	✓	✗	✓	✗
Loop check 4	✗	✓	✗	✓	✗	✓	✗	✗
Loop check 5	✗	✗	✓	✓	✗	✗	✗	✗
Loop check 6	✗	✓	✓	✓	✓	✓	✗	✗
Loop check 7	?	✓	✓	✓	✓	✓	?	✗

Table 3.1: Soundness and completeness of *Loop check* 1 to *Loop check* 7 for Datalog, restricted, *nvi*, *svo* and general logic programs

Chapter 4

Information Flow in Logic Programming

As stated earlier, data security is the science and study of methods of protecting data in computer and communication systems from unauthorized disclosure and modification by controlling information flow in the system. In some sense, an information flow should describe controls that regulate the dissemination of information. These controls are needed to prevent programs from leaking confidential data, or from disseminating classified data to users with lower security clearances.

The theory of information flow in security systems is well defined for imperative programming. Different models of information flow were proposed, namely, the Bell-LaPadula Model [3], nondeducibility and noninterference [34] models. Each model has rules about the conditions under which information can be transfered throughout the system.

Several studies [24] addressed information flow in security systems for imperative programming, but none were concerned to bring answers of what could be an information flow in security systems for logic programming.

In this chapter, we propose three definitions of information flows in Datalog logic programs. These definitions correspond to what can be observed by the user when a query $\leftarrow G(x, y)$ is run on a logic program P.

Firstly, we consider that the user only sees whether her queries succeed or fail. In this respect, we say information flows from x to y in G when there exists

constants a, b such that $\leftarrow G(a, y)$ succeeds whereas $\leftarrow G(b, y)$ fails.

Secondly, we assume that the user has also access to the sets of substitution answers computed by the interpreter with respect to her queries. As a result, in this case, there is a flow of information from x to y in G if there are constants a, b such that the substitution answers of $\leftarrow G(a, y)$ and $\leftarrow G(b, y)$ are different.

Thirdly, we suppose that the user, in addition to the substitution answers, also observes the SLD-refutation trees produced by the interpreter. If the SLD-trees of the queries $\leftarrow G(a, y)$ and $\leftarrow G(b, y)$ can be distinguished in one way or another by the user, then we will say that information flows from x to y in G. Of course, it remains to properly define what "distinguished" means in our setting. Following a traditional view in program semantics, we will base distinguishability of SLD-refutation trees on the notion of bisimilarity.

In sections 4.1, 4.2 and 4.3, several definitions of information flow in logic programming are proposed relatively for a Datalog logic program P and a goal $\leftarrow G(x, y)$ of arity 2, (which stipulates the existence of a flow from the variable x to the variable y in the goal $\leftarrow G(x, y)$). The implications between these definitions are then studied in section 4.4. Decision procedures are then given in section 4.7 for each of the previous definitions and computational issues studied for some types of logic programs.

As the theory of information flow is well studied for imperative programming, it is tempting to see what could be an information flow in logic programming, especially given the fact that there are no notions of assignment, or variable of a program. In fact, variables in logic programs behave differently from variables in conventional programming languages. They stand for an unspecified but single entity rather than for a store location in memory.

The following three definitions for information flow in logic programming are based on the following principle. The information flow that occurs when the user asks a goal to logic programs depends mainly on what parts of the computation the user sees. In the first definition, the user only sees whether goals succeed or fail. In the second definition, the user has access to the set of substitution answers computed by the program. In the third definition, the user obtains the shape of the computation trees produced by the program.

It is now time to present our three definitions of information flows in logic programs.

4.1 Information Flow based on Success / Failure

Let P be a Datalog logic program, and $G(x, y)$ be a two variables goal. We shall say that there is a flow from x to y in $G(x, y)$ with respect to successes and failures in P (in symbols $x \xrightarrow{SF}{}^{P}_{G} y$) iff there exists $a, b \in U_{L(P)}$ such that $P \cup \{G(a, y)\}$ succeeds and $P \cup \{G(b, y)\}$ fails. This intuitively means that when the user only sees the outputs of computations in terms of successes and failures, there exists two different $a, b \in U_{L(P)}$ such that this user can distinguish (without seeing what concerns a, b) between the output for $P \cup \{G(a, y)\}$ and the output for $P \cup \{G(b, y)\}$.

Example 4.1 Example of an information flow in logic programming based on success and failure

Let P_1 be the following program:
$C_1 : murderer(angel, billy) \leftarrow$;
$C_2 : murderer(george, bob) \leftarrow$

and let $G_1(x, y)$ be the following goal: $\leftarrow murderer(x, y)$
Since $P_1 \cup \{G_1(angel, y)\}$ succeeds and $P_1 \cup \{G_1(bob, y)\}$ fails, then $x \xrightarrow{SF}{}^{P_1}_{G_1} y$.

4.2 Information Flow based on Substitution Answers

Let P be a Datalog logic program, and $G(x, y)$ be a two variables goal. We shall say that there is a flow from x to y in $G(x, y)$ with respect to substitution answers in P (in symbols $x \xrightarrow{SA}{}^{P}_{G} y$) iff there exists $a, b \in U_{L(P)}$ such that $\Theta(P \cup \{G(a, y)\}) \neq \Theta(P \cup \{G(b, y)\})$. Roughly speaking, in this definition, the user only sees the outputs of computations in terms of substitution answers. As a result, there is a flow if this user can distinguish (without seeing what is about a, b) the output of $P \cup \{G(a, y)\}$ and the output of $P \cup \{G(b, y)\}$.

Example 4.2 Example of an information flow in logic programming based on substitution answers

Let P_2 be the following program:
$C_1 : murderer(angel, billy) \leftarrow;$
$C_2 : murderer(george, bob) \leftarrow$

and let $G_2(x, y)$ be the following goal: $\leftarrow murderer(x, y)$
Since $\Theta(P_2 \cup \{G_2(angel, y)\}) = \{y \mapsto billy\}$ and $\Theta(P_2 \cup \{G_2(bob, y)\}) = \emptyset$, then $x \xrightarrow{SA}{}^{P_2}_{G_2} y$.

4.3 Information Flow based on Bisimulation

Our third definition of flow is based on the notion of bisimulation between goals. Quite simply, a bisimulation is a binary relation between goals such that related goals have "equivalent" SLD-trees.

4.3.1 Bisimulation Definition

Let P be a Datalog program. A binary relation $\mathcal{Z}$ between goals is said to be a P-bisimulation iff it satisfies the following conditions; for all Datalog goals F_1, G_1 such that $F_1 \mathcal{Z} G_1$:

- $F_1 = \square$ iff $G_1 = \square$,
- For each SLD-resolvent F_2 of F_1 (i.e. $F_2 \in succ_P(F_1)$) and a clause in P, there exists a resolvent G_2 of G_1 (i.e. $G_2 \in succ_P(G_1)$) and a clause in P such that $F_2 \mathcal{Z} G_2$,
- For each SLD-resolvent G_2 of G_1 (i.e. $G_2 \in succ_P(G_1)$) and a clause in P, there exists a resolvent F_2 of F_1 (i.e. $F_2 \in succ_P(F_1)$)and a clause in P such that $F_2 \mathcal{Z} G_2$.

Above, $succ_P(G)$ denotes the set of all goals obtained from a goal G by means of a resolution step in the program P.

Proposition 4.3.1. *The identity relation Id between goals is a $P-$bisimulation.*

Proof. Let $\mathcal{Z}$ be a $P - bisimulation$ and G a goal. Suppose that $not\{G \mathcal{Z} G\}$. We will denote the right goal G in $G \mathcal{Z} G$ by G_{right} and the left goal by G_{left}. If $G_{left} = \square$, then, for sure $G_{right} = \square$ and thus, the first condition of the bisimulation holds. Let G'_{left} be a successor of G_{left}. Eventually, G_{right} will have a successor G'_{right} (which is equal to G'_{left}). With a similar reasoning, we can prove the third condition of the bisimulation. Thus, a contradiction, and $G \mathcal{Z} G$. □

Proposition 4.3.2. *If $\mathcal{Z}$ is a $P-$bisimulation, then $\mathcal{Z}^{-1}$ is also a $P-$bisimulation.*

Proof. We define the converse $\mathcal{Z}^{-1}$ by $\mathcal{Z}^{-1} = \{(H, G)/G \mathcal{Z} H\}$. Let $\mathcal{Z}$ be a $P - bisimulation$. Suppose that $G \mathcal{Z} H$, let us prove that $not\{H \mathcal{Z} G\}$. Suppose that $G = \square$, as $G \mathcal{Z} H$, then $H = \square$. Thus the first condition holds for bisimulation. Let G' be a successor of G. Since $G \mathcal{Z} H$, then, there exists a successor H' of H such that $G' \mathcal{Z} H'$. In addition, if I' is a successor of H, then there exists a

successor J' of G such that $J'\mathcal{Z}I'$. Thus, the second and third conditions hold for the bisimulation. A contradiction that leads to the fact that $H\mathcal{Z}G$. □

Proposition 4.3.3. *If $\mathcal{Z}_1$, $\mathcal{Z}_2$ are two $P-$ bisimulations, then the composition $\mathcal{Z}_1 \circ \mathcal{Z}_2$ defined by $\mathcal{Z}_1 \circ \mathcal{Z}_2 = \{(G,H)/\exists I, G\mathcal{Z}_1 I$ and $I\mathcal{Z}_2 H\}$ is also a $P-$ bisimulation.*

Proof. Suppose that $(G,H) \in \mathcal{Z}_1 \circ \mathcal{Z}_2$. Then there exists I such that $G\mathcal{Z}_1 I$ and $I\mathcal{Z}_2 H$. Since $G\mathcal{Z}_1 I$, $G = \square$ iff $I = \square$, and as $I\mathcal{Z}_1 H$, then $I = \square$ iff $H = \square$. Thus, $G = \square$ iff $H = \square$. Now, let G' be a successor of G. Since $G\mathcal{Z}_1 I$, then there exists I' successor of I such that $G'\mathcal{Z}_1 I'$. Since $I\mathcal{Z}_2 H$, then, there exists H' successor of H such that $I'\mathcal{Z}_2 H'$. Thus, $(G',H') \in \mathcal{Z}_1 \circ \mathcal{Z}_2$. With a similar reasoning, if H' is a successor of H, we can find a G' successor of G such that $(G',H') \in \mathcal{Z}_1 \circ \mathcal{Z}_2$. □

Proposition 4.3.4. *Let $(\mathcal{Z}_i)_{i\in I}$ be a family of $P-$ bisimulations, then $\bigcup_{i\in I} \mathcal{Z}_i$ is also a $P-$ bisimulation .*

As all P-bisimulations are closed under taking arbitrary unions, as shown by the proposition 4.3.4, one can show that:

Proposition 4.3.5. *There exists a maximal P-bisimulation, namely the binary relation $\mathcal{Z}^P_{max}$ between goals defined as follows: $F_1\mathcal{Z}^P_{max}G_1$ iff there exists a P-bisimulation $\mathcal{Z}$ such that $F_1\mathcal{Z}G_1$.*

It follows immediately that:

Proposition 4.3.6. *$\mathcal{Z}^P_{max}$ is an equivalence relation on the set of all goals.*

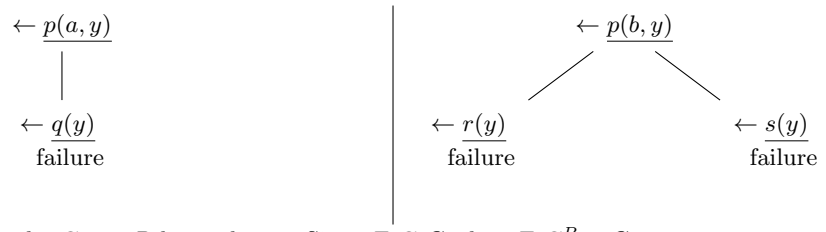
Proof. By propositions 4.3.1 – 4.3.3. □

Example 4.3 Example of bisimulation betweens logic goals

Let P be the following program:
$C_1 : p(a,y) \leftarrow q(y)$;
$C_2 : p(b,y) \leftarrow r(y)$;
$C_3 : p(b,y) \leftarrow s(y)$

and let G, H be respectively the following goals $\leftarrow p(a,y)$ and $\leftarrow p(b,y)$

Let Z be the binary relation between goals such that:
$\leftarrow p(a,y)\ Z \leftarrow p(b,y)$,
$\leftarrow q(y)\ Z \leftarrow r(y)$,
$\leftarrow q(y)\ Z \leftarrow s(y)$.



Obviously, $\mathcal{Z}$ is a *P-bisimulation*. Since $F\ \mathcal{Z}\ G$, then $F\ \mathcal{Z}^P_{max}\ G$.

4.3.2 Flow Definition

Let P be a logic program, and $G(x, y)$ be a two variables goal. We shall say that there is a flow from x to y in $G(x, y)$ with respect to the bisimulation in P (in symbols $x \xrightarrow{BI}{}^{P}_{G}\ y$) iff there exists $a, b \in U_{L(P)}$ such that $not\{P \cup \{G(a, y)\}\mathcal{Z}^P_{max}P \cup \{G(b, y)\}\}$. In this definition, there is a flow if the user, by only seeing the outputs of computations in terms of bisimulation between goals, can distinguish (without looking at a, b) the output of $P \cup \{G(a, y)\}$ and the output of $P \cup \{G(b, y)\}$.

Example 4.4 Example of an information flow in logic programming based on bisimulation

Let P_3 be the following program:
$C_1 : p(x, a) \leftarrow;$
$C_2 : p(a, b) \leftarrow q(a)$

and let $G_3(x, y)$ be the goal: $\leftarrow p(x, y)$.

Let us prove $not\{P_3 \cup \{\leftarrow p(a, y)\}\mathcal{Z}^{P_3}_{max}P_3 \cup \{\leftarrow p(b, y)\}\}$. Suppose that $P_3 \cup \{\leftarrow p(a, y)\}\mathcal{Z}^{P_3}_{max}P_3 \cup \{\leftarrow p(b, y)\}$. Since $\leftarrow q(a) \in succ_{P_3}(\leftarrow p(a, y))$, then there should be $G_3 \in succ_{P_3}(\leftarrow p(b, y))$ such that $P_3 \cup \{\leftarrow q(a)\}\mathcal{Z}^{P_3}_{max}P_3 \cup \{G_3\}$. The problem is that the only goal in $succ_{P_3}(\leftarrow p(b, y))$ is the empty goal, which cannot be bisimilar to $\leftarrow q(a)$. Hence, $not\{P_3 \cup \{\leftarrow p(a, y)\}\mathcal{Z}^{P_3}_{max}P_3 \cup \{\leftarrow p(b, y)\}\}$. Therefore $x \xrightarrow{BI}{}^{P_3}_{G_3}\ y$.

4.4 Links Between the Different Types of Information Flow

The existence of a flow with respect to substitution answers does not entail the existence of a flow with respect to successes and failures. To see this, it suffices to consider the following example.

Example 4.5 Example of an information flow in logic programming where the existence of a flow with respect to substitution answers does not entail the existence of a flow with respect to successes and failures

Let P be the following program:
$C_1 : love(angel, bob) \leftarrow$;
$C_2 : love(x, carl) \leftarrow$

and let $G(x, y)$ be the goal: $\leftarrow love(x, y)$.

Since $\Theta(P \cup \{G(angel, y)\}) = \{y \mapsto bob, y \mapsto carl\}$ and $\Theta(P \cup \{G(bob, y)\}) = \{y \mapsto carl\}$, then $x \overset{SA}{\longrightarrow}{}^{P}_{G} y$. Since $P \cup \{G(a, y)\}$ and $P \cup \{G(b, y)\}$ both succeed, then $x \overset{SF}{\not\longrightarrow}{}^{P}_{G} y$.

However, one can establish the following result.

Lemma 4.4.1. *Let P be a logic program and $G(x, y)$ be a two variables goal. If $x \overset{SF}{\longrightarrow}{}^{P}_{G} y$ then $x \overset{SA}{\longrightarrow}{}^{P}_{G} y$.*

Proof. Suppose that $x \overset{SF}{\longrightarrow}{}^{P}_{G} y$, then there exists $a, b \in U_{L(P)}$ such that $P \cup \{G(a, y)\}$ succeeds and $P \cup \{G(b, y)\}$ fails. Therefore, $\Theta(P \cup \{G(a, y)\}) \neq \emptyset$ and $\Theta(P \cup \{G(b, y)\}) = \emptyset$. Consequently, $x \overset{SA}{\longrightarrow}{}^{P}_{G} y$. □

The existence of a flow with respect to bisimulation does not entail the existence of a flow with respect to successes and failures. The next example explains why.

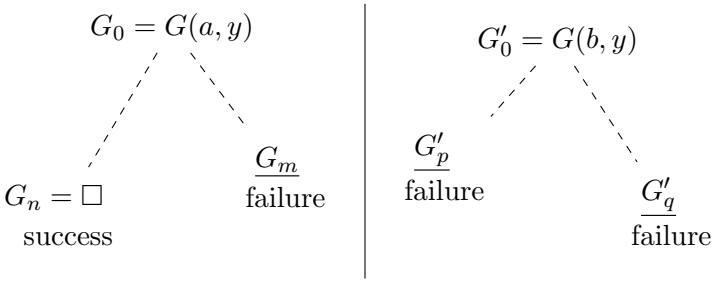
Example 4.6 Example of an information flow in logic programming where the existence of a flow with respect to bisimulation does not entail the existence of a flow with respect to successes and failures

Let P_3 and G_3 be the program and goal considered in example 4.4. We know that $x \overset{BI}{\longrightarrow}{}^{P_3}_{G_3} y$. Nevertheless, since all the goals of the form $G_3(a, y)$, with $a \in U_{L(P)}$, succeed, thus $x \overset{SF}{\not\longrightarrow}{}^{P_3}_{G_3} y$.

Nevertheless, it is worth noting at this point the following.

Lemma 4.4.2. *Let P be a logic program and $G(x,y)$ be a two variables goal. If $x \xrightarrow{SF}{}^{P}_{G} y$ then $x \xrightarrow{BI}{}^{P}_{G} y$.*

Proof. Suppose that $x \xrightarrow{SF}{}^{P}_{G} y$. Thus, there exists $a, b \in U_{L(P)}$ such that $P \cup \{G(a,y)\}$ succeeds and $P \cup \{G(b,y)\}$ fails. Suppose that $P \cup \{G(a,y)\} \mathcal{Z}^{P}_{max} P \cup \{G(b,y)\}$. Since $P \cup \{G(a,y)\}$ succeeds, then there exists an SLD-refutation $G_0 \Rightarrow \cdots \Rightarrow G_n$ of $G(a,y)$ in P. That is to say, $G_0 = G(a,y)$, $G_n = \square$ and G_i is a successor of G_{i-1} in P for $i = 1 \cdots n$.
Since $P \cup \{G(a,y)\} \mathcal{Z}^{P}_{max} P \cup \{G(b,y)\}$ in P, thus $P \cup \{G(b,y)\}$ succeeds: a contradiction. Thus, $not\{P \cup \{G(a,y)\} \mathcal{Z}^{P}_{max} P \cup \{G(b,y)\}\}$ and $x \xrightarrow{BI}{}^{P}_{G} y$.



□

4.5 Information Flow over Goals with Arity > 2

We now generalize the previous definitions by considering goals with arity higher than two. Firstly, we consider information flows between two variables. Secondly, we consider information flows between two sets of variables.
The generalization of the previous three definitions to goals with arity higher than two and by only considering information flows between two variables leads us to the following three definitions:

Definition 4.5.1. *For a logic program P and a goal $G(x_1, \cdots, x_k, \cdots, x_m, \cdots, x_p)$ of arity p*

$$x_k \xrightarrow{SF}{}^{P}_{G(x_1,\cdots,x_k,\cdots,x_m,\cdots,x_p)} x_m \text{ iff}$$
$$\exists a, a' \in U_{L(P)},\ a \neq a'$$
$$\exists c_1, \cdots, c_{k-1}, c_{k+1}, \cdots, c_{m-1}, c_{m+1}, \cdots, c_p \in U_{L(P)} \text{ such that}$$
$$P \cup \{G(c_1, \cdots, c_{k-1}, a, c_{k+1}, \cdots, c_{m-1}, x_m, c_{m+1}, \cdots, c_p)\} \text{ succeeds}$$
$$\text{and}$$
$$P \cup \{G(c_1, \cdots, c_{k-1}, a', c_{k+1}, \cdots, c_{m-1}, x_m, c_{m+1}, \cdots, c_p)\} \text{ fails}$$

Definition 4.5.2. *For a logic program P and a goal $G(x_1, \cdots, x_k, \cdots, x_m, \cdots, x_p)$ of arity p*

$$x_k \xrightarrow{SA}{}^{P}_{G(x_1,\cdots,x_k,\cdots,x_m,\cdots,x_p)} x_m \text{ iff}$$
$$\exists a, a' \in U_{L(P)},\ a \neq a'$$
$$\exists c_1, \cdots, c_{k-1}, c_{k+1}, \cdots, c_{m-1}, c_{m+1}, \cdots, c_p \in U_{L(P)} \text{ such that}$$
$$\Theta[P \cup \{G(c_1, \cdots, c_{k-1}, a, c_{k+1}, \cdots, c_{m-1}, x_m, c_{m+1}, \cdots, c_p)\}]$$
$$\neq$$
$$\Theta[P \cup \{G(c_1, \cdots, c_{k-1}, a', c_{k+1}, \cdots, c_{m-1}, x_m, c_{m+1}, \cdots, c_p)\}]$$

Definition 4.5.3. *For a logic program P and a goal $G(x_1, \cdots, x_k, \cdots, x_m, \cdots, x_p)$ of arity p*

$$x_k \xrightarrow{BI}{}^{P}_{G(x_1,\cdots,x_k,\cdots,x_m,\cdots,x_p)} x_m \text{ iff}$$
$$\exists a, a' \in U_{L(P)},\ a \neq a'$$
$$\exists c_1, \cdots, c_{k-1}, c_{k+1}, \cdots, c_{m-1}, c_{m+1}, \cdots, c_p \in U_{L(P)} \text{ such that}$$
$$not\{P \cup \{G(c_1, \cdots, c_{k-1}, a, c_{k+1}, \cdots, c_{m-1}, x_m, c_{m+1}, \cdots, c_p)\}$$
$$\mathcal{Z}_{max}$$
$$P \cup \{G(c_1, \cdots, c_{k-1}, a', c_{k+1}, \cdots, c_{m-1}, x_m, c_{m+1}, \cdots, c_p)\}\}$$

The idea behind the previous definitions is that in order to see if there is a flow from x_k to x_m, one can try to instantiate the $p-2$ other variables to some constants and to find two constants a, a' for which the instantiations of the variable x_k by a or a' leads to a success and failure for the first definition, or

different substitutions answers for the second definition or two different shapes of resolution trees for the third definition.

By considering $x_1 = x$, $x_m = y$ and $p = 2$, we find again the same information flow definitions for goals of arity two. In addition, with this generalization, the results of lemma 4.4.1 and 4.4.2 are also preserved.

Now we will generalize the previous notions to cover information flows from a set of variables to a set of variables. For this, we will proceed in 2 steps: first we give a definition of the information flow from a set of variables to a single variable, and then we generalize it from a set of variables to a set of variables.

1. **Information flow from a set of variables to a single variable**
   For a program P and a goal $G(x_1, \cdots, x_k, x_l, x_m, \cdots, x_n)$, we say that there is a flow from $\{x_1, \cdots, x_k\}$ to x_l iff one can instantiate the variables $\{x_m, \cdots, x_n\}$ by some constants and instantiate the variables $\{x_1, \cdots, x_k\}$ in two different manners and thus lead to a success and failure / different substitution answers / non bisimilar goals.

   Definition 4.5.4 (Information flow definition from a set of variables to a single variable relatively to the definition of success / failure)**.**

$$\begin{gathered}\{x_1, \cdots, x_k\} \xrightarrow{SF}{}^{P}_{G(x_1,\cdots,x_k,x_l,x_m,\cdots,x_n)} x_l \textit{ iff} \\ \exists c_m, \cdots, c_n \in U_{L(P)}, \exists a_1, \cdots, a_k, \exists a'_1, \cdots, a'_k \in U_{L(P)} \textit{ such that} \\ (a_1, \cdots, a_k) \neq (a'_1, \cdots, a'_k) \textit{ and} \\ P \cup \{G(a_1, \cdots, a_k, x_l, c_m, \cdots, c_n)\} \textit{ succeeds and} \\ P \cup \{G(a'_1, \cdots, a'_k, x_l, c_m, \cdots, c_n)\} \textit{ fails.}\end{gathered}$$

   Definition 4.5.5 (Information flow definition from a set of variables to a single variable relatively to the definition of substitution answers)**.**

$$\begin{gathered}\{x_1, \cdots, x_k\} \xrightarrow{SA}{}^{P}_{G(x_1,\cdots,x_k,x_l,x_m,\cdots,x_n)} x_l \textit{ iff} \\ \exists c_m, \cdots, c_n \in U_{L(P)}, \exists a_1, \cdots, a_k, \exists a'_1, \cdots, a'_k \in U_{L(P)} \textit{ such that} \\ (a_1, \cdots, a_k) \neq (a'_1, \cdots, a'_k) \textit{ and} \\ \Theta[P \cup \{G(a_1, \cdots, a_k, x_l, c_m, \cdots, c_n)\}] \\ \neq \\ \Theta[P \cup \{G(a'_1, \cdots, a'_k, x_l, c_m, \cdots, c_n)\}].\end{gathered}$$

   Definition 4.5.6 (Information flow definition from a set of variables to a single variable relatively to the definition of bisimulation)**.**

$$\{x_1, \cdots, x_k\} \xrightarrow{BI}{}^{P}_{G(x_1,\cdots,x_k,x_l,x_m,\cdots,x_n)} x_l \textit{ iff}$$

$$\exists c_m, \cdots, c_n \in U_{L(P)}, \exists a_1, \cdots, a_k, \exists a'_1, \cdots, a'_k \in U_{L(P)} \text{ such that}$$
$$(a_1, \cdots, a_k) \neq (a'_1, \cdots, a'_k) \text{ and}$$
$$\text{not } \{P \cup \{G(a_1, \cdots, a_k, x_l, c_m, \cdots, c_n)\}$$
$$\mathcal{Z}_{max}$$
$$P \cup \{G(a'_1, \cdots, a'_k, x_l, c_m, \cdots, c_n)\}\}.$$

2. **Generalization of the previous definition of information flow from a set of variables to a set of variables**
For a program P and a goal $G(\overline{x_1, \cdots, x_k}, \underline{x_l, \cdots, x_{m-1}}, x_m, \cdots, x_n)$, we say that there is a flow from $\{x_1, \cdots, x_k\}$ to $\{x_l, \cdots, x_{m-1}\}$ iff there is a flow from $\{x_1, \cdots, x_k\}$ to every variable in $\{x_l, \cdots, x_{m-1}\}$.

Definition 4.5.7 (Information flow definition from a set of variables to a set of variables relatively to the definition of success / failure)**.**

$$\{x_1, \cdots, x_k\} \xrightarrow{SF}{}^{P}_{G(x_1, \cdots, x_k, x_l, \cdots, x_{m-1}, x_m, \cdots, x_n)} \{x_l, \cdots, x_{m-1}\} \text{ iff}$$
$$\forall j = l, \cdots, m-1, \{x_1, \cdots, x_k\} \xrightarrow{SF}{}^{P}_{G(x_1, \cdots, x_k, x_l, \cdots, x_{m-1}, x_m, \cdots, x_n)} x_j$$

Definition 4.5.8 (Information flow definition from a set of variables to a set of variables relatively to the definition of substitution answers)**.**

$$\{x_1, \cdots, x_k\} \xrightarrow{SA}{}^{P}_{G(x_1, \cdots, x_k, x_l, \cdots, x_{m-1}, x_m, \cdots, x_n)} \{x_l, \cdots, x_{m-1}\} \text{ iff}$$
$$\forall j = l, \cdots, m-1, \{x_1, \cdots, x_k\} \xrightarrow{SA}{}^{P}_{G(x_1, \cdots, x_k, x_l, \cdots, x_{m-1}, x_m, \cdots, x_n)} x_j$$

Definition 4.5.9 (Information flow definition from a set of variables to a set of variables relatively to the definition of bisimulation)**.**

$$\{x_1, \cdots, x_k\} \xrightarrow{BI}{}^{P}_{G(x_1, \cdots, x_k, x_l, \cdots, x_{m-1}, x_m, \cdots, x_n)} \{x_l, \cdots, x_{m-1}\} \text{ iff}$$
$$\forall j = l, \cdots, m-1, \{x_1, \cdots, x_k\} \xrightarrow{BI}{}^{P}_{G(x_1, \cdots, x_k, x_l, \cdots, x_{m-1}, x_m, \cdots, x_n)} x_j$$

A similar remark applies here too, by considering $k = 1$, and $l = m - 1$, we find again the same definitions of information flows between single variables.

Example 4.7 Example of an information flow in logic programming over goals with arity > 2

Let P be the following program, representing facts in a virtual game:
$C_1 : p(john, kim, fire, earth, battle_1) \leftarrow;$
$C_2 : p(john, kim, fire, water, battle_2) \leftarrow;$
$C_3 : p(kim, romeo, earth, fire, battle_1) \leftarrow;$
$C_4 : p(kim, romeo, water, fire, battle_2) \leftarrow$

The fact C_1 in the previous program can be read as follows: kim will give $john$ the power of $earth$ only if $john$ holds the power of $fire$ in $battle_1$.

We have: $x \xrightarrow{SA}{}^{P}_{G(x,y,fire,earth,battle_1)} y$ since
$\Theta(P \cup \{G(john, y, fire, earth, battle_1)\}) = \{y \mapsto kim\}$ and
$\Theta(P \cup \{G(romeo, y, fire, earth, battle_1)\}) = \{\epsilon\}$.

Moreover, we have, $\{x, y\} \xrightarrow{SA}{}^{P}_{G(x,y,r,s,battle_1)} \{r, s\}$ because:
$\{x, y\} \xrightarrow{SA}{}^{P}_{G(x,y,r,earth,battle_1)} r$ $(\Theta(P \cup \{G(john, kim, r, earth, battle_1)\}) = \{r \mapsto fire\}$ and $\Theta(P \cup \{G(kim, romeo, r, earth, battle_1)\}) = \{\epsilon\})$
and
$\{x, y\} \xrightarrow{SA}{}^{P}_{G(x,y,fire,s,battle_1)} s$ $(\Theta(P \cup \{G(john, kim, fire, s, battle_1)\}) = \{s \mapsto earth, s \mapsto water\}$ and $\Theta(P \cup \{G(kim, romeo, fire, s, battle_1)\}) = \{\epsilon\})$

4.6 Non-transitivity of the Flow

Most of the policies of information flow in imperative programming are represented by a lattice structure, which means that if information flows from a variable x to a variable y and from y to z, then there is a flow from x to z. In such contexts, the information flow relation between program variables is transitive. It is interesting to investigate this property on the information flow of logic programs according to our definitions.

Several counter examples prove that the information flow relation according to our definitions is not transitive.

