Imperial College London
Department of Computing

Distributed Abductive Reasoning: Theory, Implementation and Application

Jiefei Ma

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College, December, 2011

Declaration of Originality

I, Jiefei Ma, declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given in the bibliography.

Abstract

Abductive reasoning is a powerful logic inference mechanism that allows assumptions to be made during answer computation for a query, and thus is suitable for reasoning over incomplete knowledge. Multi-agent hypothetical reasoning is the application of abduction in a distributed setting, where each computational agent has its local knowledge representing partial world and the union of all agents' knowledge is still incomplete. It is different from simple distributed query processing because the assumptions made by the agents must also be consistent with global constraints.

Multi-agent hypothetical reasoning has many potential applications, such as collaborative planning and scheduling, distributed diagnosis and cognitive perception. Many of these applications require the representation of arithmetic constraints in their problem specifications as well as constraint satisfaction support during the computation. In addition, some applications may have confidentiality concerns as restrictions on the information that can be exchanged between the agents during their collaboration. Although a limited number of distributed abductive systems have been developed, none of them is generic enough to support the above requirements.

In this thesis we develop, in the spirit of Logic Programming, a generic and extensible distributed abductive system that has the potential to target a wide range of distributed problem solving applications. The underlying distributed inference algorithm incorporates constraint satisfaction and allows non-ground conditional answers to be computed. Its soundness and completeness have been proved. The algorithm is customisable in that different inference and coordination strategies (such as goal selection and agent selection strategies) can be adopted while maintaining correctness. A customisation that supports confidentiality during problem solving has been developed, and is used in application domains such as distributed security policy analysis. Finally, for evaluation purposes, a flexible experimental environment has been built for automatically generating different classes of distributed abductive constraint logic programs. This environment has been used to conduct empirical investigation of the performance of the customised system.

Acknowledgements

I would like to express my deepest gratitude to my co-supervisors Dr. Alessandra Russo and Dr. Krysia Broda. This thesis would not have been possible without their advice, encouragement, support and guidance. I am also very thankful to my second supervisor Dr. Emil Lupu, whose constructive criticism and invaluable insights were very inspiring and motivational. I would like to express my wholehearted gratitude to all of my supervisors and Professor Morris Sloman for organising the financial support required for me to pursue my PhD studies.

I would also like to thank my PhD examiners Professor Marek Sergot (internal) and Professor Antonis Kakas (external) from University of Cyprus for their very constructive criticism and detailed comments that help improve this thesis.

I am grateful to the academic staff and researchers in the Department of Computing for their guidance and technical advice, in particular Professor Keith Clark, Dr. Francesca Toni, Professor Murray Shanahan and Professor Morris Sloman. I am also grateful to Professor Ken Satoh and Professor Hiroshi Hosobe of the National Institute of Informatics (Japan), and Dr. Jorge Lobo and Dr. Seraphin Calo of the IBM T. J. Watson Research Center (USA) for their helpful advice and discussions related to my PhD studies.

Many thanks to my fellow students and friends who have been part of Office 502 or in the Policy research group, namely Dalal Alrajeh, Rudi Ball, Themis Bourdenas, Domenico Corapi, Robert Craven, Luke Dickens, Changyu Dong, Anandha Gopalan, Sye-loong Keoh, Jim Kuo, Srdjan Marinovic, Encrico Scalavino, Alberto Schaeffer-Filho, Vrizlynn Thing and Ryan Wishart, for their friendship and support.

My most profound gratitude goes to my parents, MA Xincai and SHI Baolan, to whom this thesis is dedicated. They provided me with constant care, encouragement and inspiration to never give up.

The work presented in this thesis was partly funded by: the Imperial College London Deputy Rector's Award 2007–2010; the International Technology Alliance sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-06-3-0001.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivations

Abductive reasoning is a powerful mechanism for reasoning with incomplete knowledge. It can be viewed as a process of finding *explanation* for an *observation*, or as a process of generating *conditional proofs* for a *conclusion*. The conditions of the proof are *abduced* assumptions that, together with a given knowledge-base, imply the conclusion of the proof. Abduced assumptions can be viewed, within the context of a knowledge-base, as an explanation of the conclusion. Abductive Logic Programming (ALP) [KKT92] is the combination of abductive reasoning and Logic Programming, in which the knowledge-base is a logic program paired with a set of integrity constraints – queries that must never succeed – used to define constraints upon the assumptions that can be abduced. ALP, as a general *knowledge-based* problem solving method, has been used in a wide range of real world applications: in cognitive robotics [Sha05] for abducing higher-level descriptions of the world from sense data, in planning [Esh88, Sha00] for abducing action events that would result in a desired state of the world using a knowledge-base about effects of actions on features of the world, in diagnostic analysis of system specifications [CLM$^+$09, BN08] for abducing system traces that would lead to property violations, and many others [Poo88, MBD94, RMNK02].

These application problems have a corresponding formulation in the multi-agent context. For example, several robots may collaboratively try to abduce an agreed higher-level description of the state of the world, from their separate sense data, that is consistent with their collective constraints on the abduced information. Similarly, parties of a coalition network supporting joint-rescue operations for an earthquake-hit zone may collaboratively reason about the dependency of their policies to abduce circumstances of policy violations, which would obstruct the success of their rescue operation. In these *distributed knowledge-based* problem solving tasks each agent has an *incomplete knowledge* of the problem domain. A robot's sense data provides only a partial description of the state of the world, and policies of a rescue party constitute only part of the knowledge involved in a collective rescue operation. Communication overheads and confidentiality concerns often prevent solutions for these tasks being engineered by centralising the agents' knowledge into a single computation point and using existing ALP proof procedures [KM90b, DS98, FK97, KMM00, EMS+04b, KvND01]. Thus, distributed algorithms for ALP need to be developed for solving problems that require *distributed abductive reasoning*.

Distributed Abductive reasoning can be seen as a sub-type of multi-agent reasoning [Dur01], which has two basic characteristics: 1. each agent is an entity (or module) that has only partial knowledge of the world and is capable of performing reasoning individually; 2. there are some global constraints that need to be satisfied by the reasoning results of the agents. The first characteristic implies that these agents need to cooperate and exchange their local reasoning results, whereas the second characteristic implies that the coordination between the agent reasoning and answer sharing is a must. The main difference between distributed abductive reasoning and other multi-agent reasoning is that the union of all the agent knowledge may be incomplete. Thus, agents can make *shared assumptions* during their reasoning, and exchange *conditional answers* during cooperation. In addition, the shared assumptions must also satisfy the global constraints. These properties of distributed abductive reasoning give new challenges in agent coordination.

Multi-agent reasoning has been widely studied and several logic-based systems have been developed [CLM+03, II04, ACG+06, HSC07]. However, most of these systems do not consider the special properties of distributed abductive reasoning. ALIAS [CLM+03] (stands for Abductive LogIc AgentS) was the first system developed as the study of abductive reasoning to a multi-agent setting. In ALIAS, the agent reasoning is based on an distributed algorithm extended from the Kakas-Mancarella abductive proof procedure, and the agent interactions such as cooperation and answer sharing are controlled through a specifically designed coordination language called LAILA [CLMT01]. Although ALIAS has been shown to be applicable to a number of problems [Cia02, CT04], it has several limitations. First, it assumes *local consistency* of global constraints – the constraints only need to be satisfied by the shared assumptions with respect to each agent's partial knowledge, instead of the union of all agent knowledge. Secondly, the distributed abductive algorithm cannot generate non-ground answers and lacks arithmetic constraint satisfaction support, which is necessary for many abductive reasoning applications, such as planning involving time and cost, or reasoning over infinite domains. Thirdly, the aspect of confidentiality is not considered, and thus there is no restriction on what information agents can exchange during cooperation and answer sharing. Finally, since the agent interactions are explicitly specified by the LAILA language as part of an agent's knowledge base, the system must assume a fixed group of agents.

The main focus of this thesis is to develop generic agent-based distributed abductive reasoning system(s) that satisfy some or all of the following requirements:

- *Consider global consistency*: this means that the shared assumptions made by any agent must satisfy the integrity constraints by all agents with respect to the union of all agent knowledge.

- *Support Open Agent Groups*: this means that the group of agents in the system may change over time, even during agent cooperation. The distributed abductive reasoning must guarantee correctness of the final answer even if an agent joins or leaves the system during the computation of the answer.

- *Support Constraint Satisfaction*: this means that the knowledge representation language should be expressive enough to allow problem domains involving arithmetic constraints to be specified, and the distributed algorithm should support constraint satisfaction during inference.

- *Respect Confidentiality*: this means that *public* and *private* knowledge of the agents can be distinguished and that during the inference process no private knowledge of an agent can be disclosed to others.

1.2 Summary of Contributions

The contributions of this thesis can be summarised into three parts:

1. **Theory**: We have developed two distributed abductive reasoning systems, DARE and DAREC.
   (a) DARE is our early work in the study of multi-agent abductive reasoning, and is inspired by ALIAS. DARE defines the notation of distributed abductive framework, which allows distributed agent knowledge to be represented as abductive normal logic programs and integrity constraints. Similar to ALIAS, DARE deploys an algorithm that is an extension to the Kakas-Mancarella abductive proof procedure. Different from ALIAS, DARE focuses on global consistency and assumes an open group of agents. To our knowledge, DARE was the first distributed abductive reasoning system that supports both of these two properties.
   (b) Based on the experiences gained from the early study, a more powerful system DAREC (standing for DARE with Constraints) has been designed to supplant DARE. DAREC is similar to DARE in that it focuses on global consistency and extends the distributed abductive framework to distributed abductive normal *constraint* logic programs and integrity constraints. DAREC is superior to DARE in that it deploys a completely new, yet much more powerful and flexible, distributed algorithm based on the ASystem proof procedure, which supports reasoning over unbound domains and arithmetic constraints. To our knowledge, DAREC is the first distributed abductive reasoning system that can compute non-ground (conditional) answers and has (arithmetic) constraint satisfaction support. Soundness and completeness of DAREC are also proven.

(c) Finally, to focus on the confidentiality aspects and to demonstrate the flexibility of DAREC, a customisation of it called DAREC^2 (standing for DAREC with Confidentiality) has been developed. DAREC^2 inherits all properties of DAREC, and can guarantee *confidential reasoning*, i.e., no private knowledge of any agent is disclosed during or after the distributed inference process. This is achieved by two steps. First, syntactic features are added to the knowledge representation language in order to allow *public* and *private* knowledge of agents to be specified and distinguished. Secondly, special *goal selection strategies* and *agent interaction strategies*, which can affect the execution of the distributed algorithm, are developed and adopted to ensure no private information can be exchanged between agents during cooperation. The resulting system satisfies all of the requirements aforementioned in Section 1.1.

2. **Implementation:**

(a) We have produced a multi-threaded prototype of our most powerful system DAREC^2 in YAP Prolog. The distributed algorithm has been implemented as a meta-interpreter and each agent is implemented as a reasoning module. This system prototype can be used in two ways. It can be used as a stand-alone distributed abductive query processing system, i.e., reasoning modules contain distributed knowledge and respond to queries submitted to the system. Alternatively, it can be used as a decoupleable multi-purpose tool for larger multi-agent systems (MAS), e.g., each reasoning module can be embedded into an agent (with some well-known agent architecture such as BDI [RG95]) of a larger MAS to support the agent/system functionalities (e.g., distributed reasoning over BDI agents belief stores [SDDM09]).

(b) We have conducted experiments to study the performance of our distributed abductive reasoning against distributed programs with different structures and of different scales. As part of the automated test-bed we have developed a distributed abductive constraint logic program generator. This generator takes as input a set of tunable parameters and (randomly) generates as output a set of logic programs that satisfy the properties described by the input parameters. This generator is generic enough to produce example inputs for benchmarking not only our distributed abductive reasoning systems but also any other (centralised) logic programming reasoning system.

3. **Application:**

   (a) We have applied the DAREC^2 system to a real world problem of distributed security policy analysis, which is a generalisation of [CLM+09] in the distributed setting. In this problem domain, a policy-managed distributed system consists of a set of nodes, each of which has its own private policies and private domain knowledge. The analysis tasks, such as identifying conflicts in policies, need to be done in a decentralised fashion to respect confidentiality. It can be demonstrated that DAREC^2 can be used to solve these tasks seamlessly.

The research work leading to this thesis has resulted in the following publications:

- Jiefei Ma, Alessandra Russo, Krysia Broda and Keith Clark: DARE: A System for Distributed Abductive REasoning, *Journal of Autonomous Agents and Multi-Agent Systems*, 16, 271-297, 2008

- Jiefei Ma, Alessandra Russo, Krysia Broda and Keith Clark: A Dynamic System for Distributed Reasoning, *AAAI Spring Symposium, Technical Report SS-08-02*, 31-36, AAAI Press, 2008

- Robert Craven, Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil Lupu, Arosha Bandara, Seraphin Calo and Morris Sloman: Expressive Policy Analysis with Enhanced System Dynamicity, *Proceedings of the 4th International Symposium on Information, Computer and Communications Security*, 239-250, ACM, 2009

- Jiefei Ma, Alessandra Russo, Krysia Broda and Emil Lupu: Multi-agent Planning with Confidentiality (Extended Abstract), *Proceedings of the 8th International Conference on Autonomous Agents and Multi-agent Systems*, 1275-1276, 2009

- Jiefei Ma, Alessandra Russo, Krysia Broda, Hiroshi Hosobe and Ken Satoh: On the Implementation of Speculative Constraint Processing, *Post-Proceedings of the 10th International Workshop on Computational Logic in Multi-Agent Systems*, 178-195, 2009

- Jiefei Ma, Krysia Broda, Randy Goebel, Hiroshi Hosobe, Alessandra Russo and Ken Satoh: Speculative Abductive Reasoning for Hierarchical Agent Systems. *Proceedings of the 11th International Workshop on Computational Logic in Multi-Agent Systems*, 49-64, 2010

- Jiefei Ma, Alessandra Russo, Krysia Broda and Emil Lupu: Distributed abductive reasoning with constraints (Extended Abstract), *Proceedings of the 9th International Conference on Autonomous Agents and Multi-agent Systems*, 1381-1382, 2010

- Jiefei Ma, Krysia Broda, Alessandra Russo and Emil Lupu: Distributed Abductive Reasoning with Constraints, *Post-Proceedings of the 8th International Workshop on Declarative Agent Languages and Technologies*, 148-166, 2010

- Jiefei Ma, Alessandra Russo, Krysia Broda and Emil Lupu: Multi-agent Confidential Abductive Reasoning, *Technical Communications of the 27th International Conference on Logic Programming*, 175-186, 2011

1.3 Thesis Overview

This thesis is organised as follows.

Chapter 2 gives the background of abductive logic programming. Chapter 3 briefly describes our early work of developing the first distributed abductive reasoning system (DARE). We illustrate its algorithm through a distributed meeting scheduling example. We also summarise the properties and limitations of DARE. Most of the lessons we learned from developing DARE contributed to the design decisions of our new DAREC system.

Chapter 4 presents our main contribution – the general-purpose distributed abductive system DAREC. We first focus on a set of fixed agents, and give definitions of the DAREC knowledge modelling framework and the DAREC distributed reasoning algorithm (in terms of a set of inference rules). The execution of the algorithm is illustrated through an ambient intelligence example. The soundness and completeness of the algorithm are also given. We then show how a "yellow-page" directory similar to the one used in DARE can be used by DAREC to improve efficiency of the algorithm, and how the system can be extended to cope with an *open* set of agents.

Chapter 5 focuses on the confidentiality aspect of distributed abductive reasoning. We show how the flexible DAREC system is customised (into DAREC^2) to address location awareness and privacy awareness issues in the distributed knowledge modelling and to support confidential reasoning. The ambient intelligence example used in Chapter 4 is further elaborated to illustrate the extended framework and algorithm of DAREC^2. The impact of the customisations to the system's properties such as soundness and completeness, in addition to confidentiality maintenance during the execution of the algorithm, are also discussed. Finally, a Prolog-based implementation of DAREC^2 is also described.

Chapter 6 presents experimental results of the DAREC^2 system. We first describe a system that we have developed for randomly generating example knowledge bases for DAREC^2, and discuss how, by supplying different parameters, the system can also generate examples for the benchmarking of any centralised (abductive) algorithm. Although we have conducted a number of different experiments of DAREC^2 with randomly generated distributed logic programs, in this chapter we only describe and discuss three of the most interesting ones.

Chapter 7 describes a case study of distributed security policy analysis, which is an application of DAREC^2. We first describe an existing formal policy framework for modelling and analysing centralised security policies, and show how this framework is extended to model systems where policies and domain knowledge are distributed among, and are private to, different policy enforcement points. We then illustrate, through a coalition example, the usage of DAREC^2 for solving policy analysis tasks in a distributed and confidential manner.

Finally, Chapter 8 summarises the contributions of the thesis, and discusses related and future work.

Chapter 2

Background

2.1 First-Order Logic Language and Semantics

This section gives the syntax and semantics of first-order logic (FOL).

A FOL has the following basic (disjoint) elements:

- *constants*;
- *variables*;
- *function symbols* of the form f/n where f is the *function name*, and n is a positive integer called the *function arity*;
- *predicate symbols* of the form p/m where p is the *predicate name*, and m is a non-negative integer called the *predicate arity*;
- *logic connectives* $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$;
- *quantifiers* $\exists, \forall$.

By convention, constants, function names and predicate names are often strings starting with a lowercase letter, such as $a, bob, \ldots$, and variables are often strings starting with a uppercase letter, e.g., $X, Y, \ldots$.

Definition 2.1. *The* signature *(or* language*) $\mathcal{L}$ of a FOL consists of a set of constants, a set of function symbols and a set of predicate symbols.*

Given a signature, a *term* is either a constant, a variable or a function such as $f(t_1, \ldots, t_n)$ where f is a function name with arity of n and $t_1, \ldots, t_n$ are terms called the *function arguments*. In some literatures, function symbols may have zero arity. In this case functions with zero arguments are often treated as constants. A *predicate* is $p(t_1, \ldots, t_n)$ where p is a predicate name with arity of n and $t_1, \ldots, t_n$ are terms called the *predicate arguments*. Sometimes a vector of terms $t_1, \ldots, t_n$ can be abbreviated as $\vec{t}$. An *atomic formula* (or *atom* in short) is a predicate.

Definition 2.2. *Given a signature $\mathcal{L}$, a* (well-formed) formula *written in $\mathcal{L}$ is defined as follows:*

- *if A is an atom then A is a formula;*
- *if A and B are formula then so are $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ and $(A \leftrightarrow B)$;*
- *if A is a formula then so are $(\exists X.A)$ and $(\forall X.A)$ (variable X is said to be* quantified *or* bound *by a quantifier $\exists$ or $\forall$).*

The precedences (or binding powers) of the logic connectives are ordered as $\neg > \wedge > \vee > \rightarrow > \leftrightarrow$. When there is no confusion, brackets "(" and ")" may be dropped. A variable in a formula A that is not bound is called a *free* variable (of A). A formula without any free variable is *closed*; otherwise it is *open*. A closed formula is often called a *sentence*. The *universal closure* (*existential closure*) of a formula A is the sentence $\forall \vec{X}.A$ $(\exists \vec{X}.A)$ where $\vec{X}$ are the free variables of A. Sometimes it is convenient to drop the variables when they are not considered during discussions, i.e., $\forall(A)$ $(\exists(A))$. A *(logic) theory* is a set of sentences.

Definition 2.3. *An* interpretation *(or* structure*) of a FOL signature $\mathcal{L}$ has the following information:*

- *a non-empty set of objects called the* domain *(DOM);*
- *for every constant c in $\mathcal{L}$, an object in DOM;*

- *for every function symbol f/n in $\mathcal{L}$, a function that maps n objects to one object in DOM;*
- *for every predicate symbol p/n in L, a relation between n objects in DOM.*

Given an interpretation, the meaning of a term is an object in the domain, and the meaning of a closed formula is a truth value. An open formula has no meaning. An *assignment* (or *valuation*) is a function that allocates an object in the domain to each free variable. We denote the allocation of object o to a variable X with $X \mapsto o$.

Definition 2.4. *Given an interpretation I with domain DOM for a signature $\mathcal{L}$ and an assignment φ, the statement "a formula F in $\mathcal{L}$ is* true *with respect to I and φ" (denoted with $I, \varphi \Vdash F$) is defined as follows:*

- *$I, \varphi \Vdash p(t_1, \ldots, t_n)$ if and only if the relation of p between $(t_1, \ldots, t_n)$ is in DOM;*
- *$I, \varphi \Vdash \neg A$ if and only if not $I, \varphi \Vdash A$, i.e., $I, \varphi \nVdash A$;*
- *$I, \varphi \Vdash A \wedge B$ if and only if $I, \varphi \Vdash A$ and $I, \varphi \Vdash B$;*
- *$I, \varphi \Vdash A \vee B$ if and only if $I, \varphi \Vdash A$ or $I, \varphi \Vdash B$;*
- *$I, \varphi \Vdash A \rightarrow B$ if and only if $I, \varphi \Vdash B$ whenever $I, \varphi \Vdash A$;*
- *$I, \varphi \Vdash A \leftrightarrow B$ if and only if $I, \varphi \Vdash A \rightarrow B$ and $I, \varphi \Vdash B \rightarrow A$;*
- *$I, \varphi \Vdash \forall X.A$ if and only if $I, \varphi[X \mapsto o] \Vdash A$ for every object o in DOM;*
- *$I, \varphi \Vdash \exists X.A$ if and only if $I, \varphi[X \mapsto o] \Vdash A$ for some object o in DOM.*

A sentence F is *satisfiable* if there exists some interpretation in which it is true. F is *unsatisfiable* if there exists no interpretation in which it is true. F is *valid* if it is true in all possible interpretations, in which case F is called a *tautology.*

Definition 2.5. *A* model *of a sentence F is an interpretation in which F is true. A* model *of a theory $\mathcal{T}$ is an interpretation in which all the sentences in $\mathcal{T}$ are true.*

Definition 2.6. *A sentence F is said to be* entailed *by a theory $\mathcal{T}$ (denoted with $\mathcal{T} \models F$) if and only if all the models of $\mathcal{T}$ are also the models of F. $\models$ is called the* logical entailment *and $\not\models$ is the contrary.*

2.1.1 Herbrand Models

Given a theory $\mathcal{T}$ with a signature $\mathcal{L}$, there can be infinitely many possible domains and interpretations. There is a significant type of interpretation called the *Herbrand interpretation*. Let C, F and P be the set of constants, set of function symbols and set of predicate symbols of $\mathcal{L}$, respectively. The *Herbrand universe* ($\mathcal{U}$) is the set of **all** the ground terms constructed from C and F. When F is not empty, $\mathcal{U}$ is infinite. For example, let c be the only constant in C and $f/1$ be the only function symbol in F, then $\mathcal{U} = \{c, f(c), f(f(c)), \ldots\}$. The *Herbrand base* ($\mathcal{B}$) is the (possibly infinite) set of atoms constructed from $\mathcal{U}$ and P. For example, let $p/1$ be the only predicate symbol in P, then $\mathcal{B} = \{p(c), p(f(c)), p(f(f(c))), \ldots\}$.