Example 4.8 Non transitivity of the information flow for the first definition of flow based on success and failure

For the following program P_4:
$C_1 : p(a, a, a) \leftarrow$;
$C_2 : p(x, a, b) \leftarrow$

and the goal $G(x, y, z) :\leftarrow p(x, y, z)$, we have:

$x \xrightarrow[G(x,y,z)]{SF\ P_4} y$ ($P_4 \cup \{G(a, y, a)\}$ succeeds, $P_4 \cup \{G(b, y, a)\}$ fails),
$y \xrightarrow[G(x,y,z)]{SF\ P_4} z$ ($P_4 \cup \{G(a, a, z)\}$ succeeds, $P_4 \cup \{G(a, b, z)\}$ fails),
but we have not $x \xrightarrow[G(x,y,z)]{SF\ P_4} z$ (both $P_4 \cup \{G(a, a, z)\}$ and $P_4 \cup \{G(b, a, z)\}$ succeed).

Example 4.9 Non transitivity of the information flow for the first definition of flow based on substitution answers

For the following program P_5:
$C_1 : p(x, y, z) \leftarrow q(x, y), r(y, z)$;
$C_2 : q(x, y) \leftarrow$;
$C_3 : q(a, c) \leftarrow$;
$C_4 : r(y, c) \leftarrow$;
$C_5 : r(c, d) \leftarrow$

and the goal $G(x, y, z) :\leftarrow p(x, y, z)$, we have:

$x \xrightarrow[G(x,y,z)]{SA\ P_5} y$ ($\Theta(P_5 \cup \{G(a, y, c)\}) = \{y \mapsto c, \epsilon\}, \Theta(P_5 \cup \{G(b, y, c)\}) = \{\epsilon\}$),
$y \xrightarrow[G(x,y,z)]{SA\ P_5} z$ ($\Theta(P_5 \cup \{G(a, c, z)\}) = \{z \mapsto c, z \mapsto d\}, \Theta(P_5 \cup \{G(a, b, z)\}) = \{z \mapsto c\}$),
but we have not $x \xrightarrow[G(x,y,z)]{SA\ P_5} z$ ($\Theta(P_5 \cup \{G(a, c, z)\}) = \{z \mapsto c, z \mapsto d\}, \Theta(P_5 \cup \{G(b, c, z)\}) = \{z \mapsto c, z \mapsto d\}$).

Example 4.10 Non transitivity of the information flow for the first definition of flow based on bisimulation between goals

For the same previous program P_4 and the goal $G(x,y,z) :\leftarrow p(x,y,z)$, we have:

$x \xrightarrow{BI}{}^{P_4}_{G(x,y,z)} y$ (not$\{tree(P_4 \cup \{G(a,y,a)\}) \mathcal{Z}_{max} tree(P_4 \cup \{G(b,y,a)\})\}$),
$y \xrightarrow{BI}{}^{P_4}_{G(x,y,z)} z$ (not$\{tree(P_4 \cup \{G(a,a,z)\}) \mathcal{Z}_{max} tree(P_4 \cup \{G(a,b,z)\})\}$),
but we have not $x \xrightarrow{BI}{}^{P_4}_{G(x,y,z)} z$ $(tree(P_4 \cup \{G(a,a,z)\}) \mathcal{Z}_{max} tree(P_4 \cup \{G(b,a,z)\}))$.

This non-transitivity of our information flow relation can be explained by the particular role of variables in logic programming. The truth is that in imperative programs, the basic instruction is the assignment operation, whereas in logic programs, the basic instructions are the resolution rule and the unification.

4.7 Complexity Results

4.7.1 Complexity Classes

In this section, we give a brief overview of complexity concepts that will be used throughout the thesis. We refer the reader to [43, 56] for a thorough introduction in the field of complexity.

In this work, we deal most of the times with **decision problems**, i.e. problems that admit a boolean answer. Informally, we think of a Turing machine as a device able to read from and write on a semi-infinite tape, whose contents may be locally accessed and changed in a computation. Its behavior at a given moment is determined partially by the current state of the machine. For Deterministic Turing Machine (DTM), when any of the states *halt*, *yes* or *no* is reached, the tape (T) halts. We say that T **accepts** its input if T halts in *yes*. Similarly, we say that T rejects the input in the case of halting in *no*. Unlike the case of DTM, the definition of acceptance and rejection by a NonDeterministic Turing Machine (NDTM) is asymmetric. We say that a NDTM **accepts** an input if there is at least one sequence if choices leading to the state *yes*. A NDTM **rejects** an input if no sequence of choices can lead to *yes*.

Thus, for decision problems, the class P is the set of problems that can be answered by a DTM in polynomial time. The class of decision problems that can be solved by a NDTM in polynomial time is denoted by NP. $co-NP$ is the class of problems whose answer is always the complement of those in NP (Generally, for a language L, $co-L$ is the class of problems whose answer is always the complement of those in L). The class P is obviously contained both in NP and in $co-NP$. As for the class of decision problems that can be solved by a DTM in exponential time, it is denoted by $EXPTIME$. Its corresponding class for decision problems solved by a NDTM is denoted by $NEXPTIME$.

In the following, we refer to a particular type of computation called computation with *oracles*. Oracles are intuitively subroutines without cost. Given a class of decision problem C, the class P^C (NP^C) is the class of decision problems that can be solved in polynomial time by a DTM (NDTM) that uses an oracle for the problems in C, i.e. a subroutine for any problem in C that can be called several times, spending just one time-unit for each call.

The definition of *polynomial hierarchy* is based on oracle computations. The classes Σ_k^p, Π_k^p and Δ_k^p of the polynomial hierarchy are defined by:

$$\Sigma_0^p = \Pi_0^p = \Delta_0^p = P$$

and for all $k \geq 0$,

$$\Sigma_{k+1}^p = NP^{\Sigma_k^p}, \Pi_{k+1}^p = co-\Sigma_{k+1}^p, \Delta_{k+1}^p = P^{\Sigma_k^p}.$$

Notice that $\Sigma_1^p = NP, \Pi_1^p = co-NP, \Delta_1^p = P$.

An Alternating Turing Machine (ATM) is a NDTM that allows two modes of configurations:

1. A first configuration that leads to acceptance iff it is either a final accepting configuration, or (recursively) at least one of its successors leads to acceptance (called OR configuration).

2. A second configuration that leads to acceptance iff it is either a final accepting configuration, or (recursively) all of its successors leads to acceptance (called AND configuration).

The machine accepts its input iff its initial configuration with this input does. The class of decision problems that can be solved by a ATM in polynomial time is denoted by $ATIME$, and the class of decision problems that can be solved by a ATM in polynomial space is denoted by $APSPACE$. Notice that $EXPTIME = APSPACE$ [16].

Let L_1 and L_2 be two languages, and assume that there is a DTM R that halts in polynomial-time such that: for all input strings x, we have $x \in L_1$ iff $R(x) \in L_2$ (where $R(x)$ denotes the output of R on input x). Then R is called a **polynomial reduction** from L_1 to L_2 and we say that L_1 is **reducible** to L_2. Besides this notion of reduction, complexity theory also considers many other kinds of reductions, for example logarithmic-space reductions or polynomial time Turing reductions.

Let C be a set of languages. A language L is called $C - hard$, if any language $L' \in C$ is reducible to L. If L is $C-hard$ and $L \in C$, then L is called $C-complete$. If two decision problems A and B are complete for the same class, then there is always a way to solve any instance of A by solving a single instance of B and vice versa.

We now study the computational complexity of the following decision problems:

$$\pi_{SF}\begin{cases}\text{Input: A logic program } P\text{, a two variables goal } G(x,y)\\ \text{Output: Determine whether } x \xrightarrow{SF}{}^{P}_{G}\ y\end{cases}$$

$$\pi_{SA}\begin{cases}\text{Input: A logic program } P\text{, a two variables goal } G(x,y)\\ \text{Output: Determine whether } x \xrightarrow{SA}{}^{P}_{G}\ y\end{cases}$$

$$\pi_{BI}\begin{cases}\text{Input: A logic program } P\text{, a two variables goal } G(x,y)\\ \text{Output: Determine whether } x \xrightarrow{BI}{}^{P}_{G}\ y\end{cases}$$

4.7.2 Undecidability

In the general setting, our decision problems are undecidable.

Proposition 4.7.1. *The three decision problems above are undecidable.*

Proof. (π_{SF}) We will reduce the following undecidable decision problem π_1 [28] to π_{SF}:

$$\pi_1\begin{cases}\text{Input: A logic program } P\text{, a ground goal } q(a)\\ \text{Output: } P\cup\{\leftarrow q(a)\} \text{ succeeds}\end{cases}$$

Let $(P, q(a))$ be an instance of π_1 and let $(P', G(x,y))$ be the instance of π_{SF} defined by: $P' = P \cup \{G(a,y) \leftarrow q(a)\}$, where G is a new predicate symbol of arity 2 and a is a new constant. We need to show that, $P\cup\{\leftarrow q(a)\}$ succeeds iff $x \xrightarrow{SF}{}^{P'}_{G}\ y$.
($\Rightarrow$) Suppose that $P\cup\{\leftarrow q(a)\}$ succeeds. Thus $P'\cup\{G(a,y)\}$ succeeds and $P'\cup\{G(b,y)\}$ fails, consequently $x \xrightarrow{SF}{}^{P'}_{G}\ y$.
($\Leftarrow$) Suppose that $x \xrightarrow{SF}{}^{P'}_{G}\ y$, then there exists $a', b' \in U_{L(P)}$ such that $P'\cup\{G(a',y)\}$ succeeds and $P'\cup\{G(b',y)\}$ fails. Thus, $a' = a$ and $b' \neq a$. Thus, $P\cup\{\leftarrow q(a)\}$ succeeds.

(π_{SA}) A similar proof applies here.

(π_{BI}) We will reduce the following undecidable decision problem [29] to π_{BI}:

$$\pi_2\begin{cases}\text{Input: A binary logic program } P\text{, a ground goal } q(a)\\ \text{Output: The SLD-tree of } P\cup\{\leftarrow q(a)\} \text{ contains a failure branch}\end{cases}$$

Let $(P, q(a))$ be an instance of π_2 and let $(P', G(x,y))$ be the instance of π_{BI} defined by:

$$P' = P \cup \begin{cases} G(a,y) \leftarrow q(a)\\ G(b,y) \leftarrow G(b,y) \text{ for all } b \text{ in } L(P) \text{ such that } a \neq b\\ G(f(x_1,\cdots,x_n),y) \leftarrow G(f(x_1,\cdots,x_n),y) \text{ for all } f \text{ in } L(P)\end{cases}$$

Remark that for all $a' \in U_{L(P)}$, the computation tree of $P' \cup \{G(a', y)\}$ consists of a unique infinite branch. We need to show that the SLD-tree of $P \cup \{\leftarrow q(a)\}$ contains a failure branch iff $x \xrightarrow{BI}{}^{P'}_{G} y$.
($\Rightarrow$) Suppose that the SLD-tree of $P \cup \{\leftarrow q(a)\}$ contains a failure branch. Thus the SLD-tree of $P' \cup \{G(a, y)\}$ will eventually contains this failure branch while the SLD-tree of $P' \cup \{G(b, y)\}$ will have infinite branche(s). Consequently $x \xrightarrow{BI}{}^{P'}_{G} y$.
($\Leftarrow$) Suppose that $x \xrightarrow{BI}{}^{P'}_{G} y$, then there exists $a', b' \in U_{L(P)}$ such that *not* $\{P' \cup \{G(a', y)\} \mathcal{Z}_{max} P' \cup \{G(b', y)\}\}$. Hence, either a' or b' is equal to a. Thus (in the case of $a' = a$) the SLD-tree of $P \cup \{\leftarrow q(a)\}$ contains a failure branch. $\square$

4.7.3 Decidability

If one restricts the language to Datalog programs and goals then determining existence of information flows becomes decidable.

Proposition 4.7.2. *π_{SF} is EXPTIME-complete for Datalog programs.*

Proof. (*Membership*) The following algorithm decides the existence of the information flow in Datalog programs.

Require: A Datalog program P, a goal $G(x,y)$, finite Herbrand Universe $U_{L(P)} = \{a_1, \cdots, a_n\}$

Ensure: $x \xrightarrow{SF}{}^{P}_{G(x,y)} y$ for the Datalog program P and the goal g

1: $answer = false$
2: $i = 0$
3: **while** $i < n$ and not $answer$ **do**
4: $\quad i = i + 1$; $j = i$
5: $\quad$ **while** $j < n$ and not $answer$ **do**
6: $\quad\quad j = j + 1$
7: $\quad\quad$ **if** ($P \cup \{G(a_i, y)\}$ succeeds and $P \cup \{G(a_j, y)\}$ fails) or ($P \cup \{G(a_i, y)\}$ fails and $P \cup \{G(a_j, y)\}$ succeeds) **then**
8: $\quad\quad\quad answer = true$
9: $\quad\quad$ **end if**
10: $\quad$ **end while**
11: **end while**
12: return $answer$

This algorithm is deterministic and using the fact that Datalog is program complete for EXPTIME [41, 71], it follows that it can be executed in EXPTIME.

(*Hardness*) In order to prove EXPTIME-hardness, we consider the following decision problem known to be EXPTIME-hard [71]:

$$\pi_3 \begin{cases} \text{Input: A Datalog program } P, \text{ a ground atom } A \\ \text{Output: } P \cup A \ (A \text{ is a logical consequence of } P) \end{cases}$$

Let (P, A) an instance of π_3 and let $(P', g(x,y))$ be the instance of π_{SF} defined by $P' = P \cup \{g(a,y) \leftarrow A\}$, where g is a new predicate symbol. Thus $P \cup A$ iff $x \xrightarrow{SF}{}^{P'}_{g} y$.

($\Rightarrow$) Suppose that A is a logical consequence of P, thus $P' \cup \{\leftarrow g(a,y)\}$ succeeds and $P' \cup \{\leftarrow g(b,y)\}$ fails. Consequently $x \xrightarrow{SF}{}^{P'}_{g} y$.

($\Leftarrow$) Suppose that $x \xrightarrow{SF}{}^{P'}_{g} y$. Then there exists $a, b \in U_{L(P)}$ such that $P' \cup \{\leftarrow g(a',y)\}$ succeeds and $P' \cup \{\leftarrow g(b',y)\}$ fails. Hence, it follows that $a' = a$ and $b' \neq a$. Thus, $P \cup A$. □

Proposition 4.7.3. *π_{SA} is EXPTIME-complete for Datalog programs.*

Proof. A proof similar to the previous one applies here. □

Concerning π_{SF}, determining existence of flows is even in $\Delta_2 P$ if one considers binary hierarchical Datalog programs.

Proposition 4.7.4. *π_{SF} is in $\Delta_2 P$ for binary hierarchical Datalog programs.*

Proof. Let us consider the following deterministic algorithm with oracle:

Algorithm 1: function SF(P,$G(x, y)$)

Require: A binary hierarchical Datalog program P, a goal $G(x, y)$.
Ensure: $x \xrightarrow{SF}{}^{P}_{G} y$.
begin
  For all a in $U_{L(P)}$ do
  For all b in $U_{L(P)}$ do
  if *($P \cup \{G(a, y)\} \in$ SUCCESSES and $P \cup \{G(b, y)\} \in$ FAILURES)*
  then
    Accept
  else
    Reject

The oracle SUCCESSES consists in the set of all pairs (P, G) such that G succeeds in P. Restricting P to binary hierarchical programs, one can show that SUCCESSES belongs to NP. The oracle FAILURES consists in the set of all pairs (P, G) such that G fails in P. Restricting P to binary hierarchical programs, one can show that FAILURES belongs to co-NP. Hence π_{SF} is in $\Delta_2 P$. □

We do not know if π_{SA} is in $\Delta_2 P$ too for binary hierarchical programs.

Now, let us address the complexity of deciding the existence of flows with respect to our third definition.

Proposition 4.7.5. *π_{BI} is in EXPTIME for hierarchical binary Datalog programs.*

Proof. Since $EXPTIME = APSPACE$, then it suffices to demonstrate that π_{BI} is in $APSPACE$ for binary hierarchical Datalog programs. In this respect, we consider the following alternating algorithm:

The subprocedure *succ*(.,.) produces, given an hierarchical Datalog program P and a goal G a Boolean value. More precisely, *succ*(P, G) is true iff there exists

Algorithm 2: function bisim(P,G_1,G_2)

Require: A hierarchical Datalog program P, two goals G_1 and G_2.
Ensure: Deciding whether $P \cup \{G_1\}\ \mathcal{Z}^P_{max}\ P \cup \{G_2\}$.
begin
  case $(succ(P, G_1), succ(P, G_2))$
  begin
    - $(true, true)$**:**
      (∀) choose $i, j \in \{1, 2\}$ such that $i \neq j$
      (∀) choose a successor G'_i of G_i in P
      (∃) choose a successor G'_j of G_j in P
      (.) call **bisim**(P, G'_i, G'_j)
    - $(true, false)$**:** reject
    - $(false, true)$**:** reject
    - $(false, false)$**:**
    if *($G_1 = \Box$ iff $G_2 = \Box$)* **then**
      Accept
    else
      Reject

a goal G' such that G' is derived from G and P. Obviously, $succ(.,.)$ can be implemented in deterministic linear time. Concerning the procedure **bisim**, seeing that P is hierarchical, it accepts its inputs P, G_1, G_2 iff $G_1 \mathcal{Z}_{max} G_2$. Moreover, seeing that P is binary, **bisim** can be implemented in polynomial space. □

We mention that the different algorithms previously presented for goals of arity two can be generalized easily to goals with arity higher than two.

4.8 Information Flow Existence after Program Transformation

In this section, we focus our attention on the existence of the information flow after program transformation. We will be interested to check whether transformations on logic programs introduce or eliminate information flows.

Unfold/fold transformations were first proposed by Burstall and Darlington [14] in the context of a functional language. In the context of logic programming, Tamaki and Sato [69] formulated unfold/fold transformations for definite logic programs so as to preserve the equivalence of programs in the sense of the least Herbrand model semantics.

These transformations are based on a very simple idea: that of replacing equals by equals. Informally, unfolding refers to the replacement of an atom in the body of a clause by the appropriate body of some other clause. For example, given the program:
$C_1 : p(x, y) \leftarrow q(x), r(y);$
$C_2 : q(u) \leftarrow s(u, v);$
we can unfold the literal q in the C_1, to obtain:
$C'_1 : p(x, y) \leftarrow s(x, w), r(y);$
$C_2 : q(u) \leftarrow s(u, v);$

However, folding refers to the replacement of an/some atom(s) in the body of a clause by the head of some other clause. For example, given the program:
$C_1 : p(x) \leftarrow q(x), s(x, z), r(z);$
$C_2 : t(u, v) \leftarrow s(u, v), r(v);$
we can fold the literals for s and r in C_1, to yield:
$C'_1 : p(x) \leftarrow q(x), t(x, z).$

Formally,

Definition 4.8.1 (Unfolding)**.** *Let C be a clause in P of the form*
$C : A \leftarrow A_1, \cdots, A_{i-1}, A_i, A_{i+1}, \cdots, A_n.$
*and $C_1, C_2, \cdots, C_m$ be all clauses in P, whose heads are unifiable with A_i by most general unifiers $\theta_1, \theta_2, \cdots, \theta_m$. The result of **unfolding** C **at** A_i is the set of clauses $\{C'_1, \cdots, C'_m\}$ such that, for each j $(1 \leq j \leq m)$ if*
$C_j : B_j \leftarrow B_{j_1}, \cdots, B_{j_h}$ *$(h \geq 0)$ and $B_j[\theta_j] = A_i[\theta_j]$, then*
$C'j : (A \leftarrow A_1, \cdots, A_{i-1}, B_{j_1}, \cdots, B_{j_h}, A_{i+1}, \cdots, A_n)[\theta_j].$
Then, $P' = (P - \{C\}) \cup \{C'_1, \cdots, C'_m\}$. C is called the unfolded clause and $C_1, \cdots, C_m$ are called the unfolding clauses. A_i is called the unfolded atom.

Definition 4.8.2 (Folding)**.** *Let C be a clause in P of the form*
$C : A_0 \leftarrow A_1, \cdots, A_n.$ *$(n > 0)$*

and D be a clause in P' (i.e. the resulting program after the folding transformation). Let D be of the form:
$D : B_0 \leftarrow B_1, \cdots, B_k$. *($k > 0$)*
Suppose that there exists a substitution θ satisfying the following conditions:

1. *$B_1[\theta] = A_{j_1}, B_2[\theta] = A_{j_2}, \cdots, B_k[\theta] = A_{j_k}$ where $j_1, j_2, \cdots, j_k$ are all different natural numbers between 1 and n.*

2. *For each variable in the body in D, θ substitutes a distinct variable not appearing in $\{B_0[\theta], A_0, A_1, \cdots, A_n\} - \{A_{j_1}, \cdots, A_{j_k}\}$.*

Then, the result of ***folding*** *C* ***using*** *D is the clause C' with head A_0 and body $\{B_0[\theta]\} \cup \{A_1, \cdots, A_n\} - \{A_{j_1}, \cdots, A_{j_k}\}$. C is called a folded clause, D is called a folding clause and $B_0[\theta]$ the atom introduced by folding.*

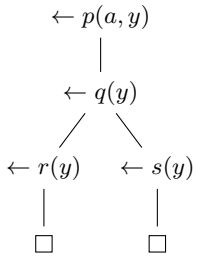
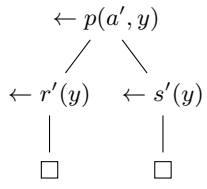
Obviously, since folding or unfolding clauses in logic programs change neither its successes, nor its failures [69], nor its substitution answers [44], the information flows based either on successes and failures or on substitution answers are preserved after applying the transformations of Tamaki and Sato. The same cannot be said for information flows based on bisimulation.

Example 4.11 Example of a program transformation that do not preserve information flows based on bisimulation.

let P_0 be the logic program containing the following clauses:
$C_1 : p(a, y) \leftarrow q(y)$;
$C_2 : q(y) \leftarrow r(y)$;
$C_3 : q(y) \leftarrow s(y)$;
$C_4 : r(y) \leftarrow$;
$C_5 : s(y) \leftarrow$;
$C_6 : p(a', y) \leftarrow r'(y)$;
$C_7 : p(a', y) \leftarrow s'(y)$;
$C_8 : r'(y) \leftarrow$;
$C_9 : s'(y) \leftarrow$

and let G be the goal $\leftarrow p(x, y)$.

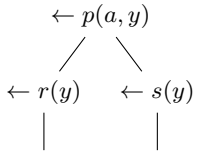
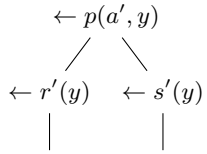
It is easy to verify that $x \xrightarrow{BI}{}^{P_0}_{G} y$. To see this, we sketch, by omitting the different substitutions, the SLD-refutation trees corresponding to the two goals $P_0 \cup \{\leftarrow p(a, y)\}$ and $P_0 \cup \{\leftarrow p(a', y)\}$.

SLD-tree$(P_0 \cup \{\leftarrow p(a,y))\}$	**SLD-tree**$(P_0 \cup \{\leftarrow p(a',y)\})$
	

Obviously, as $not\{P_0 \cup \{\leftarrow p(a,y)\}\mathcal{Z}_{max}P_0 \cup \{\leftarrow p(a',y)\}\}$, $x \xrightarrow{BI}{}^{P_0}_{G}\ y$.

By unfolding C_1, the program P_1 is obtained from P_0 by replacing C_1 with the following clauses:
$C_{10} : p(a,y) \leftarrow r(y)$;
$C_{11} : p(a,y) \leftarrow s(y)$

In the new transformed program P_1, the two SLD-refutation trees of the goals $P_1 \cup \{\leftarrow p(a,y)\}$ and $P_1 \cup \{\leftarrow p(a',y)\}$ are bisimilar as shown in the next figure.

SLD-tree$(P_1 \cup \{\leftarrow p(a,y)\})$	**SLD-tree**$(P_1 \cup \{\leftarrow p(a',y)\})$
	

Thus $x \not\xrightarrow{BI}{}^{P_1}_{G}\ y$.

A general question concerns the definition of transformations of logic programs that never introduce or eliminate information flows.

4.9 Summary

In this chapter, we have proposed three definitions of information flow in logic programs. As proved in section 4.7.2, determining whether there exists an information flow is undecidable in the general setting. Hence, a natural question was to restrict the language of logic programming as done in section 4.7.3. Table 4.1 contains the results we have obtained so far. Much remains to be done.

	General setting	Datalog programs	Binary hierarchical Datalog programs
π_{SF}	Undecidable	EXPTIME-complete	in $\Delta_2 P$
π_{SA}	Undecidable	EXPTIME-complete	in EXPTIME
π_{BI}	Undecidable	?	in EXPTIME

Table 4.1: Complexity results

Firstly, in the setting of Datalog programs, the main difficulty concerning π_{BI} comes from loops or infinite branches in SLD-refutation trees. Therefore, in order to determine, given a Datalog program P and two Datalog goals G_1 and G_2, whether $G_1 \mathcal{Z}_{max} G_2$, one can think about using loop checking techniques and considering either restricted programs, or *nvi* programs or *svo* programs.

Secondly, considering the unfold/fold transformations introduced by Tamaki and Sato [69] within the context of logic programs optimization, we showed that these transformations can introduce or eliminate information flows, and that a general question resides concerning the definition of transformations of logic programs that never introduce or eliminate flows.

Chapter 5

Goals Bisimulation

Within the context of a given programming language, it is essential to associate a semantics to programs written in it. This semantics induces an equivalence relation between programs. Hence, the decision problem "given two programs, determine whether they are equivalent" is at the heart of the study of program semantics.

In logic programming, several ideas of equivalence of logic programs are in competition. These ideas are based on the various semantics of logic programs. For example, if one considers the least Herbrand model semantics, then two given logic programs are said to be equivalent iff their least Herbrand models are equal. Such equivalence relations between programs have been intensively studied in the past [38, 39, 55, 57]. Nevertheless, in some cases, they fail to adequately provide the right tool for comparing logic programs. For instance, in the case of perpetual processes, most of them are inadequate to offer frameworks in which to reason about programs. The truth is that two perpetual processes may have the same declarative semantics (defined in terms of least model) for example and, still have very different behaviors. And the various equivalence relations considered in [31, 37, 48, 52] do not take into account the computational aspects of logic programming. The goal of this chapter is to suggest the use of equivalence relations between logic programs that take into account the shape of the SLD-trees that these programs give rise to. This idea is not new: in automata theory, for instance, many variants of the equivalence relation of bisimilarity have been defined in order to promote the idea that automata with the same trace-based semantics should sometimes not be considered as equivalent if they are not bisimilar [61].

In this chapter, in order to simplify matter, we consider logic programs of a

very simple kind: Datalog programs. Furthermore, comparing two given Datalog programs and taking into account the shape of the SLD-trees they give rise to necessitates the comparison of infinitely many SLD-trees. Thus, in a first approach, we restrict our study to the comparison of two given Datalog goals. We will say that, with respect to a fixed Datalog program P, two given goals are equivalent when their SLD-trees are bisimilar.

In this chapter, we investigate the computability and complexity of the equivalence problem between Datalog goals. In particular, we examine the complexity of the following decision problems:

- given two Datalog goals F, G and a hierarchical Datalog program P, determine if the SLD-trees of $P \cup F$ and $P \cup G$ are bisimilar.
- given two Datalog goals F, G and a restricted Datalog program P, determine if the SLD-trees of $P \cup F$ and $P \cup G$ are bisimilar.

An undecidability result concerning Prolog programs will be given on section 5.1. In section 5.2, we will address the problem of deciding whether two given goals are bisimilar with respect to a given hierarchical program. At the end of the section, computational issues will be studied. Note that for hierarchical programs, the SLD-tree for a goal $\leftarrow G$ contains only finite branches. In section 5.3, we will address the same questions as in section 5.2 by considering here restricted programs. These programs allow a specific kind of recursion in the clauses. Finally, we will conclude by proposing an open question concerning to decide whether two given goals are bisimilar with respect to a *nonvariable introducing* logic program or a *single variable occurrence* logic program.

5.1 Undecidability for Prolog Programs

As is well known, the SLD-resolution principle and the concepts of resolvent, SLD-derivation, SLD-refutation and SLD-tree have also been considered within the context of Prolog programs and Prolog goals. In addition to the predicate symbols, constants and variables composing the alphabet of Datalog, the alphabet of Prolog also includes function symbols allowing the use of terms inductively defined as follows:
(i) constants are terms;
(ii) variables are terms;
(iii) expressions of the form $f(t_1, \cdots, t_h)$, where $t_1, \cdots, t_h$ are terms and f is a function symbol of arity h, are terms.
As a result, one may also define in Prolog the binary relations between goals similar to the bisimulation defined in Datalog. Nevertheless,

Proposition 5.1.1. *It is undecidable, given a Prolog program P and Prolog goals F_1, G_1, to determine whether $P \cup \{F_1\}$ $\mathcal{Z}^P_{max}$ $P \cup \{G_1\}$.*

Proof. Let us consider the following decision problem:
(π_1) given a Prolog program P with exactly one binary clause (i.e. a clause such that its body contains at most one atom) and a Prolog goal $\leftarrow B$, determine whether the SLD-tree for $P \cup \{\leftarrow B\}$ contains an infinite branch.

Let $(P, \leftarrow B)$ be an instance of π_1. We consider new constants a, b, c and a new predicate symbol p of arity 2. Let P' be the least Prolog program containing P and the following Horn clauses:

$$p(a, y) \leftarrow B,$$
$$p(b, y) \leftarrow B,$$
$$p(b, y) \leftarrow p(c, y),$$
$$p(c, y) \leftarrow p(c, y).$$

We demonstrate that the following conditions are equivalent:
(i) the SLD-tree for $P \cup \{\leftarrow B\}$ contains an infinite branch;
(ii) $P' \cup \{\leftarrow p(a, y)\}$ $\mathcal{Z}^{P'}_{max}$ $P' \cup \{\leftarrow p(b, y)\}$.

$(i) \Rightarrow (ii)$ Suppose the SLD-tree for $P \cup \{\leftarrow B\}$ contains an infinite branch. Since P consists of exactly one binary clause, then the SLD-tree for $P \cup \{\leftarrow B\}$ is equal to a unique infinite branch. Obviously, the SLD-tree for $P' \cup \{\leftarrow p(a, y)\}$ is also equal to a unique infinite branch whereas the SLD-tree for $P' \cup \{\leftarrow p(b, y)\}$ is equal to a pair of infinite branches. Hence, $P' \cup \{\leftarrow p(a, y)\}$ $\mathcal{Z}^{P'}_{max}$ $P' \cup \{\leftarrow p(b, y)\}$.

$(ii) \Rightarrow (i)$ Suppose $P' \cup \{\leftarrow p(a, y)\}$ $\mathcal{Z}^{P'}_{max}$ $P' \cup \{\leftarrow p(b, y)\}$. Obviously, the SLD-tree for $P' \cup \{\leftarrow p(b, y)\}$ contains an infinite branch. Hence, the SLD-tree for $P' \cup \{\leftarrow p(a, y)\}$ also contains an infinite branch. Hence, the SLD-tree for $P \cup \{\leftarrow B\}$ contains an infinite branch.

Since (π_1) is undecidable [28], then it is undecidable, given a Prolog program P and Prolog goals F_1, G_1, to determine whether $P \cup \{F_1\} \; \mathcal{Z}^P_{max} \; P \cup \{G_1\}$.□ □

A question arises here, how can we restore this decision problem to decidability? One can think (as we have done) about considering specific classes of logic programs, by restricting the language of logic programming. In the setting of Datalog programs for example, the main difficulty concerning (π) comes from loops or infinite branches in SLD trees. We will address this issue in the following sections.

5.2 Decidability for Hierarchical Programs

We now study the computational complexity of the following decision problem: (π_{hie}) given an hierarchical Datalog program P and Datalog goals F_1, G_1, determine whether $P \cup \{F_1\} \; \mathcal{Z}^P_{max} \; P \cup \{G_1\}$. In this respect, let P be a hierarchical Datalog program.

Algorithm 3: function bisim1(F_1,G_1)

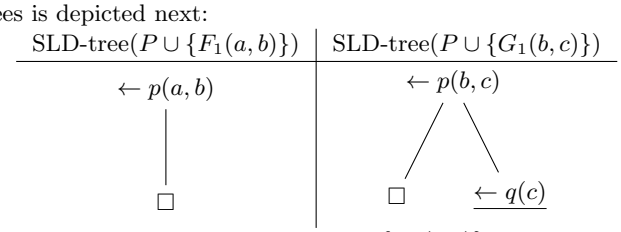
```
begin
    if bothempty(F1, G1) or bothfail(F1, G1) then
        return true
    else
        SF ⟵ successor(F1)
        SG ⟵ successor(G1)
        if SF ≠ ∅ and SG ≠ ∅ then
            SF′ ⟵ SF
            while SF′ ≠ ∅ do
                F2 ⟵ get-element(SF′)
                found-bisim ⟵ false
                SG′ ⟵ SG
                while SG′ ≠ ∅ and found-bisim = false do
                    G2 ⟵ get-element(SG′)
                    found-bisim ⟵ bisim1(F2, G2)
                if found-bisim = false then
                    return false
            SG′ ⟵ SG
            while SG′ ≠ ∅ do
                G2 ⟵ get-element(SG′)
                found-bisim ⟵ false
                SF′ ⟵ SF
                while SF′ ≠ ∅ and found-bisim = false do
                    F2 ⟵ get-element(SF′)
                    found-bisim ⟵ bisim1(G2, F2)
                if found-bisim = false then
                    return false
            return true
        else
            return false
```

In `Algorithm` 3, `bothempty(`F_1`,`G_1`)` is a Boolean function returning `true` iff $F_1 = \square$ and $G_1 = \square$, whereas `bothfail(`F_1`,`G_1`)` is a Boolean function returning `true` iff $F_1 \neq \square$, `successor(`F_1`)=`$\emptyset$, $G_1 \neq \square$ and `successor(`G_1`)=`$\emptyset$. Moreover, `successor(.)` is a function returning the set of all resolvents of its argument

with a clause of P whereas `get-element(.)` is a function removing one element from the set of elements given as input and returning it.

Example 5.1 Running the algorithm bisim1 on an hierarchical logic program

Let P be the following hierarchical logic program:
$C_1 : q(a) \leftarrow$;
$C_2 : p(a,b) \leftarrow$;
$C_3 : p(b,c) \leftarrow$;
$C_4 : p(b,y) \leftarrow q(y)$;
and $F_1(a,b) = \leftarrow p(a,b)$, $G_1(b,c) = \leftarrow p(b,c)$ two goals. Its corresponding SLD-trees is depicted next:



Note that in the SLD-tree of $P \cup \{G_1(b,c)\}$, the goal $\leftarrow q(c)$ is a failure goal.

The following is the trace execution of the `Algorithm` 3 over the goals $F_1(a,b)$ and $G_1(b,c)$.

`bisim1(`$F_1(a,b)$`,`$G_1(b,c)$`)`
- `bothempty(`$F_1(a,b)$`,`$G_1(b,c)$`)` is false
- `bothfail(`$F_1(a,b)$`,`$G_1(b,c)$`)` is false
- $SF \longleftarrow \{\square\}$
- $SG \longleftarrow \{\square, q(c)\}$
- $SF \neq \emptyset$ and $SG \neq \emptyset$
- $SF' \longleftarrow \{\square\}$
- $SF' \neq \emptyset$
- $F_2 \longleftarrow \square$, $SF' \longleftarrow \emptyset$
- $found\text{-}bisim \longleftarrow false$
- $SG' \longleftarrow \{\square, q(c)\}$
- $SG' \neq \emptyset$ and $found\text{-}bisim = false$
- $G_2 \longleftarrow \square$, $SG' \longleftarrow \{q(c)\}$
- $found\text{-}bisim \longleftarrow$ `bisim1(`$\square$`,`$\square$`)`
  - `bothempty(`$\square$`,`$\square$`)` is true
  - return $true$
- $SG' \neq \emptyset$ and $found\text{-}bisim = true$ (quit the while loop)
- $found\text{-}bisim = true$ (loop on the elements of SF')
- $SF' = \emptyset$ (continue in the second part of the algorithm)
- $SG' \longleftarrow \{\square, q(c)\}$
- $SG' \neq \emptyset$
- $G_2 \longleftarrow \square$, $SG' \longleftarrow \{q(c)\}$
- $found\text{-}bisim \longleftarrow false$
- $SF' \longleftarrow \{\square\}$
- $SF' \neq \emptyset$ and $found\text{-}bisim = false$
- $F_2 \longleftarrow \square$, $SF' \longleftarrow \emptyset$
- $found\text{-}bisim \longleftarrow$ `bisim1(`$\square$`,`$\square$`)`
  - `bothempty(`$\square$`,`$\square$`)` is true

return $true$
$SF' = \emptyset$ (continue and check the if statement)
$found\text{-}bisim = true$ (loop on the elements of SG')
$SG' \neq \emptyset$
$G_2 \longleftarrow q(c)$, $SG' \longleftarrow \emptyset$
$found\text{-}bisim \longleftarrow false$
$SF' \longleftarrow \{\Box\}$
$SF' \neq \emptyset$ and $found\text{-}bisim = false$
$F_2 \longleftarrow \Box$, $SF' \longleftarrow \emptyset$
$found\text{-}bisim \longleftarrow$ `bisim1(`$\Box$`,`$q(c)$`)`
`bothempty(`$\Box$`,`$q(c)$`)` is false
`bothfail(`$\Box$`,`$q(c)$`)` is false
$SF \longleftarrow \emptyset$, $SG \longleftarrow \emptyset$
return $false$
$SF' = \emptyset$ and $found\text{-}bisim = false$ (continue and check the if statement)
$found\text{-}bisim = false$
return $false$

Thus the algorithm `bisim1(`$F_1(a,b)$`,`$G_1(b,c)$`)` returns $false$.

In order to demonstrate the decidability of (π_{hie}), we need to prove the following lemmas for all Datalog goals F_1, G_1:

Lemma 5.2.1 (Termination)**.** `bisim1(`F_1`,`G_1`)` *terminates.*

Lemma 5.2.2 (Completeness)**.** *If* $P \cup \{F_1\}$ $\mathcal{Z}^P_{max}$ $P \cup \{G_1\}$*, then* `bisim1(`F_1`,`G_1`)` *returns* `true`*.*

Lemma 5.2.3 (Soundness)**.** *If* `bisim1(`F_1`,`G_1`)` *returns* `true`*, then* $P \cup \{F_1\}$ $\mathcal{Z}^P_{max}$ $P \cup \{G_1\}$*.*

Let $\ll$ be the binary relation on the set of all pairs of Datalog goals defined by: $(F_2, G_2) \ll (F_1, G_1)$ iff

- the SLD-tree for F_1 is deeper than the SLD-tree for F_2,
- the SLD-tree for G_1 is deeper than the SLD-tree for G_2.

The depth of an SLD-tree is the depth of its longest branch. Remark that in this section, all SLD-trees are finite.
Obviously, $\ll$ is a partial order on the set of all pairs of goals. Since P is hierarchical, then $\ll$ is well-founded.

Proof of Lemma 5.2.1. The proof is done by $\ll$-induction on (F_1, G_1). Let (F_1, G_1) be such that for all (F_2, G_2), if $(F_2, G_2) \ll (F_1, G_1)$ then `bisim1(`F_2`,`G_2`)` terminates. Since every recursive call to `bisim1` that is performed along the execution of `bisim1(`F_1`,`G_1`)` is done with respect to a pair (F_2, G_2) of goals such that $(F_2, G_2) \ll (F_1, G_1)$, then `bisim1(`$F_1$`,`$G_1$`)` terminates. □

Proof of Lemma 5.2.2. Let us consider the following property:

$$(Prop_1(F_1, G_1)) \text{ if } F_1 \mathcal{Z}^P_{max} G_1 \text{ then } \texttt{bisim1}(F_1, G_1) \text{ returns } \texttt{true}.$$

Again, we proceed by $\ll$-induction. Suppose (F_1, G_1) is such that for all (F_2, G_2), if $(F_2, G_2) \ll (F_1, G_1)$ then $Prop_1(F_2, G_2)$. Let us show that $Prop_1(F_1, G_1)$. Suppose $P \cup \{F_1\}\ \mathcal{Z}^P_{max}\ P \cup \{G_1\}$. Hence, for all successors F_2 of F_1, there exists a successor G_2 of G_1 such that $P \cup \{F_2\}\ \mathcal{Z}^P_{max}\ P \cup \{G_2\}$, and conversely. Seeing that the logic program is hierarchical, then $(F_2, G_2) \ll (F_1, G_1)$. By induction hypothesis, $Prop_1(F_2, G_2)$. Since $P \cup \{F_2\}\ \mathcal{Z}^P_{max}\ P \cup \{G_2\}$, then $\texttt{bisim1}(F_2, G_2)$ returns `true`. As a result, one sees that $\texttt{bisim1}(F_1, G_1)$ returns `true`. □

Proof of Lemma 5.2.3. It suffices to demonstrate that the binary relation $\mathcal{Z}$ defined as follows between Datalog goals is a bisimulation: $F_1\ \mathcal{Z}\ G_1$ iff $\texttt{bisim1}(F_1, G_1)$ returns `true`. Let F_1, G_1 be Datalog goals such that $F_1\ \mathcal{Z}\ G_1$. Hence, $\texttt{bisim1}(F_1, G_1)$ returns `true`. Thus, obviously, $F_1 = \square$ iff $G_1 = \square$, and the first condition characterizing bisimulations holds for $\mathcal{Z}$. Now, suppose that F_2 is a resolvent of F_1 and a clause in P. Since $\texttt{bisim1}(F_1, G_1)$ returns `true`, then there exists a resolvent G_2 of G_1 and a clause in P such that $\texttt{bisim1}(F_2, G_2)$ returns `true`, i.e. $F_2\ \mathcal{Z}\ G_2$. As a result, the second condition characterizing bisimulations holds for $\mathcal{Z}$. The third condition characterizing bisimulations holds for $\mathcal{Z}$ too, as the reader can quickly check. Thus $\mathcal{Z}$ is a bisimulation. □

As a consequence of lemmas 5.2.1 – 5.2.3, we have:

Theorem 5.2.1. `Algorithm` *3 is a sound and complete decision procedure for* (π_{hie}).

It follows that (π_{hie}) is decidable. Moreover,

Theorem 5.2.2. (π_{hie}) *is in 2EXPTIME.*

Proof. Let P be a hierarchical Datalog program and G be a goal. Let n be the maximal number of atoms in the clauses of P or in G, and t be the number of level mapping in P.

In fact, the maximal depth of a branch in an SLD-tree is equal to $n \times$ (the maximal depth of a branch at level $t-1$) which is in turn equal to $n \times (1 + n \times (1 +$ the maximal depth of a branch at level $t - 2$)). Thus, by iterating the same operation until level 1, we conclude that the maximal depth D of a branch in an SLD-tree cannot exceed $\sum_{i=1}^{t} n^i$. Remark that $\sum_{i=1}^{t} n^i \leq t \times n^t$.

Let s be the maximal number of clauses that defines a predicate. Hence, the branching degree of the SLD-tree for $P \cup \{G\}$ is bounded by s. Remark that our algorithm uses twice two nested loops. Each loop run through all the successors of a some goal. In the worst case, the number of successors of a goal will not exceed the branching degree s. In fact, for a maximal depth D, the time complexity

of the algorithm is approximately equal to $2 \times s^2 \times$ (the time complexity of the algorithm for a depth $D-1$) which is in turn equal to $4 \times s^4 \times$ (the time complexity of the algorithm for a depth $D-2$). Thus, by iterating the same operation until depth 1, one can show that for a depth D, the time complexity of our algorithm is about $2^D \times s^{2 \times D} \leq 2^{t \times n^t} \times s^{2 \times t \times n^t}$.

Note that for a hierarchical program P and a goal G, the number of nodes in the corresponding SLD-tree is about $s^{t \times n^t}$.□ □

Concerning the exact complexity of (π_{hie}), we do not know whether (π_{hie}) is 2EXPTIME-hard or (π_{hie}) is in EXPSPACE.

5.3 Decidability for Restricted Programs

We now study the computational complexity of the following decision problem: (π_{res}) given a restricted Datalog program P and Datalog goals F_1, G_1, determine whether $P \cup \{F_1\} \mathcal{Z}^P_{max} P \cup \{G_1\}$. In this respect, let P be a restricted Datalog program. In `Algorithm` 4, `bothempty(`F_i`,`G_i`)` and `bothfail(`F_i`,`G_i`)` are similar to the corresponding functions used in `Algorithm` 3, whereas `occur((`$F_1 \Rightarrow \cdots \Rightarrow F_i$`),(`$G_1 \Rightarrow \cdots \Rightarrow G_i$`))` is a Boolean function returning `true` iff there exists substitutions σ, τ such that $F_l = F_k[\sigma]$, $G_l = G_k[\tau]$ for some $1 \leq k < l \leq i$. As the reader can see, `Algorithm` 4 is very similar to `Algorithm` 3. The main difference lies in the introduction of the `occur` test in the first conditional instruction.

Example 5.2 Running the algorithm bisim2 on a restricted logic program

Let P be the following hierarchical logic program:
$C_1 : q(x) \leftarrow r(x)$;
$C_2 : r(a) \leftarrow s(a)$;
$C_3 : p(x, y) \leftarrow p(y, x)$;
and $F_1(a, b) = \leftarrow p(a, b)$, $G_1(a) = \leftarrow q(a)$, $H_1(b, a) = \leftarrow p(b, a)$ three goals. Its corresponding SLD-trees is depicted next:

SLD-tree($P \cup \{F_1(a,b)\}$)	SLD-tree($P \cup \{G_1(a)\}$)	SLD-tree($P \cup \{H_1(b,a)\}$)
$\leftarrow p(a, b)$	$\leftarrow q(a)$	$\leftarrow p(b, a)$
$\leftarrow p(b, a)$	$\leftarrow r(a)$	$\leftarrow p(a, b)$
$\leftarrow p(a, b)$	$\underline{\leftarrow s(a)}$	$\leftarrow p(b, a)$
$\leftarrow p(b, a)$		$\leftarrow p(a, b)$
$\vdots$		$\vdots$

Note that in the SLD-tree of $P \cup \{G_1(a)\}$, the goal $\leftarrow s(a)$ is a failure goal.

The following is the trace execution of the `Algorithm` 4 over the goals $F_1(a, b)$ and $G_1(a)$.

`bisim2(`$F_1(a, b)$`,`$G_1(a)$`)`
- `bothempty(`$F_1(a, b)$`,`$G_1(a)$`)` is false
- `bothfail(`$F_1(a, b)$`,`$G_1(a)$`)` is false
- `occur((`$F_1(a, b)$`),(`$G_1(a)$`))` is false
- $SF \longleftarrow \{p(b, a)\}$
- $SG \longleftarrow \{r(a)\}$
- $SF \neq \emptyset$ and $SG \neq \emptyset$

Algorithm 4: function bisim2($(F_1 \Rightarrow \cdots \Rightarrow F_i)$,$(G_1 \Rightarrow \cdots \Rightarrow G_i)$)

```
begin
    if bothempty(F_i, G_i) or bothfail(F_i, G_i) or
    occur((F_1 ⇒ ··· ⇒ F_i), (G_1 ⇒ ··· ⇒ G_i)) then
        return true
    else
        SF ⟵ successor(F_i)
        SG ⟵ successor(G_i)
        if SF ≠ ∅ and SG ≠ ∅ then
            SF' ⟵ SF
            while SF' ≠ ∅ do
                F' ⟵ get-element(SF')
                found-bisim ⟵ false
                SG' ⟵ SG
                while SG' ≠ ∅ and found-bisim = false do
                    G' ⟵ get-element(SG')
                    found-bisim ⟵ bisim2((F_1 ⇒ ··· ⇒ F_i ⇒ F'), (G_1 ⇒
                    ··· ⇒ G_i ⇒ G'))
                if found-bisim = false then
                    return false
            SG' ⟵ SG
            while SG' ≠ ∅ do
                G' ⟵ get-element(SG')
                found-bisim ⟵ false
                SF' ⟵ SF
                while SF' ≠ ∅ and found-bisim = false do
                    F' ⟵ get-element(SF')
                    found-bisim ⟵ bisim2((G_1 ⇒ ··· ⇒ G_i ⇒ G'), (F_1 ⇒
                    ··· ⇒ F_i ⇒ F'))
                if found-bisim = false then
                    return false
            return true
        else
            return false
```

$SF' \longleftarrow \{p(b,a)\}$
$SF' \neq \emptyset$
$F' \longleftarrow p(b,a)$, $SF' \longleftarrow \emptyset$
$found\text{-}bisim \longleftarrow false$
$SG' \longleftarrow \{r(a)\}$
$SG' \neq \emptyset$ and $found\text{-}bisim = false$
$G' \longleftarrow r(a)$, $SG' \longleftarrow \emptyset$
$found\text{-}bisim \longleftarrow$ `bisim2(`$((p(a,b) \Rightarrow p(b,a))$`,`$(q(a) \Rightarrow r(a)))$
`bothempty(`$p(b,a)$`,`$r(a)$`)` is false
`bothfail(`$p(b,a)$`,`$r(a)$`)` is false
`occur(`$((p(a,b) \Rightarrow p(b,a))$`,`$(q(a) \Rightarrow r(a)))$`)` is false
$SF \longleftarrow \{p(a,b)\}$
$SG \longleftarrow \{s(a)\}$
$SF \neq \emptyset$ and $SG \neq \emptyset$
$SF' \longleftarrow \{p(a,b)\}$
$SF' \neq \emptyset$
$F' \longleftarrow p(a,b)$, $SF' \longleftarrow \emptyset$
$found\text{-}bisim \longleftarrow false$
$SG' \longleftarrow \{s(a)\}$
$SG' \neq \emptyset$ and $found\text{-}bisim = false$
$G' \longleftarrow s(a)$, $SG' \longleftarrow \emptyset$
$found\text{-}bisim \longleftarrow$ `bisim2(`$((p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b))$`,`
$(q(a) \Rightarrow r(a) \Rightarrow s(a)))$
`bothempty(`$p(a,b)$`,`$s(a)$`)` is false
`bothfail(`$p(a,b)$`,`$s(a)$`)` is false
`occur(`$((p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b))$`,`$(q(a) \Rightarrow r(a) \Rightarrow s(a)))$`)` is false
$SF \longleftarrow \{p(b,a)\}$
$SG \longleftarrow \emptyset$
$SF \neq \emptyset$ and $SG = \emptyset$
return $false$
$SG' = \emptyset$ and $found\text{-}bisim = false$ (quit the while loop)
$found\text{-}bisim = false$
return $false$
$SG' = \emptyset$ and $found\text{-}bisim = false$ (quit the while loop)
$found\text{-}bisim = false$
return $false$
Thus the algorithm `bisim2(`$F_1(a,b)$`,`$G_1(a)$`)` returns $false$.

Let us now consider is the trace execution of the `Algorithm` 4 over the goals $F_1(a,b)$ and $H_1(b,a)$.
`bisim2(`$F_1(a,b)$`,`$H_1(b,a)$`)`
`bothempty(`$F_1(a,b)$`,`$H_1(b,a)$`)` is false
`bothfail(`$F_1(a,b)$`,`$H_1(b,a)$`)` is false
`occur(`$((F_1(a,b))$`,`$(H_1(b,a)))$`)` is false
$SF \longleftarrow \{p(b,a)\}$
$SG \longleftarrow \{p(a,b)\}$
$SF \neq \emptyset$ and $SG \neq \emptyset$
$SF' \longleftarrow \{p(b,a)\}$
$SF' \neq \emptyset$
$F' \longleftarrow p(b,a)$, $SF' \longleftarrow \emptyset$
$found\text{-}bisim \longleftarrow false$

$SG' \longleftarrow \{p(a,b)\}$
$SG' \neq \emptyset$ and $found\text{-}bisim = false$
$G' \longleftarrow p(a,b)$, $SG' \longleftarrow \emptyset$
$found\text{-}bisim \longleftarrow$ `bisim2((`$p(a,b) \Rightarrow p(b,a)$`),(`$p(b,a) \Rightarrow p(a,b)$`))`
  `bothempty(`$p(b,a)$`,`$p(a,b)$`)` is false
  `bothfail(`$p(b,a)$`,`$p(a,b)$`)` is false
  `occur((`$p(a,b) \Rightarrow p(b,a)$`),(`$p(b,a) \Rightarrow p(a,b)$`))` is false
  $SF \longleftarrow \{p(a,b)\}$
  $SG \longleftarrow \{p(b,a)\}$
  $SF \neq \emptyset$ and $SG \neq \emptyset$
  $SF' \longleftarrow \{p(a,b)\}$
  $SF' \neq \emptyset$
  $F' \longleftarrow p(a,b)$, $SF' \longleftarrow \emptyset$
  $found\text{-}bisim \longleftarrow false$
  $SG' \longleftarrow \{p(b,a)\}$
  $SG' \neq \emptyset$ and $found\text{-}bisim = false$
  $G' \longleftarrow p(b,a)$, $SG' \longleftarrow \emptyset$
  $found\text{-}bisim \longleftarrow$ `bisim2((`$p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b)$`),`
    `(`$p(b,a) \Rightarrow p(a,b) \Rightarrow p(b,a)$`))`
    `bothempty(`$p(a,b)$`,`$p(b,a)$`)` is false
    `bothfail(`$p(a,b)$`,`$p(b,a)$`)` is false
    `occur((`$p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b)$`),`
      `(`$p(b,a) \Rightarrow p(a,b) \Rightarrow p(b,a)$`))` is true
    return $true$
  $SG' = \emptyset$ and $found\text{-}bisim = true$ (quit the inner while loop)
  $found\text{-}bisim = true$
  $SF' = \emptyset$ (continue with the second part of the algorithm)
  $SG' \longleftarrow \{p(b,a)\}$
  $SG' \neq \emptyset$
  $G' \longleftarrow p(b,a)$, $SG' \longleftarrow \emptyset$
  $found\text{-}bisim \longleftarrow false$
  $SF' \longleftarrow \{p(a,b)\}$
  $SF' \neq \emptyset$ and $found\text{-}bisim = false$
  $F' \longleftarrow p(a,b)$, $SF' \longleftarrow \emptyset$
  $found\text{-}bisim \longleftarrow$ `bisim2((`$p(b,a) \Rightarrow p(a,b) \Rightarrow p(b,a)$`),`
    `(`$p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b)$`))`
    `bothempty(`$p(b,a)$`,`$p(a,b)$`)` is false
    `bothfail(`$p(b,a)$`,`$p(a,b)$`)` is false
    `occur((`$p(b,a) \Rightarrow p(a,b) \Rightarrow p(b,a)$`),`
      `(`$p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b)$`))` is true
    return $true$
  $SF' = \emptyset$ and $found\text{-}bisim = true$ (quit the while loop)
  $found\text{-}bisim = true$
  $SG' = \emptyset$
  return $true$
$SG' = \emptyset$ and $found\text{-}bisim = true$ (quit the while loop)
$found\text{-}bisim = true$
$SF' = \emptyset$ (continue with the second part of the algorithm)
$SG' \longleftarrow \{p(a,b)\}$
$SG' \neq \emptyset$
$G' \longleftarrow p(a,b)$, $SG' \longleftarrow \emptyset$

$found\text{-}bisim \longleftarrow false$
$SF' \longleftarrow \{p(b,a)\}$
$SF' \neq \emptyset$ and $found\text{-}bisim = false$
$F' \longleftarrow p(b,a)$, $SF' \longleftarrow \emptyset$
$found\text{-}bisim \longleftarrow$ `bisim2((`$p(b,a) \Rightarrow p(a,b)$`),(`$p(b,a) \Rightarrow p(a,b)$`))`
  `bothempty(`$p(a,b)$`,`$p(b,a)$`)` is false
  `bothfail(`$p(a,b)$`,`$p(b,a)$`)` is false
  `occur((`$p(b,a) \Rightarrow p(a,b)$`),(`$p(a,b) \Rightarrow p(b,a)$`))` is false
  $SF \longleftarrow \{p(b,a)\}$
  $SG \longleftarrow \{p(a,b)\}$
  $SF \neq \emptyset$ and $SG \neq \emptyset$
  $SF' \longleftarrow \{p(b,a)\}$
  $SF' \neq \emptyset$
  $F' \longleftarrow p(b,a)$, $SF' \longleftarrow \emptyset$
  $found\text{-}bisim \longleftarrow false$
  $SG' \longleftarrow \{p(a,b)\}$
  $SG' \neq \emptyset$ and $found\text{-}bisim = false$
  $G' \longleftarrow p(a,b)$, $SG' \longleftarrow \emptyset$
  $found\text{-}bisim \longleftarrow$ `bisim2((`$p(b,a) \Rightarrow p(a,b) \Rightarrow p(b,a)$`),`
    `(`$p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b)$`))`
    `bothempty(`$p(b,a)$`,`$p(a,b)$`)` is false
    `bothfail(`$p(b,a)$`,`$p(a,b)$`)` is false
    `occur((`$p(b,a) \Rightarrow p(a,b) \Rightarrow p(b,a)$`),`
      `(`$p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b)$`))` is true
    return $true$
  $SG' = \emptyset$ and $found\text{-}bisim = true$ (quit the while loop)
  $found\text{-}bisim = true$
  $SF' = \emptyset$ (continue with the second part of the algorithm)
  $SG' \longleftarrow \{p(a,b)\}$
  $SG' \neq \emptyset$
  $G' \longleftarrow p(a,b)$, $SG' \longleftarrow \emptyset$
  $found\text{-}bisim \longleftarrow false$
  $SF' \longleftarrow \{p(b,a)\}$
  $SF' \neq \emptyset$ and $found\text{-}bisim = false$
  $F' \longleftarrow p(b,a)$, $SF' \longleftarrow \emptyset$
  $found\text{-}bisim \longleftarrow$ `bisim2((`$p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b)$`),`
    `(`$p(b,a) \Rightarrow p(a,b) \Rightarrow p(b,a)$`))`
    `bothempty(`$p(a,b)$`,`$p(b,a)$`)` is false
    `bothfail(`$p(a,b)$`,`$p(b,a)$`)` is false
    `occur((`$p(a,b) \Rightarrow p(b,a) \Rightarrow p(a,b)$`),`
      `(`$p(b,a) \Rightarrow p(a,b) \Rightarrow p(b,a)$`))` is true
    return $true$
  $SF' = \emptyset$ and $found\text{-}bisim = true$ (quit the while loop)
  $found\text{-}bisim = true$
  $SG' = \emptyset$
  return $true$
$SF' = \emptyset$ and $found\text{-}bisim = true$ (quit the inner while loop)
$found\text{-}bisim = true$ (continue with the outer while loop)
$SG' = \emptyset$
return $true$

Thus the algorithm `bisim2(`$F_1(a,b)$`,`$H_1(b,a)$`)` returns *true*.

In order to demonstrate the decidability of (π_{res}), we need to prove the following lemmas for all Datalog goals F_1, G_1.

Lemma 5.3.1 (Termination)**.** `bisim2((`F_1`),(`G_1`))` *terminates.*

Lemma 5.3.2 (Completeness)**.** *If* $P \cup \{F_1\}\ \mathcal{Z}^P_{max}\ P \cup \{G_1\}$*, then* `bisim2((`$F_1$`),(`$G_1$`))` *returns* `true`*.*

Lemma 5.3.3 (Soundness)**.** *If* `bisim2((`F_1`),(`G_1`))` *returns* `true`*, then* $P \cup \{F_1\}\ \mathcal{Z}^P_{max}\ P \cup \{G_1\}$.

Let $\prec$ be the binary relation on the set of all pairs of SLD-derivations defined by: $((F_1 \Rightarrow \cdots \Rightarrow F_i), (G_1 \Rightarrow \cdots \Rightarrow G_i)) \prec ((F'_1 \Rightarrow \cdots \Rightarrow F'_j), (G'_1 \Rightarrow \cdots \Rightarrow G'_j))$ iff $i > j$, $(F_1 \Rightarrow \cdots \Rightarrow F_j) = (F'_1 \Rightarrow \cdots \Rightarrow F'_j)$, $(G_1 \Rightarrow \cdots \Rightarrow G_j) = (G'_1 \Rightarrow \cdots \Rightarrow G'_j)$, there exists no substitutions σ, τ such that $F'_l = F'_k[\sigma]$, $G'_l = G'_k[\tau]$ for some $1 \leq k < l \leq i$. Obviously, $\prec$ is a partial order on the set of all pairs of SLD-derivations. Let us demonstrate that $\prec$ is well-founded. Suppose $\prec$ is not well-founded. Hence, there exists infinite SLD-derivations $(F_1 \Rightarrow F_2 \Rightarrow \cdots)$, $(G_1 \Rightarrow G_2 \Rightarrow \cdots)$ such that for all substitutions σ, τ, $F_l \neq F_k[\sigma]$ or $G_l \neq G_k[\tau]$ for all $1 \leq k < l$. This contradicts the following claim.

Claim 1. *Let* $(F_1 \Rightarrow F_2 \Rightarrow \cdots)$, $(G_1 \Rightarrow G_2 \Rightarrow \cdots)$ *be infinite SLD-derivations. There exists* $M \geq 1$ *and there exists substitutions* σ, τ *such that* $F_M = F_k[\sigma]$, $G_M = G_k[\tau]$ *for some* $1 \leq k < M$.

Proof. According to [9, Corollary 4.14], there exists $N \geq 1$ such that for all $j \geq 1$, $|F_j| \leq N$ and $|G_j| \leq N$. Let $\cong$ be the binary relation on the set of all pairs of goals of size bounded by N defined by: $(F, G) \cong (F', G')$ iff

- there exists substitutions σ, τ such that $F[\sigma] = F'$, $G[\tau] = G'$,
- there exists substitutions σ', τ' such that $F'[\sigma'] = F$, $G'[\tau'] = G$.