Definition 2.7. *A* Herbrand interpretation *($\mathcal{I}$) based on the* Herbrand base *of a given signature $\mathcal{L}$ is an interpretation such that:*

1. *each constant is assigned to itself;*
2. *each function is assigned to its syntactic equivalence;*
3. *there is no restriction on how $\mathcal{I}$ may interpret the predicates.*

Usually, a Herbrand interpretation is simply represented as a set of atoms that are assigned to be *true* (i.e., an atom that is not in the set is assigned to be *false*). A Herbrand interpretation that is a model of $\mathcal{T}$ is called the *Herbrand model* of $\mathcal{T}$. The following theorem shows the significance of Herbrand models.

Theorem 2.1. *[vEK76] A theory $\mathcal{T}$ is satisfiable if and only if there is a Herbrand interpretation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{T}$.*

2.2 Logic Programming

2.2.1 Syntax

In Logic Programming (LP), we only consider a special type of sentence called a *clause*. A *literal* is either an atom A or the negation of an atom $\neg A$. The former is called a *positive* literal and the latter is called a *negative* literal. The meaning of *negation* ($\neg$) will be defined later in Section 2.2.2. A clause has the following form

$$\forall(H \leftarrow L_1 \wedge \cdots \wedge \neg L_n) \qquad (n \geq 0)$$

where H is an atom, and each L_i is a literal. Its notation can be conveniently written as a *rule* by dropping the universal quantifier and replacing $\wedge$ with ",", e.g.,

$$H \leftarrow L_1, \ldots, L_n \qquad (n \geq 0)$$

where H is called the *head* and $L_1, \ldots, L_n$ is called the *body*. A rule with empty body (i.e., $n = 0$) is called a *fact*. A rule without the head is a *denial*. A denial with non-empty body (i.e., $n > 0$) is called an *integrity constraint*, e.g.,

$$\leftarrow L_1, \ldots, L_n \qquad (n > 0)$$

A denial with empty body (i.e., $n = 0$) is equivalent to *falsity* (i.e., $\perp$).

Let R be a rule. We denote its head with *head*(R) and its body with *body*(R). In many literatures, people tend to distinguish between a rule with a head and a denial. In this thesis, unless stated otherwise, a rule usually refers to a rule with a head.

A *definite clause (rule, denial)* is a clause (rule, denial) whose body contains only positive literals. A *normal clause (rule, denial)* is a clause (rule, denial) whose body may contain negative literals.

Definition 2.8. *A* definite logic program *is a finite set of definite rules.*

Definition 2.9. *A* normal logic program *is a finite set of normal rules.*

It is obvious that any definite logic program is also a normal logic program. Thus, in the rest of this thesis "logic program" usually refers to a normal logic program.

A term or a clause is *ground* if it does not contain any variable. A (ground) instance of a clause is obtained by replacing all of its variables with ground terms. The ground instance of a given logic program Π, denoted with $ground(\Pi)$, is a (possibly infinite) set of all the possible ground instances of its clauses.

2.2.2 Semantics

Note that in a clause, though the symbol " $\leftarrow$ " is used, it does not always correspond to the (reverse of) *classical implication*. It is definitional (e.g., the body literals defines the head atom) and can be interpreted in different ways. Thus, we usually call a rule R a definition of p/n if p/n is the predicate symbol of $head(R)$. In addition, the negation "$\neg$" does not always correspond to the *classical negation*. With the *closed world assumption* (CWA) it is usually interpreted as *negation as failure* (NAF). In *extended logic programs* it is even possible to allow both classical negations and NAFs. But this type of logic programs are not considered in this thesis. Therefore, the semantics of a given logic program depends on how these symbols are interpreted. This section summarises some popular semantics of logic programs.

Semantics for Definite Logic Programs

Let us first consider definite logic programs. If " $\leftarrow''$ is considered as classical implication, then a definite logic program is a logic theory, and Herbrand models can be used. However, for every definite logic program there may be many Herbrand models, and we need to choose only one from them to represent the semantics of the program. This is the *minimal Herbrand model*, which is defined as follows.

Theorem 2.2. *[vEK76] Every definite logic program Π has at least one Herbrand model, which is equivalent to the Herbrand base.*

Theorem 2.3 (Model Intersection)**.** *[vEK76] If $\mathcal{M}_1$ and $\mathcal{M}_2$ are two Herbrand models for a (definite) logic program Π, then $\mathcal{M}_1 \cap \mathcal{M}_2$ is also a Herbrand model of Π.*

Definition 2.10 (Minimal Herbrand Model)**.** *Let Π be a definite logic program, a Herbrand model $\mathcal{M}$ of Π is said to be* minimal *if and only if there does not exists a Herbrand model $\mathcal{M}'$ of Π such that $\mathcal{M}' \subset \mathcal{M}$.*

The next theorem (following from Theorem 2.2 and Theorem 2.3) describes an important property for definite logic programs:

Theorem 2.4. *[vEK76] Every definite logic program Π has a* unique *minimal Herbrand model, which is equivalent to the intersection of all of its Herbrand models.*

Apt et. al. [AvE82] have proposed a *fixed-point approach* for computing the unique minimal Herbrand model for any given logic program. The key idea is to iteratively compute the set of atoms that are *implied* by the rules and facts in the logic program.

Definition 2.11 (Immediate Consequence Operator)**.** *Given a definite logic program Π, the immediate consequence operator T_Π is a function on the Herbrand interpretations of Π such that*

$$T_\Pi(\mathcal{I}) = \{H \mid H \leftarrow A_1, \ldots, A_n \in \mathit{ground}(\Pi) \ \texttt{and}\ \{A_1, \ldots, A_n\} \subseteq \mathcal{I}\}$$

If we start an iteration of the application of T_Π with the Herbrand interpretation $\emptyset$ (i.e., all atoms being assigned to false), then we can obtain a sequence of interpretations $\emptyset, T_\Pi(\emptyset), T_\Pi(T_\Pi(\emptyset)), \ldots,$ which can be enumerated with a standard notation:

$$\begin{array}{rcl} T_\Pi \uparrow^0 & \equiv & \emptyset \\ T_\Pi \uparrow^{i+1} & \equiv & T_\Pi(T_\Pi \uparrow^i) \end{array}$$

It can be shown [AvE82] that there is always a least fixed-point $\mathcal{I}^w$ such that $T_\Pi(\mathcal{I}^w) = \mathcal{I}^w$, and let $T_\Pi \uparrow^\omega \equiv \bigcup_{i=0}^{\infty} T_\Pi \uparrow^i$, then $T_\Pi \uparrow^\omega$ is in fact equivalent to the unique minimal Herbrand model of Π. We call this least fixed-point the *least Herbrand model* (LHM) of Π and denote it with $lhm(\Pi)$:

Theorem 2.5. *[AvE82] Let Π be a definite logic program and $lhm(\Pi)$ its least Herbrand model, then:*

- *$lhm(\Pi)$ is the least Herbrand interpretation such that $T_\Pi(lhm(\Pi)) = lhm(\Pi)$;*
- *$lhm(\Pi) = T_\Pi \uparrow^\omega$*

Semantics for Normal Logic Programs

Normal logic programs allow negation literals in the body of a rule. The immediate consequence operator can be extended such that

$$T'_\Pi(\mathcal{I}) = \{H \mid H \leftarrow A_1, \dots, A_n, \neg B_1, \dots, \neg B_m \in ground(\Pi) \texttt{ and }$$
$$\{A_1, \dots, A_n\} \subseteq \mathcal{I} \texttt{ and } \{B_1, \dots, B_m\} \cap \mathcal{I} = \emptyset\}$$

However, while the least Herbrand model (fixed-point) semantics is adequate for definite logic programs, it does not work for *all* normal logic programs. For example, the following normal logic program

$$\Pi_1 = \left\{ \begin{array}{c} p \leftarrow \neg q. \\ q \leftarrow \neg p. \end{array} \right\}$$

has two minimal Herbrand models $\{p\}$ and $\{q\}$ but it does not have a least Herbrand model (LHM). Moreover, the fixed-point computation does not terminate to give any of these (in fact, the computation oscillates between $\{\ \}$ and $\{p, q\}$).

Apt et. al. [ABW88] have identified a special class of (normal) logic programs called *stratified* (normal logic) programs, which always have LHMs.

Definition 2.12. *A normal logic program Π is* stratified *if and only if there exists a function v which maps each predicate (symbol) P of Π to a natural number such that for every rule R in Π, let P_h be the predicate of the head of R:*

- *if P_b is the predicate of a positive body literal of R, then $v(P_h) \geq v(P_b)$;*
- *if P_b is the predicate of a negative body literal of R, then $v(P_h) > v(P_b)$.*

As a remark, all definite logic programs are stratified (e.g., we can assign the same value for all the predicates). With the predicate ordering function v, a logic program Π can be partitioned into disjoint sets $\Pi_0 \cup \cdots \cup \Pi_n$ such that for each rule R in Π_i, let P be the head predicate of R, then $v(P) = i$.

Definition 2.13 (Fixed-point Semantics for Stratified Logic Programs)**.** *Let $\Pi = \Pi_0 \cup \cdots \cup \Pi_n$ be a stratified logic program, then*

$$\begin{aligned} M_0 &= T'_{\Pi_0} \uparrow^{\omega} (\emptyset) \\ &\vdots \\ M_n &= T'_{\Pi_n} \uparrow^{\omega} (M_{n-1}) \end{aligned}$$

and $lhm(\Pi) = M_n$.

For example, the logic program

$$\Pi_2 = \left\{ \begin{array}{c} p \leftarrow \neg q. \\ q \leftarrow \neg r. \end{array} \right\}$$

is stratified (i.e., $v(p) = 2$, $v(q) = 1$ and $v(r) = 0$), and it has a LHM of $\{q\}$ (e.g., $M_0 = \emptyset$, $M_1 = \{q\}$, $M_2 = \{q\}$).

However, sometimes even if a logic program is not stratified, it may still have a LHM. For example, the logic program

$$\Pi_3 = \left\{ \begin{array}{c} p(1) \leftarrow \neg q(1). \\ q(1) \leftarrow \neg p(2). \end{array} \right\}$$

is not stratified, but it has a LHM of $\{q(1)\}$. A more relaxed class of logic programs called the *locally stratified* programs can be defined similarly to stratified programs by using a function v' which assigns the ordering to the (ground) predicate instances instead of to the predicate symbols. For example, Π_3 is locally stratified as $v'(p(1)) = 2$, $v'(q(1)) = 1$ and $v'(p(2)) = 0$. As a remark, all stratified programs are locally stratified (but not vice versa).

For logic programs that are not stratified or locally stratified, fixed-point semantics does not work. We summarise below some of the most popular semantics for logic programs with negations.

Clark Completion In the *Clark Completion* semantics [Cla78], "$\neg$" in a logic program is interpreted as *negation as failure*, which means $\neg p$ *is true (or* p *is false) if every possible proof of* p *fails in finite time.* The semantics of a logic program Π is given by a logic theory obtained by the *(Clark) completion* of Π, usually denoted with $comp(\Pi)$. Let $\vec{X} = \vec{t}$ denote a list of equalities between the elements in a list of variables $\vec{X}$ and a list of terms $\vec{t}$ of equal length n, i.e., $\vec{X} = \vec{t}$ is $\bigwedge_{i \in [1,n]} X_i = t_i$. Let a clause be in a form of $p(\vec{t}) \leftarrow F$ where F represents its body. The completion of a logic program (with signature $\mathcal{L}$) is done as follows:

1. for every predicate p/n in $\mathcal{L}$ that has at least one definition in Π, let

$$p(\vec{t_1}) \leftarrow F_1 \tag{2.1}$$

$$\vdots \tag{2.2}$$

$$p(\vec{t_m}) \leftarrow F_m \tag{2.3}$$

   be all the definitions for p/n, the completion of p/n is the following sentence:

$$\forall \vec{X}.p(\vec{X}) \leftrightarrow (\exists \vec{Y_1}.\vec{X} = \vec{t_1} \wedge F'_1) \vee \cdots \vee (\exists \vec{Y_m}.\vec{X} = \vec{t_m} \wedge F'_m)$$

   where each $\vec{Y_i}$ is the set of variables appearing in F_i but not in $\vec{t_i}$, and each F'_i is obtained by replacing "," with $\wedge$ in F_i. Note that each "$\neg$" in such sentence is interpreted as the classical negation and each " $\leftrightarrow$ " is interpreted as the classical *if and only if*;

2. for every other predicate p/n in $\mathcal{L}$ that does not have a definition in Π, the completion

of p/n is the following sentence:

$$\forall \vec{X}. \neg p(\vec{X})$$

3. *comp*(Π) contains all the completions of the predicates in $\mathcal{L}$, a set of the *Clark Equality Theory* (CET) axioms [Cla78], and nothing else.

Since *comp*(Π) is a logic theory, a Herbrand model of *comp*(Π) is a model of *comp*(Π). However, given a completed program there may not be a unique minimal Herbrand model. For example, the completed program of Π_1 is

$$comp(\Pi_1) = \left\{ \begin{array}{c} p \leftrightarrow \neg q. \\ q \leftrightarrow \neg p. \end{array} \right\}$$

and there are two minimal Herbrand models: $\{p\}$ and $\{q\}$.

Stable Model The *stable model* semantics [GL88] is defined by the means of a *reduct* for the ground instance of a logic program. Let Λ be a set of (ground) atoms and Π be a (ground) logic program, the reduct of Π, denoted with Π^Λ, is obtained from Π as follows:

1. remove any clause $H \leftarrow A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m$ in Π where $\{B_1, \ldots, B_m\} \cap \Lambda \neq \emptyset$;
2. remove all the negative body literals from the remaining clauses in Π

Thus, Π^Λ is a definite logic program. Λ is a *stable model* of Π if and only if $\Lambda = lhm(\Pi^\Lambda)$.

Again, it is possible that a logic program has two minimal stable models. For example, both $\{p\}$ and $\{q\}$ are stable models of Π_1.

3-valued Semantics In the Clark completion semantics, the models of a completed logic program are in 2-valued logic, i.e., each atom can be either *true* (**t**) or *false* (**f**). Fitting [Fit85] studied the models of the same completed program in 3-valued logic, where an atom may have a third value called *unknown* (**u**). The 3-valued logic used by Fitting was one proposed by

$\wedge$	t	f	u
t	t	f	u
f	f	f	f
u	u	f	u

$\vee$	t	f	u
t	t	t	t
f	t	f	u
u	t	u	u

$\leftrightarrow$	t	f	u
t	t	f	u
f	f	f	u
u	u	u	u

Table 2.1: Kleene's 3-Valued Logic

$\leftrightarrow_k$	t	f	u
t	t	f	u
f	f	t	u
u	u	u	u

$\leftrightarrow_s$	t	f	u
t	t	f	f
f	f	t	f
u	f	f	t

Table 2.2: Kleene's Equivalence $\leftrightarrow_k$ vs. Strong Equivalence $\leftrightarrow_s$

Kleene [Kle52]. In the Kleene's 3-valued logic, the truth table of the logic connectives " $\wedge$ ", " $\vee$ " and " $\leftrightarrow$ " are shown in Table 2.1.

However, in a given completed logic program, " $\leftrightarrow''$ is interpreted as the *strong equivalence* instead of the *Kleene's equivalence* (See Table 2.2).

Fitting [Fit85] and Kunen [Kun87] also showed that all such models were fixed points of a 3-valued immediate consequence operator. An interpretation of a logic program Π is usually represented by a pair of two sets $\mathcal{I} = \langle \overline{T}, \overline{F} \rangle$, where $\overline{T}$ is a set of atoms that are assigned **t** and $\overline{F}$ is a set of atoms that are assigned **f**. Thus, let $\mathcal{B}_\Pi$ be the Herbrand base of Π, then all the atoms in $\mathcal{B}_\Pi - (\overline{T} \cup \overline{F})$ are assigned **u**. A positive literal A is true in $\mathcal{I}$ if and only if $A \in \overline{T}$, and is false if and only if $A \in \overline{F}$; otherwise it is unknown. A negative literal $\neg A$ is true in $\mathcal{I}$ if and only if $A \in \overline{F}$, and is false if and only if $A \in \overline{T}$; otherwise it is unknown.

Definition 2.14 (3-valued Immediate Consequence Operator). *Given a normal logic program* Π, *the 3-valued operator* T^3_Π *is a function on the 3-valued interpretation* $\mathcal{I} = \langle \overline{T}, \overline{F} \rangle$ *of* Π *such that*

$$T^3_\Pi(\langle \overline{T}, \overline{F} \rangle) = \left\langle \overline{T}' \overline{F}' \right\rangle$$

where

$$\overline{T}' = \{A \mid \textit{there is a clause } A \leftarrow L_1, \ldots, L_n \in \textit{ground}(\Pi) \textit{ such that all } L_i \textit{ are true in } \mathcal{I}\}$$

and

$$\overline{F}' = \{A \mid \textit{for every clause } A \leftarrow L_1, \ldots, L_n \in \textit{ground}(\Pi) \textit{ at least one } L_i \textit{ is false in } \mathcal{I}\}$$

Let $\langle \overline{T}_1, \overline{F}_1 \rangle \cup \langle \overline{T}_2, \overline{F}_2 \rangle$ denote $\langle \overline{T}_1 \cup \overline{T}_2, \overline{F}_1 \cup \overline{F}_2 \rangle$. The 3-valued fixed-point interpretation of any given normal logic program Π can be computed as:

$$\begin{aligned} T^3_\Pi \uparrow^0 &\equiv \langle \emptyset, \emptyset \rangle \; (\textit{i.e., all atoms are initially unknown}) \\ T^3_\Pi \uparrow^{i+1} &\equiv T^3_\Pi(T^3_\Pi \uparrow^i) \end{aligned}$$

and $T^3_\Pi \uparrow^\omega \equiv \bigcup_{i=0}^{\infty} T^3_\Pi \uparrow^i$ is the least 3-valued fixed-point interpretation (model) for Π. This is known as the Fitting 3-valued semantics.

Gelder, Ross and Schlipf proposed [GRS91] another type of 3-valued semantics for normal logic programs, called the *well-founded semantics*. The main difference between the two semantics is how they assign values to atoms that positively depend on themselves in a logic program. Let $A \rightarrowtail B$ denote that there is a clause in $\textit{ground}(\Pi)$ where the atom A is the head and the atom B is a positive body literal. Let $\textit{dgraph}(\textit{ground}(\Pi))$ be a directed graph where the nodes are the Herbrand base of $\textit{ground}(\Pi)$ and the edges are $\rightarrowtail$. An atom A positively depends on itself if and only if all the paths in $\textit{dgraph}(\textit{ground}(\Pi))$ starting from A will eventually visit A again. In the Fitting semantics A is assigned with $\mathbf{u}$, whereas in the well-founded semantics A is assigned with $\mathbf{f}$. For example, in the following program

$$\left\{ \begin{array}{l} p \leftarrow q. \\ q \leftarrow p. \\ r \leftarrow \neg w. \\ w \leftarrow \neg r. \end{array} \right\}$$

r and w are assigned $\mathbf{u}$ in both semantics. But p and q are assigned $\mathbf{u}$ in the Fitting semantics, and are assigned $\mathbf{f}$ in the well-founded semantics.

2.2.3 Operational Semantics of Logic Programs

A *query* for a logic program is a conjunction of literals of the form

$$\exists(L_1 \wedge \cdots \wedge L_n) \qquad (n > 0)$$

and can be conveniently written as a list, e.g.,

$$\{L_1, \ldots, L_n\}$$

Each L_i is called a *(sub-)goal.* In some literatures, it may even be written as a denial in the form of $\leftarrow L_1, \ldots, L_n$. However, it should not be confused with an integrity constraint which is also written in a denial form (i.e., a query is an existential closure of a denial, whereas an integrity constraint is a universal closure of a denial).

Given a logic program Π and a query $\mathcal{Q}$, the *querying* task is to check whether $P \models_s \mathcal{Q}$, where $\models_s$ is the logic entailment under some selected semantics.

Operationally, there are two main types of algorithms for computing the answers of a query with respect to a logic program – *top-down* and *bottom-up*. Top-down algorithms, e.g., SLDNF [AvE82], starts with the query and performs *backward* inference using the clauses in the program. In contrast, bottom-up algorithms, e.g., Answer Set Programming [Bar03], starts with a set of atoms (known as the partial model) and computes the full model by performing *forward* inference using the clauses in the program.

SLDNF stands for *Selective Linear Definite-clause resolution with Negation by Failure.* Given a query $\leftarrow L_1, \ldots, L_n$, a (successful) SLDNF computation can be described as a series of tuples $(\leftarrow L_1, \ldots, L_n, \emptyset), (G_1, \theta_1), \ldots, (\Box, \theta)$, where each G_i is a query, each θ_i is a set of variable substitutions, and $\Box$ denotes an empty query. Each tuple (G_{i+1}, θ_{i+1}) is obtained by means of the following two derivation steps:

1. the computation selects a positive literal $L_i = H$ from G_i, and resolves H with a clause

$H' \leftarrow B_1, \ldots, B_m$ where $H\theta' = H'\theta'$, then $G_{i+1} = \leftarrow (L_1, \ldots, L_{i-1}, B_1, \ldots, B_m, L_{i+1}, \ldots, L_n)\theta$, and $\theta_{i+1} = \theta_i \cdot \theta'$ (i.e., the composition of θ_i and θ');

2. the computation selects a *ground* negative literal $L_i = \neg H$ from G_i, if all the SLDNF computations for the query $\leftarrow H$ fail (finitely), then $G_{i+1} = \leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$.

After a successful SLDNF computation, the final set of substitutions θ obtained is the answer for the original given query. Note that at each step, non-ground negative literals must not be selected (in order to guarantee soundness). If at a step the current query contains only non-ground negative literals, then the whole computation *stops* and is said to be *floundered*.

2.2.4 Constraint Logic Programming

Constraint programming [Bar99] is a programming paradigm in which relations between variables are expressed as *constraints*. A constraint can be an *arithmetic constraint* (i.e., arithmetic expression connected by comparison operators) such as $X \geq Y$ and $T = T1 + 5$, or constraints connected by boolean connectives such as $(X > 4) \wedge (X < 6)$ and $\neg(X - Y \geq 2) \vee (Y < X)$. Given a set of constraints, the problem of finding the numerical assignments to the variables that can make all the constraints true is called *constraint satisfaction* (CSP). Since 1978 [Lau78] CSP has been a very hot research topic. CSP problems can be divided into different categories depending on the domains of the variables, e.g., CSP over finite domain variables, CSP over boolean variables, and CSP over variables with real/rational number domains. A large number of sophisticated and efficient algorithms (solvers) have already been developed [Kum92, Lec09] for these different type of CSPs.

Constraint Logic Programming [JM94] (CLP) is the integration of constraint programming into logic programming. In a CLP program, a rule has the form

$$H \leftarrow L_1, \ldots, L_n, C_1, \ldots, C_m$$

where $C_1, \ldots, C_m$ are constraints. During a top-down query computation process, a set of constraints is collected and is solved (incrementally) by a CLP solver, such as CLP(FD) [DCV93] for finite domain constraints, and CLP(R) [JMSY92] for constraints over real numbers.

2.3 Abductive Logic Programming

2.3.1 Abductive Reasoning

Charles Peirce has identified [Pei31] three distinguished types of reasoning: *deduction*, *abduction* and *induction*. Let $B \leftarrow A$ be a rule read as *"if A then B"*:

deduction is to *derive* the conclusion B from the given premise A and the given rule $B \leftarrow A$;

abduction is to *derive* the possible "cause" A from the given observation B and the given rule $B \leftarrow A$;

induction is to *learn* the possible rule $B \leftarrow A$ from a large example set of A–B pairs.