Obviously, $\cong$ is an equivalence relation on the set of all pairs of goals of size bounded by N. Seeing that our Datalog language has no function symbols and possesses finitely many predicate symbols and constants, $\cong$ determines a finite number of equivalence relations. Thus, one of these equivalence classes is repeated infinitely often in the sequence $(F_1, G_1), (F_2, G_2), \cdots$. Therefore, there exists $M \geq 1$ and there exists substitutions σ, τ such that $F_M = F_k[\sigma]$, $G_M = G_k[\tau]$ for some $1 \leq k < M$. □

Proof of Lemma 5.3.1. In order to show the termination of `bisim2`, it suffices to repeat the proof of the termination of `bisim1`, with $\prec$ instead of $\ll$. □

Proof of Lemma 5.3.2. Let us consider the following property: $(Prop_2((F'_1 \Rightarrow \cdots \Rightarrow F'_j), (G'_1 \Rightarrow \cdots \Rightarrow G'_j)))$ if $P \cup \{F'_j\}\ \mathcal{Z}^P_{max}\ P \cup \{G'_j\}$ then `bisim2((`$F'_1 \Rightarrow \cdots \Rightarrow F'_j$`),(`$G'_1 \Rightarrow \cdots \Rightarrow G'_j$`))` returns `true`. Again, we proceed by induction, this time with $\prec$ instead of $\ll$. Suppose $((F'_1 \Rightarrow \cdots \Rightarrow F'_j), (G'_1 \Rightarrow \cdots \Rightarrow G'_j))$ is such that for all $((F_1 \Rightarrow \cdots \Rightarrow F_i), (G_1 \Rightarrow \cdots \Rightarrow G_i))$, if $((F_1 \Rightarrow \cdots \Rightarrow F_i), (G_1 \Rightarrow \cdots \Rightarrow G_i)) \prec ((F'_1 \Rightarrow \cdots \Rightarrow F'_j), (G'_1 \Rightarrow \cdots \Rightarrow G'_j))$ then $Prop_2((F_1 \Rightarrow \cdots \Rightarrow F_i), (G_1 \Rightarrow \cdots \Rightarrow G_i))$. Let us show that $Prop_2((F'_1 \Rightarrow \cdots \Rightarrow F'_j), (G'_1 \Rightarrow \cdots \Rightarrow G'_j))$. Suppose $P \cup \{F'_j\}\ \mathcal{Z}^P_{max}\ P \cup \{G'_j\}$. Hence, for all successors F' of F'_j, there exists a successor G' of G'_j such that $P \cup \{F'\}\ \mathcal{Z}^P_{max} P \cup \{G'\}$, and conversely. Seeing that $((F'_1 \Rightarrow \cdots \Rightarrow F'_j \Rightarrow F'), (G'_1 \Rightarrow \cdots \Rightarrow G'_j \Rightarrow G')) \prec ((F'_1 \Rightarrow \cdots \Rightarrow F'_j), (G'_1 \Rightarrow \cdots \Rightarrow G'_j))$, then by induction hypothesis, $Prop_2((F'_1 \Rightarrow \cdots \Rightarrow F'_j \Rightarrow F'), (G'_1 \Rightarrow \cdots \Rightarrow G'_j \Rightarrow G'))$. Since $P \cup \{F'\}\ \mathcal{Z}^P_{max}\ P \cup \{G'\}$, then `bisim2((`$F'_1 \Rightarrow \cdots \Rightarrow F'_j \Rightarrow F'$`),(`$G'_1 \Rightarrow \cdots \Rightarrow G'_j \Rightarrow G'$`))` returns `true`. As a result, one sees that `bisim2((`$F'_1 \Rightarrow \cdots \Rightarrow F'_j$`),(`$G'_1 \Rightarrow \cdots \Rightarrow G'_j$`))` returns `true`. □

Proof of Lemma 5.3.3. It suffices to demonstrate that the binary relation $\mathcal{Z}$ defined as follows between Datalog goals is a bisimulation: $F \mathcal{Z} G$ iff there exists SLD-derivations $(F_1 \Rightarrow \cdots \Rightarrow F_i)$, $(G_1 \Rightarrow \cdots \Rightarrow G_i)$ such that $F_i = F$, $G_i = G$ and `bisim2((`$F_1 \Rightarrow \cdots \Rightarrow F_i$`),(`$G_1 \Rightarrow \cdots \Rightarrow G_i$`))` returns `true`. Let F, G be Datalog goals such that $F \mathcal{Z} G$. Hence there exists SLD-derivations $(F_1 \Rightarrow \cdots \Rightarrow F_i)$, $(G_1 \Rightarrow \cdots \Rightarrow G_i)$ such that $F_i = F$, $G_i = G$ and `bisim2((`$F_1 \Rightarrow \cdots \Rightarrow F_i$`),(`$G_1 \Rightarrow \cdots \Rightarrow G_i$`))` returns `true`. Thus, it is easy to verify that the first condition characterizing bisimulations holds for $\mathcal{Z}$ as $F_i = \square$ iff $G_i = \square$. Let F' be a resolvent of F_i and a clause in P. Since `bisim2((`$F_1 \Rightarrow \cdots \Rightarrow F_i$`),(`$G_1 \Rightarrow \cdots \Rightarrow G_i$`))` returns `true`, then there exists a resolvent G' of G_i and a clause in P such that `bisim2((`$F_1 \Rightarrow \cdots \Rightarrow F_i \Rightarrow F'$`),(`$G_1 \Rightarrow \cdots \Rightarrow G_i \Rightarrow G'$`))` returns `true`, i.e. $F' \mathcal{Z} G'$. Consequently, the second condition characterizing bisimulations holds for $\mathcal{Z}$. For the third condition characterizing bisimulations, a proof similar to the one presented for the second condition applies here. Thus $\mathcal{Z}$ is a bisimulation. □

As a consequence of lemmas 5.3.1 – 5.3.3, we have:

Theorem 5.3.1. `Algorithm` *4 is a sound and complete decision procedure for* (π_{res}).

It follows that (π_{res}) is decidable. Moreover,

Theorem 5.3.2. (π_{res}) *is in 2EXPTIME.*

Proof. Let P be a restricted Datalog program and G be a goal. Let n be the maximal number of atoms in the clauses of P or in G, p be the number of predicate symbols in P, a be the maximal arity of the predicate symbols in P, and c be

the number of constants in P. Thus, the number of variables in P is about $n \times a$, the number of ground atoms is bounded by $p \times c^a$ and the total number of atoms is bounded by $p \times (c + n \times a)^a$. Moreover, as the number of goals of size n is bounded by $p^n \times (c + n \times a)^{a \times n}$, the number of pairs of goals of size n is bounded by $p^{2 \times n} \times (c + n \times a)^{2 \times a \times n}$. Then, according to the claim of page 117. The maximal length of SLD-derivations that are considered in `Algorithm` 4 will be smaller than $M = p^{2 \times n} \times (c + n \times a)^{2 \times a \times n}$. Now, replacing D by M in the proof of Theorem 2, one can demonstrate that the time complexity of our algorithm is about $2^M \times s^{2 \times M} \leq 2^{p^{2 \times n} \times (c + n \times a)^{2 \times a \times n}} \times s^{2 \times p^{2 \times n} \times (c + n \times a)^{2 \times a \times n}}$.□ □

In practice, we can limit the tests performed by `occur` in such a way that `occur((`$F_1 \Rightarrow \cdots \Rightarrow F_i$`),(`$G_1 \Rightarrow \cdots \Rightarrow G_i$`))` returns `true` iff there exists substitutions σ, τ such that $F_i = F_k[\sigma]$, $G_i = G_k[\tau]$ for some $1 \leq k < i$.

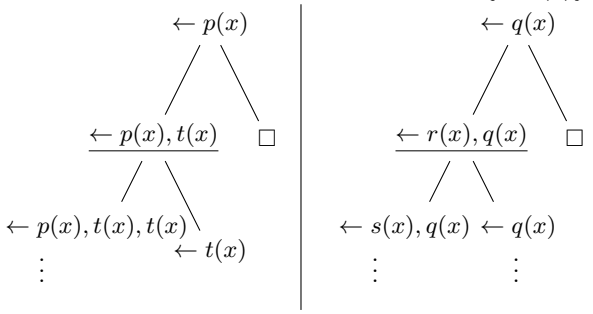
5.4 Notes on Bisimilarity for nvi and svo Goals

We tried to prove if we can decide whether two given goals are bisimilar for *nvi* programs and *svo* programs. Unfortunately, applying the techniques of loop detection developed in [9] does not seem to allow us to determine if two given goals are bisimilar with respect to an *nvi* or an *svo* logic program.

Example 5.3 Example showing that loop detection in SLD-trees for *nvi* and *svo* logic programs do not preserve bisimilarity

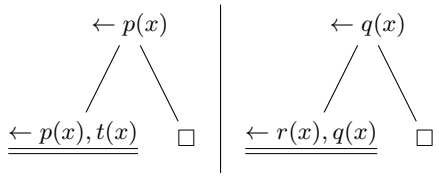
Let P be the following *nvi* program:
$C_1 : p(x) \leftarrow p(x), t(x)$;
$C_2 : p(x) \leftarrow$;
$C_3 : q(x) \leftarrow r(x), q(x)$;
$C_4 : q(x) \leftarrow$;
$C_5 : r(x) \leftarrow$;
$C_6 : r(x) \leftarrow s(x)$;
$C_7 : s(x) \leftarrow$

and let F, G be respectively the following goals $P \cup \{\leftarrow p(x)\}$ and $P \cup \{\leftarrow q(x)\}$.



Obviously, F and G are not bisimilar with respect to P. To see this, it suffices to verify that, since the goal $P \cup \{\leftarrow p(x), t(x)\}$ has an empty successor whereas all successors of the goal $P \cup \{\leftarrow r(x), q(x)\}$ are non empty, then $P \cup \{\leftarrow p(x), t(x)\}$ and $P \cup \{\leftarrow r(x), q(x)\}$ are not bisimilar with respect to P.

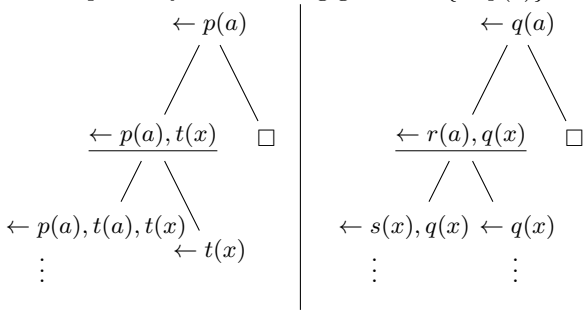
Nevertheless, if one applies the techniques of loop detection developed in [9], then one obtains the following bisimilar reduced SLD-trees:



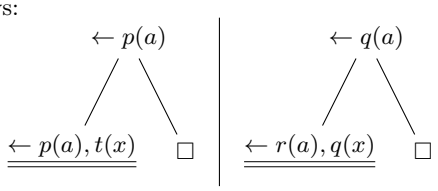
Note that the previous logic program is not a *svo* program, since the variable x is encountered more than once in the body of the clauses C_1 and C_3. Rewriting it in the following way leads to a similar result:
$C_1 : p(a) \leftarrow p(a), t(x)$;
$C_2 : p(x) \leftarrow$;
$C_3 : q(a) \leftarrow r(a), q(x)$;

$C_4 : q(x) \leftarrow;$
$C_5 : r(x) \leftarrow;$
$C_6 : r(x) \leftarrow s(x);$
$C_7 : s(x) \leftarrow$

Let F, G be respectively the following goals $P \cup \{\leftarrow p(a)\}$ and $P \cup \{\leftarrow q(a)\}$.



Again, F and G are not bisimilar with respect to P. The reduced SLD-trees obtained by applying the techniques of loop detection developed in [9] are as follows:



5.5 Summary

In this chapter, we have reused the concept of bisimulation, previously presented in section 4.3.1, between datalog goals: two Datalog goals are bisimilar with respect to a given program when their SLD-trees are bisimilar. As proved in Section 5.1, deciding whether two given goals are bisimilar with respect to a given general logic program is undecidable. Hence, a natural question is to restrict the language of logic programming as done in Section 5.2 and Section 5.3. Thus, when the given logic program is hierarchical or restricted, the problem of deciding whether two given goals are bisimilar becomes decidable in 2EXPTIME. The proof of decidability of bisimulation problem for restricted logic program that we presented in Section 5.3 is based on techniques that were developed in [9] for detecting loops in logic programming.

Chapter 6

Application

In this chapter, we will aim to provide an application of the previously presented information flow theory in logic programming in the field of inference control.

For this, we will begin, in section , by introducing the notion of **indistinguishability** of the flow in logic programming. We will extend this notion and propose the notion of **level** of goals in logic programming.

Section 6.2 will be devoted to recall what we mean about **inference control**, and talk about **preventing** inference control for **information systems**.

In section 6.3, we will formally define, **protection mechanisms**, **secure mechanisms**, **mechanisms** and **confidentiality policies**. We will end the chapter by giving an example of a secure protection mechanism for deductive databases and this using the previously exposed notions. We will give a formal proof too of the security of this protection mechanism.

6.1 Level of indistinguishability of information flow in logic programming

We will proceed in this section to refine the notion of information flow for Datalog logic programs. For this, we propose the notion of level of indistinguishability of the flow.

For a Datalog logic program P and a goal $G(x, y)$ with the variable x considered as an input variable and y as an output variable, let $\equiv$ be a binary relation over $U_{L(P)}$ of cardinality n. Let a, b be two distinct elements of $U_{L(P)}$.

- For the first definition of information flow (based on success and failure), we say that $a \equiv b$ iff both $P \cup \{\leftarrow p(a, y)\}$ and $P \cup \{\leftarrow p(b, y)\}$ succeed or both $P \cup \{\leftarrow p(a, y)\}$ and $P \cup \{\leftarrow p(b, y)\}$ do not succeed (in the sense that both goals can fail or not terminate because of the presence of loops).
- For the second definition of information flow (based on substitution answers), we say that $a \equiv b$ iff $\theta(P \cup \{\leftarrow p(a, y)\}) = \theta(P \cup \{\leftarrow p(b, y)\})$.
- As for the third definition of information flow (based on bisimulation between goals), we say that $a \equiv b$ iff $Tree(P \cup \{\leftarrow p(a, y)\})\ Z^P_{max}\ Tree(P \cup \{\leftarrow p(b, y)\}))$.

Lemma 6.1.1. *The binary relation $\equiv$ is reflexive.*

Lemma 6.1.2. *The binary relation $\equiv$ is symmetric.*

Lemma 6.1.3. *The binary relation $\equiv$ is transitive.*

Lemma 6.1.4. *$\equiv$ is an equivalence relation.*

Proof. Proof. By lemmas 6.1.1, 6.1.2 and 6.1.3. □

It is worth noting here, that the definitions presented in chapter 4 rely on the fact that for a logic program P and a goal $G(x, y)$, an information flow passes from x to y if one can find just two distinguishable equivalence classes. In the next subsection, we will use this notion of equivalence classes and its cardinality to define the level of an information flow as one of its characteristics.

6.1.1 Level of information flows in logic programs

In this section, we will present the definitions of the level of information flow based on the notion of equivalence classes.

Definition 6.1.1 (Level of a logic goal)**.** *For a Datalog logic program P and a goal $G(x, y)$, the level of the goal $G(x, y)$ is equal to the cardinality of the smallest equivalence class.*

Example 6.1 Example of a level of a goal in logic programming

Let P be the following program:
$C_1 : p(a, b) \leftarrow$;
$C_2 : p(a, c) \leftarrow$;
$C_3 : p(b, c) \leftarrow$;
$C_4 : p(c, b) \leftarrow$;
The Herbrand Universe $U_{L(P)}$ is equal to $\{a, b, c\}$.

For the definition of the flow based on success and failure, it is easy to see that:
$P \cup \{\leftarrow p(a, y)\}$ succeeds,
$P \cup \{\leftarrow p(b, y)\}$ succeeds, and
$P \cup \{\leftarrow p(c, y)\}$ succeeds.
Thus, $a \equiv b \equiv c$. Consequently, the cardinality of the equivalence class corresponding to success is equal to 3, while the one corresponding to failure is equal to 0. Thus, the level based on success and failure which corresponds to cardinality of the smallest equivalence class is equal to 0.

For the definition of the flow based on substitution answers,we have:
$\Theta[P \cup \{\leftarrow p(a, y)\}] = \{y \mapsto b, y \mapsto c\}$,
$\Theta[P \cup \{\leftarrow p(b, y)\}] = \{y \mapsto c\}$, and
$\Theta[P \cup \{\leftarrow p(c, y)\}] = \{y \mapsto b\}$.
Thus, $a \equiv c$, and $a \equiv b$. Consequently, the cardinality of each equivalence class is equal to 2. Consequently, the level based on substitution answers is equal to 2.

For the definition of the flow based on bisimulation,we have:
$Tree(P \cup \{\leftarrow p(a, y)\})\ Z^P_{max}\ Tree(P \cup \{\leftarrow p(b, y)\})\ Z^P_{max}\ Tree(P \cup \{\leftarrow p(b, y)\})$.
Thus, $a \equiv b \equiv c$. Consequently, the cardinality of the equivalence class is equal to 3. Consequently, the level based on bisimulation is equal to 3.

One can see that for an Herbrand universe $U_{L(P)}$ of cardinality n, if the level of indistinguishability is n then we have 1 equivalence class which is the class of cardinality n $\{a_1, \cdots, a_n\}$.

Theoretically, this level can be calculated for each of the three definitions of information flow previously presented. For example, for the first definition of flow based on success and failure, one can write the following algorithm:

Require: A Datalog logic program P, a goal $G(x, y)$, finite Herbrand Universe $U_{L(P)} = \{a_1, \cdots, a_n\}$
Ensure: Level of $G(x, y)$
begin
  $Level_{succ} \leftarrow 0$; counter on the number of successful goals
  $Level_{no-succ} \leftarrow 0$; counter on the number of non-successful goals
  $i \leftarrow 1$; counter on the set of the Herbrand universe
  while $i \leq n$ **do**
    if $P \cup \{\leftarrow p(a_i, y)\}$ *succeeds* **then**
      $Level_{succ} \leftarrow Level_{succ} + 1$;
    else
      $Level_{no-succ} \leftarrow Level_{no-succ} + 1$;
    $i \leftarrow i + 1$;
  return $min(Level_{succ}, Level_{no-succ})$;

Let us prove this algorithm is correct. In fact, the previous algorithm terminates since starting from $i = 1$ and after n iterations, the *while* loop will terminate. Since the Herbrand Universe contains at least one element, the while loop will be executed at least one. After one iteration, the $min(Level_{succ}, Level_{no-succ})$ will be equal to 0, despite the fact that the goal $P \cup \{G(a_1, y)\}$ succeeds or fails. Now suppose that the level is equal to $min(Level_{succ}, Level_{no-succ})$ at the iteration $i - 1 \leq n$. Let us prove now that the level calculated at iteration i will still equal to $min(Level_{succ}, Level_{no-succ})$. In the *while* loop, the values of $Level_{succ}$, $Level_{no-succ}$ and consequently of min are modified as follows:
If $(P \cup \{\leftarrow p(a_i, y)\}$ succeeds) then $Level_{succ}$ will be incremented and $min(Level_{succ}, Level_{no-succ})$ keeps its old value;
If $(P \cup \{\leftarrow p(a_i, y)\}$ does not succeeds) then $Level_{no-succ}$ will be incremented and $min(Level_{succ}, Level_{no-succ})$ keeps also its old value;
Then in both situations, the level will satisfy $min(Level_{succ}, Level_{no-succ})$. This proves the correctness of the algorithm.

Example 6.2 Running the algorithm of calculating the level based on success and failure

Let P be the same program as in example 6.1. We saw that the level is equal to 0. We will now run the algorithm on the same example and prove the same result. Recall that the Herbrand Universe is equal to $\{a, b, c\}$.
$Level_{succ} \leftarrow 0$;
$Level_{no-succ} \leftarrow 0$;

$i \leftarrow 1$; (initializing the counter)
$1 \leq 3$ (checking the while loop condition)
$P \cup \{\leftarrow p(a, y)\}$ succeeds
$Level_{succ} \leftarrow 1$; (incrementing the number of successful goals)
$i \leftarrow 2$; (incrementing the counter)
$2 \leq 3$ (checking the while loop condition)
$P \cup \{\leftarrow p(b, y)\}$ succeeds
$Level_{succ} \leftarrow 2$; (incrementing the number of successful goals)
$i \leftarrow 3$; (incrementing the counter)
$3 \leq 3$ (checking the while loop condition)
$P \cup \{\leftarrow p(c, y)\}$ succeeds
$Level_{succ} \leftarrow 3$; (incrementing the number of successful goals)
$i \leftarrow 4$; (incrementing the counter)
$4 \not\leq 3$ (quit the while loop)
return $min(0, 3)$;

Thus the level calculated based on success and failure is equal to 0.

As for the second definition of information flow based on substitution answers, one can write the following algorithm:

Require: A Datalog logic program P, a goal $G(x, y)$, finite Herbrand Universe $U_{L(P)} = \{a_1, \cdots, a_n\}$, m the total number of possible substitution answers of the y variable for all the goals $P \cup \{\leftarrow G(a_1, y)\}, \cdots, P \cup \{\leftarrow G(a_n, y)\}$.
Ensure: Level of $G(x, y)$
begin
  table $T[m]$; table of counters, all initialized to 0, corresponding to the m different substitution answers. The indexes of the table T are the m different substitution answers and the corresponding value represent the total number of occurrence of that substitution answer
  $i \leftarrow 1$; counter on the set of the Herbrand Universe
  $sub \leftarrow \theta[P \cup \{p(a_1, y)\}]$;we initialize the *sub* by the first substitution answer of y
  while $i \leq n$ **do**
    $tmps \leftarrow \Theta[P \cup \{\leftarrow p(a_i, y)\}]$; as $P \cup \{\leftarrow p(a_i, y)\}$ can have multiple substitution answers, *tmps* is the table containing these substitution answers
    $j \leftarrow 1$; counter on the set of the substitution answers for the goal $P \cup \{\leftarrow p(a_i, y)\}$
    while $j <= count(tmps)$ **do**
      $T[tmps[j]] \leftarrow T[tmps[j]] + 1$;
      if $T[tmps[j]] < T[sub]$ **then**
        $sub \leftarrow tmps[j]$;
      $j \leftarrow j + 1$;
    $i \leftarrow i + 1$;
  return $T[sub]$;

To prove the correctness of this algorithm, remark first that the Herbrand Universe contains at least one element and thus starting from $i = 1$ and after n iterations, the outer *while* loop will terminate. Seeing that each goal in a Datalog logic program has a finite number of substitution answers, the inner *while* loop will also terminate. After one iteration of the outer and the inner while loop, $T[sub]$ will be equal to 1 (easy to verify), despite the fact if the goal $P \cup \{G(a_1, y)\}$ has one or multiple substitution answers.
Now suppose that the level is equal to $T[sub]$ at the iteration $i - 1 \leq n$. Let us prove now that the level calculated at iteration i will still equal to $T[sub]$.
In the *while* loops, the value of *sub* is modified as follows:
If $(T[tmps[j]] < T[sub])$ then the value of *sub* is replaced by $tmps[j]$;
If $(T[tmps[j]] \geq T[sub])$ then *sub* keeps its old value;

Then in both situations, $T[sub] \leq T[tmps[j]]$. This proves the correctness of the algorithm.

Example 6.3 Running the algorithm of calculating the level based on substitution answers

Let P be the same program as in example 6.1. We saw that the level is equal to 2. We will now run the algorithm on the same example and prove the same result. Recall that the Herbrand Universe is equal to $\{a, b, c\}$, and that the total number of possible and different substitution answers for all the possible goals $P \cup \{\leftarrow p(a, y)\}$, $P \cup \{\leftarrow p(b, y)\}$ and $P \cup \{\leftarrow p(c, y)\}$ is equal to 2, namely, $y \mapsto b$ and $y \mapsto c$. The execution trace is as follows:

$T[y \mapsto b] = 0$, $T[y \mapsto c] = 0$;
$i \leftarrow 1$; (initializing the counter)
$sub \leftarrow y \mapsto b$; (corresponding to $\theta[P \cup \{\leftarrow p(a, y)]$)
$1 \leq 3$ (checking the outer while loop condition)
$tmps[1] \leftarrow y \mapsto b$, $tmps[2] \leftarrow y \mapsto c$, (corresponding to $\Theta[P \cup \{\leftarrow p(a, y)]$)
$j \leftarrow 1$; (initializing the counter on the set of substitution answers)
$1 \leq 2$ (checking the inner while loop condition)
$T[y \mapsto b] \leftarrow 1$;
$T[y \mapsto b] \not< T[y \mapsto b]$; (skip the if body)
$j \leftarrow 2$; (increment the counter)
$2 \leq 2$ (checking the inner while loop condition)
$T[y \mapsto c] \leftarrow 1$;
$T[y \mapsto c] \not< T[y \mapsto b]$; (skip the if body)
$j \leftarrow 3$; (increment the counter)
$3 \not\leq 2$ (quit the inner while loop)
$i \leftarrow 2$; (increment the counter)
$2 \leq 3$ (checking the outer while loop condition)
$tmps[1] \leftarrow y \mapsto c$ (corresponding to $\Theta[P \cup \{\leftarrow p(b, y)]$)
$j \leftarrow 1$; (initializing the counter on the set of substitution answers)
$1 \leq 1$ (checking the inner while loop condition)
$T[y \mapsto c] \leftarrow 2$;
$T[y \mapsto c] \not< T[y \mapsto b]$; (skip the if body)
$j \leftarrow 2$; (increment the counter)
$2 \not\leq 1$ (quit the inner while loop)
$i \leftarrow 3$; (increment the counter)
$3 \leq 3$ (checking the outer while loop condition)
$tmps[1] \leftarrow y \mapsto b$ (corresponding to $\Theta[P \cup \{\leftarrow p(c, y)]$)
$j \leftarrow 1$; (initializing the counter on the set of substitution answers)
$1 \leq 1$ (checking the inner while loop condition)
$T[y \mapsto b] \leftarrow 2$;
$T[y \mapsto b] \not< T[y \mapsto b]$; (skip the if body)
$j \leftarrow 2$; (increment the counter)
$2 \not\leq 1$ (quit the inner while loop)
$i \leftarrow 4$; (increment the counter)
$4 \not\leq 3$ (quit the outer while loop)

return $T[y \mapsto b] = 2$;

Thus the level calculated based on substitution answers is equal to 2.

As for calculating the level based on the definition of bisimulation, one can use a similar algorithm to the one presented above jointly with the sound and complete algorithm presented in chapter 5 that decides if two goals are bisimilar for hierarchical and restricted Datalog logic programs.

6.1.2 Specification of information flows in Datalog logic programs

In order to present the notion of specification, we will motivate it first by giving an example.

Consider a system composed by a deductive database (represented as facts in logic programming) and a user who can run queries (in the form of logic goals).

Example 6.4 Example showing the specification of an information flow in a Datalog logic program - part I

Let P be the following program representing:

- the three floors and its corresponding departments in a hospital
  $hospital(floor_1, cancerology) \leftarrow$;
  $hospital(floor_2, cardiology) \leftarrow$;
  $hospital(floor_2, urology) \leftarrow$;
  $hospital(floor_3, gynaecology) \leftarrow$
- and some of the patients location in the hospital
  $location(Ana, floor_1) \leftarrow$;
  $location(Bob, floor_1) \leftarrow$;
  $location(Carl, floor_2) \leftarrow$;
  $location(David, floor_2) \leftarrow$;
  $location(Elianor, floor_3) \leftarrow$

The goal is to prevent the user to know exactly for example the existence of the exact departments on each floor or the exact location of some specific patient.

Definition 6.1.2. *For a Datalog logic program P and a two goals $F(x, y)$ and $G(x, y)$, we say that:*

- *the goal $F(x, y)$ is critical iff the level of $F(x, y)$ is equal to 1.*

- *the goal $F(x, y)$ is weaker than $G(x, y)$ iff the level of $F(x, y)$ is greater than the level of $G(x, y)$.*
- *the goal $F(x, y)$ is stronger than $G(x, y)$ iff the level of $F(x, y)$ is smaller than the level of $G(x, y)$.*

Example 6.5 Example showing the specification of an information flow in a Datalog logic program - part II

It is easy to verify that for the definition of flow of information based on substitution answers, there is three equivalence classes for the goal $P \cup \{\leftarrow hospital(x, y)\}$ of cardinalities 1 and 2. Thus the level of $P \cup \{\leftarrow hospital(x, y)\}$ is equal to 1. Whereas, for the goal $P \cup \{\leftarrow location(x, y)\}$, one can count three equivalence classes too, two of cardinality 2 and one of cardinality 1. So, the level of $P \cup \{\leftarrow location(x, y)\}$ is equal to 1. Consequently, both goals are critical.

Lemma 6.1.5. *For a Datalog logic program P and a goal $F(x, y)$, if $F(x, y)$ is critical, then the output variable y reveals information about the variable x.*

Proof. By definition, if a goal $F(x, y)$ is critical then the level of $F(x, y)$ is equal to 1. Moreover, when a level of a goal is equal to 1, it means that the cardinality of the corresponding equivalence class of the goal $F(x, y)$ is equal to 1. Thus, the variable x will be uniquely identified. □

Example 6.6 Example showing the specification of an information flow in a Datalog logic program - part III

As both goals $P \cup \{\leftarrow hospital(x, y)\}$ and $P \cup \{\leftarrow location(x, y)\}$ are critical, the output variable y can conveys information as we can see next. Suppose that a user runs the goal $P \cup \{\leftarrow hospital(floor_1, y)\}$, than the corresponding output variable y will be unified uniquely with *cancerology*. Thus the disclosure of this information will render the identification of the probable disease of the patient residing on the first floor very easy.

Whereas, if the user runs the goals $P \cup \{\leftarrow location(Ana, y)\}$ and $P \cup \{\leftarrow location(Bob, y)\}$, the corresponding output variable y will be unified uniquely with $floor_1$, and the user will still know that both *Ana* and *Bob* are sharing the same floor but without knowing anything about their diseases. Moreover, let us suppose that the user runs the goals $P \cup \{\leftarrow location(Carl, y)\}$ or $P \cup \{\leftarrow location(David, y)\}$, the y variable will be unified with *cardiology* and *urology*. So *Carl* and *David* can be both in the same *cardiology* or *urology* department or each one of them in a different department. A natural question arises here, **what should the system do when it detects that some queries are critical**.

As for the other forms of specifications, one can verify that the level of the goal $P \cup \{\leftarrow location(Carl, y)\}$ is greater than the one of $P \cup \{\leftarrow location(Elianor, y)\}$ and thus the last goal is stronger than the former one.

6.2 Preventive Inference Control for Information Systems

As stated earlier in section 2.3.3, inference control deals with both **enabling** and **preventing** information **gain**, based on actual inferences from observations.

In this section, we focus on security mechanisms for preventing a participant from gaining information about hidden parts of an **information system** from just observing selected visible parts. Note that, previously, in section 2.3.2, we examined constructs (assignments, procedure call, sequential and parallel control structures) of computing systems regarding their potential for information gain.

Preventive inference control for information systems must ensure a trade-off between availability and confidentiality: for any particular user, on the one hand, the system should return useful answers to queries issued by the user according to that user's legitimate purposes and needs, and on the other hand, the system should hide any further information that it keeps available for other users.

As stated in section 2.3.3, preventing inference control can be achieved either by **dynamic monitoring** or **static verification**. In figure 6.1, we outline a generic framework for dynamic monitoring of query sequences, based on a **logic-oriented**, **model-theoretic** approach to complete information systems.

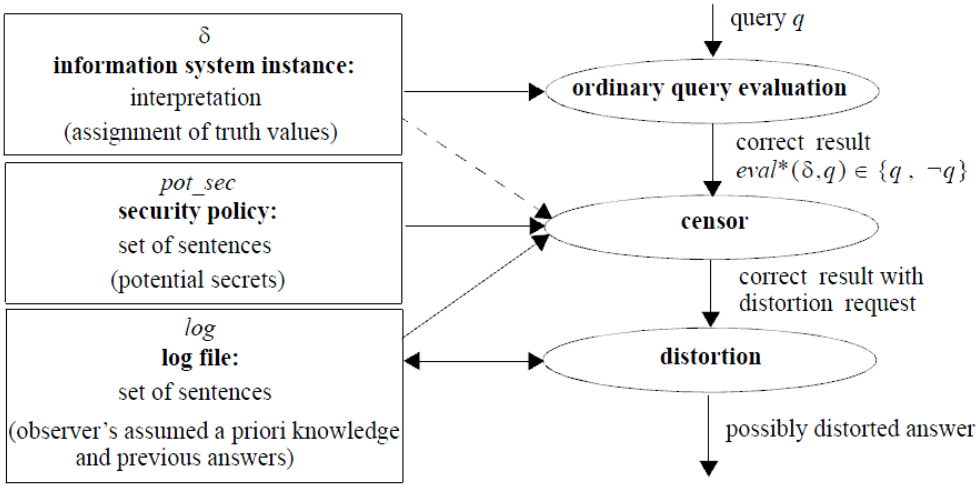


Figure 6.1: Inference control by dynamic monitoring of query sequences, based on a logic-oriented model of information systems.

- The information system maintains data in form of **instance**s. An instance is an **interpretation** (noted δ) which assigns a truth value to every query/goal q of some logic (for this, we assume the existence of a Boolean operator $model-of$ defined in such a way that $\delta\ model-of\ q$ denotes the pertinent truth value).
- A **query/goal** q is a sentence in the pertinent logic.

- Given an instance δ and a query q, the **ordinary query evaluation** of the information system returns the pertinent truth value $false$ or $true$, i.e., $eval(\delta, q) := \delta\ model - of\ q$, or **confirms** the queried sentence or returns its **negation**, i.e., $eval^*(\delta, q) := if\ \delta\ model - of\ q\ then\ q\ else\ \neg q$.

- An **observer** might be a single user or a group of users expected to collaborate. An observer can issue any query sequence $Q =< q_1, q_2, \cdots, q_i, \cdots >$. Given an instance δ, the systems query evaluation returns the corresponding sequence $< eval^*(\delta, q_1), eval^*(\delta, q_2), \cdots, eval^*(\delta, q_i), \cdots >$.

- A **security policy** is specified as a set of **potential secrets**: $pot - sec = \{ps_1, \cdots, ps_k\}$. To declare a sentence as a potential secret requires that the observer should not be able to infer that the sentence is true in the stored instance δ, even if this is actually the case.

- We assume also that the observer have some a priori knowledge, and when issuing a query sequence, he retains all previously returned answers. Accordingly, the system maintains a **log** file that reflects the system's view of the observer's knowledge: log is initialized with the a priori knowledge prior and suitably updated after each of the observer's queries is processed.

- The decision to permit an observation is taken in two steps:

  1. A **censor** inspects whether the ordinary answer would be harmful with respect to the security policy.
  2. The ordinary answer is **distorted** (by refusing to answer or by lying or by using a combination of these two methods) if it is requested by the censor. It is shown that the refusal method, the lying method and the combined method always preserve confidentiality under the respective precondition.

The following example summarizes all these concepts and shows that different behaviors for a simple example with a query sequence of length two.

Example 6.7 Example for an inference control prevention for a propositional information system

Universe of discourse:	propositional atoms			$\{p, q, s_1, s_2, s_3\}$
information system instance:	interpretation	δ	:=	$\{p, \neg q, s_1, s_2, \neg s_3\}$
security policy:	potential secrets	pot-sec	:=	$\{s_1, s_2, s_3\}$
	true potential secrets			$\{s_1, s_2\}$
a priori knowledge:		prior	:=	$\{p \Rightarrow s_1 \vee s_2, p \wedge q \Rightarrow s_3\}$

query	ordinary answer	refusal method	lying method	combined method
p	p	p correct answer	$\neg p$ lie protects disjunction of potential secrets	p correct answer
q	$\neg q$	- refusal protects false potential secret s_3, due to false answer	$\neg q$ correct answer	$\neg q$ correct answer since false answer does not have to be considered

6.3 Secure and Precise Security Mechanisms

We have presented in chapter 2 definitions of security in terms of states of systems. A question arises here: is it possible to devise a generic procedure for developing a mechanism that is both secure and precise using the notion of information flow for logic programs?

For this, we will consider here logic programs as a set of clauses, having all the same predicate definition. The atoms in this logic programs have several input positions but one single output position. Recall that data is *brought in* to a clause through the input positions, and *sent out* through the output positions, review Definitions 3.3.1, 3.3.3 for a formal explanation of argument positions, input and output positions.

Example 6.8 Example of a transformation of logic program having multiple predicate definitions into an equivalent logic program with just one predicate definition

Let P be the following Datalog logic program:
$C_1 : q(a, b) \leftarrow;$
$C_2 : r(b, a) \leftarrow;$
$C_3 : p(x, y) \leftarrow q(x, z), r(z, y);$

Let $\alpha, \beta, \gamma, \delta, \zeta$ and κ be following argument positions of all the variables in the clause C_3: $\alpha =< C_3, 0, p, 1 >$, $\beta =< C_3, 0, p, 2 >$, $\gamma =< C_3, 1, q, 1 >$, $\delta =< C_3, 1, q, 2 >$, $\zeta =< C_3, 2, r, 1 >$ and $\kappa =< C_3, 2, r, 2 >$. Let α, γ and ζ be in $I(C_3)$, β, δ and κ in $O(C_3)$. Recall that $I(C_3)$ denotes the input positions of the clause C_3 and $O(C_3)$ denotes the output positions of the clause C_3.

Seeing that the program P have three different predicate definitions, namely, q, r and p, we can rewrite the program in such a way to have only one predicate definition:
Let P' be P's equivalent program:
$C'_1 : t(q, a, b) \leftarrow;$
$C'_2 : t(r, b, a) \leftarrow;$
$C'_3 : t(p, x, y) \leftarrow t(q, x, z), t(r, z, y)$
P' has now one predicate definition, namely t. t has 3 arguments. The first two are input arguments and the last one is an output argument. It is easy to see that P and P' are equivalent according to the least fixpoint semantics. Thus, one can rewrite any logic program like P into an equivalent logic program having one predicate definition. In the next, we will only consider logic program composed simply of facts, and we will denote a program P by its predicate definition.

We will represent logic programs as abstract functions:

Definition 6.3.1. *(Logic programs as abstract functions) For a logic program P denoted by its predicate definition $t(I_1, \cdots, I_n, O)$, where $I_1, \cdots, I_n$ are input*

positions and O one output position, let p be the function $p : I_1 \times \cdots \times I_n \times O \rightarrow R$. Then p is the function with n ***inputs positions*** *$i_k \in I_k, 1 \leq k \leq n$, and* ***one output position*** *$o \in O$, and* ***one result*** *$r \in R$. O is the set of substitution answers associated to the output position o. Depending on each definition of information flow, R can be equal to $\{success, failure\}$, or to the set of substitution answers corresponding to the output position o, or to the SLD-tree of the goal $P \cup \{\leftarrow t(i_1, \cdots, i_n, o)\}$.*

With respect to the Lemma 6.1.5, we assume that the result $r \in R$ of the function $p(i_1, \cdots, i_n, o)$ conveys information about the input variables $i_1, \cdots, i_n$.

Dealing with confidentiality, a natural question arises here, whether if the result of $p(i_1, \cdots, i_n, o)$ contains any information that could violate the policy. For this, **protection mechanisms** are proposed. A protection mechanism produces for every input, that do not violate the policy, the same value as for p, and for inputs that would impart confidential information an error message. For this, let E be the set of results from a program p that indicate errors.

Definition 6.3.2. *(Protection mechanism) Let p be a function $p : I_1 \times \cdots \times I_n \times O \rightarrow R$. A protection mechanism m is a function $m : I_1 \times \cdots \times I_n \rightarrow R \cup E$ for which, when $i_k \in I_k, 1 \leq k \leq n$, $o \in O$, either*

- $m(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$ *or*
- $m(i_1, \cdots, i_n) \in E$

Example 6.9 Protection mechanism, confidentiality policy, secure mechanism-Part I

We consider here a logic program P, with one input position and one output position, that contains the age of some individuals.
$age(ann, 56) \leftarrow$;
$age(billy, 27) \leftarrow$;
$age(carl, 34) \leftarrow$

The program P is represented by the function: $age : I, O \rightarrow R$. Queries would be of the form $P \cup \{\leftarrow age(ann, A)\}$ for example.

A protection mechanism would be for example to answer correctly (in the terms of the different information flow definitions) every query whenever its input position variable corresponds to one of the Herbrand Universe constants. More formally, let m be the following function:
$m : I \rightarrow R \cup E$ for which:

- $m(i) = age(i, o)$ when $i \in U_{L(P)}$, $(o \in O)$
- $m(i) = Error$, otherwise.

Now we define a confidentiality policy.

Definition 6.3.3. *(Confidentiality policy) A confidentiality policy for the logic program* $p : I_1 \times \cdots \times I_n \times O \to R$ *is a function* $c : I_1 \times \cdots \times I_n \to J_1 \times \cdots \times J_n$*, where* $J_1 \subseteq I_1, \cdots, J_n \subseteq I_n$*.*

Informally, the sets $J_i, 1 \leq i \leq n$ corresponds to sets of inputs that may be revealed. The function c acts as a filter by bearing leakage of confidential inputs by seeing that the complements of J_i with respect to I_i represent the sets of inputs that must be kept confidential.

Example 6.10 Protection mechanism, confidentiality policy, secure mechanism-Part II

Let c be the confidentiality policy that bears leaking information about *ann* for example; thus, for $c : I \to J$, where $I = \{ann, billy, carl\}$ and $J = \{billy, carl\}$, $c(billy) = billy, c(carl) = carl$ and $c(ann)$ is *undefined*.

Now we define what we hear about a **secure** mechanism.

Definition 6.3.4. *(Secure mechanism) Let* $c : I_1 \times \cdots \times I_n \to J_1 \times \cdots \times J_n$ *be a confidentiality policy for a program* p*. Let* $m : I_1 \times \cdots \times I_n \to R \cup E$ *be a security mechanism for the same program* p*. Then the mechanism* m *is* ***secure*** *(i.e. confidential) if and only if there is a function* $m' : J_1 \times \cdots \times J_n \to R \cup E$ *such that, for all* $i_k \in I_k, 1 \leq k \leq n, m(i_1, \cdots, i_n) = m'(c(i_1, \cdots, i_n))$*.*

Example 6.11 Protection mechanism, confidentiality policy, secure mechanism-Part III

Let us check if this security mechanism is secure and this for the first two definitions of information flow previously presented:

	Information flow Success/ failure	Information flow Substitution answers
Query: $P \cup \{\leftarrow age(billy, A)\}$	*success*	$\theta = \{A \mapsto 27\}$
Protection mec: $m(billy)$	*success*	$\theta = \{A \mapsto 27\}$
Sec. mec: $m(c(billy))$	*success*	$\theta = \{A \mapsto 27\}$
Query: $P \cup \{\leftarrow age(diana, A)\}$	*failure*	$\theta = \{\}$
Protection mec: $m(diana)$	*error*	*error*
Sec. mec: $m(c(diana))$	*error*	*error*
Query: $P \cup \{\leftarrow age(ann, A)\}$	*success*	$\theta = \{56\}$
Protection mec: $m(ann)$	*success*	$\theta = \{56\}$
Sec. mec: $m(c(ann))$	*error*	*error*

In this example, we have showed for three queries the result of the protection mechanism, and checked if it is secure. Even if for the first two queries, the results show that this is case, the third query reported a discordance. Thus, the protection mechanism presented in this example is not secure. We will present later in this section a mechanism that is secure.

Despite the fact that a secure mechanism ensures that the policy is obeyed, it may disallow actions that do not violate it and thus be overly restrictive. Next we define the notion of precision which measures the degree of **overrestrictiveness**.

Definition 6.3.5. *Let m_1 and m_2 be two distinct protection mechanisms for the logic program p under the policy c. In the rest, o is an output position.*

Then m_1 is as precise as m_2 ($m_1 \succ m_2$) provided that, for all inputs $(i_1, \cdots, i_n)$, if $m_2(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$, then $m_1(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$.

We say that m_1 is more precise than m_2 ($m_1 \gg m_2$) if ($m_1 \succ m_2$) and there is an input $(i'_1, \cdots, i'_n)$ such that $m_1(i'_1, \cdots, i'_n) = p(i'_1, \cdots, i'_n, o)$ and $m_2(i'_1, \cdots, i'_n) \neq p(i'_1, \cdots, i'_n, o)$.

$m_1 \gg m_2$ implies that m_1 never gives a violation notice when m_2 does not. This implies that the utility of m_1 is at least as high as of m_2.

Lemma 6.3.1. *1. The relation $\succ$ is reflexive and transitive on the protection mechanisms for a given p and c.*

2. The relation $\gg$ is a strict ordering on the protection mechanisms for a given p and c.

Proof. 1. $\succ$ is reflexive and transitive on the protection mechanisms for a given p and c since,

- $\succ$ is *reflexive*: let m_1 be a protection mechanism for p and c, then $m_1 \succ m_1$ because for all the inputs $(i_1, \cdots, i_n)$, if $m_1(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$, then obviously $m_1(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$.

- $\succ$ is *transitive*: let m_1, m_2 and m_3 be three protection mechanisms for p and c such that $m_1 \succ m_2$ and $m_2 \succ m_3$. $m_2 \succ m_3$ means that for all inputs $(i_1, \cdots, i_n)$, if $m_3(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$, then $m_2(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$ and $m_1 \succ m_2$ means that for all inputs $(i_1, \cdots, i_n)$, if $m_2(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$, then $m_1(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$. Thus, for all inputs $(i_1, \cdots, i_n)$, if $m_3(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$, then $m_1(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$. This establishes that $m_1 \succ m_3$.

2. $\gg$ is a strict ordering on the protection mechanisms for a given p and c since,

   - $\gg$ is *irreflexive*: let m_1 be a protection mechanism for p and c, then $m_1 \not\gg m_1$ since there is no an input $(i'_1, \cdots, i'_n)$ such that $m_1(i'_1, \cdots, i'_n) = p(i'_1, \cdots, i'_n, o)$ and $m_1(i'_1, \cdots, i'_n) \neq p(i'_1, \cdots, i'_n, o)$.
   - $\gg$ is *asymmetric*: let m_1 and m_2 be two protection mechanisms for p and c such that $m_1 \gg m_2$. $m_1 \gg m_2$ implies that $m_1 \succ m_2$ and there is an input $(i'_1, \cdots, i'_n)$ such that $m_1(i'_1, \cdots, i'_n) = p(i'_1, \cdots, i'_n, o)$ and $m_2(i'_1, \cdots, i'_n) \neq p(i'_1, \cdots, i'_n, o)$. Thus, $m_2 \not\succ m_1$, since there will be the input $(i'_1, \cdots, i'_n)$ for which $m_1(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$, but $m_2(i_1, \cdots, i_n) \neq p(i_1, \cdots, i_n, o)$. Thus $\gg$ is asymmetric.
   - $\gg$ is *transitive*: let m_1, m_2 and m_3 be three protection mechanisms for p and c such that $m_1 \gg m_2$ and $m_2 \gg m_3$. $m_2 \gg m_3$ implies that $(m_2 \succ m_3)$ and there is an input $(i'_1, \cdots, i'_n)$ such that $m_2(i'_1, \cdots, i'_n) = p(i'_1, \cdots, i'_n, o)$ and $m_3(i'_1, \cdots, i'_n) \neq p(i'_1, \cdots, i'_n, o)$. $m_1 \gg m_2$ means that $(m_1 \succ m_2)$ and there is an input $(i"_1, \cdots, i"_n)$ such that $m_2(i"_1, \cdots, i"_n) = p(i"_1, \cdots, i"_n, o)$ and $m_3(i"_1, \cdots, i"_n) \neq p(i"_1, \cdots, i"_n, o)$. Thus, there is an input $(i'_1, \cdots, i'_n)$ such that $m_1(i'_1, \cdots, i'_n) = p(i'_1, \cdots, i'_n, o)$ and $m_3(i'_1, \cdots, i'_n) \neq p(i'_1, \cdots, i'_n, o)$. Seeing the fact that the relation $\succ$ is transitive, the relation $\gg$ is thus transitive.

□

Example 6.12 Comparison of protection mechanisms

For the same previous example, let $m_1 : I \rightarrow R \cup E$ be the following protection mechanism

- $m_1(i) = p(i, o)$ when $i \in U_{L(P)}$,
- $m_1(i) = Error$, otherwise.

and $m_2 : I \rightarrow R \cup E$ a protection mechanism that uses a counter *cn* on the number of queries already asked. *cn* is initialized to 1, and incremented by 1 on every query ran against the program.

- $m_2(i) = p(i, o)$ when $cn\%2 = 0$,
- $m_2(i) = Error$, otherwise.

Suppose that the user asks the following set of queries : $\{P \cup \{\leftarrow age(billy, A), P \cup \{\leftarrow age(ann, A), P \cup \{\leftarrow age(david, A)\}$. Next, we will be interested only with the second definition of flow, i.e. based on substitution answers, as it could be easily generalized to the other definitions.
The corresponding answers are as follows: $\theta = \{A \mapsto 27\}$, $\theta = \{A \mapsto 56\}$ and $\theta = \{\}$.
For the first protection mechanism, the answers would be respectively, $\theta = \{A \mapsto 27\}$, $\theta = \{A \mapsto 56\}$ and $error$, while for the second protection mechanism, the answers would be $\theta = \{A \mapsto 27\}$, $error$ and $\theta = \{\}$ respectively.

In this example, m_2 is not as precise as m_1, because $m_1(ann) = p(ann, o)$ and $m_2(ann) \neq p(ann, o)$. m_1 is not as precise as m_2, because $m_2(david) = p(david, o)$ and $m_1(david) \neq p(david, o)$. Thus, one cannot establish that $m_1 \gg m_2$ or $m_2 \gg m_1$. A question arises here, what about combining two protection mechanisms?

Combining two protection mechanisms form a new mechanism that is as precise as the two original ones.

Definition 6.3.6 (Union of protection mechanisms)**.** *Let m_1 and m_2 be protection mechanisms for the program p. Then their union $m_3 = m_1 \cup m_2$ is defined as*

$$m_3(i_1, \cdots, i_n) \begin{cases} = p(i_1, \cdots, i_n) \text{ when } m_1(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o) \text{ or} \\ m_2(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o) \\ = m_1(i_1, \cdots, i_n) \text{ otherwise.} \end{cases}$$

One can see that the previous definition is not symmetric, since $m_1 \cup m_2 \neq m_2 \cup m_1$.

Example 6.13 Union of protection mechanisms

For the same logic program in example 6.9, let $m_1 : I \to R \cup E$ be the following protection mechanism

- $m_1(i) = p(i, o)$ when $i \in U_{L(P)} \backslash \{ann\}$,
- $m_1(i) = Error$, otherwise.

and $m_2 : I \to R \cup E$ the following one:

- $m_2(i) = p(i, o)$ when $i \in U_{L(P)}$,
- $m_2(i) = Error$, otherwise.

Then,

$m_3(ann)$	$=$	$m_1(ann) \cup m_2(ann)$	$=$	$m_2(ann)$	$=$	$p(ann, o)$
$m_3(billy)$	$=$	$m_1(billy) \cup m_2(billy)$	$=$	$m_1(billy)$	$=$	$p(billy, o)$
$m_3(carl)$	$=$	$m_1(carl) \cup m_2(carl)$	$=$	$m_1(carl)$	$=$	$p(carl, o)$

Note that here $m_2 \gg m_1$.

From this definition and the definitions of secure and precise, we have:

Theorem 6.3.1 (Union of secure protection mechanisms)**.** *Let m_1 and m_2 be secure protection mechanisms for a program p and policy c. Then $m_1 \cup m_2$ is also a secure protection mechanism for p and c. Furthermore, $m_1 \cup m_2 \succ m_1$ and $m_1 \cup m_2 \succ m_2$.*

From secure protection mechanisms $m_1, m_2 \cdots$, one can define the secure protection mechanism $m^* = m_1 \cup m_2 \cup \cdots$ such that $m^* \succ m_1, m^* \succ m_2, \cdots$. Thus, we have the following generalization of the previous Theorem:

Theorem 6.3.2. *For any program p and security policy c, there exists a precise, secure mechanism m^* such that, for all secure mechanisms m associated with p and c, $m^* \succ m$.*

Proof. Let $m = \{m' | m'$ is a secure protection mechanism for p and $c\}$. Let m^* be $\cup_{n \in m} n$. Then by Theorem 6.3.1, $m^* \succ n$ for any secure protection mechanism m^*; hence, m^* is maximal. □

m^* is the mechanism that ensures security while minimizing error messages.

Example 6.14 Secure protection mechanism

We consider here a logic program P , with one input position and one output position, that contains the salary of some individuals in euros.
$salary(abby, 2500) \leftarrow$;
$salary(bob, 2500) \leftarrow$;
$salary(carla, 2400) \leftarrow$

The program P is represented by the function: $salary : I, O \rightarrow R$.

Let c be the confidentiality policy that bears leaking information about all input variables, namely $abby, bob$ and $carla$. Thus, for $c : I \rightarrow J$, where $I = \{abby, bob, carla\}$ and $J = \{\}$, $c(abby)$ is *undefined*, $c(bob)$ is *undefined* and $c(carla)$ is *undefined*.

A trivial protection mechanism for example, would be in this case to not answer any query. Formally, let m be the following function:
$m : I \rightarrow R \cup E$ for which:

- $m(i) = no\ answer$, where $i \in U_{L(P)}$.

Obviously, this protection mechanism is secure, but what about the existence of mechanisms that are both secure and precise in the sense that the mechanism ensures security while minimizing the number of denials of legitimate actions. For this, we will use the notion of level of a flow, previously presented in section 6.1, and the context exposed in section 6.2, to define our secure protection mechanism. In this example, we allow the observer to see the query sequences issued by the users and to have some a priori knowledge by retaining all previously returned

answers. **Note that in the query sequences visualized by the observer, the input parameters are kept hidden**.

Recall that for every query $\leftarrow p(i, o\}$ in a program p, we associate an **equivalence class**, denoted $\bar{o}$, and thus a cardinality. In the program p above, $abby \equiv bob$, since $\Theta(P \cup \{\leftarrow p(abby, o\}) = \Theta(P \cup \{\leftarrow p(bob, o\}) = \{o \mapsto 2500\}$, and consequently the $card(\overline{2500}) = 2$. One can see that $card(\overline{2400}) = 1$.

For our protection mechanism, we associate to each equivalent class of cardinality higher or equal than 1, a random number $\alpha > 1$.

As long as the queries are asked, the system counts the number of queries asked in each equivalence class. If the level associated to the query is equal to 1, the protection mechanism will respond by *no answer*. Also, when the number of queries corresponding to an equivalence class is equal to its associated random number α, the protection mechanism will respond by *no answer*. Otherwise, the protection mechanism will answer the query by giving its substitution answer sets. Formally, let m be the following protection mechanism:
$m : I \rightarrow R \cup E$ for which:

- $m(i) = no\ answer$, if for $P \cup \{\leftarrow p(i, o)\}$, $card(\bar{o}) = 1$,
- $m(i) = no\ answer$, if for $P \cup \{\leftarrow p(i, o)\}$, $nc_{\bar{o}} = \alpha_{\bar{o}}$,
- $m(i) = p(i, o)$, otherwise.

Above, $nc_{\bar{o}}$ is a counter corresponding to the number of queries already asked associated to the equivalence class of the goal $P \cup \{\leftarrow p(i, o)\}$.

Let the random numbers associated be as follows:
$\alpha_{\overline{2500}} = 1$, and
$\alpha_{\overline{2400}} = 1$.

As stated earlier, the observer can visualize query sequences with the input parameter hidden (in the next shown in red) and can have **some** a priori knowledge. An a priori knowledge that should not contain any information about the random numbers α, because an omniscient observer can easily violate the confidentiality policy, as shown next:
Suppose that the observer sees the following query sequences: $\{P \cup \{\leftarrow salary(abby, o)\}$, $P \cup \{\leftarrow salary(abby, o)\}, P \cup \{\leftarrow salary(bob, o)\}, P \cup \{\leftarrow salary(carla, o)\}\}$.

	Query 1: $P \cup \{\leftarrow salary(abby, o)\}$	Query 2: $P \cup \{\leftarrow salary(abby, o)\}$	Query 3: $P \cup \{\leftarrow salary(bob, o)\}$	Query 4: $P \cup \{\leftarrow salary(carla, o)\}$
Protection mechanism result	No answer	$\{o \mapsto 2500\}$	$\{o \mapsto 2500\}$	No answer
Why	Since $\alpha_{\overline{2500}} = 1$, the protection mechanism will not answer the **first** query of this equivalence class $\overline{2500}$.	This is the second query in the class $\overline{2500}$, the protection mechanism will reply by giving the substitution answer.	The is the third query in the class $\overline{2500}$, the protection mechanism will reply by giving the substitution answer.	Since $\alpha_{\overline{2400}} = 1$, the protection mechanism will not answer the **first** query of this equivalence class $\overline{2400}$.
Observer inference	As it is the first query, the observer cannot deduce anything about the input parameter. In fact, the input parameter could be any element of $\{abby, bob, carla\}$.	By giving the substitution answer of a query belonging to the equivalence class $\overline{2500}$, the observer is now sure that the **first query** concerns the same class $\overline{2500}$.	The observer learns nothing more from the third query.	As the observer knew that a previously asked query concerned the equivalence class $\overline{2500}$ and returned *no answer*, the current returned result *no answer* necessarily concerns the equivalence class $\overline{2400}$, and thus, **the observer is able to state that the hidden input parameter for this query is *carla*.**

Thus, the random numbers α associated to each equivalence class should not be disclosed to the observer.

Let us now show that the protection mechanism is secure. For this, we need to define what is meant by '**the observer can infer the exact value of the hidden input parameter**'. For this, we associate to the query sequences issued by the user, a vector of the observed substitution answers. Formally, for the query sequence $Q = \{P \cup \{\leftarrow p(a_1, o_1\}, P \cup \{\leftarrow p(a_2, o_2\}, \cdots, P \cup \{\leftarrow p(a_n, o_n\}\}$, we associate the observed returned results in terms of substitution answers $\Theta = (\theta_1, \theta_2, \cdots, \theta_n)$.

We say that the observer can guess the exact value of a hidden input parameter a_i, if for the corresponding θ_i, and in **all** the query vectors $\{P \cup \{\leftarrow p(a_1, o_1\}, P \cup \{\leftarrow p(a_2, o_2\}, \cdots, P \cup \{\leftarrow p(a_n, o_n\}\}$, a_i have the same value.

Let us show now that our mechanism is secure.
Suppose that the logic program is composed by at least two facts (the case where the logic program is composed by 1 fact is trivial, as the hidden input parameter is unique). Suppose that there exists a substitution answer θ_i for which a_i have the same value in **all** the query vectors $\{P \cup \{\leftarrow p(a_1, o_1\}, P \cup \{\leftarrow p(a_2, o_2\}, \cdots, P \cup \{\leftarrow p(a_n, o_n\}\}$, that is $\theta(p(a_i, o_i)) = \theta_i$.
Thus, for this θ_i, there is a unique associated input parameter a_i. Furthermore, there is a unique fact of the form $p(a_i, b) \leftarrow$ with $\theta(p(a_i, o_i)) = \{o_i \mapsto b\}$.
But, according to the protection mechanism; for each equivalence class of cardinal 1, the associated returned answer by the mechanism is *no answer*. So, in this case,

$\theta_i = \epsilon$ (ϵ is used to note the *no answer* returned value).
Thus, seeing that for $\theta_i = \epsilon$, there is **one** associated input parameter a_i, this means that the **logic program is composed by one fact only**, because according to the mechanism, a *no answer* should be returned to **one** of the queries in each equivalence class (or to all the queries for equivalence classes with cardinality equal to 1). Note that for $\theta_i = \epsilon$, there should be a number of distinct a_i equal to the number of different equivalence classes in the program. This contradicts the fact that the logic program is composed by at least two facts.

6.4 Summary

In this chapter, we refined the notion of information flow for Datalog logic programs, by proposing the notion of level of indistinguishability of the flow. We proposed an equivalence relation between the elements of the Herbrand universe relatively for a Datalog logic program P. We showed that the notion of indistinguishability proposed, coincides with the one presented earlier in chapter 4 since practically it suffices to find two indistinguishable classes to state that an information flow passes from x to y in the goal $P \cup \{G(x, y)\}$. Based on the notion of equivalence classes, we proposed also the definition of the level of an information flow. Algorithms were proposed to calculate this level, and this for the first two definitions of information flow, namely, based on success/failure and substitution answers. We then discussed the specifications of the flow and give an example to emphasize the fact that the result returned by the query can convey confidential information.

To control this, we focused on the notion of inference control and we proposed definitions of protection mechanisms, secure mechanisms, precise mechanisms and confidentiality policies. We ended by giving an example of a secure and precise protection mechanism that prohibits any undesirable inferences meanwhile and minimizes the number of denials of legitimate actions.

Chapter 7

Evaluation and Future Work

In the final chapter, we summarize the work presented in this thesis, and outline possible directions for further research.

7.1 Summary of the Thesis and Conclusion

Our main objective in this thesis was to provide a model of interaction in deductive databases that preserves the database security while minimizing the number of denials answers. We went on building a secure and precise mechanism for such databases. Obviously, as the role of a deductive database is to make deductions based on rules and facts stored in it, we headed toward the use of first-order logic programming, typically a language like Datalog, that is used to specify facts, rules and queries.

We started by presenting, in chapter 2, the basic components for data security. As we are mostly interested in confidentiality, it was natural to explore security policies that deal with this aspect. We reviewed two extensions of confidentiality policies, namely, the noninterference and the nondeducibility policies because they tackle the confidentiality issue by observing execution traces in the system. We then turned on how to implement security in systems. We mentioned three main methods, namely, access control, information flow and inference control, skipping willingly cryptography (as it is out of the scope of this thesis). While access control mechanism describes the conditions under which a system is secure, information flow checking mechanisms ensure that information flows in the system comply with its security policy.

In chapter 3, we looked at the relevant background material needed in our thesis, the first-order logic programming. We reviewed the notions of substitutions, unification, SLD-refutation and SLD-trees, and showed the limitations of such resolutions since searching for a proof of a goal can be incomplete. To remedy this problem, we reviewed loop checking techniques which modifies the computation mechanism by adding a capability of pruning. A pruning based on an occurrence of some similar subgoal in the SLD-derivation. We ended by an overview of control flow and dependence analysis in logic programming, highlighting the fact that these analyzes do not substitute the notion of information flow in security systems.

To fulfill our aim, and as the notion of information flow in security systems was never addressed in logic programming, we proposed, in chapter 4, three definitions of what could possibly be an information flow in logic programming. These definitions, based on traces, correspond to what can be observed by a user when she runs a query on a logic program. These definitions were based respectively, on success/failure, substitution answers and bisimulation. We explored the links between these definitions and prove the non-transitivity of the flow. To decide of the existence of a flow relatively for a logic program and a goal, we gave complexity results and showed undecidability of the flow for general logic programs and decidability for other restricted types of logic programs. The problem of deciding of the existence of the flow in Datalog programs relatively for the definitions of flow based on success/failure and substitutions answers is EXPTIME-complete.