Informally, abduction could be viewed as the reverse process of deduction: while deduction can be used to predict the effects of given causes, abduction can be used to explain the effects by the causes [Esh88, Sha89]. Abduction is therefore particularly suitable for reasoning over *incomplete knowledge*. Consider the following example (derived from [Pea87]),

$$\left\{ \begin{array}{l} shoes\_wet \leftarrow walked\_on\_grass, grass\_wet. \\ grass\_wet \leftarrow rained. \\ grass\_wet \leftarrow sprinkler\_on. \end{array} \right\}$$

Given the above *background knowledge*, suppose that we are given the observation that the shoes are wet. With abduction we can derive the following possible explanations: either someone walked on the grass and it rained, or someone walked on the grass and the sprinkler was on. It is possible that someone walked on the grass while it rained and the sprinkler was on. But we often prefer the minimal set of sufficient explanations as the results of abduction. The explanations are also called *assumptions*, e.g., we do not know whether indeed it rained or not, but if we assume it did then we can prove the observation is correct (together with another assumption – someone indeed walked on the grass). Sometimes we want to restrict the possible explanations for an observation independently from the background knowledge. This can be done via *integrity constraints*. For example, if we add $\leftarrow walked\_on\_grass, rained$ (read as *"it is impossible that one walks on the grass during/after raining"*) to the example, then only the explanation having sprinkler on will be accepted as the result of abduction.

2.3.2 Abductive Logic Programs

Abductive Logic Programming (ALP) [KKT92, KM90a] is the combination of logic programming and abduction. The background knowledge and the integrity constraints are modelled as logic programs. The observations are modelled as queries. Explanations or assumptions are usually formed from a selected set of *ground atoms*.

Definition 2.15 (Abductive (Logic) Framework). *An abductive (logic) framework is a tuple $\langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ where*

- Π *is a normal logic program called the* background knowledge
- $\mathcal{AB}$ *is the set of* abducible predicates*;*
- $\mathcal{IC}$ *is a set of integrity constraints.*

In ALP, predicates are divided into two disjoint sets: *abducible* and *non-abducible*. An atom with abducible predicate is called an *abducible atom* (or *abducible* in short). An atom with non-abducible predicate is called a *non-abducible atom* (or *non-abducible* in short). Sometimes we may abuse the notation of $\mathcal{AB}$ to represent the set of all ground abducible atoms constructed from the abducible predicates. In most literatures, without lost of generality it is assumed that no abducible atom may appear as the head of a rule in the background knowledge, i.e., no abducible atom is defined. Any abductive framework with a defined abducible can always be transformed into one without. Consider the following example where a logic program contains a rule

$$a \leftarrow p.$$

where a is an abducible. A new logic program can be obtained by replacing every occurrence of a in the old framework with a new non-abducible a_def, and adding a new rule $a\_def \leftarrow a$. Since only non-abducibles can appear as the head of a rule, they are sometimes called *defined atoms* (or *defined* in short).

Definition 2.16 (Abductive Explanation). *Given an abductive logic framework $\langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ and a query $\mathcal{Q}$, the tuple $\langle \Delta, \theta \rangle$ is an abductive explanation for $\mathcal{Q}$ if*

1. *Δ is a set of abducible atoms and θ is a set of variable substitutions, i.e., $\Delta\theta \subseteq \mathcal{AB}$;*
2. *$\Pi \cup \Delta\theta \models_s \mathcal{Q}\theta$*
3. *$\Pi \cup \Delta\theta$ is* consistent *with $\mathcal{IC}$*

where $\models_s$ is the logical entailment under a selected semantics.

The second condition means that the abductive explanation and the background knowledge must be able to prove the query. The third condition means that the abductive explanation and the background knowledge must be consistent with the integrity constraints. Many literature defines *consistency* as $\Pi \cup \Delta \models_s \mathcal{IC}$.

2.3.3 Semantics for Abduction

Like normal logic programs, many semantics have been proposed and studied for abductive logic programs. Widely used semantics include the *generalised stable model* semantics and the *three-valued completion* semantics.

Generalised Stable Model

In [KM90c], Kakas and Mancarella proposed an extension to the stable model semantics for abductive logic programs. Given an abductive (logic) framework $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$, the *negation transformed framework* [EK89] is $\mathcal{F}^* = \langle \Pi^*, \mathcal{AB}^*, \mathcal{IC}^* \rangle$, where

- Π^* is a definite logic program obtained from Π by replacing each negative literal $\neg p(\vec{t})$ with a new positive literal $p^*(\vec{t})$ where p^* is a new predicate;
- similarly $\mathcal{IC}'$ is a set of definite integrity constraints obtained from $\mathcal{IC}$ by replacing each negative literal $\neg p(\vec{t})$ with a new positive literal $p^*(\vec{t})$, where p^* is a new predicate;
- $\mathcal{AB}^*$ extends $\mathcal{AB}$ with the set of new predicates introduced above;
- $\mathcal{IC}^*$ is the union of $\mathcal{IC}'$ and the set of integrity constraints $\leftarrow p(\vec{t}), p^*(\vec{t})$ for each p^* in $\mathcal{AB}^*$.

Definition 2.17. *Given a negation transformed framework $\mathcal{F}^* = \langle \Pi^*, \mathcal{AB}^*, \mathcal{IC}^* \rangle$ and a set of ground abducible atoms $\Delta \subseteq \mathcal{AB}^*$, a model M is a* generalised stable model *for $\mathcal{F}^*$ and Δ if and only if*

- *M is a stable model of $\Pi^* \cup \Delta$;*
- *I is true in M for each $I \in \mathcal{IC}^*$*

Three-valued Completion

Abduction through predicate completion was first introduced by Console et al. [CDT91]. In [Teu96], Teusink generalised Fitting (3-valued) semantics for abductive logic programs, based on the completion of abductive logic programs. Let $\langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ be an abductive framework. The completion of Π, denoted as $comp(\Pi)$, is obtained similar to the Clark completion except that $\forall \vec{X}. \neg a(\vec{X})$ is not in the $comp(\Pi)$ for any abducible predicate a even though a does not appear as the head of any rule in the program. The completion of abducibles is done separately.

Let $\mathcal{AB}$ be the set of all (ground) abducible atoms constructable from the abductive framework. Given a set of (ground) abducible atoms Δ, the two-valued interpretation of $\mathcal{AB}$ with respect to Δ (called the *completion* of Δ), denoted with $\mathcal{I}^{\Delta}$, is defined as

$$\mathcal{I}^{\Delta} = \{A \mid A \in \Delta\} \cup \{\neg A \mid A \in \mathcal{AB} \texttt{ and } A \notin \Delta\}$$

Definition 2.18. *Given an abductive logic framework $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ and a set of (ground) abducible atoms $\Delta \subseteq \mathcal{AB}$, let comp$(\Pi)$ be the completion for Π and $\mathcal{I}^{\Delta}$ be the completion of Δ, a model M is a* 3-valued model *for $\mathcal{F} + \Delta$ if and only if*

- *M is a three-valued (Fitting) model of comp$(\Pi) \cup \mathcal{I}^{\Delta}$;*
- *I is true in M for each $I \in \mathcal{IC}^*$*

2.3.4 Abductive Proof Procedures

Over the past two decades, various proof procedures have been developed for abductive logic framework, such as the Kakas-Mancarella proof procedure (KM) [KM90b], IFF [FK97], SLD-NFA [DS92, DS98] for normal abductive logic programs, and ACLP [KMM00], CIFF [EMS+04b, MTS+09], ASystem [KvND01] for constraint abductive logic programs. Among them, KM is probably the earliest influential one, and ASystem is known as the latest and fastest implementation. In the next section, we will briefly describe these two proof procedures.

Kakas-Mancarella Proof Procedure

The Kakas-Mancarella proof procedure (KM) is based on the generalised stable model semantics, i.e., it treats negative literals as abducibles (called the *non-base abducibles*, in contrast to the *base* abducibles with abducible predicates). In addition, it assumes the additional requirement that each integrity constraint must contain at least one abducible.

KM was first described in [KM90b] and then was re-formulated in several literatures. We describe it here following the convention used in [Ton95]. Both base and non-base abducibles are called *assumptions*. $\overline{L}$ denotes the complement of a literal L, i.e., if $L = p(\vec{t})$ then $\overline{L} = \neg p(\vec{t})$, and vice versa. The execution of KM interleaves the *abductive derivations* and the *consistent derivations*.

Let $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ be an abducible framework. An abductive derivation with respect to a *safe goal selection strategy* Ξ is a sequence $(\mathcal{G}_1, \Delta_1), \ldots, (\mathcal{G}_n, \Delta_n)$, where $\mathcal{G}_i$ $(1 \leq i \leq n)$ is the set of remaining goals and Δ_i is a set of (ground) assumptions. Ξ is *safe* in the sense that it selects an assumption goal only if it is ground. For $i = 1, \ldots, n-1$, Ξ selects a goal G from $\mathcal{G}_i$. Let $\mathcal{G}_{i^-} = \mathcal{G}_i - \{G\}$, then $(\mathcal{G}_{i+1}, \Delta_{i+1})$ is obtained according to one of the following rules:

(A1) if L is not an assumption (i.e., a positive non-abducible), let $H \leftarrow B$ be a clause in Π such that $L = H\theta$, then $\mathcal{G}_{i+1} = B\theta \cup \mathcal{G}_i^- \theta$ and $\Delta_{i+1} = \Delta_i$;

(A2) if L is an assumption such that $L \in \Delta_i$, then $\mathcal{G}_{i+1} = \mathcal{G}_i^-$ and $\Delta_{i+1} = \Delta_i$;

(A3) if L is an assumption such that $L \notin \Delta_i$ and $\overline{L} \notin \Delta_i$, and if there exists a *successful* consistency derivation $(L, \Delta_i \cup \{L\}), \ldots, (\emptyset, \Delta')$, then $\mathcal{G}_{i+1} = \mathcal{G}_i^-$ and $\Delta_{i+1} = \Delta'$.

A *successful abductive derivation* is an abductive derivation from $(\mathcal{G}_1, \Delta_1)$ to $(\emptyset, \Delta_n)$ $(n \geq 1)$.

A consistent derivation with respect to Ξ is a sequence $(A, \Delta_1), (F_1, \Delta_1), \ldots, (F_n, \Delta_n)$, where F_1 is the set of all denials of the form $\leftarrow \phi$ obtained by resolving A with the integrity constraints in $\mathcal{IC} \cup \{\leftarrow P, \neg P \mid P \textit{ is an atom in ground}(\Pi)\}$ (i.e., removing the L from the body of every instantiated integrity constraint that contains L). If none of ϕ is empty, then for $i = 1, \ldots, n-1$, Ξ selects a denial $\leftarrow \phi$ from F_i and a literal L from ϕ. Let $\phi^- = \phi - \{L\}$ and $F_i^- = F_i - \{\leftarrow \phi\}$, then (F_{i+1}, Δ_{i+1}) is obtained according to one of the following rules:

(C1) if L is not an assumption, let F' be the set of all denials of the form $\leftarrow B\theta, \phi^-\theta$ obtained for each clause $H \leftarrow B$ in Π such that $L = H\theta$, then $F_{i+1} = F' \cup F_i^-$ and $\Delta_{i+1} = \Delta_i$;

(C2) if L is an assumption such that $L \in \Delta_i$ and $\phi^- \neq \emptyset$, then $F_{i+1} = \{\leftarrow \phi^-\} \cup F_i^-$ and $\Delta_{i+1} = \Delta_i$;

(C3) if L is an assumption such that $\overline{L} \in \Delta_i$, then $F_{i+1} = F_i^-$ and $\Delta_{i+1} = \Delta$;

(C4) if L is an assumption such that $L \notin \Delta_i$ and $\overline{L} \notin \Delta_i$, then

1. if there exists a successful abductive derivation from $(\{\overline{L}\}, \Delta_i)$ to $(\emptyset, \Delta')$, then $F_{i+1} = F_i^-$ and $\Delta_{i+1} = \Delta'$;
2. otherwise, if $\phi^- \neq \emptyset$, then $F_{i+1} = \{\leftarrow \phi^-\} \cup F_i^-$ and $\Delta_{i+1} = \Delta_i$.

A *successful consistency derivation* is a consistency derivation from (A, Δ) to $(\emptyset, \Delta_n)$ $(n \geq 1)$.

If during an abductive derivation or consistent derivation the set of remaining goals contains only non-ground abducibles, then it *flounders* and reports *error*.

Thus, given a query $\mathcal{Q}$, if there is a successful abductive derivation from $(\mathcal{Q}, \emptyset)$ to $(\emptyset, \Delta)$, then Δ is the abductive explanation computed by KM for $\mathcal{Q}$ with respect to $\mathcal{F}$.

Under the generalised stable model semantics, KM is sound (Theorem 1 in [KM90b]) for locally stratified programs, and is complete[KM90b] (Theorem 2) for programs that are *allowed* (i.e., to avoid floundering) and *acyclic* (i.e., to avoid looping).

Definition 2.19 (Allowedness)**.** *[Top87] A logic program is* allowed *if for each clause any variable appearing in an abducible body literal also appears in a non-abducible body literal.*

Lemma 2.1. *[KM90b] Let $\mathcal{F}$ be an allowed abductive framework and $\mathcal{Q}$ be a ground query, then no abductive derivation or consistency derivation of KM resulting from $\mathcal{Q}$ flounders.*

Definition 2.20 (Level Mapping)**.** *[AB91] Let Π be a logic program and $\mathcal{B}_\Pi$ be the Herbrand base of Π. A level mapping $|.|$ is a function that maps each atom $P \in \mathcal{B}_\Pi$ and its negation to a natural number, i.e., $|.| : \mathcal{B}_\Pi \mapsto \mathfrak{N}$, and $|P| = |\neg P|$.*

Definition 2.21 (Acyclic Programs)**.** *[AB91] A logic program Π is* acyclic *if and only if there exists a level mapping $|.|$ such that for each clause $H \leftarrow L_1, \ldots, L_n$ in ground(Π), $|H| > |L_i|$ for each $i \in [1, n]$.*

ASystem

ASystem [KvND01] extends its predecessor SLDNFA [DS98] by adding finite domain constraint satisfaction support. It also adopts [vN04] some of the properties of other abductive systems, such as formulating the proof procedure as a state rewriting process like in IFF [FK97] with a set of inference rules inherited from SLDNFA, and collecting arithmetic constraints along the derivation and solving them using an external constraint satisfaction solver like in ACLP [KMM00]. ASystem adopts the three-valued completion semantics (for abductive logic programs). In addition to having constraint satisfaction support, ASystem has two more advantages over KM: it allows non-ground abducibles to be collected (i.e., computing non-ground explanations) and performs *constructive negation* instead of *negation as failure*. To see the difference between these two kinds of negations, consider the following program

$$\left\{ \begin{array}{l} p(X) \leftarrow \neg q(X). \\ q(1). \end{array} \right\}$$

The query $p(X)$ has no answer with negation as failure, while it has an answer $X \neq 1$ with constructive negation. We briefly describes next the ASystem proof procedure [vN04].

In ASystem, atoms can be *abducibles*, *non-abducibles* (or *defined atoms*), *(in-)equality* and (finite domain) constraints. A goal can be either a literal or a denial of the form $\forall \vec{X}. \leftarrow L_1, \ldots, L_n$ $(n > 0)$, where $\vec{X}$ are the universally quantified variables appearing in the denial, i.e., all other variables appearing in the denial are existentially quantified implicitly.

Definition 2.22 (**ASystem (Computational) State).** *An ASystem (computational) state is a tuple $S = \langle \mathcal{G}, \mathcal{ST} \rangle$ where G is a set of goals and $\mathcal{ST}$ is a tuple of four stores $(\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C})$:*

- *Δ is a set of collected (possibly non-ground) abducibles;*
- *$\mathcal{N}$ is a set of collected denials (sometimes called the* dynamic integrity constraints*);*
- *$\mathcal{E}$ is a set of collected (in-)equalities;*
- *$\mathcal{C}$ is a set of collected finite domain constraints.*

Definition 2.23 (**Meaning of An ASystem State).** *Given an ASystem state $S = \langle \mathcal{G}, \mathcal{ST} \rangle$, each goal in $\mathcal{G}$ and each element in the four stores of $\mathcal{ST}$ can be viewed as a first order formulate, and every free variable appearing in S is existentially quantified with the scope of the whole state.*

The meaning of S, denoted with $\mathcal{M}(S)$, is the conjunction of all the formulas in S, i.e.,

$$\mathcal{M}(S) = \bigwedge_{F \in \mathcal{G}} F \wedge \bigwedge_{F \in \mathcal{ST}} F$$

A tuple of four empty stores $\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ is often denoted with $\mathcal{ST}^{\emptyset}$. Given a query $\mathcal{Q}$, the *initial state* is $\langle \mathcal{Q} \cup \mathcal{IC}, \mathcal{ST}^{\emptyset} \rangle$ (note that $\mathcal{IC}$ becomes part of the initial goals). A *successful state* is one that has an empty set of goals (i.e., $\mathcal{G} = \emptyset$) and the four stores (i.e., $\mathcal{ST}$) are *consistent*. In this case an ASystem answer $\langle \Delta, \theta \rangle$ can be extracted, where θ is a set of substitutions induced by $\mathcal{ST}$.

Definition 2.24. *An ASystem* derivation tree *for a query $\mathcal{Q}$ with respected to a goal selection strategy Ξ is a tree in which*

- *each node is an ASystem state;*
- *the root node is the initial state;*
- *the children of a node are all the states that can be constructed from that node for the Ξ-selected goal F according to a suitable* inference rule*.*

The set of *inference rules* [vN04] are summarised below.

Given an abductive framework $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$, let $S_i = \langle \mathcal{G}_i, (\Delta_i, \mathcal{N}_i, \mathcal{E}_i, \mathcal{C}_i) \rangle$ be an ASystem state, and let F be a goal selected by Ξ from $\mathcal{G}_i$ and thus $\mathcal{G}_i^- = \mathcal{G}_i - \{F\}$. A child state $S_{i+1} = \langle \mathcal{G}_{i+1}, (\Delta_{i+1}, \mathcal{N}_{i+1}, \mathcal{E}_{i+1}, \mathcal{C}_{i+1}) \rangle$ is obtained by modifying S_i according to the application of an inference rule on F. In the rule specification, only state component modifications are described and `OR` denotes alternative modifications to S_i.

There are five rules for a selected positive goal F:

(D1) if $F = p(\vec{u})$ is a non-abducible, let $p(\vec{v}_j) \leftarrow \Phi_j$ $(j = 1, ..., n)$ be n rules in Π, then:

- $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_1\} \cup \Phi_1 \cup \mathcal{G}_i^-$

OR $\vdots$

OR $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_n\} \cup \Phi_n \cup \mathcal{G}_i^-$

(A1) if $F = a(\vec{u})$ is an abducible, let $a(\vec{v}_j)$ $(j = 1, \ldots, n)$ be n abducibles in Δ_i, then:

- $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_1\} \cup \mathcal{G}_i^-$

OR $\vdots$

OR $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_n\} \cup \mathcal{G}_i^-$

OR $\Delta_{i+1} = \{F\} \cup \Delta_i$ and $\mathcal{G}_{i+1} = R_{\Delta_i} \cup R_{\mathcal{N}_i} \cup \mathcal{G}_i^-$, where

$*$ $R_{\Delta_i} = \{\leftarrow \vec{u} = \vec{v} \mid j = 1, \ldots, n\}$,

$*$ $R_{\mathcal{N}_i} = \{\forall \vec{X}. \leftarrow \vec{u} = \vec{w}, \Phi \mid \forall \vec{X}. \leftarrow a(\vec{w}), \Phi \in \mathcal{N}_i\}$

(C1) if F is a (finite domain) constraint, let $\mathcal{C}_{new} = \{F\} \cup \mathcal{C}_i$:

- if $\mathcal{C}_{new}$ is consistent, then $\mathcal{C}_{i+1} = \mathcal{C}_{new}$ and $\mathcal{G}_{i+1} = \mathcal{G}_i^-$

(E1) if F is an (in-)equality, let $\mathcal{E}_{new} = \{F\} \cup \mathcal{E}_i$:

- if $\mathcal{E}_{new}$ is consistent, then $\mathcal{E}_{i+1} = \mathcal{E}_{new}$ and $\mathcal{G}_{i+1} = \mathcal{G}_i^-$

(N1) if $F = \neg p(\vec{u})$ (i.e., is a negative literal), then:

- $\mathcal{G}_{i+1} = \{\leftarrow p(\vec{u})\} \cup \mathcal{G}_i^-$

If the selected goal F is a denial $\forall \vec{X}. \leftarrow \Gamma$ where $\Gamma \neq \emptyset$, then Ξ further selects a literal L from Γ and thus $\Gamma^- = \Gamma - \{L\}$. There are also five rules for the selected literal L:

(D2) if $L = p(\vec{u})$ is a non-abducible, then:

- $\mathcal{G}_{i+1} = \{\forall \vec{Y}. \leftarrow \Gamma^+ \mid p(\vec{v}) \leftarrow \Phi \in \Pi \texttt{ and } \vec{Y} = \vec{X} \cup vars(p(\vec{v})) \cup vars(\Phi) \texttt{ and } \Gamma^+ = \{\vec{u} = \vec{v}\} \cup \Phi \cup \Gamma^-\} \cup \mathcal{G}_i^-$

(A2) if $L = a(\vec{u})$ is an abducible, let $F' = \vec{X}. \leftarrow a(\vec{u}), \Gamma^-$ then:

- $\mathcal{G}_{i+1} = \{\forall \vec{X}. \leftarrow \Gamma^+ \mid a(\vec{v} \in \Delta_i \texttt{ and } \Gamma^+ = \{\vec{u} = \vec{v}\} \cup \Gamma^-\} \cup \mathcal{G}_i^-$, and $\mathcal{N}_{i+1} = \{F'\} \cup \mathcal{N}_i$

(C2) if L is a (finite domain) constraint such that $vars(L) \cap \vec{X} = \emptyset$ (i.e., it does not contain any universal variable), let $\bar{(}L)$ be the *negated constraint* of L [1], then:

- if $\mathcal{C}^+ = \{\overline{L}\} \cup \mathcal{C}_i$ is consistent, then $\mathcal{G}_{i+1} = \mathcal{G}_i^-$ and $\mathcal{C}_{i+1} = \mathcal{C}^+$;

OR if $\mathcal{C}^+ = \{L\} \cup \mathcal{C}_i$ is consistent, then $\mathcal{G}_{i+1} = \{\forall \vec{X}. \leftarrow \Gamma^-\} \cup \mathcal{G}_i^-$ and $\mathcal{C}_{i+1} = \mathcal{C}^+$

(E2) if L is an equality of the form $t = s$, let $\vec{E}$ be *equational solved form* of $t = s$ (i.e., set of substitutions that are obtained by unifying s with t and are in the form of equation whose left-hand side is a universal variable if it contains one), then $\mathcal{G}_{i+1} = \{\forall \vec{X}. \leftarrow \vec{E}, \Gamma^-\} \cup \mathcal{G}_i^-$. Furthermore, if $\vec{E}$ contains exactly one equation $Y = r$ where Y is a variable and r is a term, let θ be an substitution $\{Y/r\}$,

– if Y is a universal variable (i.e., $Y \in \vec{X}$), let $\vec{X}^- = \vec{X} - \{Y\}$, then

  - $\mathcal{G}_{i+1} = \{\forall \vec{X}^-. \leftarrow \Gamma^-\theta\} \cup \mathcal{G}_i^-$

– if Y is an existential variable (i.e., $Y \notin \vec{X}$),

  - if $vars(r) \cap \vec{X} = \emptyset$ and $\mathcal{E}_i^+ = \{Y \neq r\} \cup \mathcal{E}_i$ is consistent, then $\mathcal{G}_{i+1} = \mathcal{G}_i^-$ and $\mathcal{E}_{i+1} = \mathcal{E}_i^+$

  OR $\mathcal{G}_{i+1} = \{\forall \vec{X}. \leftarrow \Gamma^-\theta\} \cup \mathcal{G}_i^-$

(N2) if $L = \neg p(\vec{u})$ (i.e., is a negative literal) such that $varsL \cap \vec{X} = \emptyset$, then

- $\mathcal{G}_{i+1} = \{p(\vec{u})\} \cup \mathcal{G}_i^-$

- $\mathcal{G}_{i+1} = \{\leftarrow p(\vec{u}), \forall \vec{X}. \leftarrow \Gamma^-\} \cup \mathcal{G}_i^-$

Note that rules **C2** and **N2** require that the selected literal does not contain any universal variable. This should be guaranteed by a *safe* goal selection strategy Ξ. Recall that in KM, Ξ is safe if it does not select any non-ground (base or non-base) abducibles. In ASystem, Ξ is safe if it selects a constraint or negative literal from the body of a denial if the literal does not contain any universal variable. If during an ASystem derivation a denial goal contains only constraints or negative literals with universal variables, the derivation flounders and **fails**.

[1]The negated constraint $\overline{L}$ of a finite domain constraint L is obtained by switching the operator $(\{<, \geq\}, \{>, \leq\})$ between the two expressions, e.g., $\overline{X > Y} \equiv X \leq Y$

The ASystem proof procedure has been proven [vN04] *sound* and *weakly complete* with respected to the three-valued completion semantics [Teu96] for abductive logic programs. It is weakly complete because *termination condition* is required.

Theorem 2.6. *[vN04] Let $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ be an abductive framework, if $\langle \Delta, \theta \rangle$ is an ASystem answer extracted from a successful state and $I^{\Delta\theta}$ is the completed abducibles, then*

1. $\Pi + I^{\Delta\theta} \models_3 \mathcal{Q}\theta$
2. $\Pi + I^{\Delta\theta} \models_3 \mathcal{IC}$

where $+$ and $\models_3$ denote operators of the three-valued completion semantics for abductive logic programs [Teu96].

Theorem 2.7. *[vN04] Given an abductive framework $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ and a query $\mathcal{Q}$. Suppose $\mathcal{Q}$ has a* finite *ASystem derivation tree W.*

1. *if all branches of W are failed (e.g., due to inconsistency in store), then $\Pi \models_3 \forall(\neg\mathcal{Q})$;*
2. *if $\Pi \cup \exists(\mathcal{Q})$ is satisfiable, then W contains a successful branch.*

The termination condition required for ASystem is the same one required for SLDNFA, which is based on abductive non-recursiveness for the abductive programs studied by Verbaeten [Ver99].

Theorem 2.8. *[vN04] Given an abductive framework $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$, if Π is semi-acyclic [AB91] with respect to a level mapping $|.|$ and Π is abductive non-recursive [Ver99], then for all bounded queries [AB91] $\mathcal{Q}$ with respect to $|.|$, the ASystem proof procedure is terminating with respect to $\mathcal{Q}$.*

Chapter 3

Early Work – Distributed Abductive REasoning (DARE)

3.1 Introduction

Abduction is a powerful inference mechanism that can generate conditional proofs. The conditions are abduced assumptions, which together with a given knowledge base, will imply the conclusion of the proof. The abduced conditions can be viewed as an answer, or as an explanation, in the context of the knowledge base, of the conclusion. In abductive logic programming, the knowledge base is represented as a logic program paired with a set of integrity constraints. In distributed abduction, we focus on the problems where knowledge and constraints are distributed over a group of agents that co-operate to produce the proof. Each agent has its own knowledge base and consistency constraints. The abduced conditions for the collective proof may come from different agents but they *must* satisfy the relevant consistency constraints of *all* the agents who *have contributed to the abductive proof*. We call this subset of the agents in the group, who have contributed, the proof *cluster*. Moreover, because we have in mind applications where the group of agents is open, where agents can join and leave the group at will, we have developed a distributed abductive inference algorithm, DARE, that can opportunistically make use of new agents that arrive whilst a proof is in progress, dynamically extending the current proof cluster. It can also recover if an agent that has been contributing leaves the group, dynamically reducing the current cluster and discarding any sub-proofs to which the agent may have contributed.

This early work on DARE was inspired by ALIAS [CLM+03], and hence shares some common features with ALIAS. Both DARE and ALIAS are distributed extensions of the Kakas-Mancarella proof procedure (KM) [KM90b, KKT92] for a single logic program knowledge base. However, there are significant differences between DARE and ALIAS. In general terms, the openness of the DARE architecture builds upon the directory mechanism that allows helper agents to be recruited on-the-fly as and when they join the agents group. In contrast, in the ALIAS system the knowledge about agents' abilities is predefined as part of the background knowledge of an agent using the LAILA [CLMT01] language. Because of this dynamic feature of the system, the DARE abductive algorithm uses a more elaborate global consistency check whereby new agents can be "recruited" for the consistency to be maintained within a cluster.

We will illustrate our DARE algorithm using a multi-agent meeting scheduling example. It is the problem of finding the names of a subset of agents from an open group of agents who can attend a meeting together and the time of that meeting. The problem is posed as a conjecture to conditionally prove that there is a time at which some subset of the people, where each is a representative of a particular interest group with particular expertise, can meet. The abduced conditions are the names of the people who can attend. Each person is represented by an agent that has knowledge about that person's current commitments and their constraints about which other people they are not prepared to meet. These constraints are used to filter out unacceptable combinations of abduced names of attendees. Although quite simple, the application illustrates the dynamic nature of the algorithm. At any moment in time the current cluster of agents collaborating in trying to find the abductive proof are the putative attendees. This subset changes not only as a result of their respective constraints but also because agents can leave and join the wider group of available agents, and hence the current proof cluster, whilst the proof is in progress.

The rest of the chapter is organised as follows. Section 3.2 gives a centralised version of the multi-agent meeting scheduling example and illustrates the use of KM. Section 3.3 extends the example with distributed settings and presents the DARE algorithm. Section 3.4 and Section 3.5 discuss properties and limitations of this early work. Section 3.6 summaries the lessons learned and the motivation for subsequent work.

3.2 The Centralised Meeting Scheduling Example

Throughout this chapter, standard abductive logic programming notations are used, and only normal logic programs are considered. Terminology of the Kakas-Mancarella proof procedure (KM) is adopted. For example, positive abducibles are considered base abducibles, whereas negative literals are considered non-base abducibles. Both base and non-base abducibles can be collected as *abduced assumptions*. To illustrate how KM works, a simple scheduling problem example is considered.

Example 3.1. *This simple centralised scheduling problem is about finding the names of people, within a given organisation, that can attend a meeting and the day in the week of that meeting. Let $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ be an abductive logic framework and $\mathcal{Q}$ be a query defined below.*

$$\Pi = \left\{ \begin{array}{l} conveneMeeting(X) \leftarrow studentCanAttend(X), tutorCanAttend(X), \neg weekend(X). \\ studentCanAttend(X) \leftarrow studentName(dan), free(dan, X). \\ studentCanAttend(X) \leftarrow studentName(ben), free(ben, X). \\ tutorCanAttend(X) \leftarrow tutorName(pat), free(pat, X). \\ free(dan, monday). \\ free(ben, tuesday). \\ free(pat, monday). \\ free(pat, tuesday). \\ weekend(saturday). \\ weekend(sunday). \end{array} \right\}$$

$$\mathcal{IC} = \left\{ \leftarrow tutorName(pat), studentName(dan). \right\}$$

$$\mathcal{AB} = \{tutorName, studentName\}$$

and

$$\mathcal{Q} = \{conveneMeeting(X)\}$$

In this example, the background knowledge Π states that a meeting can be convened for a day X, different from Saturday and Sunday, provided that a tutor and a student can attend the meeting on that day. The other rules and facts in Π together state that student Dan can attend on Monday, student Ben can attend on Tuesday and tutor Pat can attend both Monday and Tuesday. The integrity constraint, however, specifies that Pat and Dan cannot both attend the meeting. Hence, the only consistent abductive answer for the given query is for X bound to Tuesday and both Pat and Ben to attend the meeting.

The execution of KM takes as input the above abductive framework and query, and starts an abductive derivation to prove the initial goal

$$G_0 = \mathcal{Q} = \{conveneMeeting(X)\}$$

with initially an empty set $\Delta_0 = \emptyset$ of abduced assumptions (collected base or non-base abducibles). The abductive derivation unifies $conveneMeeting(X)$ with the head of the first rule in Π and collects the body literals of this rule as new goals to prove, i.e.,

$$G_1 = \{studentCanAttend(X), tutorCanAttend(X), \neg weekend(X)\}$$

and $\Delta_1 = \Delta_0$.

The first goal $studentCanAttend(X)$ is then removed from G_1 after the abductive derivation unifies it with the head of the second rule in Π. The body literals of this second rule are added in front of current list of remaining goals so giving

$$G_2 = \{studentName(dan), free(dan, X), tutorCanAttend(X), \neg weekend(X)\}$$

and $\Delta_2 = \Delta_1$.

At this point the next goal is $studentName(dan)$ and it is an abducible, so the abductive derivation adds it temporarily to Δ_2 and a consistency derivation is activated with the temporary set $\Delta' = \Delta_2 \cup \{studentName(dan)\}$ of abduced assumptions. If the consistency derivation is successful then the abductive derivation process continues its proof on the remaining goals $G_3 = \{free(dan, X), tutorCanAttend(X), \neg weekend(X)\}$ and the set Δ_3 of abducibles given by Δ' possibly extended with assumptions accumulated during the consistency derivation (i.e., KM interleaves between the abductive derivation and the consistency derivation).

The consistency derivation takes the new temporary set $\Delta^{'}$ of abduced assumptions and considers all the integrity constraints that include the new assumption $studentname(dan)$. In our example, there is only one such integrity constraint. The resolvent of the integrity constraint with the abducible gives the denial $\leftarrow tutorName(pat)$, which means the sub-goal $tutorName(pat)$ **has to fail** for the integrity constraint to be satisfied. $tutorName(pat)$ is itself abducible but not included in Δ', so it cannot be proved and the consistency derivation succeeds adding its complement $\neg tutorName(pat)$ [1] to Δ'.

The abductive derivation can then continue its proof with

$$\Delta_3 = \{studentName(dan), \neg tutorName(pat)\}$$

and

$$G_3 = \{free(dan, X), tutorCanAttend(X), \neg weekend(X)\}$$

[1]Since $\neg tutorName(pat)$ is a non-base abducible, in some literatures it may be annotated as $tutorName*(pat)$.

Continuing the abductive derivation, the variable X gets unified with Monday, the proof of $tutorCanAttend(monday)$ generates a new set of goals $tutorName(pat), free(pat, X)$. But the abductive derivation then **fails** because the abducible goal $tutorName(pat)$ cannot be proved as its complement is already abduced in Δ_3. At this point, since there is no other rule in Π for proving $tutorCanAttend(monday)$, the abductive derivation **backtracks** to the previous branching point of its proof, which is where the set of goals was

$$G_1 = \{studentCanAttend(X), tutorCanAttend(X), \neg weekend(X)\}$$

and $\Delta_1 = \Delta_0 = \emptyset$, in order to find another way to prove $studentCanAttend(X)$. This corresponds to choosing the second rule for this non-abducible and starting the abductive derivation again. Continuing in a similar way, it is easy to see that the answer to the initial query $\mathcal{Q}_0 = \{conveneMeeting(X)\}$ is

$$\Delta = \{studentName(ben), tutorName(pat), \neg student(pat)\}$$

with unification $X = tuesday$.

3.3 DARE Framework and Algorithm

Example 3.1 in the previous section shows a very simple snapshot of a scheduling problem. In a real context, the background knowledge would be much bigger and the computation cost much higher if expressed as a single agent (i.e., centralised) process. A distributed representation of the knowledge base among *personal agents*, namely an agent process that has knowledge about the person's commitments and their constraints, would instead allow for more efficient computations. Alternative abductive computations for a given goal (e.g. $studentName(X)$) could, in such a multi-agent context, be fired in parallel. So in case of failure of an abductive derivation, alternative abductive answers, already computed in parallel, can be directly *fetched* and used to continue a proof. To this aim, KM would need to be extended to allow for abductive derivations over distributed knowledge and consistency derivations over distributed integrity constraints. This section presents a *distributed* abductive inference algorithm, referred to as the DARE algorithm, that computes abductive derivations by integrating *local* (i.e. agent-based) explanations to goals, computed by individual agents, whilst preserving the consistency of the combined answer within the context of the proof cluster (i.e. set of agents involved in the proof). The algorithm extends the KM illustrated in the previous section to allow abductive derivations over multiple agents (A_i), equipped with individual background knowledge Π_i and integrity constraints IC_i, and consistency derivations over the integrity constraints of the agents involved in the computation. The abductive computation of the DARE algorithm is *cluster-based*: its aim is to identify a set of missing information Δ that is consistent with the integrity constraints of the agents in a given proof cluster, and that together with the knowledge base of these agents *explain* a given (set of) goals. A proof cluster is formed dynamically during the reasoning process as and when agent specific knowledge and related integrity constraints are needed during an abductive (resp. consistency) derivation. For the DARE algorithm to work, the existence of a *shared predicate ontology* among agents is assumed, which includes a given set $\mathcal{AB}$ of *abducible predicates*. Predicates in the shared ontology are considered to be *global* over the agents, whereas those not in the shared ontology are assumed to be renamed uniquely with respect to each agent that defines them. It is also assumed that agents share the same set $\mathcal{AB}$ of abducible predicates. Thus, the total knowledge of each agent (A_i) is represented as an abductive framework $\mathcal{F}_i = \langle \Pi_i, \mathcal{AB}, \mathcal{IC}_i \rangle$.

3.3.1 Overview

Within the DARE system, each agent is capable of performing *local* abductive reasoning to explain (sub-)goals using its own background knowledge and integrity constraints; it can communicate with other agents to ask for help in explaining information that is outside the realm of its own knowledge. The background knowledge of a given agent can in fact use in its rules predicates that are defined in other agents. Incompleteness of information is therefore not only related to abducible predicates but also to positive non-abducible predicates. Whereas for the first type of information, an agent is allowed to make assumptions in order to continue its proof, for the case of positive non-abducible predicates, an agent can ask for help to any agent who has *advertised* that predicate to be part of its reasoning capability. Note that not all information (or non-abducible predicates) of an agent needs necessarily to be public. To preserve a certain level of encapsulation of information, an agent has the ability to announce the information that it will provide proof for. Advertised non-abducible predicates must be part of the pre-defined shared ontology. Together with the background knowledge, an agent also has its own integrity constraints. These are always kept private and never exported to other agents. Given the encapsulation of the integrity constraints and the fact that agents collaborate to compute a (global) abductive answers, a natural question is then how can the DARE algorithm guarantee consistency of the abductive answer with respect to the integrity constraints of the other agents involved in the proof. A local consistency check on locally abduced assumptions is clearly not sufficient to assure that such assumptions would be consistent with the integrity constraints of any other agent that can subsequently join an abductive proof. A more sophisticated process for checking consistency is therefore needed. Before defining our DARE algorithm, we formalise the notions of DARE distributed abductive context and DARE distributed explanation as a generalisation of the notions of abductive framework and abductive explanation given in Section 3.2 to the case of a cluster of agents.

Definition 3.1 ((DARE) Distributed Abductive Context). *Let $\Sigma = \{A_1, \ldots, A_n\}$ be a non-empty group of agents and let Π_i and $\mathcal{IC}_i$ be the normal logic program and set of denial clauses that define the background knowledge and integrity constraints of agent A_i, respectively, for each $A_i \in \Sigma$. A (DARE) distributed abductive context is the tuple* $\mathtt{DAC} = \left\langle \Sigma, \widehat{\Pi}, \mathcal{AB}, \widehat{\mathcal{IC}}, \mathcal{Q}, A_{init} \right\rangle$ *where A_{init} is the agent in Σ that receives a top-level (i.e., initial) query $\mathcal{Q}$, $\widehat{\Pi} = \{\Pi_1, \ldots, \Pi_n\}$ and $\widehat{\mathcal{IC}} = \{\mathcal{IC}_1, \ldots, \mathcal{IC}_n\}$, and $\mathcal{AB}$ is a set of abducible predicates.*

A distributed abductive context can evolve as agents may join or leave the current group, but it is assumed that agent A_{init} belongs to the context at all time. This is formally defined as follows.

Definition 3.2 (Evolved (DARE) Distributed Abductive Context). *Let* $\mathtt{DAC} = \langle \Sigma, \widehat{\Pi}, \mathcal{AB},$

$\widehat{\mathcal{IC}}, \mathcal{Q}, A_{init}\rangle$ *be a (DARE) distributed abductive context. An* evolved *(DARE) distributed abductive context is the tuple* $\mathtt{DAC}' = \left\langle \Sigma', \widehat{\Pi}', \mathcal{AB}, \widehat{\mathcal{IC}}', \mathcal{Q}, A_{init} \right\rangle$*, with* A_{init} *belonging to the* Σ'*.*

In an evolved distributed abductive context $\mathtt{DAC}'$, the group of agents does not need to be the same as that of the starting context $\mathtt{DAC}$, with the exception of the initial agent A_{init}. Within the scope of this chapter it is also assumed that the program Π_i and the integrity constraints $\mathcal{IC}_i$ of an agent A_i do not evolve. In fact, any evolution from Π_i to Π'_i or from $\mathcal{IC}_i$ to $\mathcal{IC}'_i$ can be considered as the leaving of agent A_i followed by the joining of agent A'_i, providing that $A_i \neq A_{init}$.

Definition 3.3 ((DARE) Distributed Explanation)**.** *Let* $\mathtt{DAC} = \left\langle \Sigma, \widehat{\Pi}, \mathcal{AB}, \widehat{\mathcal{IC}}, \mathcal{Q}, A_{init} \right\rangle$ *be a (DARE) distributed abductive context, let* $\mathcal{C} \subseteq \Sigma$ *be a cluster of agents and let* $\mathcal{L}^{\mathcal{C}}_{\mathcal{AB}}$ *denote the set of all ground literals with abducible predicates in* $\mathcal{AB}$ *and all negative ground literals with non-abducible predicates that appear in* $\bigcup_{A_j \in \mathcal{C}} (\Pi_j \cup \mathcal{IC}_j)$*. Then a* (DARE) abductive explanation *of* $\mathtt{DAC}$ with respect to the cluster $\mathcal{C}$ *is a set of ground literals* $\Delta \subseteq \mathcal{L}_{\mathcal{AB}}$ *for which there exists a ground instance* $\mathcal{Q}\theta$ *of the query* $\mathcal{Q}$ *such that*

- $(\bigcup_{A_j \in \mathcal{C}} \Pi_j) \cup \Delta \models \mathcal{Q}\theta$*, and*
- $(\bigcup_{A_j \in \mathcal{C}} \Pi_j) \cup \Delta \models \bigcup_{A_j \in \mathcal{C}} \mathcal{IC}^{\Delta}_j$

where $\mathcal{IC}^{\Delta}_j$ *is the set of integrity constraints in* A_j *whose body literals unify with a literal in* Δ*.*

3.3.2 Distributed Algorithm

The DARE algorithm uses two main phases, called respectively *global abductive derivation* (`GAD`) and *global consistency derivation* (`GCD`), with two supporting phases, called respectively *local abductive derivation* (`LAD`) and *local consistency derivation* (`LCD`), to compute an abductive explanation. The full description of these four phases is given at the end of this section.

The global abductive derivation is the *top-level* reasoning process that initially takes in input a (set of) goal(s) G and an agent A in the system and starts a derivation process for proving

G. This derivation takes one literal (goal) L at a time from G and tries to abductively prove it. If L is defined in A, the global abductive derivation proceeds locally as a KM abductive derivation (Rule (2) of GAD). If L is not defined in A and it is a positive non-abducible predicate, other agents, among those able to prove it, are invoked for help (Rule (3) of GAD). Whenever a new agent joins an abductive derivation, a global consistency check is performed to make sure that its addition to the proof cluster does not violate the current set of assumed abducibles (Rule (1) of GAD). If L is a ground base abducible and $\overline{L}$ (i.e. the complement of L) has already been assumed in Δ, then the current global abductive derivation **fails** and backtracks to any earlier branching point (if any) in the proof (Rule (4) of GAD). If L is a ground base abducible already included in Δ then the global abductive derivation continues its abductive proof on the remaining literals in the given initial goal G (see Rule (5) of GAD). This is also the case for L non-base ground abducible already assumed in Δ. But, if the literal L to prove is a ground base abducible not yet assumed, then it can be temporarily added to the set Δ and checked for consistency (Rule (6) of GAD). This is the point where in KM the abductive derivation calls a consistency derivation. In the DARE algorithm, this corresponds to passing the newly extended set of assumptions $\{L\} \cup \Delta$ to all the agents in the current proof cluster to make sure that this new set of assumptions is still consistent with their integrity constraints. If the literal L to prove is a non-base ground abducible not included in Δ, then the current agent A can temporarily add it to Δ and check for consistency first locally (i.e. whether it can actually prove the complement of this literal), and then globally over the proof cluster to verify that the integrity constraints of the agents collaborating in the proof are still consistent with the newly extended set of assumptions $\{L\} \cup \Delta$ (Rule (7) of GAD). The overall process of the distributed abductive derivation is diagrammatically represented in Figure 3.1.

Whereas the global abductive derivation succeeds when the given set G of goals has been reduced to the empty set of literals (as all of its literals have been abductively proved), the global consistency phase takes as input the current set of assumptions Δ as its goal, and checks it for consistency with respect to the integrity constraints of all the agents in the current cluster. In essence this means considering one element L in Δ, choosing one agent A in the cluster, and resolving L with the integrity constraints in A; if at least one of these integrity constraints fails,

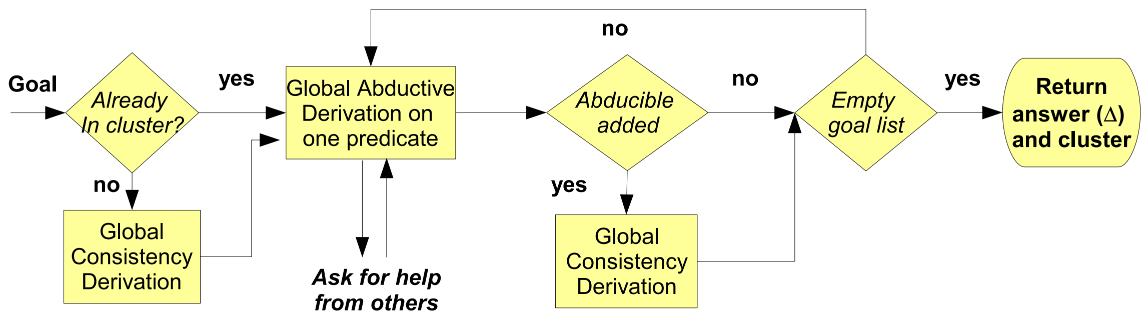

Goal
Already In cluster?
yes
no
Global Consistency Derivation
Global Abductive Derivation on one predicate
Ask for help from others
Abducible added
no
yes
Global Consistency Derivation
Empty goal list
no
yes
Return answer (Δ) and cluster

Figure 3.1: Global Abductive Derivation

then the global derivation fails. Otherwise, the consistency check passes to the next agent in the cluster, and so on through the entire cluster and for each literal in the given set Δ of abduced information. The global consistency process uses two additional supporting derivations – `LAD` and a `LCD`. The resolution of the chosen literal L with the integrity constraints of the particular agent A in the cluster is handled by that agent via a `LAD` but with the KM abductive derivation step applied only between L and its integrity constraints instead of L and its rules (see Rule (2) of `GCD` and step (3) of `LAD`). While checking the consistency of L with its integrity constraints the agent A may make further assumptions which need themselves to be checked for consistency over the cluster. A round of consistency checks over the cluster terminates when all the agents have been considered once. If at the end of a round no additional assumptions have been made during the `LAD`s then the global consistency check terminates successfully. Of course, the global consistency derivation fails as soon as a round of consistency checks does not finish as the initial set of assumption Δ does not satisfy the integrity constraints in one of the agents in the cluster. The GCD process is diagrammatically represented in Figure 3.2.