The problem became in $\Delta 2P$ for binary hierarchical Datalog programs and this relatively for the first definition of flow based on success/failure. We devoted chapter 5 to goals bisimulation. We showed that deciding if two goals are bisimilar is undecidable in the general setting, and it became decidable in 2EXPTIME for hierarchical and restricted logic programs.

Finally in chapter 6, using all the results obtained, we tackled our objective. We first extended the notion of flow to the notion of indistinguishability. Definition of the level of logic goals was proposed. We reformulated the concepts of protection mechanisms, secure mechanisms, precise mechanisms and confidentiality policies in the logic setting. We finally ended by giving (we proved it formally too) a secure protection mechanism for deductive databases using the previously exposed notions.

7.2 Future Work

Future works can be directed in three main research directions. First toward deciding of the existence of the flow in other types of logic programs and deciding of bisimulation in pure Datalog programs, second toward the implementation of the information flow in a logic programming framework, and third toward embedding our security mechanism framework in real-time databases.

7.2.1 On more and more Formal Works

Following the results already obtained in this thesis, one can think about finding and exploring other types of logic programs for which the question of the existence of the flow could be decidable too. Meanwhile, we are still investigating the decidability and the complexity of the existence of the flow relatively to bisimulation in Datalog programs without considering loop checking techniques.

7.2.2 On Implementation

We have presented in chapters 4 and 5 several definitions of what could be an information flow in logic programs. We have presented algorithms too on how to decide of this flows and this in different settings. Implementing these algorithms constitutes a way to justify our results. Although this implementation is relatively easy for the first two proposed definitions of the flow, it is more difficult in the case of bisimulation. Recall that to decide if two goals are bisimilar, we need to keep track in the memory of all the encountered resolvents in the SLD-derivation, even if the branches contained in the SLD-tree are not infinite. Even if the implementation is quite possible, it is not too interesting seeing the amount of space required to decide of the existence of the flow. We think that a better implementation could be achieved using the techniques exposed in section 3.2.9 or by first conducting a deep investigation on how to relax the conditions on how to decide the bisimulation of two goals while maintaining the soundness and completeness of the algorithms.

7.2.3 On Real-Time Databases

In chapter 6, we gave a secure and precise security mechanism for deductive databases. However, as real-time databases are gaining more and more attention, it is tempting to see if we can embed our security mechanism framework in these databases, and how using our framework can enforce its security policies, or more to bring changes on our framework in order to balance real-time requirements with security.

Bibliography

[1] K. R. Apt, R. N. Bol, and J. W. Klop. On the safe termination of prolog programs. Technical report, Austin, TX, USA, 1989. [cited at p. 51, 175]

[2] K. R. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982. [cited at p. 46]

[3] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. *The MITRE Corporation Bedford MA Technical Report M74244*, 1(M74-244):42, 1973. [cited at p. 77, 179]

[4] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. *Proc 10*, 1(MTR-2997):118121, 1976. [cited at p. 23, 169]

[5] P. Besnard. On infinite loops in logic programming. Research Report RR-1096, INRIA, 1989. [cited at p. 51, 53, 67, 175]

[6] K. J. Biba. Integrity considerations for secure computer systems. *Proceedings of the 4th annual symposium on Computer architecture*, 5(7):135–140, 1977. [cited at p. 26, 170]

[7] J. Biskup. *Security in Computing Systems: Challenges, Approaches and Solutions*. Springer Publishing Company, Incorporated, 1st edition, 2008. [cited at p. 37]

[8] R. N. Bol. Towards more efficient loop checks. In *Proceedings of the 1990 North American conference on Logic programming*, pages 465–479, Cambridge, MA, USA, 1990. MIT Press. [cited at p. 51, 69, 175, 177]

[9] R. N. Bol, K. R. Apt, and J. W. Klop. An analysis of loop checking mechanisms for logic programs. *Theor. Comput. Sci.*, 86(1):35–79, August 1991. [cited at p. 49, 51, 52, 53, 62, 64, 65, 67, 68, 117, 120, 121, 122, 175, 177, 185]

[10] P. A. Bonatti, S. Kraus, and V. S. Subrahmanian. Foundations of secure deductive databases. *Knowledge and Data Engineering, IEEE Transactions on*, 7(3):406 –422, jun 1995. [cited at p. 8]

[11] J. Boye, J. Paakki, and J. Maluszynski. *Synthesis of Directionality Information for Functional Logic Programs*, pages 165–177. Springer LNCS 724, 1993. [cited at p. 71, 72]

[12] D. R. Brough and A. Walker. Some practical properties of logic programming interpreters. In *FGCS*, pages 149–156, 1984. [cited at p. 51, 53, 54, 56, 58, 175, 176]

[13] M. Bruynooghe and G. Janssens. *An instance of abstract interpretation integrating type and mode inferencing*, pages 669–683. The MIT Press, Cambridge, MA, 1988. [cited at p. 70, 177]

[14] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, January 1977. [cited at p. 99, 182]

[15] S. Ceri, G. Gottlob, and G. Wiederhold. Interfacing relational databases and prolog efficiently. *Proceedings of the 2nd International Conference on Expert Database Systems*, pages 141–153, 1986. [cited at p. 8]

[16] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981. [cited at p. 93]

[17] J-H. Chang and A. M. Despain. *Semi intelligent backtracking of Prolog based on static data dependency analysis*, pages 10–21. 1985. [cited at p. 70, 73, 177]

[18] J-H. Chang, A. M. Despain, and D. DeGroot. *AND-Parallelism of Logic Programs Based on a Static Data Dependency Analysis*, pages 218–226. 1985. [cited at p. 70, 73, 177]

[19] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987. [cited at p. 26, 170]

[20] K. L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977. [cited at p. 48, 174]

[21] M. A. Covington. Eliminating unwanted loops in prolog. *SIGPLAN Not.*, 20(1):20–26, January 1985. [cited at p. 51, 53, 59, 175, 176]

[22] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, 1988. [cited at p. 70, 177]

[23] S. K. Debray and D. S. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, 1989. [cited at p. 70, 177]

[24] D. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982. [cited at p. 28, 34, 77, 170]

[25] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977. [cited at p. 28, 170]

[26] P. J. Denning. Third generation computer systems. *ACM Comput. Surv.*, 3(4):175–216, December 1971. [cited at p. 27, 170]

[27] P. Deransart and J. Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2(2):119–155, 1985. [cited at p. 70, 177]

[28] P. Devienne, P. Lebègue, A. Parrain, J-C. Routier, and J. Würtz. Smallest Horn clause programs. *The Journal of Logic programming*, 27:227–267, 1996. [cited at p. 94, 106, 180, 183]

[29] P. Devienne, P. Lebègue, and J-C. Routier. Halting problem of one binary Horn clause is undecidable. *STACS 93*, 665:48–57, 1993. [cited at p. 94, 181]

[30] J. S. Fenton. Memoryless subsystems. *Computer Journal*, 17(2):143–147, 1974. [cited at p. 33]

[31] M. Gabbrielli, G. Levi, and M.C. Meo. Observable behaviors and equivalences of logic programs. *Information and Computation*, 122(1):1 – 29, 1995. [cited at p. 103]

[32] H. Gallaire, J. Minker, and J-M. Nicolas. Logic and databases: A deductive approach. *ACM Comput. Surv.*, 16(2):153–185, June 1984. [cited at p. 8]

[33] A. V. Gelder. Efficient loop detection in prolog using the tortoise-and-hare technique. *The Journal of Logic Programming*, 4(1):23 – 31, 1987. [cited at p. 51, 69, 175, 177]

[34] J. Goguen and J. Meseguer. Security policies and security models. *Security and Privacy, IEEE Symposium on*, 0:11, 1982. [cited at p. 77, 179]

[35] J. A. Goguen and J. Meseguer. *Security Policies and Security Models*, volume pages, pages 11–20. IEEE, 1982. [cited at p. 25, 169]

[36] G. S. Graham and P. J. Denning. Protection: principles and practice. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, 1972. ACM. [cited at p. 27, 170]

[37] J. Harland. On normal forms and equivalence for logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 146–160. ALP, MIT Press, 1992. [cited at p. 103]

[38] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer Berlin - Heidelberg, 1980. [cited at p. 103]

[39] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. [cited at p. 103]

[40] C. J. Hogger. *Essentials of logic programming.* Oxford University Press, Inc., New York, NY, USA, 1990. [cited at p. 42, 46]

[41] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986. [cited at p. 96, 181]

[42] M. Jarke, J. Clifford, and Y. Vassiliou. An optimizing prolog front-end to a relational query system. *SIGMOD Rec.*, 14(2):296–306, June 1984. [cited at p. 8]

[43] D. S. Johnson. Handbook of theoretical computer science (vol. a). chapter A catalog of complexity classes, pages 67–161. MIT Press, Cambridge, MA, USA, 1990. [cited at p. 92]

[44] T. Kawamura and T. Kanamori. Preservation of stronger equivalence in unfold/fold logic program transformation. *Theoretical Computer Science*, 75:139–156, 1990. [cited at p. 100, 182]

[45] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, January 1974. [cited at p. 27, 170]

[46] L. J. LaPadula and D. E. Bell. Secure computer systems: A mathematical model. *Journal of Computer Security*, 4(May 1973):239263, 1973. [cited at p. 23, 169]

[47] D. Li. *A PROLOG database system.* Electronic & electrical engineering research studies. Research Studies Press New York, Letchworth, Hertfordshire, England, 1984. [cited at p. 8]

[48] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Trans. Comput. Logic*, 2(4):526–541, October 2001. [cited at p. 103]

[49] S. B. Lipner. Non-discretionary controls for commercial applications. In *IEEE Symposium on Security and Privacy*, pages 2–10, 1982. [cited at p. 26, 170]

[50] J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987. [cited at p. 42, 46, 48]

[51] G. Luo, G. V. Bochmann, B. Sarikaya, and M. Boyer. Control-flow based testing of prolog programs. In *Software Reliability Engineering, 1992. Proceedings., Third International Symposium on*, pages 104 –113, oct 1992. [cited at p. 70]

[52] M. J. Maher. Foundations of deductive databases and logic programming. chapter Equivalences of logic programs, pages 627–658. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. [cited at p. 103]

[53] C. S. Mellish. Some global optimizations for a prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985. [cited at p. 70, 177]

[54] C. S. Mellish. *Abstract Interpretation of Prolog Programs*, pages 181–197. Number 484. Ellis Horwood, 1987. [cited at p. 70, 177]

[55] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267 – 310, 1983. [cited at p. 103]

[56] C. M. Papadimitriou. *Computational complexity.* Addison-Wesley, Reading, Massachusetts, 1994. [cited at p. 92]

[57] D. Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Berlin / Heidelberg, 1981. 10.1007/BFb0017309. [cited at p. 103]

[58] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):181–186, 1976. [cited at p. 43, 173]

[59] D. Poole and R. Goebel. On eliminating loops in prolog. *SIGPLAN Not.*, 20(8):38–40, August 1985. [cited at p. 50, 175]

[60] D. Sahlin. Mixtus: An automatic partial evaluator for full prolog. *New Generation Computing*, 12(1):7–51, March 1993. [cited at p. 51, 175]

[61] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, May 2009. [cited at p. 103]

[62] Y. Shen. An extended variant of atoms loop check for positive logic programs. *New Generation Computing*, 15(2):187–203, June 1997. [cited at p. 51, 52, 175]

[63] Y. Shen, L. Yuan, and J. You. Loop checks for logic programs with functions. *Theoretical Computer Science*, 266(12):441 – 461, 2001. [cited at p. 52]

[64] J. C. Shepherdson. Negation as failure ii. *The Journal of Logic programming*, 2(3):185–202, 1985. [cited at p. 48, 174]

[65] D. E. Smith, M. R. Genesereth, and M. L. Ginsberg. Controlling recursive inference. *Artif. Intell.*, 30(3):343–390, December 1986. [cited at p. 50, 51, 175]

[66] R. E. Smith. *Constructing a High Assurance Mail Guard*. 1994. [cited at p. 35]

[67] M. Spivey. *An introduction to logic programming through Prolog*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. [cited at p. 47]

[68] D. Sutherland. *A Model of Information*, pages 175–183. IEEE, 1986. [cited at p. 25, 169]

[69] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *In S.-. Trnlund, editor, Proceedings of The Second International Conference on Logic Programming*, pages 127–139, 1984. [cited at p. 99, 100, 102, 182]

[70] P. Van Roy, B. Demoen, and Y. D. Willems. *Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection and Determinism*, volume 250, pages 111–125. Springer LNCS 250, 1987. [cited at p. 70, 177]

[71] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 137–146. ACM, 1982. [cited at p. 96, 181]

[72] L. Vielle. Recursive query processing: the power of logic. *Theoretical Computer Science*, 69(1):1 – 53, 1989. [cited at p. 51, 52, 175]

[73] R. Warren, M. V. Hermenegildo, and S. K. Debray. On the practicality of global flow analysis of logic programs. In *ICLP/SLP*, pages 684–699, 1988. [cited at p. 70, 73, 177]

List of Figures

List of Tables

List of Examples

Index

Résumé en Français

Chapitre 1: Objectifs et résultats

Dans ce chapitre, nous commençons par situer notre travail de recherche puis nous résumons le contenu de cette thèse en présentant la méthodologie adoptée consistant à appliquer la notion du flux de l'information pour les systèmes de sécurité en programmation logique.

Motivation

La théorie des BD déductives a largement été abordée par la communauté scientifique qui sy est intéressé, notamment grâce à lexistence de nombreux systèmes supportant la notion de requête, de raisonnement et, au développement dapplications base de langages logiques. Dans les années 70, les travaux se sont penchés sur l'établissement des fondements théoriques de ces bases de données. C'est dans les années 80 que les recherches se sont axées sur le développement de deux systèmes très proches des bases de données déductives, à savoir les bases de données relationnelles et la programmation logique. Plusieurs travaux ont mis le point sur le couplage de systèmes logiques tels que Prolog avec des bases de données relationnelles. Ce progrès est démontré par la réalisation de prototypes de systèmes offrant un niveau de généralité, de performance et de robustesse permettant le développement d'applications complexes. Ainsi, plusieurs domaines dapplication intégrant efficacement ces systèmes et dépassant celui des BD traditionnelles ont été identifiés. Une de ces applications est la nécessité de maintenir la sécurité dans ces bases de données. Dans les bases de données où différents utilisateurs sont autorisés à accéder à différents éléments de données, des mécanismes doivent être mis en place pour que tous les utilisateurs puissent accéder aux données auxquelles ils possèdent une autorisation daccès sans pour autant violer la politique de sécurité. Par exemple, des modèles d'interaction entre les utilisateurs et la base de données, proposés dans la littérature, permettent à la base de données de mentir à l'utilisateur. Ainsi, notre objectif dans cette thèse est de fournir un modèle d'interaction dans les bases de données déductives qui préserve la sécurité des base de données tout en minimisant le nombre de refus de réponses.

Méthodologie

Dans le but d'atteindre notre but, nous avons utilisé la programmation logique pour représenter les bases de données déductives. Nous avons adapté la notion du flux de linformation bien connue en programmation impérative à la programmation logique. Une difficulté provient du fait que les variables dans les programmes logiques représentent une entité et non un emplacement de stockage dans la mémoire. Pour cela, nous étions obligé de proposer de nouvelles définitions qui s'adaptent à la programmation logique. Une fois ces définitions proposées, nous nous sommes posé le problème de la décidabilité de lexistence de ces flux. Malheureusement, cette question est indécidable dans le cas général. Ceci nous a poussé à prendre en considération dautres types

de langages logiques pour lesquels la question de lexistence de flux est décidable bien que, dans certains cas, nous soyons dans lincapacité daffirmer si deux buts logiques sont bisimilaires du fait que les arbres de résolution correspondants sont infinis. Nous nous sommes servi des techniques de détection de boucles pour décider si deux buts sont bisimilaires. Nous avons également étudié cette question pour plusieurs types de programmes logiques, partant du fait que le problème est indécidable en général. Finalement, nous étions capable de présenter un mécanisme sûr et certain pour protéger les bases de données déductives, tout en proposant des définitions de ce qu'est une politique de confidentialité et un mécanisme sûr et certain.

Résultat

Dans ce qui suit, nous dressons un récapitulatif des principales contributions de cette thèse :

- **Flux de l'information en programmation logique** : nous proposeons trois définitions du flux de l'information. Ces définitions correspondent à ce qui peut être observé par l'utilisateur quand un but $\leftarrow G(x, y)$ est posé à un programme logique P. Nous considèrons tout d'abord que l'utilisateur peut voir si ses requêtes réussissent ou échouent. Nous dirons que l'information passe de x vers y dans G s'il existe deux constantes a, b telles que $P \cup \{\leftarrow G(a, y)\}$ réussit alors que $P \cup \{\leftarrow G(b, y)\}$ échoue. Ensuite, nous supposons que l'utilisateur a accès à l'ensemble des substitutions réponses des requêtes. Dans ce cas, un flux existe de x vers y dans G s'il existe deux constantes a, b tel que les substitutions réponses de $P \cup \{\leftarrow G(a, y)\}$ et $P \cup \{\leftarrow G(b, y)\}$ sont différentes. Dernièrement, nous supposons que l'utilisateur observe les arbres de résolution des buts. Si les deux arbres de résolution de $P \cup \{\leftarrow G(a, y)\}$ et $P \cup \{\leftarrow G(a, y)\}$ peuvent être distingués par l'utilisateur, alors nous dirons que l'information passe de x vers y dans G.
- **Non décidabilité/décidabilité du flux en programmation logique** : Pour un programme logique P et un but à deux variables $\leftarrow G(x, y)$, il est indécidable, dans le cas général, de dire si un flux passe de x vers y et ceci pour les trois définitions du flux. Le problème devient dans EXPTIME pour les programmes Datalog (pour la définition du flux basée sur réussite/échec et substitutions réponses) et dans $\Delta 2P$ pour les programmes Datalog hiérarchiques (pour la définition du flux basée sur réussite/échec).
- **Bisimulation de buts** : Nous avons prouvé que décider si deux buts sont bisimilaires est indécidable pour les programmes Prolog et que ce problème devient décidable en 2EXPTIME pour les programmes Datalog hiérarchiques et restricted.
- **Contrôle d'inférence préventive pour les bases de données déductives** : nous proposons un mécanisme sûr et certain pour les bases de données déductives basé sur la notion du flux de l'information et des niveaux.

Certains documents de cette thèse ont été publiés dans des conférences et des revues. La notion de flux d'informations dans la programmation logique a été décrite dans [JIAF 2011], et le résultat sur la bisimulation a été publié dans [JeLIA 2012].

Organisation de la thèse

Le reste de la thèse est organisé comme suit:
Dans le chapitre 2, nous présentons les fondements de la sécurité des données. Nous explorons les politiques de sécurité en évoquant des politiques de confidentialité, non-interférence, non-déducibilité et d'intégrité. Nous décrivons quelques implémentations de mécanismes de sécurité à savoir le contrôle d'accès, le flux de l'information et le contrôle d'inférence.
Dans le chapitre 3, nous passons en revue la programmation logique du premier ordre. Nous présentons le mécanisme de détection de boucles ainsi que quelques techniques correctes et complètes de ce mécanisme. Nous terminons ce chapitre en introduisant le contrôle de flux et lanalyse de dpendance en programmation logique.

Dans le chapitre 4, nous donnons trois définitions du flux en programmation logique basées respectivement sur la réussite/échec, les substitutions réponses et la bisimulation. Nous explorons les liens entre les différentes définitions et nous prouvons que le fux est non transitif. Nous démontrons l'indécidabilité de l'exitence du flux pour le cas général des programmes logiques et donnons des résultats de complexité et de décidabilité pour d'autres types de programmes.
Dans le chapitre 5, nous prouvons l'indécidabilité de la bisimulation des buts logiques pour les programmes Prolog. Ensuite, nous montrons la décidabilité de la bisimulation des buts pour les programmes Datalog hiérarchiques et *restricted*.
Dans le chapitre 6, nous proposons la notion de niveaux du flux. Nous appliquons cette notion pour prévenir les inférences illicites dans les systèmes d'information tout en présentant un mécanisme de sécurité sûr et certain.
Dans le chapitre 7, nous concluons avec un résumé de cette thèse et nous présentons quelques travaux futurs.

Chapitre 2: Sécurité des données

La sécurité des données est la science et l'étude des méthodes de protection des données dans les systèmes informatiques contre la divulgation non autorisée et la modification. L'un des aspects de la sécurité des données est le contrôle du flux de l'information dans le système. Une politique de contrôle de flux d'information devrait décrire les règles qui régissent la diffusion de l'information. Ces règles sont nécessaires pour empêcher les programmes de divulguer de données confidentielles. Nous introduirons tout d'abord, les notions de bases des sécurité des données, à savoir les conditions de confidentialité, d'intégrité et de disponibilité. Nous discutons ensuite des politiques de sécurité (les politiques de confidentialité et d'intégrité) qui identifient les menaces et définissent les exigences pour maintenir un système sécurisé. L'implémentation de la sécurité peut être réalisée par cryptographie et/ou par partage de droits et d'informations. Nous nous intéressons plutôt au deuxième aspect en présentant les mécanismes de contrôle d'accès et de contrôle d'inférence. Nous consacrons une partie pour les mécanismes de vérification du flux de l'information dans la programmation impérative. Nous discutons ainsi des mécanismes qui sont basés sur la compilation et l'exécution pour déterminer si un flux d'information dans un programme impératif peut violer une politique donnée.

Composants de base

La sécurité des données est basée sur trois aspects principaux: la confidentialité, l'intégrité et la disponibilité:

- **Confidentialité** représente la dissimulation de l'information. L'importance de cet aspect découle de la nécessité de garder le secret des informations dans des domaines sensibles tels que le gouvernement et l'industrie. Par exemple, tous les établissements gardent les dossiers personnels secrets.

  Différents mécanismes supportent la confidentialité:

  - Mécanismes de contrôle d'accès comme la cryptographie.
  - Mécanismes dépendants du système: Ces mécanismes empêchent les processus d'accéder illicitement à l'information.

- **Intégrité** empêche toute modification non autorisée des données ou des ressources.

  Principalement, deux mécanismes soutiennent l'intégrité:

  - **Mécanismes de prévention**: ils cherchent à maintenir l'intégrité des données en:
    - **bloquant toute tentative non autorisée de modifier les données** ou
    - **bloquant toute tentative de modifier les données d'une manière non autorisée**.
  - **Mécanismes de détection**: ils signalent simplement que l'intégrité des données n'est plus digne de confiance.

- **Disponibilité** renvoie à la capacité d'utiliser l'information ou la ressource désirée.

Formellement, pour dire ce qui est et ce qui n'est pas autorisé en sécurité des données, les politiques de sécurité ont été introduites.

Politiques de sécurité

Une politique de sécurité définit ce que signifie **sûr** pour un système. Les politiques de sécurité peuvent être informelles ou très mathématiques. En fait, les politiques peuvent être présentées mathématiquement, comme une liste d'états permis (sûrs) et d'états interdits (non sûrs). En pratique, les politiques sont rarement aussi précises, elles sont décrites en langage naturel.

Fondements des politiques de sécurité

Nous considèrons un système informatique comme un **automate à états finis** avec un ensemble de fonctions de transition qui permettent de changer d'état. Alors:
Politique de sécurité: Une politique de sécurité est une déclaration (généralement écrite en langage naturel) qui partitionne les états du système en un ensemble d'états **sûrs / autorisés** et un ensemble d'états **non sûrs / non autorisés**.

Une politique de sécurité définit le contexte dans lequel nous pouvons définir un système sécurisé. Plus précisément:
Système sûr: Un système sûr est un système qui est au départ dans un état autorisé et qui ne peut pas se retrouver dans un état non autorisé au bout d'une ou plusieurs transitions.

Violation de la sécurité: Une violation de la sécurité se produit lorsqu'un système entre dans un état non autorisé.

Une politique de sécurité prend en compte tous les aspects pertinents de la confidentialité, l'intégrité et la disponibilité. Quant à la confidentialité, elle recense les états dans lesquels des informations confidentielles sont divulguées. Cela inclut la transmission illicite de l'information, appelé flux d'informations. En ce qui concerne l'intégrité, une politique de sécurité identifie les moyens autorisés dans lesquels les informations peuvent être modifiées, et les entités autorisées à les modifier. En ce qui concerne la disponibilité, une politique de sécurité décrit les services qui doivent être disponibles.

Une politique de sécurité définit les informations qui doivent être protégées. Elle a en fait une forme non procédurale. Par exemple, une politique de sécurité peut indiquer que l'utilisateur n'a pas à obtenir des informations "top secret". En revanche, un mécanisme de protection définit comment l'information doit être protégée, il a une forme procédurale. Par exemple, un mécanisme de protection peut vérifier chaque opération effectuée par l'utilisateur.

Dans le reste de cette section, nous allons présenter brièvement les politiques de confidentialité, d'intégrité, de la non-interférence et de non-déducibilité.

Politiques de confidentialité

L'objectif principal d'une politique de confidentialité est d'empêcher la divulgation non autorisée d'informations confidentielles. Les politiques de confidentialité sont aussi appelées **politiques du flux de l'information**. Parmi la multitude de politiques de confidentialité, nous notons:

- Le **modèle de Bell-LaPadula**: Le type le plus simple de classification de confidentialité est un ensemble de **classes de securité** disposées suivant un ordre linéaire total. Ces niveaux représentent des niveaux de sensibilité. Plus le niveau est haut, plus l'information est sensible. Le but du modèle de Bell-LaPadula [4, 46] est d'empêcher l'accès en lecture par des utilisateurs aux objets d'un niveau de sécurité plus élevé.
- **Non-interférence**: Un système est sécurisé si des groupes de sujets ne peuvent pas interférer les uns avec les autres. Goguen et Meseguer [35] ont utilisé cette approche pour définir les politiques de sécurité. Une politique de sécurité, fondée sur la non-interférence, décrira les états dans lesquels des interférences interdites ne se produisent pas.
- **Nondéducibilité**: Goguen et Meseguer [35] ont caractérisé la sécurité en termes de transitions d'états. Si les transitions d'état causées par des commandes de haut niveau interfèrent avec une séquence de transitions causées par des commandes de bas niveau, le système n'est pas non-interférence-sûr. Notons que, étant donné un ensemble de sorties de bas niveau, aucun sujet de bas niveau ne devrait être en mesure de déduire quoi que ce soit sur les sorties de haut niveau. Sutherland [68] a réexaminé cette question de la façon suivante: il considère un système comme une "boîte noire" avec deux ensembles

d'entrées, un classifié haut et l'autre classifié bas. La boîte dispose également de deux sorties, une fois encore, une classifiée haute et l'autre basse. Si un observateur, autorisé uniquement comme bas, peut observer une séquence d'entrées basses et de sorties basses, et de déduire des informations sur les entrées ou les sorties classifiées hautes, l'information a fui alors de haut vers bas.

Politiques d'intégrité

Comme la plupart des sociétés et des entreprises se préoccupent davantage de la précision des données que de la divulgation, les politiques d'intégrité se penchent polutôt sur l'intégrité de l'information que sur la confidentialité. Trois politiques d'intégrité ont été proposées dans la littérature, à savoir les modèles d'intégrité de Biba [6], de Lipner [49] et de Clark-Wilson [19]. Les politiques d'intégrité tiennent compte des concepts tels que la séparation des tâches, la séparation des fonctions et d'audit, de concepts qui sont au-delà des fonctions des politiques de confidentialité.

Implémentation

L'implémenation de la sécurité peut être réalisée par plusieurs moyens: mécanismes basés sur la cryptographie, et mécanismes basés sur le partage des droits et d'information. Dans le reste de cette section, nous nous intéressons uniquement aux second mécanisme. Nous parlons ainsi brièvement des mécanismes de contrôle d'accès, du flux de l'information et des mécanismes de contrôle d'inférence.

Contrôle d'accès

La matrice de contrôle d'accès [26, 36, 45] est l'outil le plus précis qui peut décrire l'état actuel de protection d'un système. Elle caractérise les droits de chaque sujet (entité active, comme un processus) par rapport à toute autre entité.

En bref, l'ensemble des entités protégées est appelé l'ensemble d'objets O. L'ensemble de sujets S est l'ensemble des objets actifs, tels que les processus et les utilisateurs. Dans le modèle de matrice de contrôle d'accès, la relation entre ces entités est capturée par une matrice A avec des droits tirés d'un ensemble de droits R dans chaque entrée $a[s, o]$, où $s \in S$, $o \in O$, et $a[s, o] \subseteq R$.

Le sujet s a l'ensemble des droits $a[s, o]$ sur l'objet o. L'ensemble d'états de protection du système est réprésenté par le triplet (S, O, A). Il convient de noter que l'outil de la matrice de contrôle d'accès en sécurité informatique n'est pas utilisé dans la pratique, en raison des exigences d'espace (la plupart des systèmes ont des milliers d'objets et pourrait avoir des milliers de sujets aussi).

Flux de l'information

Une politique de sécurité régule les flux d'information dans un système. C'est l'une des tâches du système de s'assurer que ces flux d'information ne violent pas les contraintes de la politique de sécurité. Dans cette section, nous passerons en revue les principaux mécanismes proposés dans la littérature pour la programmation impérative afin de vérifier les flux d'information, à savoir les mécanismes de compilation et les mécanismes d'exécution.

Dans ce qui suit, pour un objet dans le système, nous écrirons x à la fois pour le nom et pour la valeur de x, et $\underline{x}$ pour sa classe de sécurité. Nous allons utiliser la notation $\underline{x} \rightsquigarrow \underline{y}$ pour dire que l'information peut passer d'un élément de la classe x à un élément de la classe y. Une politique est représentée par un ensemble de tels énoncés.

Mécanismes basés sur la compilation [24, 25]

Le but d'un mécanisme basé sur la compilation consiste à vérifier que les flux d'informations à travers un programme sont autorisés, en déterminant si les flux pourraient violer une politique

d'information donnée.
Certification des instructions: Un ensemble d'instructions est certifié par rapport à une politique de flux d'informations si le flux d'information dans cet ensemble d'instructions ne viole pas la politique.
Par exemple, considérons l'instruction $if\ then\ else$, où x, y, a et b sont des variables:
$if\ x = 1\ then \qquad y := a; \qquad else \qquad y := b;$
De toute évidence, l'information passe de x et a à y ou de x et b à y.
Supposons maintenant que la politique stipule que: $\underline{a} \rightsquigarrow \underline{y}$, $\underline{b} \rightsquigarrow \underline{y}$, et $\underline{x} \rightsquigarrow \underline{y}$, alors le flux d'information est sûr.

Mécanismes basés sur l'exécution

En ce qui concerne les mécanismes basés sur l'exécution, le but est d'empêcher les flux d'informations qui violent la politique. Pour les flux explicites, et avant l'exécution de l'instruction $y = f(x_1, \cdots, x_n)$, le mécanisme basé sur l'exécution vérifie que $lub(\underline{x_1}, \cdots, \underline{x_n}) \rightsquigarrow y$. Ainsi, dans le cas où la condition est vraie, l'affectation se poursuit, sinon elle échoue. En ce qui concerne la vérification des flux implicites, c'est un peu plus compliqué parce que parfois, les instructions peuvent être incorrectement certifiées comme conforme à la politique de confidentialité.

Contrôle d'inférence

Nous disons qu'un **observateur** peut atteindre un **gain d'information** si l'observateur, en observant un message, peut convertir sa **connaissance a priori** en une **connaissance a posteriori** strictement augmentée. Les capacités de calcul et les ressources de calcul disponibles de l'observateur décident si ce gain peut être atteint ou non.