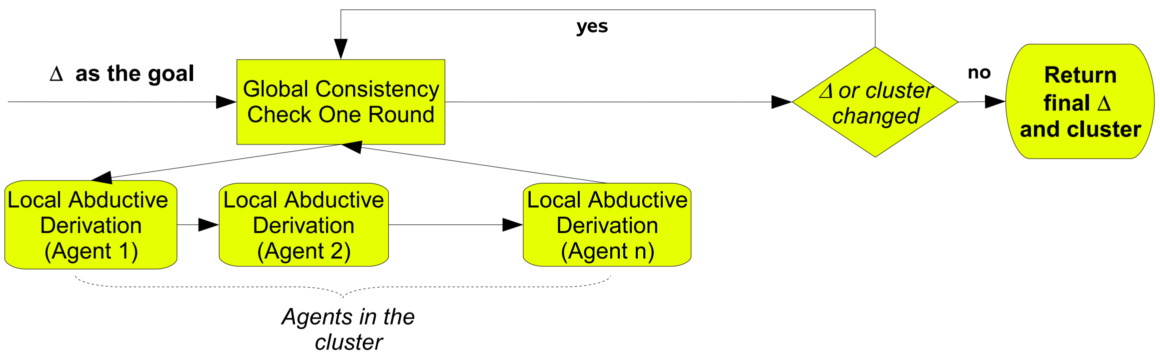

yes
Δ as the goal
Global Consistency Check One Round
Δ or cluster changed
no
Return final Δ and cluster
Local Abductive Derivation (Agent 1)
Local Abductive Derivation (Agent 2)
Local Abductive Derivation (Agent n)
Agents in the cluster

Figure 3.2: Global Consistency Derivation

At the beginning of a `GCD`, an agent A from the cluster is chosen which starts a `LAD` taking as goals the current set Δ that has to be checked for consistency and an empty set of temporary assumptions. The first literal L is then resolved with all the integrity constraints in A that contain L. The set of resolvents is then passed to a `LCD` in A. These become *must-fail* goals. The `LCD` proceeds in a similar way as the KM consistency derivation. The main difference in this case is Rule (5) of `LCD`. When the literal L to fail is a non-base ground abducible not yet assumed, a global abductive derivation is called with goal $\overline{L}$. In this case, although the `GAD` is activated from within the context of a consistency check that is cluster-based, agents outside the cluster can be invoked for help. This can be seen as a form of collaborative reasoning for constraint satisfaction, whereby a current cluster of agents can dynamically expand to include agents with reasoning capabilities that enable successful termination of local consistency checks that would otherwise fail.

This brings us to the following main features of the DARE algorithm. Firstly, clusters can expand during both global abductive and global consistency derivations. This expansion occurs in particular cases: in the global abductive derivation when positive non-abducible literals need to be proved and the current agent either fails to do so or does not have reasoning capability (i.e. rules) for that type of information whereas other agents in the group do. These other agents can then take part in the derivation process by joining the current cluster, provided that their integrity constraints do not contradict the current set Δ of assumptions; and in the local consistency derivation, on the other hand, the task of "failing a non-based abducible with non-abducible predicate" is, in a sense, similar to the task of succeeding a positive non-abducible predicate. The local consistency derivation behaves in a similar way to the global abductive derivation case. If the current agent cannot fail a non-base abducible, it can ask agents outside the cluster to help in proving the complement of such an abducible in the attempt to successfully complete its local consistency check. The second feature of this algorithm is *negation as failure*. Its semantics is strictly related to the concept of cluster. Successfully proving a negative non-abducible predicate for a given cluster $\mathcal{C}$ means that all the agents in $\mathcal{C}$ are not able to prove its complement. Negation as failure has therefore, in this case, the same meaning as in [Cla78] but with respect to a background knowledge given only by that of the agents in the cluster. On the other hand, successful failure of proving a negative non-abducible means finding an agent in the system who is able to prove the non-abducible. This reflects the standard concept that failing to prove a negated literal is in essence equivalent to finding a proof for its complement.

Finally, the full algorithm with the four types of derivations is defined below. In the following, Δ is the set of ground abducibles that are collected during the proof and $\mathcal{C}$ the set of agents in the cluster formed during the proof. The output of a *successful* DARE computation is a final set Δ and associated cluster $\mathcal{C}$ of the final distributed abductive context $\left\langle \Sigma_{fin}, \widehat{\Pi}, \mathcal{AB}, \widehat{IC}, \mathcal{Q}, A_{init} \right\rangle$. The algorithm starts with the agent A_{init} performing a global abductive derivation for the query $G_{init} = \mathcal{Q}$ with $\Delta_{init} = \emptyset$ and $\mathcal{C}_{init} = \emptyset$. The first step in this abductive derivation will add agent A_{init} to the cluster (Rule (1) of GAD).

Global Abductive Derivation

Let A be the current agent, Δ be the current set of abduced literals, $\mathcal{C}$ be the current cluster and G the current goal ($\{L_1, \ldots, L_k\}$). If G is (reduced to) the empty goal $\emptyset$ then the global abductive derivation **succeeds** and Δ and $\mathcal{C}$ are returned. Otherwise, $G^{'}$ is obtained by removing a literal L from G, and Δ' and $\mathcal{C}'$ are obtained while applying one of the following rules:

1. If $A \notin \mathcal{C}$: if there exists a global consistency check on Δ with $\mathcal{C}'' = A \cup \mathcal{C}$, and Δ' and C' are obtained after the global consistency derivation, then A continues the global abductive derivation on G with Δ' and C'.

2. If L is a non-abducible: if a rule whose head can match with L exists in A, and the instantiated body is B, then A continues the global abductive derivation on $B \cup G'$ with $\Delta' = \Delta$ and $\mathcal{C}' = \mathcal{C}$.

3. If L is a non-abducible and Rule (2) does not apply, then if there exists a global abductive derivation on $\{L\}$ with Δ and $\mathcal{C}$ by a helper agent H (whether in the group or in the current cluster), and Δ' and $\mathcal{C}'$ (Rule (1) and this rule together imply that $H \in \mathcal{C}'$) are obtained after the derivation, then A continues the global abductive derivation on G' with Δ' and $\mathcal{C}'$.

4. If L is a ground abducible and $\overline{L}$ is in Δ, the derivation fails.

5. If L is a ground abducible and L is in Δ, then the current agent continues the global abductive derivation on G' with $\Delta' = \Delta$ and $\mathcal{C}' = \mathcal{C}$.

6. If L is a ground base abducible and neither L nor $\overline{L}$ is in Δ, if there exists a global consistency derivation on $\{L\} \cup \Delta$ with $\mathcal{C}$, and Δ' and $\mathcal{C}'$ are obtained after the global consistency derivation, then the current agent continues the global abductive derivation on G' with Δ' and $\mathcal{C}'$.

7. If L is a non-base ground abducible and $L \notin \Delta$ then if there exists a local consistency derivation on $\{\leftarrow \overline{L}\}$ with $\Delta'' = \Delta \cup \{L\}$ and $\mathcal{C}$ by A, and if there exists a global consistency derivation on Δ''' with $\mathcal{C}'''$, where Δ''' and $\mathcal{C}'''$ are obtained after the local consistency derivation, then A continues the global abductive derivation on G' with Δ' and $\mathcal{C}'$, where Δ' and $\mathcal{C}'$ are obtained after the global consistency derivation.

Global Consistency Derivation

The global consistency derivation consists of one or more consecutive *consistency check rounds* to make sure the current Δ is consistent among the agents in the current cluster $\mathcal{C}$. The agents in $\mathcal{C}$ are labelled as $A_1, \dots, A_n$. For each round of consistency check:

1. Let Δ_0 be the input to A_1.

2. Δ_{k-1} is passed to A_k. If there exists a local abductive derivation by A_k on $G = \Delta_{k-1}$ with $\Delta = \emptyset$, and Δ_k and $\mathcal{C}_k$ are obtained after the derivation:

   - If $k < n$, then Δ_k is passed to A_{k+1} with cluster $\mathcal{C} = \mathcal{C}_k$
   - If $k = n$, then the current consistency round **succeeds**.

If one consistency check round succeeds and $\Delta_0 = \Delta_n$, then the global consistency derivation **succeeds** with Δ_n and $\mathcal{C}_n$. If one consistency check round succeeds but $\Delta_0 \subset \Delta_n$, then start another consistency check round on Δ_n and $\mathcal{C}_n$.

Local Abductive Derivation

The local abductive derivation of an agent A is to make sure that the goals passed to A in Rule (2) of `GCD` (a set of abducibles) do not violate any integrity constraint of A. Since the set of goals contains only abducibles, the local abductive derivation by A can be simulated as an operation such that A "re-abduces" one abducible from the goal, and checks its related local integrity constraints. If a local abductive derivation succeeds, the obtained new set of abducibles will contain the original goal and will be consistent with the integrity constraints of A. Formally, let A be the current agent and Δ be the current set of abducibles, and $\mathcal{C}$ be the current cluster. The local abductive derivation **succeeds** if G is empty, and Δ will be returned with its associated cluster. Otherwise, G' is obtained by removing a literal L from G, and Δ' is obtained while applying one of the following rules:

1. If L is a ground abducible and $\overline{L} \in \Delta$, the derivation **fails**.
2. If L is a ground abducible and $L \in \Delta$, then A continues the local abductive derivation on G' with $\Delta' = \Delta$ and $C' = C$.
3. If L is a ground abducible and $L \notin \Delta$ and $\overline{L} \notin \Delta$. Let $\mathcal{IC}^L$ be the set of integrity constraints containing L, and F be the set obtained by removing L from each constraint in $\mathcal{IC}^L$. If there exists a successful local consistency derivation on F with $\Delta \cup \{L\}$ and $\mathcal{C}$, and Δ' and $\mathcal{C}'$ are obtained after the local consistency derivation, then A continues the local abductive derivation on G' with Δ' and $\mathcal{C}'$.

Local Consistency Derivation

In the local consistency check, let A be the current agent and $\mathcal{C}$ be the current cluster. Let F be the set of denial goals to be checked for consistency (i.e. **must-fail** goals). The local consistency check **succeeds** if F is empty, and **fails** if F contains $\leftarrow$ *emptyset*. Otherwise, let $F' \cup \{G\} = F$ and G' is obtained by removing a body literal L from G, Δ' and $\mathcal{C}'$ are obtained while applying one of the following rules:

1. If L is a non-abducible: A resolvent of L is $B \cup G'$ where B is the instantiated body of a rule in the (local) agent whose head can match with L. Let $\mathcal{S}$ be the set of all resolvents of L, A continues the local consistency derivation on $\mathcal{S} \cup F'$ with $\Delta' = \{\overline{L}\} \cup \Delta$ and $C' = C$.

2. If L is a ground abducible and $L \in \Delta$, then A continues the local consistency derivation on $F' \cup \{G'\}$ with $\Delta' = \Delta$ and $C' = C$.

3. If L is a ground abducible and $\overline{L} \in \Delta$, then A continues the local consistency derivation with F' and $\Delta' = \Delta$ and $C' = C$.

4. If L is a ground base abducible and $L \notin \Delta$ and $\overline{L} \notin \Delta$, then A continues the consistency derivation with F' and $\Delta' = \Delta \cup \overline{L}$ and $C' = C$.

5. If L is non-base ground abducible and $L \notin \Delta$, if there exists a successful global abductive derivation on $\{\overline{L}\}$ with Δ and $\mathcal{C}$, then A continues the local consistency derivation on F' with Δ' and C' where Δ' and C' are obtained from the global abductive derivation.

Correctness of the DARE Algorithm

The correctness of the algorithm requires showing that given a distributed abductive context $DAC = \left\langle \Sigma, \widehat{\Pi}, \mathcal{AB}, \widehat{\mathcal{IC}}, \mathcal{Q}, A_{init}, \right\rangle$, and given a successful global abductive derivation, by A_{init}, of $\mathcal{Q}$ with a final set Δ of abduced assumptions and a final cluster $\mathcal{C}$ of agents of an evolved abductive context DAC', then Δ is an abductive explanation of DAC' with respect to the cluster $\mathcal{C}$ for the goal $\mathcal{G}$. Formally,

Theorem 3.1. *Let DAC be an initial distributed abductive context with goal $\mathcal{G}$. Let Δ be the output of a global abductive derivation for an instance $\mathcal{Q}\theta$, returned by A_{init} and computed by the cluster $\mathcal{C} = \{A_1, \ldots, A_n\}$ of a final evolved abductive context DAC_{fin} such that $A_{init} \in \mathcal{C}$. Then Δ is an abductive explanation of DAC_{fin} with respect to $\mathcal{C}$ such that:*

- $(\bigcup_{A_j \in \mathcal{C}} \Pi_j) \cup \Delta \models_s \mathcal{Q}\theta$, *and*
- $(\bigcup_{A_j \in \mathcal{C}} \Pi_j) \cup \Delta \models_s \bigcup_{A_j \in \mathcal{C}} \mathcal{IC}_j^{\Delta}$

where $\models_s$ is the logic entailment under the stable model semantics and $\mathcal{IC}_j^{\Delta}$ is the set of integrity constraints in A_j whose body literals unify with a literal in Δ.

An informal proof of this property by induction on the number of agents in the final cluster is given in [MRBC08].

3.3.3 Distributed Meeting Scheduling Example

This section illustrates the DARE algorithm via the example of distributed meeting scheduling. Building upon the initial example given in Section 3.2, let us consider now the case of a scheduling problem among a certain number of agents. The initial group of agents includes a convener, a tutor, a lecturer, several students, a nursery and a timetabler. The *Convener* is the agent responsible for organising the meeting. A second lecturer (A_8 below) may join during the execution of the DARE algorithm. The student, lecturer and tutor agents are equipped with various local rules that define when they may attend meetings and integrity constraints to specify who they are (not) prepared to meet with. The shared ontology includes all predicates with the exception of $free$, day, $tired$, $teaching$, $teachingJuniors$ and $teachingSeniors$, which are uniquely renamed with the index of the agent that defines them. All shared predicates are advertised, and the base abducible predicates are $\mathcal{AB} = \{studentName, lecturerName, tutorName\}$ (as it was also the case in the Example 3.1).

The (initial) distributed abductive context of our example is the tuple $\mathtt{DAC} = \left\langle \{\Sigma, \widehat{\Pi}, \mathcal{AB}, \widehat{\mathcal{IC}}, \mathcal{Q}, A_1 \right\rangle$ where $\Sigma = \{A_1, \ldots, A_8\}$ as described below, and the initial query $\mathcal{Q}$ is $\{conveneMeeting(T)\}$.

A_1 **(Convener)**

$$\Pi_1 = \left\{ \begin{array}{l} conveneMeeting(T) \leftarrow day1(T), studentCanAttend(T), \\ \qquad\qquad tutorCanAttend(T), lecturerCanAttend(T). \\ day1(tuesday). \\ day1(wednesday). \\ day1(thursday). \\ day1(friday). \end{array} \right\}$$

A_2 **(Tutor)**

$$\Pi_2 = \left\{ \begin{array}{l} tutorCanAttend(T) \leftarrow tutorName(pat), free2(T). \\ free2(X) \leftarrow nursery(X). \end{array} \right\}$$

$$\mathcal{IC}_2 = \left\{ \begin{array}{l} \leftarrow tutorName(pat), studentName(dan). \\ \leftarrow tutorName(pat), lecturerName(joe). \end{array} \right\}$$

A_3 **(Student)**

$$\Pi_3 = \left\{ \begin{array}{l} studentCanAttend(T) \leftarrow studentName(ben), free3(T). \\ free3(monday). \\ free3(thursday). \\ free3(friday). \end{array} \right\}$$

A_4 **(Student)**

$$\Pi_4 = \left\{ \begin{array}{l} studentCanAttend(T) \leftarrow studentName(dan), free4(T). \\ free4(monday). \\ free4(wednesday). \end{array} \right\}$$

A_5 **(Nursery)**

$$\Pi_5 = \left\{ \begin{array}{l} nursery(wednesday). \\ nursery(friday). \end{array} \right\}$$

A_6 **(Lecturer)**

$$\Pi_6 = \left\{ \; lecturerCanAttend(T) \leftarrow lecturerName(joe), freeFromTeaching(T, joe). \; \right\}$$

A_7 **(Timetabler)**

$$\Pi_7 = \left\{ \begin{array}{l} freeFromTeaching(T, X) \leftarrow \neg teaching7(T, X). \\ teaching7(T, X) \leftarrow teachingJuniors7(T, X). \\ teaching7(T, X) \leftarrow teachingSeniors7(T, X). \\ teachingSeniors7(thursday, joe). \\ teachingJuniors7(wednesday, rob). \end{array} \right\}$$

A_8 **(Lecturer)**

$$\Pi_8 = \left\{ \begin{array}{l} lecturerCanAttend(T) \leftarrow lecturerName(rob), \neg tired8(T), \\ \qquad\qquad freeFromTeaching(T, rob). \\ tired8(thursday). \end{array} \right\}$$

Typically, the convener is the agent that makes the initial query $\{conveneMeeting(T)\}$ with an empty set $\Delta_1 = emptyset$ of assumptions and a cluster consisting of only himself (i.e $\mathcal{C}_1 = \{A_1\}$). A solution to this query will include a cluster $\mathcal{C}$ of the agents contributing to the computation, an instance value for T, and an abductive explanation Δ of the given `DAC` with respect to the cluster $\mathcal{C}$. If the only available agents are $A_1 \ldots A_7$, then the above `DAC` has no solution, since the tutor Pat does not wish to meet with the lecturer Joe, and the abductive context does not include any other lecturer. If A_8 joins the group then a possible solution is

$$\Delta = \{lecturerName(rob), tutorName(pat), studentName(ben)\}$$

for the unification $T = friday$, and the cluster $\mathcal{C} = \{A_1, A_2, A_3, A_5, A_7, A_8\}$. The tutor Pat is only willing to work with Rob and Ben. The only common free day for Pat and Ben is Friday and Rob can also meet on Friday. This solution is computed by the DARE algorithm in the following way.

1. Agent 1 (A1) can prove $day1(tuesday)$ but not $studentCanAttend(tuesday)$. Therefore it applies step 3 of the GAD and recruits agents A_3 and A_4 who then carry out their own step 1 of GCD. In this moment of the proof, there are not yet any abduced assumptions, so step 1 in each of these agents succeeds trivially, and two different clusters $\mathcal{C}_1 = \{A1, A3\}$ and $\mathcal{C}_2 = \{A1, A4\}$ are formed.

2. Student agent A_3 (respectively A_4) starts its GAD process, matching $studentCanAttend(tuesday)$ to the head of one of its program clauses to derive the sub-goal $studentName(ben)$ (respectively $studentName(dan)$), to which step 6 of the GAD is applied. For example, agent A_3 adds $studentName(ben)$ to Δ and attempts a GCD, which requires each agent in its cluster, $\mathcal{C}_1$, to check that the abduced assumption does not violate any integrity constraint of the agents in $\mathcal{C}_1$ (i.e. A_1 and A_3). This is clearly the case as neither agent in $\mathcal{C}_1$ has integrity constraints. Similarly for agent A_4 except that $studentName(dan)$ is abduced.

3. The student agents A_3 and A_4 continue independently their GAD on their next sub-goal $free3(tuesday)$ and $free4(tuesday)$ respectively. Neither student agent can succeed this subgoal, so A_1 backtracks and requests help for $studentCanAttend(wednesday)$. This time A_4 succeeds and returns $\Delta = \{studentName(dan)\}$ after a successful GCD.

4. Agent A_1 continues its GAD with the current set $\Delta = \{studentName(dan)\}$ of abducibles. It requests help from A_2 to prove the sub-goal $tutorCanAttend(wednesday)$. A_2 temporarily abduces $tutorName(pat)$ and activates a GCD which fails because of the first integrity constraint of A_2.

5. Agent A_1 again backtracks and solves $day1(thursday)$, which leads to A_3 succeeding with $\Delta = \{studentName(ben)\}$. A similar computation as before follows, but this time using A_2. This results in the new abducible $\neg lecturerName(joe)$ being added to Δ, so far given by $\{studentName(ben), tutorName(pat)\}$, for the second integrity constraint of A_2 to be satisfied. However, the $free2(thursday)$ sub-goal of A_2 will fail.

6. Agent A_1 backtracks yet again and solves the sub-goal $day1(friday)$. This time A_2 succeeds to show $tutorCanAttend(friday)$, with the set $\Delta = \{studentName(ben), tutorName(pat), \neg lecturerName(joe)\}$ of abducibles and cluster $\mathcal{C} = \{A1, A2, A3, A5\}$.

7. Agent A_1 requests now help for its sub-goal $lecturerCanAttend(friday)$ and A_6 starts a new GAD. Note that when performing step 1 (on joining the cluster) of its GAD, the corresponding GCD succeeds, but in step 4 it fails since $\neg lecturerName(joe)$ is in Δ. The top-level goal fails as there are not more solutions for $day1(T)$ in A_1.

8. Suppose now that Agent A_8 joins the group before the failure. A_1 will notice this and request its help to solve $lecturerCanAttend(friday)$. As part of a derivation, A_8 adds to Δ the abduced assumption $lecturerName(rob)$ giving $\Delta = \{lecturerName(rob), studentName(ben), tutorName(pat), \neg lecturerName(joe)\}$, which does not violate any integrity constraints of any agents in the current cluster $C = \{A1, A2, A3, A5, A8\}$. A_8 continues then and applies step 7 (of its GAD) to solve $\neg tired8(friday)$. This requires adding $\neg tired8(friday)$ to Δ and checking that $tired8(friday)$ is not provable by any agent in the cluster, which in this case succeeds trivially since the predicate $tired8$ is local to just agent A_8.

9. Agent 8 next solves the sub-goal $freefromTeaching(friday, rob)$ by recruiting agent A_7. This gives the set of assumptions

$$\begin{aligned}\Delta = \{&studentName(ben), tutorName(pat), \neg lecturerName(joe),\\ &lecturerName(rob), \neg tired8(friday)\}\end{aligned}$$

and the binding $T = friday$ which are returned to A_1. The first GAD process is thus terminated.