Il est du devoir du contrôle d'inférence d'assurer que les observations accessibles ne sont pas nuisibles. Ceci peut être réalisé par une **surveillance dynamique** ou une **vérification statique**.

- La **surveillance dynamique** inspecte chaque événement et vérifie si des inférences nocives peuvent être observées par le participant. La surveillance dynamique conserve une trace des observations antérieures afin de **bloquer** toute observation critique.
- La **vérification statique** analyse globalement et à l'avance, tous les événements possibles et observables par le participant. La vérification statique inspecte donc tous les événements possibles afin de vérifier s'il existe des inférences nocives et ainsi de **bloquer** ces observations critiques.

Notons que la surveillance dynamique et la vérification statique ont besoin d'une spécification des exigences de la sécurité, c'est à dire d'une **politique de sécurité** qui capte la notion pertinente de **nocivité**.

Notez que lorsque l'observateur est **informé** d'un blocage, que ce soit explicitement ou implicitement par un raisonnement, la reconnaissance d'un blocage peut constituer aussi une information nuisible.

Pour les **refus** explicitement notifiés, l'observateur peut, dans certains cas, déterminer la raison de ce refus, et, par conséquent, trouver l'événement caché.

Chapitre 3: Programmation logique

Dans ce chapitre, nous examinerons la **logique du premier ordre** et nous passerons en revue des notions de **substitutions**, **unification**, **SLD-resolution, SLD-trees**. Les problèmes des SLD-resolutions sont exposés révélant parfois que la recherche d'une SLD-réfutation peut entraîner la non-terminaison. Une non-terminaison due à la présence de boucles. Nous exposerons des **techniques de détection de boucles** en programmation logique. Après avoir présenté les concepts de base des détections des boucles, différentes techniques seront exposées. Des remarques sur des détecteurs de boucles plus efficaces seront aussi abordées. Après avoir exposé la notion du flux de l'information en programmation impérative, il était intéressant de vérifier si ce type de flux existe en programmation logique. Nous exposerons très brièvement les notions d'**analyse du contrôle du flux** et de **dépendance de données** soulignant le fait que la notion de flux d'information précédemment présentée n'est pas couverte par ces analyses dans la programmation logique.

Programmation logique du premier ordre

Dans cette section, nous examinerons les aspects de la programmation logique du premier ordre à savoir: les termes, substitutions, unification et SLD-résolution.

Termes, programmes, buts, et substitutions

Termes et littéraux: Soit X un ensemble de variables et Ω un ensemble de constantes. Les Termes et littéraux du langage de programmation sont définis inductivement par la grammaire $\mathbb{T}_\Omega(X) = x|f(t_1, \cdots, t_n)$ où $x \in X$, $f \in \Omega$, et $t_1, \cdots, t_n \in \mathbb{T}_\Omega(X)$ pour $n \geq 0$. Un terme ground est un terme ne contenant pas de variables.

Univers d'Herbrand: Soit L un langage du premier ordre. L'Univers d'Herbrand U_L de L est l'ensemble de tous les grounds termes qui peuvent être formés à partir des constantes et des symboles de fonction apparaissant dans L.

Programmes définis: Une clause est une formule $A \leftarrow B_1, \cdots, B_n$, pour $n \geq 0$. Le littéral A est la tête de la clause et la constante la plus à gauche dans A est nommé symbole de prédicat. Quand le littéral A est égal à $p(t_1, \cdots, t_n)$, où p est un symbole de prédicat et $t_1, \cdots, t_n$ des termes, A est appelé un **p-litéral**. Une clause avec un p-litéral en sa tête est appelée **p-clause**. Le sous-ensemble de toutes les p-clauses d'un programme est appelé **procédure / définition de prédicat** de p. Les littéraux $B_1, \cdots, B_n$ forment le **corps** de la clause. Quand $n = 0$, on écrira $A \leftarrow$ pour dire que c'est un **fait**. Finallement, un programme logique est un esnemble de clauses.

But: Un but est une formule $\leftarrow B_1, \cdots, B_n$ où $B_1, \cdots, B_n$ sont des littéraux, pour $n \geq 0$. Quand $n = 0$, nous le noterons par $\square$ pour dire que c'est la clause vide.

Le cadre opérationnel d'un langage logique du premier ordre nécessite une procédure pour déterminer si un but réussit ou pas. La procédure de preuve qui en résulte doit faire correspondre des littéraux dans un but à des clauses du programme. Ainsi, les **substitutions** lient les variables aux termes:

Substitutions: Soit X et Y deux ensembles de variables. Une substitution $\phi : X \rightarrow \mathbb{T}_\Omega(Y)$ est une fonction totale faisant correspondre les variables aux termes. Pour $X = \{x_1, \cdots, x_n\}$ et $n \geq 0$, nous repreśentons ϕ en utilisant la notation $\{x_1 \mapsto s_1, \cdots, x_n \mapsto s_n\}$ où les variables x_i correspondent aux termes s_i, pour $1 \leq i \leq n$. L'ensemble Y est donné par $\cup_{i=1}^{n}\ vars\ s_i$, où la fonction $vars : \mathbb{T}_\Omega(Y) \rightarrow \mathrm{P}Y$ prend un terme et retourne l'ensemble de variables contenues dans ce terme.

Le résultat d'un calcul dans un langage logique est une substitution générée par le procédé sus-mentionné. L'instance d'un terme dans le cadre d'une substitution est obtenu selon cette définition:

Application de substitutions: Soit $\phi = \{x_1 \mapsto s_1, \cdots, x_n \mapsto s_n\} : X \rightarrow \mathbb{T}_\Omega(Y)$, nous

définissons une fonction $[\phi] : \mathbb{T}_\Omega(X) \to \mathbb{T}_\Omega(Y)$ qui détermine l'instance d'un terme t sous "ϕ" comme suit:

$$t[\phi] \begin{cases} \phi x & if\ t = x \in X \\ f(t_1[\phi], \cdots, t_n[\phi]) & if\ t = f(t_1, \cdots, t_n) \end{cases}$$

Unification

L'appariement des littéraux dans un but avec les clauses d'un programme est connu sous le nom d'unification [58]. La substitution créée par l'algorithme d'unification est la plus générale que possible dans le sens que tous les autres unificateurs sont une instance de celui-ci.

mgu: Soit $D = \{(s_1, t_1), \cdots, (s_n, t_n)\}$ un ensemble de couple de termes de $\mathbb{T}_\Omega(X)$, pour $n \geq 1$, et soit $\phi : X \to \mathbb{T}_\Omega(Y)$ une substitution. On dit que ϕ est un unifieur de D si $s_i[\phi] = t_i[\phi]$, pour tout $1 \leq i \leq n$. En plus, ϕ est un mgu si, pour tout autre unifieur $\psi : X \to \mathbb{T}_\Omega(Y)$ de D, il existe une substitution $\sigma : Y \to \mathbb{T}_\Omega(Y)$, telel que $\psi = \sigma \circ \phi$.

Nous nous demandons alors, comment vérifier si un but est une conséquence logique d'un programme. La recherche systématique de la preuve d'un but est centrée autour du processus de la résolution.

SLD-Résolution

La résolution est appelée règle d'inférence car elle permet de déduire des informations concernant un but et un programme. La résolution linéaire implique la résolution d'un littéral dans un but avec une clause du programme, produisant un nouveau but que nous continuons à résoudre de la même manière. La méthode de sélection des clauses du programme est appelée **search strategy** et la méthode de sélection du littéral dans le but est appelée la **computation rule**. La preuve d'un but peut être visualisée sous la forme d'une structure arborescente dans laquelle chaque noeud marqué avec un but et chaque branche représente le résultat d'une étape de résolution. Une méthode de résolution basée sur la recherche de preuves qui limite la taille d'un arbre de recherche est appelée SLD-résolution.

SLD-résolution: Soient $C = A \leftarrow B_1, \cdots, B_m$, pour $m \geq 0$, une clause, et $G = \leftarrow A_1, \cdots, A_n$, pour $n \geq 1$, un but tel que A_1 et A sont unifiables avec un mgu ϕ. Alors le **SLD-resolvant** de C et G est le but $G_0 = (\leftarrow B_1, \cdots, B_m, A_2, \cdots, A_n)[\phi]$.

Une séquence de SLD-résolution constitue une SLD-dérivation.

SLD-dérivation: Pour un programme P et un but G, une SLD-dérivation de $P \cup \{G\}$ est une séquence de triplets $(G_1, C_1, \phi_1), \cdots, (G_n, C_n, \phi_n)$, pour $n \geq 1$ (noté aussi $G_1 \Rightarrow_{C_1,\phi_1} \Rightarrow \cdots \Rightarrow_{C_n,\phi_n} G_n$), tels que C_n représente une 'variante' d'une clause dans P, G_n est un but, et ϕ_n est une substitution. De plus, pour $1 \leq i < n$, G_{i+1} est dérivé du G_i et de C_i via substitution ϕ_i par une SLD-résolution.

La méthode basique pour rechercher une preuve d'un but $G = \leftarrow A_1, \cdots, A_n$ est de résoudre sans cesse les littéraux dans G avec des clause de P jusqu'à ce que ce processus échoue ou dérive le but vide $\square$. La définition de ce processus, appelé **SLD-réfutation**, est comme suit:

SLD-réfutation: Pour un programme P et un butl G, une **SLD-réfutation** de G est une SLD-dérivation de $P \cup \{G\}$ commençant apr G et terminant par $G_n = \square$, pour $n \geq 1$. Si les substitutions des SLD-résolutions dans la dérivation sont $\phi_1, \cdots, \phi_n$ alors la **substitution réponse de la réfutation** est $\phi = \phi_n \circ \cdots \circ \phi_1$.

L'arbre de recherche formé à partir d'une SLD-résolution est connu sous le nom de **SLD-tree**, et nous la définissons de la façon suivante:

SLD-tree: Soient P un programme et G un but. Un **SLD-tree** pour $P \cup \{G\}$ est un arbre dont les noeuds sont étiquetés par des buts et les branches sont étiquetées par une substitution de telle façon que les conditions suivantes sont satisfaites:

1. Le noeud racine de l'arbre est G.

2. Soit $G' = \leftarrow A_1, \cdots, A_n$, pour $n \geq 1$, un noeud dans l'arbre.Pour chaque clause $A \leftarrow B_1, \cdots, B_m$, où $m \geq 0$, telle que A_1 et A sont unifiables avec un mgu ϕ, le noeud fils $\leftarrow (B_1, \cdots, B_m, A_2, \cdots, A_n)[\phi]$. Nous étiquetons l'arc connectant G' à un fils avec la substitution de la SLD-résolution.

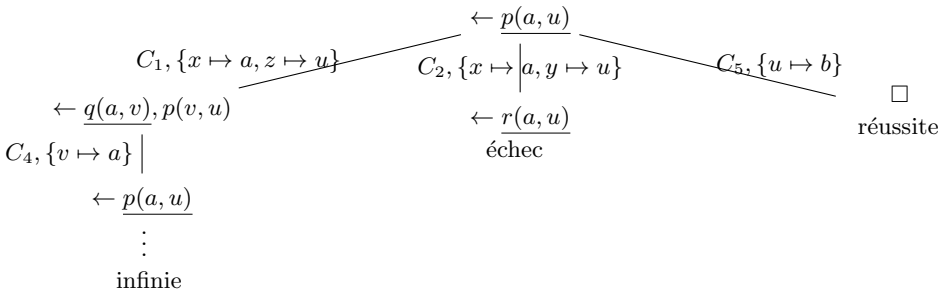
3. Les noeuds libellés avec le but vide $\square$ n'ont pas de fils.

Chaque séquence de noeuds dans un SLD-tree est soit une SLD-réfutation ou une SLD-dérivation.

Dans cette thèse, nous allons nous intéresser aux types de programmes logiques suivants: **Datalog**, **hiérarchique**, **restricted**, *nvi* and *svo*.
Datalog est une version simplifiée de Prolog. Tout programme Datalog doit satisfaire la condition suivante: chaque variable qui se produit dans la tête d'une clause doit également se produire dans le corps de la même clause. Clark [20] et Shepherdson [64] ont introduit la notion des programmes hiérarchiques. Un programme Datalog P est dit hiérarchique ssi il existe une correspondance l associant un entier positive $l(p)$ à chaque symbole de prédicat p dans P et tel que pour toutes les clauses $A_0 \leftarrow A_1, \cdots, A_n$ dans P, si chaque A_i est un litéral de la forme $p_i(t_1, \cdots, t_{k_i})$ alors $l(p_0) > l(p_1), \cdots, l(p_n)$. Le graphe de dépendance d'un programme Datalog P est un graphe (N, E) tel que N est l'ensemble de tous les symboles de prédicat dans P et E est la relation d'adjacence définue sur N: pEq ssi P contient une clause $A_0 \leftarrow A_1, \cdots, A_n$ tel que A_0 est un litéral de la forme $p(\cdots)$ et pour un $1 \leq i \leq n$, A_i est un litéral de la forme $q(\cdots)$. Soit E^* la clôture réflexive et transitive de E. Un programme Datalog P est dit **restricted** ssi pour toutes les clauses $A_0 \leftarrow A_1, \cdots, A_n$ dans P et pour tout $1 \leq i \leq n-1$, si A_0 est de la forme $p(\cdots)$ et A_i est de la forme $q(\cdots)$, alors not qE^*p. Un programme P est **nonvariable introducing** (*nvi*) si pour chaque clause $A \leftarrow B_1, \cdots, B_n$ dans P, chaque variable qui apparaît dans $B_1, \cdots, B_n$ apparaît aussi dans A. Un programme P est **single variable occurrence** (*svo*) si pour chaque clause $A \leftarrow B_1, \cdots, B_n$ dans P, aucune variable dans $B_1, \cdots, B_n$ n'apparaît plus qu'une fois.

Le problème de la SLD-Résolution

La recherche d'une SLD-réfutation peut parfois être beaucoup plus difficile, entrainant souvent une non-terminaison. L'exemple suivant montre ce phénomène.
Considérons le programme logique P suivant:
$C_1 : p(x, z) \leftarrow q(x, y), p(y, z)$;
$C_2 : p(x, y) \leftarrow r(x, y)$;
$C_3 : r(b, b) \leftarrow$;
$C_4 : q(a, a) \leftarrow$;
$C_5 : p(a, b) \leftarrow$
et le but G: $\leftarrow p(a, u)$.



En fait, la SLD-résolution peut être incomplète lors de la recherche d'une preuve pour un but. Ainsi, nous pouvons tenter de détecter toute branche infinie dans un SLD-tree. Mal-

heureusement, ce problème est indécidable. Cependant, plusieurs heuristiques ont été proposées dans la littérature pour éviter cette non-terminaison:

- Poole & Goebel [59] ont proposé d'appliquer certaines techniques de reformulation sur la spécification initiale du programme. Cependant, le programme qui en résulte peut être différent de celui d'origine.
- Apt *et al.* [1], Bol *et al.* [9], Bol [8], Smith *et al.* [65], Van Gelder [33], Vieille [72], Besnard [5], Convington [21], Sahlin [60], Brough & Walker [12], Shen [62] ont proposé de modifier le mécanisme de calcul en ajoutant une capacité de coupure. Ainsi, à un certain moment, l'interpréteur est obligé d'arrêter sa recherche. Ces mécanismes sont appelés des mécanismes de détection de boucle, car ils sont fondés sur l'exclusion de certains types de répétitions dans les SLD-dérivations.

Détection de boucles

Le but principal d'un détecteur de boucles est de réduire l'espace de recherche pour les interpréteurs afin d'obtenir un espace de recherche fini.

Concepts de base

Le but d'un détecteur en boucle est de tailler chaque SLD-tree infini en un sous-arbre fini contenant la racine. Les définitions suivantes ont été introduites par Bol *et al.* [9].
Sous-dérivation: Soit P un programme logique, G un but, et L un ensemble de SLD-dérivations de $P \cup \{G\}$. Définir: $RemSub(L) = \{D \in L | L$ ne contient pas une sous-dérivation de $D\}$.
L est **sous-dérivation libre** si $L = RemSub(L)$.
Détecteur de boucle simple: Un détecteur de boucle simple est un ensemble calculable L de SLD-dérivations finies tel que L est fermé sous des variantes et est sous-dérivation libre.
Pruned SLD-tree: Soient P un programme, G un but, T le SLD-tree de $P \cup \{G\}$, et L un détecteur en boucle. En appliquant L à T, nous obtenons un nouveau SLD-tree $f_L(P \cup \{G\})$ qui consiste en T avec tous les noeuds dans {G' | la SLD-dérivation du but G à G' est dans L} coupés.

Une des propriétés les plus importantes lors de l'utilisation des détecteurs de boucle est de ne pas perdre des résultats de réussite ou toute autre solution individuelle. De plus, vu que le but d'un détecteur de boucle est de réduire un espace infini de recherche à un ensemble fini, la seconde propriété importante est donc de tailler toute dérivation infinie:
Un détecteur de boucle L est **faiblement correct** si: pour tout programme P, but G, et SLD-tree T de $P \cup \{G\}$, si T contient une branche de réussite, alors $f_L(P \cup \{G\})$ contient une branche de réussite.
Un détecteur de boucle L est **correct** si pour tout programme P et but G, et SLD-tree T de $P \cup \{G\}$: si T contient une branche de réussite avec une substitution réponse σ, alors $f_L(P \cup \{G\})$ contient une branche de réussite avec une substitution réponse σ' telle que $G[\sigma'] \leqslant G[\sigma]$.
Un détecteur de boucle L est **complete** si toute SLD-dérivation infinie est coupée par L.

Un détecteur de boucle idéal serait à la fois faiblement correct et complet. Malheureusement, vu que les programmes logiques ont la pleine puissance de la théorie récursive, il n'y a pas de détecteurs de boucle qui sont à la fois faiblement correct et complet, même pour les programmes logiques sans fonctions [9].

Détection de boucles basée sur l'application répétée des clauses

Brough & Walker [12] ont proposé de couper les SLD-dérivations chaque fois qu'une clause est utilisée plus d'une fois pour résoudre un but.
Loop check 1=$RemSub(\{D | D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$ tel que, il existe $i, 0 \leq i < k$, $C_i = C_k\})$.

Loop check 1, basée sur une condition très simple, est trop restrictive, car elle peut éliminer toutes les branches de réussite. En ce sens, nous démontrons que:

1. Loop check 1 est non correct par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, restricted, *nvi*, *svo* et les programmes logiques en général.
2. Loop check 1 est complet par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, restricted, *nvi*, *svo* et les programmes logiques en général.

Détection de boucles basée sur des buts répétés

Brough & Walker [12] ont proposé de couper les SLD-dérivations chaque fois qu'un sous-but est égal à l'un des buts ancêtres.
Loop check 2=$RemSub(\{D|D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$ tel que il existe $i, 0 \leq i < k$, $G_k = G_i\})$.
Loop check 2 n'est pas très satisfaisante car elle ne taille pas toutes les branches infinies. Nous démontrons que:

1. Loop check 2 est correct par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, restricted, *nvi*, *svo* et les programmes logiques en général.
2. Loop check 2 est incomplet par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, restricted, *nvi*, *svo* et les programmes logiques en général.

Détection de boucles basée sur l'application répétée des clauses et des buts répétés

Brough & Walker [12] ont proposé de couper les SLD-dérivations à chaque fois qu'un sous-but est égal à l'un des buts ancêtres et les clauses utilisées pour résoudre les sous-buts sont les mêmes que celles utilisées pour résoudre les buts ancêtres.
Loop check 3=$RemSub(\{D|D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$ tel que il existe $i, 0 \leq i < k$,

- $G_k = G_i$
- $C_k = C_i$

$\})$.
Voyant que *Loop check* 3 est moins restrictive que *Loop check* 2, le problème majeur de son incomplétude reste insoluble.

1. Loop check 3 est correct par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, restricted, *nvi*, *svo* et les programmes logiques en général.
2. Loop check 3 est incomplet par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, restricted, *nvi*, *svo* et les programmes logiques en général.

Détection de boucles basée sur des atomes répétés à travers des variantes syntaxiques

Convington [21] a proposé de couper les SLD-dérivations à chaque fois qu'un atome dans un sous-but est une variante d'un atome dans un de ses buts ancêtres.
Loop check 4=$RemSub(\{D|D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$ tel que il existe $i, 0 \leq i < k$, où $G_i = A_{i_1}, \cdots, A_{i_n}$ et $G_k = A_{k_1}, \cdots, A_{k_m}$, il existe $1 \leq j \leq n$, $1 \leq l \leq m$ et une substitution τ, tel que: $A_{k_l} = A_{i_j}[\tau]\})$.
Le champ d'application de *Loop check* 4 est plutôt limité. Nous démontrons que:

1. Loop check 4 est non correct par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, restricted, *nvi*, *svo* et les programmes logiques en général.

2. Loop check 4 est complet par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, restricted, *nvi*, *svo*.
3. Loop check 4 est incomplet par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes logiques en général.

Détection de boucles basée sur des buts égaux

Bol *et al.* [9] ont proposé de couper les SLD-dérivations de la forme $(G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots)$ quand il apparaît deux buts G_i et G_j $(0 \leq i < j)$ tels que G_j est une variante de G_i.
Loop check 5=$RemSub(\{D | D = (G_0 \Rightarrow_{C_1,\theta_1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k)$, tel que, pour $i, 0 \leq i < k$, il existe une substitution τ, tel que: $G_k = G_i[\tau]\})$.
Nous démontrons que:

1. Loop check 5 est non correct par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, *nvi*, *svo* et les programmes logiques en général.
2. Loop check 5 est correct par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes restricted.
3. Loop check 5 est complet par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes restricted.
4. Loop check 5 est incomplet par rapport à la règle de sélection de l'atome le plus à gauche pour les programmes Datalog, *nvi*, *svo* et les programmes logiques en général.

Remarques sur des détecteurs de boucles efficaces

Tous les détecteurs de boucles présentés comparent les buts dérivés dans la dérivation et coupent la dérivation quand un but suffisamment similaire est rencontré. Théoriquement, un sous-but est comparé à tous les buts qui le précède dans la dérivation. En pratique, ces détecteurs de boucles sont trop coûteux vu qu'un détetceur de boucle peut effectuer $\frac{1}{2}|D||D-1|$ comparaisons pour une SLD-dérivation D finie.

Deux approches ont été proposées dans la littérature pour réduire le nombre de comparaisons effectuées par un détecteur de boucle:

1. **La technique de tortoise-and-hare**: Van Gelder [33] a proposé de comparer tous les buts G_k à un seul but, à savoir le but à mi-chemin de la dérivation $G_{k/2}$. Il montre que cette technique préserve la correction mais pas la complétude du détecteur de boucle.
2. **La technique de sélection**: Bol [8] a proposé non seulement de sélectionner le but qui est à mi-chemin de la dérivation pour le comparer à G_k, mais de sélectionner un nombre infini de buts et de comparer chacun à G_k. Il montre que cette technique préserve la correction du détecteur de boucle ainsi que sa complétude (pour des détecteurs de boucle et de programmes logiques bien spécifiques).

"Flux" en programmation logique

En général, dans les programmes logiques, les concepts d'arguments d'**entrée** et de **sortie** n'existent pas. On dit aussi que les programmes logiques ne sont pas dirigés. En outre, les unifications de sous-buts dans les programmes logiques peuvent procéder en deux directions: continuer après un succès et chainage en arrière après un échec. Ainsi, afin de générer un code exécutable des programmes logiques qui soit aussi efficace, les chercheurs ont étudié différentes dépendances dans les programmes logiques comme l'**information sur le mode** [23, 53, 70, 73], **l'inférence du mode** [13, 22, 54] et les **dépendances de données** [17, 18, 27].

Nous présenterons l'analyse de flux de données et de contrôle du flux pour la programmation logique, soulignant le fait que la notion du flux de l'information dans les systèmes de sécurité déjà présentée dans le chapitre 2 n'est pas couverte par ces analyses pour les programmes logiques.

Analyse du contrôle du flux

Le contrôle du flux dans les programmes logiques se réfère à l'ordre dans lequel le but est évalué. Ces contrôles du flux ne sont pas si évidents que dans les programmes impératifs. Ceci est dû au fait que les flux de contrôle sont cachés dans les programmes logiques. La sémantique de Prolog implique plusieurs types de chainage arrière, et de recherche d'un sous-but pour lequel le contrôle est transféré après chainage arrière.

Analyse de dépendance de données

Une dépendance de données dans les programmes logiques existe lorsqu'une clause se réfère à une variable / argument dans la définition de la même clause ou dans la tête de la définition d'une autre clause.

Chapitre 4: Flux de l'information en programmation logique

La théorie du flux de l'information est bien définie pour les programmes impératifs. Différents modèles du flux sont proposés, à savoir le modèle du Bell-Lapadula [3], le modèle de non-déducibilité et de non-interférence [34]. Cependant aucune étude ne s'est penchée sur ce que pourrait être un flux de l'information (dans le sens déjà présenté) en programmation logique. Dans ce cahpitre, nous proposons trois définitions du flux de l'information pour les programmes Datalog. Ces définitions correspondent à ce qui peut être observé par l'utilisateur lorsqu'une requête $\leftarrow G(x,y)$ est posée pour un programme logique P. Dans la première définition, l'utilisateur ne voit que si les buts réussissent ou échouent. Dans la seconde définition, l'utilisateur a accès à l'ensemble des substitutions-réponses calculées par le programme. Dans la troisième définition, l'utilisateur obtient la forme des arbres de calcul produits par le programme.

Flux de l'information basée sur réussite/échec

Soit P un programme Datalog, et $G(x,y)$ un but à deux variables. Nous dirons qu'il y a un flux de x vers y dans $G(x,y)$ par rapport à la réussite et l'échec dans P (en symbole $x \xrightarrow{SF}{}^{P}_{G} y$) ssi il existe $a, b \in U_{L(P)}$ tel que $P \cup \{G(a,y)\}$ réussit et $P \cup \{G(b,y)\}$ échoue.

Flux de l'information basée sur les substitutions réponses

Soit P un programme Datalog, et $G(x,y)$ un but à deux variables. Nous dirons qu'il y a un flux de x vers y dans $G(x,y)$ par rapport aux substitutions réponses dans P (en symbole $x \xrightarrow{SA}{}^{P}_{G} y$) ssi il existe $a, b \in U_{L(P)}$ tel que $\Theta(P \cup \{G(a,y)\}) \neq \Theta(P \cup \{G(b,y)\})$.

Flux de l'information basée sur la bisimulation

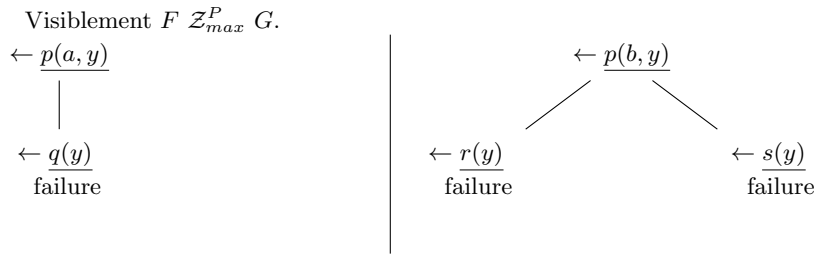
Une bisimulation est une relation binaire entre les buts de telle sorte que les buts reliés ont des arbres de résolution "équivalents".

Définition de la bisimulation

Soit P un programme Datalog. Une relation binaire $\mathcal{Z}$ entre les buts est une P-bisimulation ssi elle satisfait les conditions suivantes; pour tous les buts Datalog F_1, G_1 tel que $F_1 \mathcal{Z} G_1$:

- $F_1 = \square$ ssi $G_1 = \square$,
- Pour chaque SLD-résolvant F_2 de F_1 (c-à-d $F_2 \in succ_P(F_1)$) et une clause dans P, il existe un résolvant G_2 de G_1 (c-à-d $G_2 \in succ_P(G_1)$) et une clause dans P telle que $F_2 \mathcal{Z} G_2$,
- Pour chaque SLD-résolvant G_2 de G_1 (c-à-d $G_2 \in succ_P(G_1)$) et une clause dans P, il existe un résolvant F_2 de F_1 (c-à-d $F_2 \in succ_P(F_1)$) et une clause dans P telle que $F_2 \mathcal{Z} G_2$.

$succ_P(G)$ dénote l'ensemble de tous les buts obtenus depuis le but G moyennant une étape de résolution dans le programme P.
Nous démontrons qu'il existe une P-bisimulation maximale $\mathcal{Z}^P_{max}$ entre les buts définie comme suit: $F_1 \mathcal{Z}^P_{max} G_1$ ssi il existe une P-bisimulation $\mathcal{Z}$ telle que $F_1 \mathcal{Z} G_1$. Il en résulte que $\mathcal{Z}^P_{max}$ est une relation d'équivalence sur l'ensemble des buts.
Comme exemple, soit P le programme suivant:
$C_1 : p(a,y) \leftarrow q(y)$;
$C_2 : p(b,y) \leftarrow r(y)$;
$C_3 : p(b,y) \leftarrow s(y)$
et soient G, H respectivement les buts $\leftarrow p(a,y)$ et $\leftarrow p(b,y)$



Soit P un programme Datalog, et $G(x,y)$ un but à deux variables. Nous dirons qu'il y a un flux de x vers y dans $G(x,y)$ par rapport aux bisimulations dans P (en symbole $x \xrightarrow{BI}{}^{P}_{G} y$) ssi il existe $a, b \in U_{L(P)}$ tel que $not\{P \cup \{G(a,y)\} \mathcal{Z}^P_{max} P \cup \{G(b,y)\}\}$.

Liens entre les différentes définitions du flux de l'information

L'existence d'un flux par rapport à la subtitution réponse n'implique pas l'existence d'un flux par rapport à la réussite/échec. Cependant nous pouvons démontrer que pour un programme logique P et un but $G(x,y)$ à deux variables, si $x \xrightarrow{SF}{}^{P}_{G} y$ alors $x \xrightarrow{SA}{}^{P}_{G} y$.
Parallèlement, l'existence d'un flux par rapport à la bisimulation n'implique pas l'existence d'un flux par rapport à la réussite/échec. Cependant nous pouvons démontrer que pour un programme logique P et un but $G(x,y)$ à deux variables, si $x \xrightarrow{SF}{}^{P}_{G} y$ alors $x \xrightarrow{BI}{}^{P}_{G} y$.

Non-transitivité du flux

La plupart des politiques de contrôle du flux de l'information dans la programmation impérative sont représentées par une structure en treillis, ce qui signifie que s'il y a un flux d'information de la variable x à la variable y et de la variable y à z, alors il existe un flux de x vers z. Dans de tels contextes, la relation du flux de l'information entre les variables du programme est dite transitive. Dans notre cas et selon les définitions proposées, la relation du flux de l'information en programmation logique n'est pas transitive.

Résultats de complexité

Nous nous sommes intéressés l'étude de la complexité algorithmique des problèmes de décision suivants:

$$\pi_{SF} \begin{cases} \text{Entrée: Un programme logique } P, \text{ un but à deux variables } G(x,y) \\ \text{Sortie: Décider si } x \xrightarrow{SF}{}^{P}_{G} y \end{cases}$$

$$\pi_{SA} \begin{cases} \text{Entrée: Un programme logique } P, \text{ un but à deux variables } G(x,y) \\ \text{Sortie: Décider si } x \xrightarrow{SA}{}^{P}_{G} y \end{cases}$$

$$\pi_{BI} \begin{cases} \text{Entrée: Un programme logique } P, \text{ un but à deux variables } G(x,y) \\ \text{Sortie: Décider si } x \xrightarrow{BI}{}^{P}_{G} y \end{cases}$$

Indécidabilité

Dans le cas général, les trois problèmes de décision sont indécidables.
Idée de la preuve:
(π_{SF}) Réduire π_1 qui est indécidable [28] à π_{SF}:

$$\pi_1 \begin{cases} \text{Entrée: Un programme logique } P, \text{ un but sans variable } q(a) \\ \text{Sortie: } P \cup \{\leftarrow q(a)\} \text{ réussit} \end{cases}$$

(π_{SA}) Preuve similaire s'applique ici.

(π_{BI}) Réduire π_2 qui est indécidable [29] à π_{BI}:

$$\pi_2 \begin{cases} \text{Entrée: Un programme logique binaire } P, \text{ un but sans variable } q(a) \\ \text{Sortie: L'arbre de résolution de } P \cup \{\leftarrow q(a)\} \text{ contient une branche d'échec} \end{cases}$$

Décidabilité

Pour Datalog, nous démontrons que π_{SF} est EXPTIME-complet.
Idée de la preuve:
(*Appartenance*) Algorithme qui décide de l'existence du flux dans les programmes Datalog.

Require: Un programme Datalog P, un but $G(x, y)$, Univers d'Herbrand fini $U_{L(P)} = \{a_1, \cdots, a_n\}$
Ensure: $x \xrightarrow{SF}{}^{P}_{G(x,y)} y$ pour le programme Datalog P et le but G
1: $answer = faux$
2: $i = 0$
3: **while** $i < n$ et non $answer$ **do**
4: $i = i + 1$; $j = i$
5: **while** $j < n$ et non $answer$ **do**
6: $j = j + 1$
7: **if** ($P \cup \{G(a_i, y)\}$ réussit et $P \cup \{G(a_j, y)\}$ échoue) ou ($P \cup \{G(a_i, y)\}$ échoue et $P \cup \{G(a_j, y)\}$ réussit) **then**
8: $answer = true$
9: **end if**
10: **end while**
11: **end while**
12: return $answer$

Cet algorithme est deterministe et, sachant que les programmes Datalog sont complets pour EXPTIME [41, 71], il en résulte que cet algorithme peut être exécuté en EXPTIME.

(*Hardness*) Réduire π_3 qui est EXPTIME-hard [71] à π_{SF}:

$$\pi_3 \begin{cases} \text{Entrée: Un programme Datalog } P, \text{ un atome ground } A \\ \text{Sortie: } P \cup A \ (A \text{ est une conséquence logique de } P) \end{cases}$$

Pour Datalog aussi, nous démontrons que π_{SA} est EXPTIME-complet.
Pour les programmes Datalog binaires hiérarchiques, π_{SF} est dans $\Delta_2 P$.
Idée de la preuve:
Considérons l'algorithme déterministe avec oracle:

Require: Un programme Datalog binaires hiérarchiques P, un but $G(x, y)$.
Ensure: $x \xrightarrow{SF}{}^{P}_{G} y$.
1: Pour chaque a dans $U_{L(P)}$ faire
2: Pour chaque b dans $U_{L(P)}$ faire
3: **if** ($P \cup \{G(a, y)\} \in$ SUCCESSES and $P \cup \{G(b, y)\} \in$ FAILURES) **then**
4: Accept
5: **else**
6: Reject
7: **end if**

Les oracles SUCCESSES représentent l'ensemble des couples de buts (P, G) telles que G réussit dans P. En restreignant P aux programmes Datalog binaires hiérarchiques, nous démontrons que SUCCESSES appartient à NP. Les oracles FAILURES représentent l'ensemble des couples de buts (P, G) telles que G échoue dans P. En restreignant P aux programmes Datalog binaires hiérarchiques, nous démontrons que SUCCESSES appartient à co-NP. Alors π_{SF} est dans $\Delta_2 P$. π_{BI} est dans EXPTIME pour les programmes Datalog binaires hiérarchiques.
Idée de la preuve:
nous démontrons que π_{BI} est dans $APSPACE$, vu que $EXPTIME = APSPACE$. Soit l'algorithme alternant suivant qui peut être implémenté en espace polynomial:

Require: Un programme Datalog binaire hiérarchique P, deux buts G_1 et G_2.
Ensure: Décider si $P \cup \{G_1\} \; \mathcal{Z}^P_{max} \; P \cup \{G_2\}$.
case $(succ(P, G_1), succ(P, G_2))$
- $(true, true)$**:**
  (∀) choisir $i, j \in \{1, 2\}$ tel que $i \neq j$
  (∀) choisir un successeur G'_i de G_i dans P
  (∃) choisir un successeur G'_j de G_j dans P
  (.) call **bisim**(P, G'_i, G'_j)
- $(true, false)$**:** reject
- $(false, true)$**:** reject
- $(false, false)$**:**
if $(G_1 = \square$ iff $G_2 = \square)$ **then**
  Accept
else
  Reject
end if

Existence du flux après transformation du programme

Nous nous intéressons à vérifier si les transformations (unfold/fold proposées tout d'adord par Burstall et Darlington [14] dans le contexte des langages functionels et puis par Tamaki et Sato [69] pour la programmation logique) sur les programmes logiques peuvent introduire ou éliminer des flux.
Sachant que le fait d'appliquer une transformation de fold ou de unfold sur une clause ne modifie ni la réussite, ni l'échec [69], ni les substitutions réponses [44] dans un programme logique, les flux d'information basés soit sur réussite/échec ou sur les substitutions réponses seront conservés après application de ces transformations. Cependant, nous ne pouvons pas en dire autant des flux de l'information basés sur la bisimulation. Des exemples montrent qu'un flux peut apparaître pour ensuite disparaître.

Chapitre 5: Bisimulation de buts

Dans le contexte d'un langage de programmation donné, il est indispensable d'associer une sémantique aux programmes. Cette sémantique induit une relation d'équivalence entre les programmes. Par conséquent, le problème de décision consistant à déterminer si deux programmes donns sont équivalents est au coeur de l'étude de la sémantique des programmes. Notre but est de proposer une utilisation des relations d'équivalence entre les programmes logiques qui tiennent en compte la forme des SLD-tree.
Dans ce chapitre, nous considérons tout d'abord les programmes Datalog. Nous dirons que, par rapport à un programme Datalog P, deux buts sont équivalents si leurs SLD-trees sont bisimilaires.
Dans ce chapitre, nous examinerons la complexité des problèmes de décision suivants:

- Etant donné deux buts Datalog F, G et un programme Datalog hiérarchique P, dire si les arbres SLD de $P \cup F$ et de $P \cup G$ sont bisimilaires.
- Etant donné deux buts Datalog F, G et un programme Datalog restricted P, dire si les arbres SLD de $P \cup F$ et de $P \cup G$ sont bisimilaires.

Indécidabilité pour les programmes Prolog

Il est indécidabile, étant donné un programme Prolog P et des buts Prolog F_1, G_1, de décider si $P \cup \{F_1\} \; \mathcal{Z}^P_{max} \; P \cup \{G_1\}$.
Idée de la preuve:
Réduire π_4 qui est indécidable [28] à notre problème de décision:

$$\pi_4 \begin{cases} \text{Entrée: Un programme binaire Prolog } P, \text{ un but Prolog } A \\ \text{Sortie: L'arbre de résolution de } P \cup A \text{ contient une branche infinie} \end{cases}$$

Une question se pose ici, comment pouvons-nous rétablir la décidabilité de ce problème de décision? Nous pouvons penser à restreindre le langage du programme logique. Ainsi, dans le cadre des programmes Datalog par exemple, la principale difficulté vient des branches infinies dans les arbres de résolution. Nous aborderons cette question dans les sections suivantes.

Décidabilité pour les programmes hiérarchiques

(π_{hie}): Soit P un programme Datalog hiérarchique et F_1, G_1 deux buts Datalog, décider si $P \cup \{F_1\} \; \mathcal{Z}^P_{max} \; P \cup \{G_1\}$.
Idée de la preuve:

function bisim1(F_1,G_1)

```
1: if bothempty(F1, G1) or bothfail(F1, G1) then
2:    return true
3: else
4:    SF ⟵ successor(F1)
5:    SG ⟵ successor(G1)
6:    if SF ≠ ∅ and SG ≠ ∅ then
7:       SF' ⟵ SF
8:       while SF' ≠ ∅ do
9:          F2 ⟵ get-element(SF')
10:         found-bisim ⟵ false
11:         SG' ⟵ SG
12:         while SG' ≠ ∅ and found-bisim = false do
13:            G2 ⟵ get-element(SG')
14:            found-bisim ⟵ bisim1(F2, G2)
15:         end while
```

16: **if** $found\text{-}bisim = false$ **then**
17: return $false$
18: **end if**
19: **end while**
20: $SG' \longleftarrow SG$
21: **while** $SG' \neq \emptyset$ **do**
22: $G_2 \longleftarrow get\text{-}element(SG')$
23: $found\text{-}bisim \longleftarrow false$
24: $SF' \longleftarrow SF$
25: **while** $SF' \neq \emptyset$ and $found\text{-}bisim = false$ **do**
26: $F_2 \longleftarrow get\text{-}element(SF')$
27: $found\text{-}bisim \longleftarrow bisim1(G_2, F_2)$
28: **end while**
29: **if** $found\text{-}bisim = false$ **then**
30: return $false$
31: **end if**
32: **end while**
33: return $true$
34: **else**
35: return $false$
36: **end if**
37: **end if**

Nous démontrons que l'algorithme `bisim1(`F_1`,`G_1`)` se termine et que si $P \cup \{F_1\}$ $\mathcal{Z}^P_{max}$ $P \cup \{G_1\}$, alors `bisim1(`F_1`,`G_1`)` retourne `true`, et que si `bisim1(`F_1`,`G_1`)` retourne `true`, alors $P \cup \{F_1\}$ $\mathcal{Z}^P_{max}$ $P \cup \{G_1\}$. Il en résulte que (π_{hie}) est décidable. Nous démontrons de même que (π_{hie}) est dans 2EXPTIME.

Décidabilité pour les programmes restricted

(π_{res}): Soit P un programme Datalog restricted et F_1, G_1 deux buts Datalog, décider si $P \cup \{F_1\}$ $\mathcal{Z}^P_{max}$ $P \cup \{G_1\}$.

Idée de la preuve:

function bisim2($(F_1 \Rightarrow \cdots \Rightarrow F_i)$**,**$(G_1 \Rightarrow \cdots \Rightarrow G_i)$**)**

1: **if** $bothempty(F_i, G_i)$ or $bothfail(F_i, G_i)$ or $occur((F_1 \Rightarrow \cdots \Rightarrow F_i), (G_1 \Rightarrow \cdots \Rightarrow G_i))$ **then**
2: return $true$
3: **else**
4: $SF \longleftarrow successor(F_i)$
5: $SG \longleftarrow successor(G_i)$
6: **if** $SF \neq \emptyset$ and $SG \neq \emptyset$ **then**
7: $SF' \longleftarrow SF$
8: **while** $SF' \neq \emptyset$ **do**
9: $F' \longleftarrow get\text{-}element(SF')$
10: $found\text{-}bisim \longleftarrow false$
11: $SG' \longleftarrow SG$
12: **while** $SG' \neq \emptyset$ and $found\text{-}bisim = false$ **do**
13: $G' \longleftarrow get\text{-}element(SG')$
14: $found\text{-}bisim \longleftarrow bisim2((F_1 \Rightarrow \cdots \Rightarrow F_i \Rightarrow F'), (G_1 \Rightarrow \cdots \Rightarrow G_i \Rightarrow G'))$
15: **end while**
16: **if** $found\text{-}bisim = false$ **then**
17: return $false$
18: **end if**

```
19:       end while
20:       SG' ⟵ SG
21:       while SG' ≠ ∅ do
22:          G' ⟵ get-element(SG')
23:          found-bisim ⟵ false
24:          SF' ⟵ SF
25:          while SF' ≠ ∅ and found-bisim = false do
26:             F' ⟵ get-element(SF')
27:             found-bisim ⟵ bisim2((G1 ⇒ ··· ⇒ Gi ⇒ G'), (F1 ⇒ ··· ⇒ Fi ⇒ F'))
28:          end while
29:          if found-bisim = false then
30:             return false
31:          end if
32:       end while
33:       return true
34:    else
35:       return false
36:    end if
37: end if
```

Nous démontrons que l'algorithme `bisim2((`F_1`),(`G_1`))` se termine et que si $P \cup \{F_1\}$ $\mathcal{Z}^P_{max}$ $P \cup \{G_1\}$, alors `bisim2((`F_1`),(`G_1`))` retourne `true`, et que si `bisim2((`F_1`),(`G_1`))` retourne `true`, alors $P \cup \{F_1\}$ $\mathcal{Z}^P_{max}$ $P \cup \{G_1\}$. Il en résulte que (π_{res}) est décidable. Nous démontrons de même que (π_{res}) est dans 2EXPTIME.

Notes sur la bisimilarité pour les buts *nvi* et *svo*

Nous avons essayé de prouver la décidabilité de la bisimilarité de deux buts donnés pour les programmes *nvi* et *svo*. Malheureusement, l'application des techniques de détection de boucles développées dans [9] ne semble pas nous permettre de déterminer si deux buts sont bisimilaires par rapport à un programme logique *nvi* et *svo*.

Chapitre 6: Application

Dans ce chapitre, nous allons fournir une application dans le domaine du contrôle de l'inférence en nous basant sur la théorie du flux de l'information dans la programmation logique déjà présenté. Nous introduisons la notion de niveau du flux et nous définirons formellement les notions de mécanisme, mécanisme sûr, mécanisme certain et politique de sécurité pour les bases de données déductives et ceci en se basant sur les notions déjà évoquées.

Niveau d'indiscernabilité du flux de l'information en programmation logique

Pour un programme Datalog P et un but $G(x, y)$ avec la variable x considérée comme une variable d'entrée et y comme une variable de sortie, soit $\equiv$ une relation binaire sur $U_{L(P)}$ de cardinalité n. Soient a, b deux éléments distincts de $U_{L(P)}$.

- Pour la première définition du flux (basée sur réussite/échec), on dit que $a \equiv b$ ssi les deux buts $P \cup \{\leftarrow p(a, y)\}$ et $P \cup \{\leftarrow p(b, y)\}$ réussissent ou les deux buts $P \cup \{\leftarrow p(a, y)\}$ et $P \cup \{\leftarrow p(b, y)\}$ ne réussissent pas.
- Pour la deuxième définition du flux (basée sur les substitutions réponses), on dit que $a \equiv b$ ssi $\theta(P \cup \{\leftarrow p(a, y)\}) = \theta(P \cup \{\leftarrow p(b, y)\})$.
- Pour la troisième définition du flux (basée sur la bisimulation entre les buts), on dit que $a \equiv b$ ssi $Tree(P \cup \{\leftarrow p(a, y)\})\ Z^P_{max}\ Tree(P \cup \{\leftarrow p(b, y)\}))$.

Nous démontrons que $\equiv$ est une relation d'équivalence.

Pour un programme Datalog P et un but $G(x, y)$, le niveau du but $G(x, y)$ est égal au cardinal de la plus petite classe d'équivalence.
Théoriquement, ce niveau peut être calculé pour chacune des trois définitions du flux de l'information présentées précédemment. Par exemple, pour la première définition de flux basée sur réussite/échec, nous pouvons écrire l'algorithme suivant:

Require: Un programme Datalog P, un but $G(x, y)$, un Univers d'Herbrand fini $U_{L(P)} = \{a_1, \cdots, a_n\}$
Ensure: Niveau de $G(x, y)$
1: $Level_{succ} \leftarrow 0$; compteur sur le nombre des buts qui réussissent
2: $Level_{no-succ} \leftarrow 0$; compteur sur le nombre des buts qui ne réussissent pas
3: $i \leftarrow 1$; compteur sur l'ensemble de l'Univers d'Herbrand
4: **while** $i \leq n$ **do**
5: **if** $P \cup \{\leftarrow p(a_i, y)\}$ réussit **then**
6: $Level_{succ} \leftarrow Level_{succ} + 1$;
7: **else**
8: $Level_{no-succ} \leftarrow Level_{no-succ} + 1$;
9: **end if**
10: $i \leftarrow i + 1$;
11: **end while**
12: return $min(Level_{succ}, Level_{no-succ})$;

Quant à la seconde définition du flux de l'information basée sur les substitutions réponses, nous pouvons écrire l'algorithme suivant:

Require: Un programme Datalog P, un but $G(x, y)$, un Univers d'Herbrand fini $U_{L(P)} = \{a_1, \cdots, a_n\}$, m le nombre total de substitutions réponses possible de la variable y et ceci pour tous les buts $P \cup \{\leftarrow G(a_1, y)\}, \cdots, P \cup \{\leftarrow G(a_n, y)\}$.
Ensure: Niveau de $G(x, y)$
1: *table* $T[m]$; table de compteurs, tous initialisés à 0, correspondant aux m substitutions réponses différentes. Les index dans la table T sont les m différentes substitutions réponses et

les valeurs correspondantes représentent le nombre total d'occurrences de cette substitution réponse
2: $i \leftarrow 1$; compteur sur l'ensemble de l'Univers d'Herbrand
3: $sub \leftarrow \theta[P \cup \{p(a_1, y)\}]$;nous initialisons sub par la première substitution réponse de y
4: **while** $i \leq n$ **do**
5: $tmps \leftarrow \Theta[P \cup \{\leftarrow p(a_i, y)\}]$; comme $P \cup \{\leftarrow p(a_i, y)\}$ peut avoir plusieurs substitution réponses, $tmps$ est la table contenant ses substitutions réponses
6: $j \leftarrow 1$; Compteur sur l'ensemble des substitutions réponses pour le but $P \cup \{\leftarrow p(a_i, y)\}$
7: **while** $j <= count(tmps)$ **do**
8: $T[tmps[j]] \leftarrow T[tmps[j]] + 1$;
9: **if** $T[tmps[j]] < T[sub]$ **then**
10: $sub \leftarrow tmps[j]$;
11: **end if**
12: $j \leftarrow j + 1$;
13: **end while**
14: $i \leftarrow i + 1$;
15: **end while**
16: return $T[sub]$;

En ce qui concerne le calcul du niveau basé sur la définition de la bisimulation, nous pouvons utiliser un algorithme similaire à celui présenté ci-dessus conjointement avec un des algorithmes (correct et complet) présentés dans le chapitre précédent qui décide si deux buts sont bisimilaires pour les prgrammes Datalog hiérarchiques et restricted.
Pour un programme Datalog P et deux buts $F(x, y)$ et $G(x, y)$, on dit que:

- le but $F(x, y)$ est critique ssi le niveau de $F(x, y)$ est égal à 1.
- le but $F(x, y)$ est plus faible que le but $G(x, y)$ ssi le niveau de $F(x, y)$ est plus grand que le niveau de $G(x, y)$.
- le but $F(x, y)$ est plus fort que le but $G(x, y)$ ssi le niveau de $F(x, y)$ est plus petit que le niveau de $G(x, y)$.

Pour un programme Datalog P et un but$F(x, y)$, si $F(x, y)$ est critique alors la variable de sortie y révèle des informations sur la variable x.

Contrôle d'inférence préventive des systèmes d'information

Le contrôle d'inférence préventive des systèmes d'information doit assurer un compromis entre la disponibilité et la confidentialité des informations: pour un utilisateur en particulier, le système doit retourner des réponses utiles aux requêtes émises et légitimes par cet utilisateur tout en cachant toute information complémentaire qu'il garde à sa disposition pour les autres utilisateurs.

Mécanismes de sécurité sûrs et précis

Nous nouns intéressons à savoir s'il est possible de mettre au point une procédure pour l'élaboration d'un mécanisme qui est à la fois sûr et précis en utilisant la notion de flux de l'information pour les programmes logiques.
Pour cela, nous allons représenter les programmes logiques comme des fonctions abstraites:
Pour un programme logique P représenté par sa définition de prédicat $t(I_1, \cdots, I_n, O)$, où $I_1, \cdots, I_n$ sont des positions d'entrée et O une position de sortie, soit p la fonction $p : I_1 \times \cdots \times I_n \times O \rightarrow R$. Alors p est une fonction avec n **positions d'entrée** $i_k \in I_k, 1 \leq k \leq n$, et **une position de sortie** $o \in O$, et **un résultat** $r \in R$. O est l'ensemble de substitutions réponses associé à la position de sortie o. Selon chaque définition du flux de l'information, R

peut être égal à {*réussite,échec*}, ou à l'ensemble des substitutions réponses correspondant à la position de sortie o, ou à l'arbre SLD du but $P \cup \{\leftarrow t(i_1, \cdots, i_n, o)\}$.

En tenant compte de la confidentialité des informations, une question se pose: est-ce que le résultat de $p(i_1, \cdots, i_n, o)$ contient des renseignements qui pourraient violer la politique de confidentialité. Pour cela, les mécanismes de protection sont proposés. Un mécanisme de protection produit, pour chaque entrée qui ne viole pas la politique, la même valeur que p, et pour les entrées qui laissent fuire des informations confidentielles un message d'erreur. Pour cela, soit E l'ensemble des résultats d'un programme p qui indiquent des erreurs.
Soit p une fonction $p : I_1 \times \cdots \times I_n \times O \to R$. Un mécanisme de protection m est une fonction $m : I_1 \times \cdots \times I_n \to R \cup E$ pour laquelle, quand $i_k \in I_k, 1 \leq k \leq n$, $o \in O$, soit

- $m(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$ ou
- $m(i_1, \cdots, i_n) \in E$

Pour la politique de confidentialité:
Une politique de confidentialité pour un programme logique $p : I_1 \times \cdots \times I_n \times O \to R$ est une fonction $c : I_1 \times \cdots \times I_n \to J_1 \times \cdots \times J_n$, où $J_1 \subseteq I_1, \cdots, J_n \subseteq I_n$.

Pour le mécanisme sûr:
Soit $c : I_1 \times \cdots \times I_n \to J_1 \times \cdots \times J_n$ une politique de confidentialité pour le programme logique p. Soit $m : I_1 \times \cdots \times I_n \to R \cup E$ un mécanisme de sécurité pour le même programme logique p. Alors le mécanisme m est **sûr** (c-à-d confidentiel) ssi il existe une fonction $m' : J_1 \times \cdots \times J_n \to R \cup E$ tel que, pour tout $i_k \in I_k, 1 \leq k \leq n, m(i_1, \cdots, i_n) = m'(c(i_1, \cdots, i_n))$.

Soient m_1 et m_2 deux mécanismes pour le programme p sous la politique c. Alors m_1 est aussi précis que m_2 ($m_1 \succ m_2$) sachant que pour toutes les entrées $(i_1, \cdots, i_n)$, si $m_2(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$, alors $m_1(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o)$.
On dit que m_1 est plus précis que m_2 ($m_1 \gg m_2$) si ($m_1 \succ m_2$) et il existe une entrée $(i'_1, \cdots, i'_n)$ telle que $m_1(i'_1, \cdots, i'_n) = p(i'_1, \cdots, i'_n, o)$ et $m_2(i'_1, \cdots, i'_n) \neq p(i'_1, \cdots, i'_n, o)$.
Nous démontrons que la relation $\succ$ est réflexive et transitive, et que la relation $\gg$ est un ordre strict sur les mécanismes de protection pour un programme donné p et une politique de confidentialité c.

Soient m_1 et m_2 des mécanismes de protection pour le programme p, alors leur union $m_3 = m_1 \cup m_2$ est définie par:

$$m_3(i_1, \cdots, i_n) \begin{cases} = p(i_1, \cdots, i_n) \text{ quand } m_1(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o) \text{ ou} \\ m_2(i_1, \cdots, i_n) = p(i_1, \cdots, i_n, o) \\ = m_1(i_1, \cdots, i_n) \text{ autrement.} \end{cases}$$

Nous pouvons voir que la définition précédente n'est pas symétrique, vu que $m_1 \cup m_2 \neq m_2 \cup m_1$.

A partir de cette définition et la définition de mécanisme sûr et précis, nous avons: Etant donnés m_1 et m_2 deux mécanismes sûrs pour le programme p et la politique c, $m_1 \cup m_2$ est aussi un mécanisme sûr pour le programme p et c. De plus, $m_1 \cup m_2 \succ m_1$ et $m_1 \cup m_2 \succ m_2$.

Pour tout programme p et politique de confidentialité c, il exitse un mécanisme sûr et précis m^* tel que, pour tous les mécanismes sûrs m associés à p et c, $m^* \succ m$.

Chapitre 7: Evaluation et travaux futurs

Dans le dernier chapitre, nous résumons les travaux présentés dans cette thèse et définissons des orientations possibles pour des recherches plus poussées.

Résumé de la thèse et conclusion

Notre principal objectif dans cette thèse était de fournir un modèle d'interaction dans les bases de données déductives qui préserve la sécurité de la base de données tout en minimisant le nombre de refus de réponses. Nous nous sommes penchés sur la construction d'un mécanisme sûr et précis pour ces bases de données. Vu que le rôle d'une base de données déductive est de faire des déductions basées sur des règles et des faits qui y sont stockés, nous avons utilisé la logique du premier ordre, un langage comme Datalog, qui est utilisé pour décrire des faits, des règles et des requêtes.

Nous avons commencé par présenter, dans le chapitre 2, les éléments de base de la sécurité. Comme nous nous sommes principalement intéressés à la confidentialité, il était naturel d'étudier les politiques de sécurité qui traitent de cet aspect. Nous avons examiné deux extensions des politiques de confidentialité, à savoir les politiques de non-inférence et de non-déducibilité parce qu'elles abordent la question de la confidentialité en se basant sur les traces d'exécution dans le système. Nous nous sommes ensuite penchés sur la façon d'implémenter la sécurité dans les systèmes. Nous avons mentionné trois méthodes, à savoir, le contrôle d'accès, le flux de l'information et le contrôle de l'inférence, en omettant volontairement la cryptographie. Les mécanismes de contrôle d'accès décrivent les conditions dans lesquelles un système est sécurisé alors que les mécanismes de vérification des flux de l'information assure que l'information circulant dans le système est conforme à sa politique de sécurité.

Dans le chapitre 3, nous avons présenté la programmation logique du premier ordre. Nous avons passé en revue les notions de substitutions, unification, SLD réfutation et des arbres SLD, et nous avons montré les limites de ces résolutions vu que la réfutation d'un but peut être incomplète. Pour remédier à ce problème, nous avons examiné les techniques de détection de boucles qui modifient le mécanisme de calcul en y ajoutant une capacité de stopper la réfutation se basant sur une occurrence d'un sous-but similaire dans la SLD-dérivation. Nous avons terminé par une vue d'ensemble du contrôle de flux et de l'analyse de dépendance dans la programmation logique, tout en soulignant le fait que ces analyses ne peuvent se substituer à la notion de flux de l'information connue dans les systèmes de sécurité.

Pour atteindre notre objectif, et vu que la notion de flux de l'information connue dans les systèmes de sécurité n'a jamais été abordée en programmation logique, nous avons proposé, dans le chapitre 4, trois définitions de ce qui pourrait être un flux de l'information en programmation logique. Ces définitions, basées sur les traces, correspondent à ce qui pourrait être observé par un utilisateur quand il lance une requête sur un programme logique. Ces définitions sont basées respectivement sur la réussite / échec, les substitutions réponses et la bisimulation.

Nous avons exploré les liens entre ces définitions et nous avons prouvé la non-transitivité du flux. Pour décider de l'existence d'un flux relativement à un programme logique et un but, nous avons fourni des résultats de complexité et nous avons montré l'indécidabilité du flux pour les programmes logiques en général et la décidabilité pour d'autres types de programmes logiques. Le problème de décider de l'existence du flux dans les programmes Datalog relativement aux définitions de flux basées sur réussite / échec et substitutions réponses est EXPTIME-complet. Le problème est devenu dans $\Delta 2P$ pour les programmes Datalog binaires hiérarchiques et ceci relativement à la première définition du flux basée sur réussite / échec. Nous avons consacré le chapitre 5 aux bisimulation de buts. Nous avons montré que décider si deux buts sont bisimilaires est indécidable dans le cas général, et qu'il devient décidable en 2EXPTIME pour les programmes logiques hiérarchiques et restricted.

Enfin dans le chapitre 6, en utilisant tous les résultats obtenus, nous avons étendu la notion de flux à la notion d'indiscernabilité. Une définition du niveau des buts logiques a été proposée. Nous avons reformulé les notions de mécanismes de protection, de mécanismes sûrs, et de mécanismes précis et les notions de politiques de confidentialité dans le cadre de la programmation logique. Nous avons enfin établi (nous l'avons formellement prouvé) un mécanisme sûr pour les bases de données déductives en utilisant les notions précédemment exposées.

Travaux futurs

Les travaux futurs peuvent être adressés en trois principaux axes de recherche: le premier vers le fait de décider de l'existence du flux de l'information dans d'autres types de programmes logiques et de décider de la bisimulation dans les programmes Datalog, le second vers l'implémentation du flux de l'information dans un cadre de programmation logique, et le troisième vers l'intégration de notre mécanisme de sécurité dans les bases de données en temps réel.

Travaux formels

À la suite des résultats déjà obtenus dans cette thèse, nous pouvons penser à trouver et à explorer d'autres types de programmes logiques pour lesquels la question de l'existence du flux pourraient être décidable aussi. En attendant, nous enquêtons toujours sur la décidabilité et la complexité de l'existence du flux relativement à la bisimulation dans les programmes Datalog et ceci sans tenir en compte des techniques de détection de boucles.

Implémentation

Nous avons présenté dans les chapitres 4 et 5, plusieurs définitions de ce qui pourrait être un flux d'information dans les programmes logiques. Nous avons présenté des algorithmes sur la façon de décider de ce flux et ceci dans des contextes différents. L'implémentation de ces algorithmes constitue un moyen de consolider nos résultats. Bien que cette implémentation soit relativement facile pour les deux premières définitions proposées du flux, elle reste plus difficile dans le cas de la bisimulation. Rappelons que pour décider si deux buts sont bisimilaires, nous avons besoin de garder une trace dans la mémoire de tous les résolvants déjà rencontrés dans la dérivation, même si les branches contenues dans l'arbre SLD ne sont pas infinies.

Même si l'implémentation est tout à fait possible, celle-ci risque de ne pas être très intéressante vu la quantité d'espace nécessaire pour décider de l'existence du flux. Nous pensons qu'une meilleure implémentation pourrait être réalisée en utilisant les techniques exposées dans la section 3.2.9 ou bien en menant d'abord une recherche approfondie sur la façon d'assouplir les conditions pour décider de la bisimulation de deux buts tout en conservant le fait que nos algorithmes soient corrects et complets.

Les bases de données en temps réel

Dans le chapitre 6, nous avons élaboré un mécanisme de sécurité sûr et précis pour les bases de données déductives. Cependant, comme les bases de données en temps réel bénéficient d'une attention de plus en plus accrue, il serait tentant de s'intéresser à la possibilité d'intégrer notre mécanisme de sécurité dans ces bases afin de renforcer leurs politiques de sécurité. Il serait également intéressant de réfléchir aux changements à apporter à notre système afin d'ajuster la sécurité aux exigences du temps réel.