Suppose now that Pat has no integrity constraints and Dan will only attend the meeting if Rob does. This can be expressed as an integrity constraint for Dan (i.e. in A_4) by $\leftarrow studentName(dan), \neg lecturerName(rob)$. When $studentName(dan)$ is abduced as part of solving the sub-goal $studentCanAttend(wednesday)$ in A_4, the GCD will require $\neg lecturerName$

(rob) to fail, which means the query $lecturerName(rob)$ to succeed. Since $lectureName$ is an abducible predicate, $lecturerName(rob)$ will succeed by being added to Δ. At some point in the GAD of the top-level goal, the sub-goal $lecturerCanAttend(wednesday)$ will succeed by abducing $lecturerName(joe)$. If this enriched set of abducibles were not desirable (i.e. only the name of agents involved in the proof should be part of the scheduling solution) an additional integrity constraint $\leftarrow lecturerName(X), lecturerName(Y), \neg X = Y.$ could be added to agent A_1 to capture that at most one lecturer should attend. In this case since a GCD goes always through all the agents in the current cluster, the GCD initiated by the the GAD in agent A_6 on the sub-goal $lecturerName(joe)$ would fail when checking agent A_1, and be forced to backtrack to find another solution for $lecturerCanAttend(wednesday)$. Since this is impossible, further backtracking would be performed eventually finding Rob as the solution with T bound to Friday.

3.4 DARE Implementation

DARE has been implemented in Qu-Prolog [CRZA05] and uses Pedro [RC10], a publish/subscribe server, to support inter-agent communications. The system assumes the communication between agents to be safe and reliable, namely that the messages sent between two agents cannot be lost or corrupted, and each agent is rational and trusted by the others. As its main purpose is to support coordinating collaborative reasoning, the handling of various network attacks or fatal network failures is not considered, and the system does not allow for the possibility of malicious agents interfering in the collaboration between other agents.

As mentioned already, the DARE system consists of an open group of agents. The group can dynamically change even during a particular inference process. This can result in the agents involved in the proof, the current proof cluster, to change. When a new agent joins, it will notify all the existing agents and publish all of its advertisements (i.e., non-abducibles in the shared ontology that are defined in the new agent's background knowledge) through the Pedro server. Each agent maintains a local "yellow-page" directory storing all other agents' advertisements.

Upon receiving a joining notification from the new agent, each existing agent will send its own advertisements to the new agent so that the new agent can initialise its own directory. This directory can be used by its owner agent to identify potential helpers during a distributed inference process.

A query can be submitted to any of the agents, A_i say, in the system, and the abductive answer, if any, is returned by that *same* agent. The reasoning process starts within agent A_i. It tries to construct an abductive answer using only its own knowledge by invoking a meta-interpreter that implements the DARE algorithm. But during its abductive reasoning process, it can "ask for help" from other agents in the group. Each query answer returned by the agent is associated with the *cluster* of agents that have *contributed* to its proof. The main features of the DARE agent architecture are its high level inter-agent communication, the internal concurrency of the agents and the parallel search for alternative abductive proofs. The agents are internally concurrent because they comprise several distinct time-shared threads of computation that co-ordinate via internal thread-to-thread messages and a shared blackboard. The parallelism arises because the agents can be distributed over a network of host computers allowing the different agents requested to help with a sub-proof to search for sub-proofs in parallel. The detailed multi-threaded DARE agent architecture is described in [MRBC08, MBCR08].

3.4.1 Impact of Openness

Allowing agents to join or leave the system during an inference process will not affect the soundness of the final proof for a query. If an agent joins or leaves the system but not the current proof cluster for a query, its background knowledge and integrity constraints are not considered in the current proof and hence the proof is not affected. Once an agent joins a proof cluster, the DARE algorithm will force it to check for consistency of the already abduced assumptions before it can contribute to the inference process. If an agent in the current cluster leaves before the proof is complete, the current DARE system implementation will discard any sub-proof of a condition (i.e., sub-goal) provided by that agent, and will "backtrack" to the point where a sub-proof of that agent was first used (usually this is the point where the leaving agent joined the cluster).

However, the DARE algorithm is not complete (i.e., there exists some answers that the algorithm cannot compute) due to the openness of agent group and the depth-first inference strategy. Consider the following example involving two agents A_1 and A_2 whose background knowledge are given as:

Π_1	Π_2
$p \leftarrow q.$	$q \leftarrow a.$
$p \leftarrow r.$	
$r \leftarrow w.$	$w \leftarrow b.$

where a and b are abducibles. Suppose that initially the system only has A_1, and the query p is submitted to A_1. A_1 first tries to prove p by using the first rule in Π_1. However, it fails to prove q and no helper is available for q (yet). A_2 then tries the second rule in Π_1 and obtains a new goal r. Suppose at this time A_2 joins the system. Then by the time A_1 is attempting to prove w (for r), A_1 can ask for help from A_2 and a successful proof will be obtained for p with the answer $\{b\}$ and the cluster $\{A_1, A_2\}$. However, in the union of the background knowledge $\Pi_1 \cup \Pi_2$ there is another answer, i.e., $\{a\}$, but this answer cannot be computed by the DARE algorithm. This is known as the situation where "the helper agent joins the system too late".

3.4.2 Termination

There are two causes that will lead to non-terminating behaviour of the DARE algorithm.

Cyclic Sub-goals Outsourcing

Consider the following example involving a group of three agents A_1, A_2 and A_3 whose background knowledge are given as:

Π_1	Π_2	Π_3
$p \leftarrow q.$	$p \leftarrow r.$	$p \leftarrow w.$

where all predicates are non-abducible in the shared ontology. Thus, all three agents will advertise p. Suppose a query (goal) p is sent to A_1. A_1 cannot prove it by proving q so it outsources p to A_2 and A_3. A_2 (A_3) cannot prove p either by proving r (w), therefore A_2 (A_3) outsources p to A_1 and A_3 (A_1 and A_2). This generates a non-terminating loop of outsourcing requests. The DARE implementation overcomes this problem by having an agent pass a *DontAsk* in addition to the sub-goal and other proof information to its helpers. The *DontAsk* list contains the identifier of the current agent, and those of the agents it is asking for a sub-proof. In the case where a helper agent fails to prove the given goal, it will not *forward* the goal to any agent in the *DontAsk* list. For example, in the previous scenario, when A_1 outsources p to A_2 and A_3, the list $DontAsk = \{A_1, A_2, A_3\}$ is also passed to the two helpers. When A_2 and A_3 fail to prove p, they will not outsource it to anyone else.

However, the *DontAsk* list technique cannot avoid the following looping situation, which is caused by the *cycles* in the union of all the agent background knowledge:

Π_1	Π_2
$p \leftarrow q.$	$q \leftarrow p.$

In this scenario, the loop of outsourcing requests is between A_1 asking A_2 for help with q and A_2 asking A_1 for help with p. This problem is inherited from KM, whose termination depends on the acyclic condition of the logic program representing the background knowledge.

Rogue Agents that Oscillates between Joining and Leaving

Another non-terminating situation is caused by the openness of the DARE system. Consider the following example with two agents A_1 and A_2 whose background knowledge are given as:

Π_1	Π_2
$p \leftarrow q, w.$ $w.$	$q \leftarrow a.$

where a is an abducible. Suppose initially the system contains both agents and A_1 is given a query p. A_1 asks for help from A_2 for q, and A_2 returns an answer a, which is used by A_1 to continue its proof. However, suppose A_2 leaves while A_1 is trying to prove w. This causes A_1 to "backtrack" its inference to the point where it received an answer from A_2 and tries to find an alternative helper. If at this point A_2 joins the system again (on time), then A_1 will ask it again. Therefore, if A_2 keeps leaving and joining the system like this A_1 will never be able to progress its proof. Although this situation can be avoided by imposing a limit on the number of times an agent can join the system during an inference process, the DARE implementation does not enforce it.

3.5 Limitations of DARE

There are several limitations of the DARE system and algorithm, in addition to incompleteness. This section will give a summary and brief discussion of them.

From the efficiency point of view, the DARE algorithm has performed a considerable amount of redundant computation during the global consistency derivations. For example, in each consistency check round, a set of abducibles is passed between the agents one by one to check for consistency with respect to their local integrity constraints. And if at the end of a consistency check round the set of abducibles has been expanded, a new round will be initiated. It can be observed that the changes to the set of abducibles between consistency check rounds are *incremental*. Thus, each agent should check only those abducibles that it has not seen before at each round. We argue that this limitation is implementation specific, as we can tag each abducible with the identifier of the agents that have checked it for consistency, to avoid any agent having to check "self-tagged" abducibles during the global consistency derivation.

Another limitation is the inflexibility of the DARE algorithm. The algorithm interleaves four derivations, and each new derivation initiated requires inter-agent communication. Consider the situation where an agent A_1 has a rule $p \leftarrow a, b, q$ in Π_1, where a and b are abducibles and q is a non-abducible that no agent can prove. Suppose that the current cluster has a number of agents, each of which has integrity constraints for both a and b, respectively, but can be satisfied after long computation. Given a query (goal) p, due to the adopted depth-first goal selection strategy, A_1 will first assume a which causes a long but successful global consistency derivation, and then assume b which causes another long but successful global consistency derivation, and finally try but fail to solve q, which makes the previous two global consistency derivations unnecessary. Observing from this example, if we allow the global abductive derivation by an agent to continue until all the abducibles are collected, then only one global consistency derivation needs to be invoked and communication overhead may be significantly reduced. Thus, it is desirable to allow *flexible* interleaving between the four derivations during the DARE algorithm execution.

The abductive computation is *cluster-based* instead of *system-based*, i.e., the proof and answer for a query are based on the knowledge of a dynamically selected sub-group of agents in the system, instead of all the agents in the system. An agent is "invited" to join a cluster only if it can help provide sub-proofs for some goals of the cluster's inference process. In some situations, we may want to explicitly add some agents to a cluster for the sake of consistency check for the assumptions made by the cluster. In the current implementation, the extra consistency check agents can be added into the initial cluster of the top-level global abductive derivation. For example, if we want to achieve *system-wide* consistency or abductive computation, we can start the global abductive derivation for a given query with the set of all agents in the system as the initial cluster. However, this is in fact against the openness assumption of the system, which assumes that the group of the agents is unknown at the beginning.

Finally, since the DARE distributed abductive framework considers only normal logic programs and the DARE algorithm performs negation as failure (as in SLDNF), the DARE system is not applicable in many problem domains, in particular those involve reasoning over arithmetic constraints and/or need to compute and exchange non-ground answers, such as planning and scheduling with time and cost. This limitation will be discussed in more detail in 4.1.

3.6 Conclusion

This chapter describes our early work of developing a distributed abductive reasoning (DARE) system by proposing a new distributed abductive algorithm and the architecture of its multi-threaded distributed Qu-Prolog implementation. The DARE algorithm extends the Kakas-Mancarella abductive proof procedure by allowing the background knowledge and integrity constraints to be distributed over a group of agents of which a dynamically selected sub-group co-operate to produce a proof. The system is *open* in that it allows new agents to join or leave the group as they wish at any time. The abduced conditions for a collective proof can come from different agents but they are guaranteed to be consistent with the integrity constraints of all the agents who have contributed to the proof.

The DARE system has several potential applications such as the illustrated multi-agent scheduling and the previously mentioned multi-robot planning and collaborative interpretation of sensor data. We have also developed a distributed abductive planner [MRBL09] as an extension to DARE, based in the abductive Event Calculus planner in [Sha00], for supporting collaborative planning in the context of multiple robots. Because of the openness feature of the DARE system the distributed abductive planner will allow *plan repair* and *plan recovery* to allow the computation of executable plans, even when agents leave the system. For example, suppose that a set of robots have collaboratively computed a plan (i.e., a sequence of actions as the assumed abducibles) for a specific goal (i.e., an abductive query) using DARE, and suppose one of the agents crashes or leaves before the plan can be executed, then the plan will no longer be valid. In this case, DARE can backtrack to the point where this agent first joined the collaborative planning, and re-plan (i.e., resume the distributed abductive inference) from there without the agent. Implementing such a distributed abductive planner with plan repair support is our future work.

However, DARE has a number of limitations, in particular the inflexibility of the algorithm execution and the lacking of arithmetic constraint support, preventing it from being applied in a larger problem domains. In our later work, we focused on addressing such limitations, and developed a new distributed abductive reasoning system that is much more generic and powerful, yet flexible. This system will be described in the following chapters.

Chapter 4

Distributed Abductive REasoning with Constraints (DAREC)

4.1 Introduction

The DARE system allows agents to dynamically form clusters and perform collaborative abduction over their distributed knowledge. The distributed proof procedure of DARE is based on the Kakas-Mancarella proof procedure (KM), and hence inherits the limitation of not being able to make non-ground assumptions (i.e., assume non-ground abducibles). Note that in KM, negative non-abducible literals are treated as (non-base) abducibles. Consequently, DARE cannot accept non-ground negative queries. For example, consider a planning domain with two agents α and β with $\Pi_\alpha = \{can\_fly(Pilot, Day) \leftarrow free(Pilot), \neg storm(Day).\}$ and $\Pi_\beta = \{storm(wed).\}$, respectively, and with $\mathcal{IC}_\alpha = \mathcal{IC}_\beta = \emptyset$ where $free$ is the only abducible predicate. Given a query $can\_fly(X, Y)$, no answer can be computed by α and β in DARE, because after reducing $can\_fly(X, Y)$ to $free(X), \neg storm(Y)$, α cannot progress further assuming $free(X)$ or $\neg storm(Y)$, as they are not ground. One solution to this issue is to change

Π_α to be

$$\Pi'_\alpha = \left\{ \begin{array}{l} can\_fly(Pilot, Day) \leftarrow pilot(Pilot), free(Pilot), day(Day), \neg storm(Day). \\ pilot(p1). \\ pilot(p2). \\ day(mon). \\ \ldots \\ day(sun). \end{array} \right\}$$

This guarantees $free(X)$ and $\neg storm(Y)$ to be eventually grounded and thus assumed by α. However, this solution may cause combinatorial explosion of agent interactions during the collaborative reasoning, e.g., there are $2 \times 7 = 14$ ways to ground $free(X)$ and $\neg storm(Day)$ in this example. In addition, if we are dealing with applications with unbound domains (e.g., the names of the pilots are unknown in the example), then we may not be able to guarantee the grounding of abducibles at all. Ideally, we would like to compute succinct answers of the form $free(X), Y \neq wed$ for the query $can\_fly(X, Y)$ with respect to Π_α and Π_β. To do this, the abductive proof procedure needs to perform *constructive negation* instead of *negation as failure* for negative literals. Another limitation of DARE is the lack of support for finite domain and arithmetic constraints solving. Many applications of abduction, such as planning and scheduling, require reasoning with constraints over time and cost.

Furthermore, operationally the execution of the DARE distributed proof procedure is somewhat too rigid. For example, the collection of a new abducible is immediately followed by a global consistency derivation (`GCD`), which consists of multiple rounds of consistency checks by all the agents in the cluster. Little control can be imposed on the execution to reduce agent communications. Consider the following example of three agents α, β and γ, with

$$\Pi_\alpha = \left\{ \begin{array}{l} p \leftarrow q, r. \\ q \leftarrow a. \end{array} \right\} \quad \mathcal{IC}_\beta = \{\leftarrow a, c.\} \quad \mathcal{IC}_\gamma = \{\leftarrow a, d.\}$$

and

$$\mathcal{IC}_\alpha = \Pi_\beta = \Pi_\gamma = \emptyset$$

where a is the only abducible. Suppose that α has to solve the goal p and the current cluster contains all three agents, and a left to right goal selection strategy is adopted. Then during α's local abductive derivation (`LAD`), the abducible a can be selected before r, causing then a `GCD` among three agents. Although the `GCD` is successful, it is unnecessary if α had first chosen r in its `LCD`, i.e, r would fail and the `LCD` would backtrack without the need to check a with others.

The focus of this chapter is to present a new distributed abductive reasoning system, called DAREC, that overcomes the limitations of DARE, thus supporting a wider class of distributed knowledge-based problem solving tasks. Specifically, DAREC differs from DARE in the following ways:

- DAREC allows the reasoning of inequalities over logical terms (e.g., $f(a, X) \neq f(Y, b)$ gives either $Y \neq a$ or $X \neq b$) and the reasoning of finite domain constraints (i.e., a type of Constraint Logic Programming constraints), by using an external inequality solver and finite domain constraint solvers.

- DAREC handles negative literal goals as integrity constraints instead of (non-base) abducibles, and performs constructive negation. For example, a negative literal goal containing universal or existential variables (e.g., $\forall Day.\neg storm(Day)$ or $\exists Day.\neg storm(Day)$) can be treated as an integrity constraint (e.g., $\forall Day. \leftarrow storm(Day)$ or $\exists Day. \leftarrow storm(Day)$). Given a fact $storm(wed)$, the computation of $\forall Day. \leftarrow storm(Day)$ returns no answer (i.e., fails), and that of $\exists Day. \leftarrow storm(Day)$ succeeds and gives $Day \neq wed$.

- DAREC focuses collaborative abduction among a *fixed* set of agents, i.e., the *cluster* is always equal to the set of all the agents in the system.

In addition, DAREC has a completely new distributed proof procedure, which is based on ASystem [KvND01] instead of KM. The collaborative abduction can be seen as a distributed state rewriting/search process, where each *state* contains intermediate computational results such as the remaining goals, the assumed abducibles with their constraints (e.g., inequalities and CLP constraints), and the dynamically collected integrity constraints (e.g., derived from negative goals) that need to be satisfied by all agents. This process involves local computations by the agents and the interactions between agents through state passing:

1. after a query is sent to the system, an initial state containing only the query is created;

2. for any given state, an agent can apply a set of local abductive inference rules to rewrite it into a set of new states, using its local knowledge and integrity constraints;

3. if a state generated during the local computation by an agent contains a goal that the agent cannot reduce using its local knowledge, or contains an assumed abducible or a dynamically collected integrity constraint that has not been checked by all the agents, then the agent can pass the state to other agents for further processing;

4. if a state is generated without any remaining goal or unchecked abducible or unchecked collected integrity constraint, then an answer can be extracted from it.

The coordination between the local computations is controlled by the *agent interaction strategy* (e.g., when to pass a state) and the *agent selection strategy* (e.g., whom to pass a state to), which are application dependent and customisable in order to optimise the agent collaboration (e.g., reducing the number of interactions and communications).

The rest of this chapter is organised as follows. Section 4.2 defines the DAREC framework for knowledge representation. Section 4.3 describes the DAREC distributed proof procedure and illustrates it with a simple running example. Section 4.4 proves the soundness and completeness of the system. Section 4.5 discusses several potential extensions to the system. Finally, Section 4.6 concludes this chapter.

4.2 Distributed Framework for Fixed Agent Systems

To model a distributed abductive reasoning problem for a given multi-agent system in DAREC, a logical framework based on the DARE *distributed abductive context* is used. The framework has the following assumptions:

1. It refers to *fixed* set of agents, each of which has a *unique ID* and represented as an *abductive framework*. This assumption allows us to model distributed knowledge of the multi-agent system.

2. All predicates are *global* to the agents, and the set of abducible predicates is agreed by all agents. These two assumptions ensure that agents talk in the same language and cannot generate hypotheses that are provable by others.

3. Any two agents can send peer-to-peer messages to each other. This ensures that the communication graph for the multi-agent system is *fully connected*, and hence eliminates the need to consider message routing problems.

Similarly to DARE, our main focus is on the correctness of the distributed abductive reasoning system. It is further assumed that the agents and the communication channel are reliable, i.e. there is no corruption or loss of messages.

Recall that an agent's abductive framework is $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$, where Π is a finite set of rules called the *background knowledge*, $\mathcal{IC}$ is a finite set of denials called the *integrity constraints*, and $\mathcal{AB}$ is the set of abducible predicates. Π and $\mathcal{IC}$ together constitute the agent's local knowledge (or *local expertise*). When it is necessary, we may use the agent's identifier, say i, to suffix the agent's framework and its components, i.e. $\mathcal{F}_i$, Π_i, $\mathcal{AB}_i$ and $\mathcal{IC}_i$. The *global expertise* (in contrast to local expertise) of a set of agents is represented by the notation of *global abductive framework*.

Definition 4.1 ((DAREC) Global Abductive Framework). *The (DAREC) global abductive framework for a system of abductive agents, is a tuple* $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$*, where* Σ *is the set of* *all agent identifiers and* $\widehat{\mathcal{F}}$ *is the set of abductive agent frameworks, i.e.* $\{\mathcal{F}_i \mid i \in \Sigma\}$*. For any pair of agents* $i, j \in \Sigma$, $\mathcal{AB}_i = \mathcal{AB}_j$.

Note that the DARE$\mathcal{C}$ global abductive framework differs from the DARE distributed abductive context in the following ways. First, it is independent of the query and the initial agent who receives the query (i.e., they are not included in the tuple). Secondly, since the set of agents is fixed, we do not consider *evolved* global abductive framework (until Section 4.5.2).

Consider the following ambient intelligent system in a sheltered home for elderly people, where mobile or embedded devices are used for monitoring in house security and aiding the daily life of the occupants.

Example 4.1. *Ann and Bob live in the same care home, where a number of sensing devices are installed. For example, a corridor sensor (***cor***) detects movements along the corridor, and a window monitor (***wm***) can check which window(s) of the house are open/closed. There is also a home controller (***hm***) that can respond to events taking place inside the house, such as setting off an alarm if an intruder is detected, or notifying a nurse when a resident is in difficulty. Bob has a mental condition. Unless taking regular medication, he tends to wander around the house instead of staying in his room. So Bob is always carrying a personal device (***bob***) that logs his medication intakes. Ann is in good health and can leave the house when necessary, e.g. going to a dental appointment. Ann also carries a personal device (***ann***) which keeps her calendar and appointments. All the sensing and personal devices (except the base sensors, like the corridor sensor, which merely generate detected event notifications to* **hm***) have reasoning capability. About 12pm on Monday,* **cor** *detects movement and informs* **hm***.* **hm** *then needs to collaborate with various devices to explain the event before taking appropriate action.*

The above system can be modelled using a global abductive framework, where each device is an abductive agent, e.g.:

$\mathcal{F}_{bob}$: Bob cannot be walking in the corridor if he has taken medicine in the past 2 hours. His most recent intake is at 11am.

$$\left[\begin{array}{l} \Pi_{bob} = \left\{ \begin{array}{l} takenMedicine(11). \end{array} \right\} \\ \mathcal{IC}_{bob} = \left\{ \begin{array}{l} \leftarrow walkInCorridor(bob, T), takenMedicine(T1), \\ \qquad T - 2 \leq T1, T1 \leq T. \end{array} \right\} \end{array}\right]$$

$\mathcal{F}_{ann}$: Ann has a dental appointment from 11am to 1pm.

$$\left[\begin{array}{l} \Pi_{ann} = \left\{ \begin{array}{l} appointment(dental, 11, 13). \\ out(ann, T) \leftarrow \\ \qquad appointment(A, T1, T2), T1 \leq T, T \leq T2. \end{array} \right\} \\ \mathcal{IC}_{ann} = \emptyset \end{array}\right]$$

$\mathcal{F}_{wm}$: The window monitor has the status information of the windows on different floors. Any open window (we assume that the fact *open*(Win) is asserted/retracted from the background knowledge whenever the window Win is open/closed) on the 1^{st} floor is a possible point of entry for a potential intruder.

$$\left[\begin{array}{l} \Pi_{wm} = \left\{ \begin{array}{l} pointOfEntry(T) \leftarrow \\ \qquad open(W), floor(W, 1). \\ open(w1). \\ floor(w1, 1). \\ floor(w2, 2). \end{array} \right\} \\ \mathcal{IC}_{wm} = \emptyset \end{array}\right]$$

$\mathcal{F}_{hm}$: The home controller has knowledge about possible causes to known events. For example, movement in the corridor can be of either an occupant or an intruder.

$$\left[\begin{array}{l} \Pi_{hm} = \left\{\begin{array}{l} movement(cor, T) \leftarrow \\ \qquad occupant(X), walkInCorridor(X, T). \\ movement(cor, T) \leftarrow \\ \qquad pointOfEntry(T), walkInCorridor(intruder, T). \\ occupant(X) \leftarrow \\ \qquad X \in \{ann, bob\}. \end{array}\right\} \\ \mathcal{IC}_{hm} = \left\{ \leftarrow walkInCorridor(X, T), X \neq intruder, out(X, T). \right\} \end{array}\right]$$

In order to explain the notified event of movement in the corridor ($movement(cor, 12)$), **hm** needs to find out who could be walking in the corridor at 12. If it is Bob, then a nurse needs to be notified (remotely). If it is an intruder, then the alarm needs to be set off. If it is Ann, no action needs to be taken. All agents have the single abducible predicate $walkInCorridor$, i.e. $\mathcal{AB}_{hm} = \mathcal{AB}_{wm} = \mathcal{AB}_{ann} = \mathcal{AB}_{bob} = \{walkInCorridor\}$. Thus, the global abductive framework for Example 4.1 is $\langle\{hm, wm, ann, bob\}, \{\mathcal{F}_{hm}, \mathcal{F}_{wm}, \mathcal{F}_{ann}, \mathcal{F}_{bob}\}\rangle$.

Definition 4.2 ((DARE*C*) Global Abductive Answer). *Given a (DAREC) global abductive framework* $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$ *and a query* $\mathcal{Q}$, *let* $\widehat{\Pi} = \bigcup_{i\in\Sigma} \Pi_i$, *let* $\widehat{\mathcal{IC}} = \bigcup_{i\in\Sigma} \mathcal{IC}_i$, *and let* $\widehat{\mathcal{AB}} = \bigcup_{i\in\Sigma} \mathcal{AB}_i$. *A pair* $\langle\Delta, \theta\rangle$ *is a (DAREC) global abductive answer for* $\mathcal{Q}$ *if and only if:*

- $\Delta\theta \subseteq \widehat{AB}$;
- $\widehat{\Pi} \cup \Delta\theta \models \mathcal{Q}\theta$;
- $\widehat{\Pi} \cup \Delta\theta \models \widehat{\mathcal{IC}}$

where θ *is the variable substitutions over the variables in* $\mathcal{Q}$, *and* $\models$ *is the logical entailment of a selected semantics for the logic program formed by* $\widehat{\Pi} \cup \widehat{\mathcal{IC}}$.

4.3 Distributed Algorithm

4.3.1 Overview

Given a global abductive framework and a query, a *global abductive task* of the agents is to collaboratively compute the global abductive answers for the query. A new distributed algorithm has been developed for such a task in DAREC. Operationally, the distributed computation by the agents is a sequence of coordinated local abductive computations, i.e.,

distributed abduction = local abduction + coordination

The local abduction by an agent is a top-down (goal-directed) abductive inference, which is extended from the ASystem proof procedure [KvND01] and can be described as a state rewriting and search process. Each *computational state* (or *state* in brief) encapsulates the intermediate computation results of a global abductive task, and contains information such as *remaining goals*, *assumed abducibles* (i.e., *hypotheses made*), *collected arithmetic constraints* and the *collected consistency constraints* (in denial form) that need to be satisfied by all the agents. Thus, each state is sufficient to describe a new global abductive task that can subsume the original one, i.e., every answer for the task described by the state is also an answer for the original task. The first state of a global abductive task is called the *initial state*, and contains only the query. A *solved state* is one that contains no remaining goals and the collected constraints are all checked and satisfied by every agent, and from which global abductive answers can be extracted. Each agent's local abduction starts with a given *root state*, and involves a series of *inference steps*. Each inference step replaces a state to a (possibly empty) set of states, each given by a goal selected from the state and the application of an *inference rule*. The objective of a local abduction is to search for solved states, and its execution is influenced by the *goal selection strategy* adopted by the agent.

Differently from centralised abduction, a local abductive inference step may allow an agent to collect a (non-abducible) goal as a *delayed goal* when it does not have sufficient background knowledge to reduce it. Every new abducible or consistency constraint collected by the agent

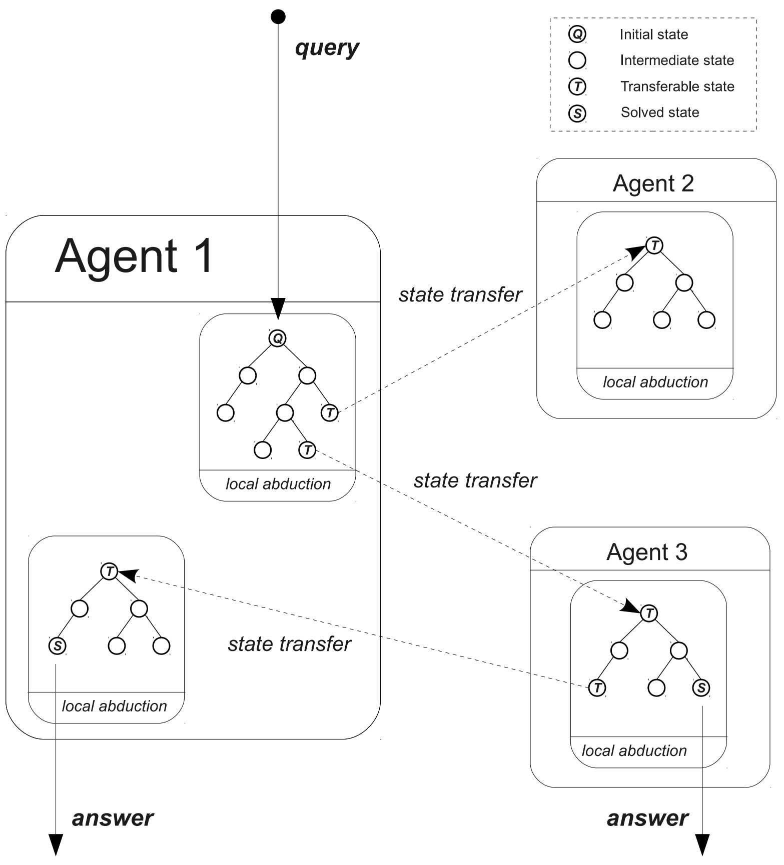


Figure 4.1: Distributed Abduction

has to be checked by all other agents. Thus, during local abduction a special type of non-solved (*intermediate*) states may be generated, which contain *delayed goals* or collected abducibles/-consistency constraints yet to be checked by other agents. These states are called *transferable* states. For a generated transferable state, the owner agent can either process further the state with a local inference step (e.g., to reduce remaining goals), or pass it to another agent (e.g., to deal with the delayed goals), according to an *agent interaction strategy* and an *agent selection strategy*. Note that each transferable state sufficiently describes a global abductive task, thus the recipient agent can start its own local abduction (with the received state being the root state) to continue the global search process independently. Let us remark that during the agents' collaboration for a given global abductive task, each agent may receive several states and perform corresponding local abductions simultaneously, and any answer found from any of these local abductions is an answer for the given task. Figure 4.1 visualises an example of distributed abduction, where each of the local abductions is represented as a *tree* of states.

4.3.2 Notations of State + Local Abductive Derivation

In this section, we give formal definitions for the data structures used in the distributed algorithm. As previously mentioned, a state may contain hypotheses and constraints that need to be checked by all agents at least once. In order to record who has checked what, we introduce the concept of tagging for literals or denials:

Definition 4.3 (Tagging). *A tag is a pair* (L, S)*, where* L *is either a literal or a denial, and* S *is a (possibly empty) set of agent identifiers. Given a set* τ *of tags, a literal or denial* L *is* tagged *by an agent* α *(or* α *has tagged* L*) if and only if* $(L, S) \in \tau$ *and* $\alpha \in S$*.*

Sometimes we also use L^S to denote that a literal or denial L is tagged by the agents in S. Tags are used during distributed abduction in one of the following ways:

- A goal may be tagged by the agents who have delayed it. Such information can be used to prevent an agent from delaying the same goal more than once, and hence avoid the non-progressive cyclic state-passing between agents. For example, suppose a system has two agents α and β. Given a query containing only one goal p, we need to stop the agent interactions cycling between α delays p then sends a state to β, and β delays p then sends a state to α.

- An abducible may be tagged by the agents who have not checked its consistency with respect to their local integrity constraints.

- A denial (collected consistency constraint) may be tagged by agents who have not checked its satisfiability.

Examples of these usages will be given in Section 4.3.3.

Definition 4.4 ((DAREC) Computational State). *A (DAREC) computational state (or state in brief) is a tuple* $\Theta = \left\langle (\mathcal{G}, \mathcal{G}^d), \mathcal{ST}, \tau \right\rangle$*, where*

- *each element in $\mathcal{G}$ is a remaining goal, and can be either a literal or a denial of the form $\forall\vec{X}. \leftarrow \phi_1, \ldots, \phi_n$ $(n > 0)$ where $\vec{X}$ is the set of universally quantified variables of the denial* [1]*;*
- *each element in $\mathcal{G}^d$ is a delayed goal and must be a non-abducible;*
- *$\mathcal{ST}$ is a tuple of four stores $(\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C})$, where*
  - *Δ is a set of abducibles;*
  - *$\mathcal{N}$ is a set of denials $\forall\vec{X}. \leftarrow \phi_1, \ldots, \phi_n$ $(n > 0)$, where the ordering of $\phi_1, \ldots, \phi_n$ matters and ϕ_1 is either an abducible or non-abducible;*
  - *$\mathcal{E}$ is a set of (in-)equalities;*
  - *$\mathcal{C}$ is a set of CLP constraints;*
- *τ is a set of tags. All free variables appearing in the state Θ are existentially quantified within the scope of the whole state.*

A denial in $\mathcal{N}$ is called a *consistency constraint*, and its first body literal is called its *constrained literal*. More specifically, such a denial is called an *abducible (consistency) constraint* (on ϕ) if its constrained literal ϕ is an abducible, or is called a *non-abducible (consistency) constraint* (on ϕ) if ϕ is a non-abducible. Intuitively, an abducible constraint must be satisfied by all the agents so that for every assumed instance of its constrained abducible, the rest of the denial body must not be provable by any agent. For example, let $\forall X. \leftarrow a(X), p(X)$ be an abducible constraint. If the abducible $a(1)$ is assumed by the agents, then $p(1)$ must not be provable by any agent. A non-abducible constraint must be satisfied by all the agents so that for every instance of its constrained non-abducible that is provable by some agent, the rest of the denial body must not be provable by any of the agents. For example, let $\forall X. \leftarrow p(X), q(X)$ be a non-abducible constraint. If $p(X)$ is provable by some agent with $X = 1$, then $q(1)$ must not be provable by any agent.

According to the cases in which tags can be used, only positive non-abducible goals, collected abducibles, and non-abducible constraints of a state may be tagged. Note that other types

[1] i.e., $\vec{X}$ are variables appearing in the denial and are within the scope of the whole denial.

of goals, such as abducibles and arithmetic constrains can be reduced by any agent regardless of their background knowledge, and hence we do not allow them to be delayed. Note also that negative goals are always converted into denial goals (as described in the corresponding local inference rules in Section 4.3.3), and denial goals cannot be delayed as they are in fact constraints.

Note that a DAREC computational state is similar to an ASystem computational state, with only the following differences:

- an ASystem state does not contain *delayed* goals;
- the denial set in an ASystem state contains only *abducible constraints*;
- nothing in an ASystem state can be tagged.

As mentioned earlier, a DAREC computational state can be of 3 different types. These are defined as follows.

Definition 4.5 (Initial State). *The initial state of a global abductive task for a query $\mathcal{Q}$ is $\langle(\mathcal{Q},\emptyset),(\emptyset,\emptyset,\emptyset,\emptyset),\emptyset\rangle$.*

To simplify notation, sometimes we use $\mathcal{ST}^{\emptyset}$ to denote the four empty stores of a state, i.e., $\mathcal{ST}^{\emptyset} = (\emptyset,\emptyset,\emptyset,\emptyset)$.

Definition 4.6 (Solved State). *A state $\Theta = \big\langle(\mathcal{G},\mathcal{G}^d),(\Delta,\mathcal{N},\mathcal{E},\mathcal{C}),\tau\big\rangle$ is a solved state if $\mathcal{G} = \mathcal{G}^d = \emptyset$, $\mathcal{E}\cup\mathcal{C}$ is consistent and no element in $\Delta\cup\mathcal{N}$ is tagged according to τ.*

After a solved state $\langle(\emptyset,\emptyset),(\Delta,\mathcal{N},\mathcal{E},\mathcal{C}),\tau\rangle$ is obtained during distributed abduction, we can extract the DAREC answers $\langle\Delta,\theta\rangle$ from it where θ is the set of variable substitutions induced by $\mathcal{E}\cup\mathcal{C}$.

Definition 4.7 (Transferable State). *A state $\Theta = \big\langle(\mathcal{G},\mathcal{G}^d),(\Delta,\mathcal{N},\mathcal{E},\mathcal{C}),\tau\big\rangle$ is a transferable state of an agent α's local abduction if one of the following conditions is satisfied:*

- *$\mathcal{G}^d$ is not empty;*

- *an abducible in Δ is tagged by some agent(s) according to τ;*
- *a non-abducible constraint in $\mathcal{N}$ is tagged by some agent(s) according to τ.*

Note that a transferable state can contain both delayed goals and tagged abducibles/constraints. Also a transferable state may be further processed by the same agent (in contrast to passing it to another agent) if it still contains remaining goals. The decision of whether to pass or to process that state is based on the agent interaction strategy being adopted by the agent.

The search space of an agent's local abduction, given a goal selection strategy and an agent interaction strategy, can be described as a *local abductive derivation tree*:

Definition 4.8 (Local Abductive Derivation Tree). *Given a goal selection strategy Ξ and an agent interaction strategy Υ, the local abductive derivation tree for a local abduction by an agent α a tree where:*

- *the root node is the root state of the local abduction,*
- *the children nodes are the states obtained by the application of a local inference rule to the parent node according to Ξ and Υ.*

.

4.3.3 Local Abductive Inference Rules

In this section, we will describe the set of local abductive inference rules of DAREC . There are ten rules and they are extended from the rules of the ASystem [vN04] (which are summarised in Section 2.3.4) with the following three features: (i) option to delay non-abducible goals, (ii) update of tags, and (iii) handling of non-abducible constraints. Below is a table summary of the rules. We will describe them in order.

Reduction of non-denial goals		
Name:	Comment:	Relation to ASystem Rules
LD1	Resolve Non-abducible	*extended* **D1**
LA1	Resolve Abducible	*extended* **A1**
LC1	Reduce CLP Constraint	*reformulated* **C1**
LE1	Reduce (In-)equality	*reformulated* **E1**
LN1	Rewrite Negation	*reformulated* **N1**
Reduction of denial goals		
Name:	Comment:	Relation to ASystem Rules
LD2	Resolve Denial through Non-abducible	*extended* **D2**
LA2	Resolve Denial through Abducible	*extended* **A2**
LC2	Reduce Denial through CLP Constraint	*reformulated* **C2**
LE2	Reduce Denial through Equality	*reformulated* **E2**
LN2	Rewrite Denial through Negation	*reformulated* **N2**

As we mentioned in Section 4.3.1, at each inference step a remaining goal is selected from a given non-solved state, and a (possibly empty) set of next states are generated according to a corresponding local inference rule. Here we assume that a *safe* goal selection strategy Ξ (as the one defined for ASystem) is adopted, which never selects a non-safe goal that will cause floundering.

Given a global abductive framework $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$, let $\Theta_i = \langle (\mathcal{G}_i, \mathcal{G}_i^d), (\Delta_i, \mathcal{N}_i, \mathcal{E}_i, \mathcal{C}_i), \tau_i \rangle$ be a state yet to be processed in the local abductive derivation by an agent $\alpha \in \Sigma$, where $\mathcal{G}_i \neq \emptyset$. Suppose a goal ϕ is selected from $\mathcal{G}_i$ according to Ξ, and let $\mathcal{G}_i^- = \mathcal{G}_i - \{\phi\}$.

If ϕ is not a denial goal, one of the following five local rules is chosen according to the type of ϕ. For the sake of simplicity, only changes to the state components are described, and `OR` denotes alternative modifications to Θ_i. Thus, the set of next states contains all the possible states modified from Θ_i.

Inference Rule (LD1). *If $\phi = p(\vec{u})$ is a non-abducible that is not tagged by α, let $p(\vec{v}_j) \leftarrow \Phi_j$*

$(j = 1, \ldots, n)$ be n rules in Π_α, then:

– *(local reduction)* $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_1\} \cup \Phi_1 \cup \mathcal{G}_i^-$, *and* $\tau_{i+1} = \tau_i - \{\langle \phi, \mathcal{S} \rangle\}$

`OR` $\vdots$

`OR` *(local reduction)* $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_n\} \cup \Phi_n \cup \mathcal{G}_i^-$, *and* $\tau_{i+1} = \tau_i - \{\langle \phi, \mathcal{S} \rangle\}$

`OR` *(delay goal)* $\mathcal{G}_{i+1} = \mathcal{G}_i^-$, $\mathcal{G}_{i+1}^d = \mathcal{G}_i^d \cup \{\phi\}$, *and*

  – *if* $\langle \phi, \mathcal{S} \rangle \in \tau_i$, *then* $\tau_{i+1} = \{\langle \phi, \{\alpha\} \cup \mathcal{S} \rangle\} \cup (\tau_i - \{\langle \phi, \mathcal{S} \rangle\})$;

  – *otherwise* $\tau_{i+1} = \{\langle \phi, \{\alpha\} \rangle\} \cup \tau_i$

This rule allows α to either resolve a non-abducible goal with a rule in Π_α, or to delay it for other agents to solve later. In the case where α delays ϕ, ϕ will become tagged by α. Note that if α has delayed ϕ once (i.e, ϕ is already tagged by α), then it is not allowed to delay ϕ again. This is important for eliminating the looping situations caused by some goal being infinitely delayed. Consider the following example. Let $\Sigma = \{\alpha, \beta\}$, and let us suppose that α applied **LD1** to delay a goal, say p, in some state Θ_1 and sent Θ_1 to β. Suppose also that during β's local abduction, β applied **LD1** to delay p too and sent the new state Θ_2 back to α. Then when α selects p again during its new local abduction, p has already been tagged by α and hence α is not allowed to delay p again or send the state back to β.

Inference Rule. (LA1) *If $\phi = a(\vec{u})$ is an abducible, let $a(\vec{v}_j)(j = 1, \ldots, n)$ be n abducibles in Δ_i, then:*

– *(reuse assumption)* $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_1\} \cup \mathcal{G}_i^-$

`OR` $\vdots$

`OR` *(reuse assumption)* $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_n\} \cup \mathcal{G}_i^-$

`OR` *(new assumption)* $\Delta_{i+1} = \{\phi\} \cup \Delta_i$ *and* $\mathcal{G}_{i+1} = R_\Delta \cup R_\mathcal{N} \cup R_\mathcal{IC} \cup \mathcal{G}_i^-$, *where*

  – $R_\Delta = \{\leftarrow \vec{u} = \vec{v}_j \mid j = 1, \ldots, n\}$,

- $R_{\mathcal{N}} = \{\forall \vec{X}. \leftarrow \vec{u} = \vec{w}, \Phi \mid \forall \vec{X}. \leftarrow a(\vec{w}), \Phi \in \mathcal{N}_i\}$,

- $R_{\mathcal{IC}} = \{\forall \vec{X}. \leftarrow \vec{u} = \vec{w}, \Phi^- \mid \leftarrow \Phi \in \mathcal{IC}_\alpha \texttt{ and } \Phi^- = \Phi - \{a(\vec{w})\} \texttt{ and } \vec{X} = vars(\Phi)\}$,

and $\tau_{i+1} = \{\langle \phi, \Sigma - \{\alpha\}\rangle\} \cup \tau_i$

This rule allows α to reuse an abducible already assumed, or to assume new abducible (make new assumption). In the latter case (i.e. moving ϕ to Δ_i), three sets of new denial goals are generated: R_Δ ensures that ϕ is different from any existing abducible in Δ_i, R_N ensures that by adding ϕ to Δ_i all the abducible constraints in $\mathcal{N}_i$ are still satisfiable, and $R_{\mathcal{IC}}$ ensures that ϕ is consistent with α's local integrity constraints $\mathcal{IC}_\alpha$. New tags $\langle \phi, \Sigma - \{\alpha\}\rangle$ indicate that all other agents need to check ϕ with their local integrity constraints.

Example (Agent α applies **LA1**). [2]

[2] In order to simplify the state tuple notation, we use L^S to denote that a literal or a denial L in a state is tagged by a set of agents S, and drop the tags component τ when describing a state tuple.

$\Theta_i : \textit{given } \Sigma = \{\alpha, \beta\}, \mathcal{IC}_\alpha = \{\leftarrow a(V), w(V)\}$
$\mathcal{G}_i = \{a(X)\}$
$\mathcal{G}_i^d = \emptyset$
$\Delta_i = \{a(Z)^{\{\alpha,\beta\}}\}$
$\mathcal{N}_i = \{\forall Y. \leftarrow a(Y), r(Y)\}$
$\mathcal{E}_i = \emptyset$
$\mathcal{C}_i = \emptyset$

$\Downarrow$ $\Downarrow$

Θ_{i+1} :
$\mathcal{G}_{i+1} = \emptyset$
$\mathcal{G}_{i+1}^d = \mathcal{G}_i^d$
$\Delta_{i+1} = \Delta_i$
$\mathcal{N}_{i+1} = \mathcal{N}_i$
$\mathcal{E}_{i+1} = \{X = Z\}$
$\mathcal{C}_{i+1} = \mathcal{C}_i$

Θ'_{i+1} :
$\mathcal{G}'_{i+1} = \left\{ \begin{array}{l} X \neq Z, \\ \forall V. \leftarrow X = V, w(V), \\ \forall Y. \leftarrow X = Y, r(Y) \end{array} \right\}$
${\mathcal{G}_{i+1}^d}' = \mathcal{G}_i^d$
$\Delta'_{i+1} = \{a(X)^{\{\beta\}}, a(Z)^{\{\alpha,\beta\}}\}$
$\mathcal{N}'_{i+1} = \mathcal{N}_i$
$\mathcal{E}'_{i+1} = \mathcal{E}_i$
$\mathcal{C}'_{i+1} = \mathcal{C}_i$

In this example, Θ_{i+1} is obtained by reusing $a(Z)$, and Θ'_{i+1} is obtained by making a new assumption $a(X)$ and dynamically generating the new goals $R_\Delta = \{X \neq Z\}$, $R_\mathcal{N} = \{\forall Y. \leftarrow X = Y, r(Y)\}$, and $R_\mathcal{IC} = \{\forall V. \leftarrow X = V, w(V)\}$. ■ *End of example.*

Inference Rule (LC1). *If ϕ is a constraint, let $\mathcal{C}_{new} = \{\phi\} \cup \mathcal{C}_i$:*

– *if $\mathcal{C}_{new}$ is consistent, then $\mathcal{C}_{i+1} = \mathcal{C}_{new}$ and $\mathcal{G}_{i+1} = \mathcal{G}_i^-$.*

Inference Rule (LE1). *If ϕ is an (in-)equality, let $\mathcal{E}_{new} = \{\phi\} \cup \mathcal{E}_i$:*

– *if $\mathcal{E}_{new}$ is consistent, then $\mathcal{E}_{i+1} = \mathcal{E}_{new}$ and $\mathcal{G}_{i+1} = \mathcal{G}_i^-$.*

Inference Rule (LN1). *If $\phi = \neg p(\vec{u})$, then:*

– $\mathcal{G}_{i+1} = \{\leftarrow p(\vec{u})\} \cup \mathcal{G}_i^-$.

Local rules **LC1** and **LE1** collect a consistent constraint or a consistent (in)equality to the corresponding stores. Local rule **LN1** simply converts a negative goal into a denial goal, so that it can be processed by one of the remaining five local rules when it is selected.

If ϕ is a denial goal of the form $\forall\vec{X}. \leftarrow \Gamma$ where Γ is not empty, suppose that a (safe) literal φ is selected from Γ according to Ξ, and let $\Gamma^- = \Gamma - \{\varphi\}$. One of the following five local rules is chosen according to φ, and the next state Θ_{i+1} is obtained after the application of the chosen rule. Note that if Γ is empty, no literal can be selected, and hence no inference rule is applicable. In this case, the set of possible next states is empty.

Inference Rule (LD2). *If $\varphi = p(\vec{u})$ is a non-abducible, let $F = \forall\vec{X}. \leftarrow p(\vec{u}), \Gamma^-$, then:*

– $\mathcal{G}_{i+1} = \{\forall\vec{Y}. \leftarrow \Gamma^+ \mid p(\vec{v}) \leftarrow \Phi \in \Pi_\alpha \text{ and } \vec{Y} = \vec{X} \cup vars(p(\vec{v})) \cup vars(\Phi) \text{ and } \Gamma^+ = \{\vec{u} = \vec{v}\} \cup \Phi \cup \Gamma^-\} \cup \mathcal{G}_i^-$, *and* $\mathcal{N}_{i+1} = \{F\} \cup \mathcal{N}_i$, *and* $\tau_{i+1} = \{\langle F, \Sigma - \{\alpha\}\rangle\} \cup \tau_i$.

Every denial goal is a global constraint, and must be satisfied by all agents in the system. If the denial goal is to be reduced by α through a selected non-abducible literal, then α alone may not be able to generate all the necessary new denial goals, which together imply the original denial goal, because α may not have all the knowledge (i.e., *definitions*) about the non-abducible. Thus, the denial goal has to be collected and tagged, so that it can be checked by other agents later.

Example. *Agent α applies* **LD2**

$\Theta_i : given\ \Sigma = \{\alpha, \beta\}, \Pi_\alpha = \left\{ \begin{array}{l} p(U) \leftarrow r(U), \\ p(V) \leftarrow w(V) \end{array} \right\}$
$\mathcal{G}_i = \{\leftarrow p(X), q(X)\}$ $\mathcal{G}_i^d = \emptyset$ $\Delta_i = \emptyset$ $\mathcal{N}_i = \emptyset$ $\mathcal{E}_i = \emptyset$ $\mathcal{C}_i = \emptyset$

$$\Downarrow$$

Θ_{i+1} :
$\mathcal{G}_{i+1} = \left\{ \begin{array}{l} \forall U. \leftarrow X = U, r(U), q(X), \\ \forall V. \leftarrow X = V, w(V), q(X) \end{array} \right\}$ $\mathcal{G}_{i+1}^d = \mathcal{G}_i^d$ $\Delta_{i+1} = \Delta_i$ $\mathcal{N}_{i+1} = \{\leftarrow p(X), q(X)^{\{\beta\}}\}$ $\mathcal{E}_{i+1} = \mathcal{E}_i$ $\mathcal{C}_{i+1} = \mathcal{C}_i$

In this example, $\forall U. \leftarrow X = U, r(U), q(X)$ *and* $\forall V. \leftarrow X = V, w(V), q(X)$ *are new denial goals obtained by resolving the non-abducible* $p(X)$ *in the denial* $\leftarrow p(X), q(X)$ *with* Π_α*. The denial is tagged by* β *and moved to* $\mathcal{N}_{i+1}$*.* ■ *End of example.*

Inference Rule (LA2). *If* $\varphi = a(\vec{u})$ *is an abducible, let* $F = \forall \vec{X}. \leftarrow a(\vec{u}), \Gamma^-$*, then:*

– $\mathcal{G}_{i+1} = \{\forall \vec{X}. \leftarrow \Gamma^+ \mid a(\vec{v}) \in \Delta_i \text{ and } \Gamma^+ = \{\vec{u} = \vec{v}\} \cup \Gamma^-\} \cup \mathcal{G}_i^-$*, and* $\mathcal{N}_{i+1} = \{F\} \cup \mathcal{N}_i$.

When a (global constraint) denial goal is to be reduced through an abducible, α can generate all the necessary new denial goals using **LA2**, because α can see all the abducibles collected in the state. However, this denial goal still needs to be kept in $\mathcal{N}_{i+1}$, so that if a new instance of the abducible is assumed later in the local abduction, additional denial goals can be generated.

But note that the denial goal collected by **LA2** does not need to be tagged, as any agent can generate all the necessary additional denial goals for it when a new abducible is assumed.

Example. *Agent α applies* **LA2**

$$\begin{array}{|l|}\hline \Theta_i : \textit{given } \Sigma = \{\alpha, \beta\} \\ \hline \mathcal{G}_i = \{\forall X. \leftarrow a(X), p(X)\} \\ \mathcal{G}_i^d = \emptyset \\ \Delta_i = \{a(Y)^{\{\alpha\}}, a(Z)^{\{\alpha\}}\} \\ \mathcal{N}_i = \emptyset \\ \mathcal{E}_i = \emptyset \\ \mathcal{C}_i = \emptyset \\ \hline \end{array}$$

$$\Downarrow$$

$$\begin{array}{|l|}\hline \Theta_{i+1} : \\ \hline \mathcal{G}_{i+1} = \left\{ \begin{array}{l} \forall X1. \leftarrow X1 = Z, p(X1), \\ \forall X2. \leftarrow X2 = Y, p(X2) \end{array} \right\} \\ \mathcal{G}_{i+1}^d = \mathcal{G}_i^d \\ \Delta_{i+1} = \Delta_i \\ \mathcal{N}_{i+1} = \{\forall X. \leftarrow a(X), p(X)\} \\ \mathcal{E}_{i+1} = \mathcal{E}_i \\ \mathcal{C}_{i+1} = \mathcal{C}_i \\ \hline \end{array}$$

In this example, $\forall X1. \leftarrow X1 = Z, p(X1)$ and $\forall X2. \leftarrow X2 = Y, p(X2)$ are new denial goals obtained by resolving the abducible in the denial $\forall X. \leftarrow a(X), p(X)$ with the two abducibles in Δ_i. The denial is also moved to $\mathcal{N}_{i+1}$ but not tagged by any agent (unlike in **LD2***).* ■
<u>*End of example.*</u>

Inference Rule (LC2). *If φ is a CLP constraint where $vars(\varphi) \cap \vec{X} = \emptyset$ (i.e., it does not contain any universal variable of ϕ), let $\overline{\varphi}$ be the* negated *constraint of φ* [3]*, then:*

[3]The negated constraint $\overline{\varphi}$ of a finite domain constraint φ is obtained by switching the operator ($\{<, \geq\}$, $\{>, \leq\}$) between the two expressions, e.g., $\overline{X > Y} \equiv X \leq Y$.

- *(falsify constraint) if* $\mathcal{C}^+ = \{\overline{\varphi}\} \cup \mathcal{C}_i$ *is consistent, then* $\mathcal{G}_{i+1} = \mathcal{G}_i^-$ *and* $\mathcal{C}_{i+1} = \mathcal{C}^+$ `OR`
- *(satisfy constraint) if* $\mathcal{C}^+ = \{\varphi\} \cup \mathcal{C}_i$ *is consistent, then* $\mathcal{G}_{i+1} = \{\forall \vec{X}. \leftarrow \Gamma^-\} \cup \mathcal{G}_i^-$ *and* $\mathcal{C}_{i+1} = \mathcal{C}^+$.

Inference Rule (LE2). *If* φ *is an equality of the form* $t = s$,

1. *if* $t = p(\vec{u})$ *and* $s = p(\vec{v})$*, then:* $\mathcal{G}_{i+1} = \{\forall \vec{X}. \leftarrow \vec{u} = \vec{v}, \Gamma^-\} \cup \mathcal{G}_i^-$;
2. *if* $t = p(\vec{u})$ *and* $s = q(\vec{v})$*, then:* $\mathcal{G}_{i+1} = \mathcal{G}_i^-$;
3. *if* $t \in vars(s)$ *or* $s \in vars(t)$*, then:* $\mathcal{G}_{i+1} = \mathcal{G}_i^-$;
4. *if* $t \in \vec{X}$ *(or* $s \in \vec{X}$*), let* $\theta = \{t/s\}$ *(or* $\theta = \{s/t\}$*) be a variable substitution, then:* $\mathcal{G}_{i+1} = \{\forall \vec{Y}. \leftarrow \Gamma^-/\theta\} \cup \mathcal{G}_i^-$*, where* $\vec{Y} = \vec{X} - \{t\}$ *(or* $\vec{Y} = \vec{X} - \{s\}$*);*
5. *if* t *is an existential variable of* ϕ *(i.e.,* $t \notin \vec{X}$*) and* s *does not contain universal variable of* ϕ *(i.e.,* $vars(s) \cap \vec{X} = \emptyset$*):*
   - *(falsify equality:) if* $\mathcal{E}^+ = \{t \neq s\} \cup \mathcal{E}_i$ *is consistent, then* $\mathcal{G}_{i+1} = \mathcal{G}_i^-$ *and* $\mathcal{E}_{i+1} = \mathcal{E}^+$ `OR`
   - *(satisfy equality:) if* $\mathcal{E}^+ = \{t = s\} \cup \mathcal{E}_i$ *is consistent, then* $\mathcal{G}_{i+1} = \{\forall \vec{X}. \leftarrow \Gamma^-\} \cup \mathcal{G}_i$ *and* $\mathcal{E}_{i+1} = \mathcal{E}^+$;
6. *symmetric to the above case (5) with* $s \notin \vec{X}$ *and* $vars(t) \cap \vec{X} = \emptyset$.

Note that the condition of **LC2** where φ does not contain any universal variable of ϕ is guaranteed by the safe goal selection strategy Ξ. In **LE2**, cases (1)-(4) implement the Clark Equality Theory (CET) [Cla78]. Cases (5)-(6) are similar to local **LC2**, i.e. treating the equality as an arithmetic constraint. Note that if one side of the equality is an existential variable, then the other side must not contain any universal variable if it is a non-variable term. This condition is also guaranteed by Ξ. Note also that inequality $t \neq s$ is not handled by **LE2**, as it is a shorthand for $\neg(t = s)$, which is handled by the local rule **LN2**.

Inference Rule (LN2). *If* $\varphi = \neg\psi$ *is a negative literal and* ψ *does not contain any universal variable of* ϕ *(i.e.,* $vars(\psi) \cap \vec{X} = \emptyset$*), then:*

- *(double negation)* $\mathcal{G}_{i+1} = \{\psi\} \cup \mathcal{G}_i^-$ `OR`
- *(single negation)* $\mathcal{G}_{i+1} = \{\leftarrow \psi, \forall \vec{X}. \leftarrow \Gamma^-\} \cup \mathcal{G}_i^-$.

There are two ways to satisfy a denial goal through a negative literal φ $(= \neg\psi)$:

1. prove its negation ψ as a positive goal, or
2. make sure the rest of the denial body is not satisfiable.

In the second case, we also want to make sure ψ is not provable by any agent in order to avoid duplicated solutions found in the former case. Again, Ξ guarantees the condition where ϕ does not contain any universal variable.

Example. *Agent α applies* **LN2**

$$\begin{array}{|l|}\hline \Theta_i : \textit{given } \Sigma = \{\alpha, \beta\} \\ \hline \mathcal{G}_i = \{\text{“}\forall Z. \leftarrow \neg p(X,Y), q(Y,Z)''\} \\ \mathcal{G}_i^d = \emptyset \\ \Delta_i = \emptyset \\ \mathcal{N}_i = \emptyset \\ \mathcal{E}_i = \emptyset \\ \mathcal{C}_i = \emptyset \\ \hline \end{array}$$

$$\Downarrow \qquad\qquad\qquad \Downarrow$$

$$\begin{array}{|l|}\hline \Theta_{i+1} : \\ \hline \mathcal{G}_{i+1} = \{p(X,Y)\} \\ \mathcal{G}_{i+1}^d = \mathcal{G}_i^d \\ \Delta_{i+1} = \Delta_i \\ \mathcal{N}_{i+1} = \mathcal{N}_i \\ \mathcal{E}_{i+1} = \mathcal{E}_i \\ \mathcal{C}_{i+1} = \mathcal{C}_i \\ \hline \end{array} \qquad \begin{array}{|l|}\hline \Theta'_{i+1} : \\ \hline \mathcal{G}'_{i+1} = \left\{ \begin{array}{l} X \neq Z, \\ \leftarrow p(X,Y), \\ \forall Z. \leftarrow q(Y,Z) \end{array} \right\} \\ {\mathcal{G}_{i+1}^d}' = \mathcal{G}_i^d \\ \Delta'_{i+1} = \Delta_i \\ \mathcal{N}'_{i+1} = \mathcal{N}_i \\ \mathcal{E}'_{i+1} = \mathcal{E}_i \\ \mathcal{C}'_{i+1} = \mathcal{C}_i \\ \hline \end{array}$$

In this example, Θ_1 is obtained by case one and Θ_2 is obtained by case two. ■ End of example.

4.3.4 Coordination

Agent Interaction and State Transfer

A *state transfer* is the operation of a transferable state being passed from one agent to another. After a transferable state is received, it needs to be pre-processed by the recipient before it can be used by the recipient as the root of a new local abduction. This pre-processing is captured by the following rule.

Transfer Rule (TR). *Let $\Theta_t = \langle(\mathcal{G}, \mathcal{G}^d), (\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C}), \tau\rangle$ be the state received by an agent α after a state transfer, then $\Theta_0 = \langle(\mathcal{F} \cup \mathcal{G}^d \cup \mathcal{G}, \emptyset), (\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C}), \tau'\rangle$ is the root state for the new local abduction by α , where F and τ' are obtained as follows. Let Δ^{new} be the set of α-tagged abducibles, i.e., $\{A \mid A \in \Delta \texttt{ and } (A, \mathcal{S}) \in \tau \texttt{ and } \alpha \in \mathcal{S}\}$, and let $\mathcal{N}^{new}$ be the set of α-tagged non-abducible constraints, i.e, $\{D \mid D \in \mathcal{N} \texttt{ and } (D, \mathcal{S}) \in \tau \texttt{ and } \alpha \in \mathcal{S}\}$:*

1. *given an abducible $A = a(\vec{u})$, the set of resolvents of A with $\mathcal{IC}_\alpha$ is $R_A(\mathcal{IC}_\alpha) = \{\forall\vec{Y}. \leftarrow \vec{u}{=}\vec{v}, \Phi^- \mid \leftarrow \Phi \in \mathcal{IC}_\alpha \texttt{ and } \Phi = \{a(\vec{v})\} \cup \Phi^- \texttt{ and } \vec{Y} = vars(\Phi)\}$;*

2. *given a non-abducible constraint $D = \forall\vec{X}. \leftarrow p(\vec{u}), \Phi$ where $p(\vec{u})$ is its constrained non-abducible, the set of resolvents of the non-abducible constraint with Π_α is $R_D(\Pi_\alpha) = \{\forall\vec{X}\vec{Y}. \leftarrow \vec{u} = \vec{v}, \Phi', \Phi \mid p(\vec{v}) \leftarrow \Phi' \in \Pi_{ag} \texttt{ and } \vec{Y} = vars(p(\vec{v})) \cup vars(\Phi')\}$;*

3. *F is the union of all the resolvents of the α-tagged abducibles and α-tagged non-abducible constraints, i.e., $\bigcup_{A\in\Delta^{new}} R_A(\mathcal{IC}_\alpha) \cup \bigcup_{D\in\mathcal{N}^{new}} R_D(\Pi_\alpha)$;*

4. *τ' is obtained by removing α from the tags τ, i.e., $\{(L, \mathcal{S}) \mid (L, \mathcal{S}) \in \tau \texttt{ and } L \notin (\Delta^{new} \cup \mathcal{N}^{new})\} \cup \{(L, \mathcal{S}') \mid (L, \mathcal{S}) \in \tau \texttt{ and } L \in (\Delta^{new} \cup \mathcal{N}^{new}) \texttt{ and } \mathcal{S}' = \mathcal{S} \setminus \{\alpha\}\}$.*

The application of this rule forces the transferable state recipient agent to perform *pending checks* of the state, i.e., check new assumptions (abducibles collected by others) against its local integrity constraints (i.e., Step 1 and 3) and continue the reduction of global constraints through a non-abducible (selected by another agent) using its local background knowledge (i.e., Step 2 and 3). It also moves the goals delayed by the sender to the pending goals set (i.e., by the definition of Θ_0), so that they can be selected and reduced by the recipient agent. The tags by the recipient agent to the abducibles or non-abducible constraints are removed after the processing (i.e., Step 4), and hence they do not need to be checked again in the future. After the processing of the state, the recipient agent can start a new local abduction with the new state independently from the sender.

Example. *Recipient Agent α applies* **TR**

$\Theta_i : given\ \Sigma = \{\alpha, \beta\}, \Pi_\alpha = \{p(U,V) \leftarrow r(U), w(V)\}, \mathcal{IC}_\alpha = \{\leftarrow a(Z), \neg q(Z)\}$
$\mathcal{G}_i = \emptyset$
$\mathcal{G}_i^d = \{q(X)^{\{\beta\}}\}$
$\Delta_i = \{a(X)^{\{\alpha\}}\}$
$\mathcal{N}_i = \{\forall Y. \leftarrow p(X,Y)^{\{\alpha\}}\}$
$\mathcal{E}_i = \emptyset$
$\mathcal{C}_i = \emptyset$

$$\Downarrow$$

Θ_{i+1} :
$\mathcal{G}_{i+1} = \left\{ \begin{array}{l} \forall Z. \leftarrow X = Z, \neg q(X), \\ \forall YUV. \leftarrow X = U, Y = V, r(X), w(Y), \\ q(X)^{\{\beta\}} \end{array} \right\}$
$\mathcal{G}_{i+1}^d = \emptyset$
$\Delta_{i+1} = \{a(X)^{\emptyset}\}$
$\mathcal{N}_{i+1} = \{\forall Y. \leftarrow p(X,Y)^{\emptyset}\}$
$\mathcal{E}_{i+1} = \mathcal{E}_i$
$\mathcal{C}_{i+1} = \mathcal{C}_i$

In this example, $\Delta^{new} = \Delta_i$ and $\mathcal{N}^{new} = \mathcal{N}_i$ for α. Therefore, $\forall Z. \leftarrow X = Z, \neg q(X)$ is *generated by checking $a(X)$ with $\mathcal{IC}_\alpha$ and $\forall YUV. \leftarrow X = U, Y = V, r(X), w(Y)$ is generated by the reduction of $\forall Y. \leftarrow p(X,Y)$ through $p(X,Y)$ using Π_α. Note that the tags for the elements in Δ_i and $\mathcal{N}_i$ are removed.* ■ *End of example.*

Agent Interaction Strategy

During local abduction, an agent can process a transferable state either by selecting a remaining goal to reduce locally, or by passing it to another agent. The decision is controlled by the *agent interaction strategy* that is adopted by the agent. Agent interaction strategies can be arbitrary, but have two extremes: a *lazy* strategy, which requires the agent to solve as many pending goals as possible before sending it out, and an *eager* strategy, which encourages the agent to send out state as soon as a goal is delayed, or a new abducible or a non-abducible constraint is collected. None of these strategies can guarantee optimal performance in all applications. For example, suppose in the local abduction by an agent α, there is a state with two goals p, q, where p is only defined by an agent β and q is only defined by α. In the case where p requires long computation and can succeed, but q requires little computation and will fail, then the *lazy* strategy will give better performance. This is because by delaying p and trying to solve q first, the state will become a failure state and no agent interaction is needed (i.e., no need for the computation of p). However, in the case where p requires little computation and will fail, but q requires long computation and can succeed, the *eager* strategy may do better as the agent interaction can avoid the computation of q.

In the following description of our distributed algorithm, unless stated otherwise, we will assume the lazy interaction strategy is used.

Agent Selection Strategy

After an agent decides to send out a transferable state, it needs to identify and select a state recipient. A recipient candidate for a transferable state can be an agent who defines but has not delayed a delayed goal in the state, or who tags an abducible or a non-abducible constraint in the state. A transferable state may have several recipient candidates, and the sender agent must select one of them to send the state to. Such a decision is controlled by the *agent selection strategy* of the sender agent, which is often application dependent and will affect the performance of the collaboration. For example, if no agent discloses what non-abducibles are defined in its background knowledge, then the selection of a state recipient may be either randomised or through some task allocation protocols such as *Auction* or *Contract-Nets*. Otherwise, if the agents *advertise* all of the non-abducibles which they have definitions for in their background knowledge, then the state recipient may be decided via a *Matchmaking* process over the delayed goals. Furthermore, for a tagged abducible or a tagged non-abducible constraint, only those agents who have an integrity constraint containing the abducible or have a definition for the non-abducible in the local background knowledge can generate additional goals (See Rule **TR**). This information may be used while deciding a suitable state recipient in order to reduce the depth of agent interactions.

In the following description of our distributed algorithm, we will assume by default that no agent advertisement is available, and a *uniform agent selection strategy* can be adopted. This is defined as follows.

Definition 4.9 (Uniform Agent Selection Strategy). *Given a global abductive framework $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$, suppose that the agent identifiers Σ can be sorted lexicographically. Let $\Theta = \left\langle (\mathcal{G}, \mathcal{G}^d), \mathcal{ST}, \tau \right\rangle$ be a transferable state. The uniform agent selection strategy selects an agent $\beta \in \Sigma$, such that*

- <u>request help for delayed goals:</u> *if $\mathcal{G}^d \neq \emptyset$, and there is no delayed goal $L \in \mathcal{G}^d$ such that $\langle L, \Sigma \rangle \in \tau$ (i.e., L has been delayed by every agent once), then let $\Omega = \bigcup_{[\phi \in (\mathcal{G} \cup \mathcal{G}^d) \text{ and } \langle \phi, \mathcal{S} \rangle \in \tau]} (\Sigma \setminus \mathcal{S})$, and β is the first agent in the list obtained by lexicographically sorting Ω;*
- <u>request consistency check for abducibles and non-abducible constraints:</u> *if $\mathcal{G}^d = \emptyset$, and $\Omega = \bigcup_{\langle \phi, \mathcal{S} \rangle \in \tau} \mathcal{S}$ is not empty, then β is the first agent in the list obtained by lexicographically sorting Ω.*

Given a transferable state, if a (non-abducible) goal has been tagged (delayed) by all the agents, there will be no recipient candidate, and hence the transferable state can be discarded. This is intended for avoiding the situation of *indefinitely asking for help*. Note that discarding such states will not cause the loss of solutions during agent collaboration, because by the definition of **LD1**, any agent that has tagged the non-abducible also has tried tried to reduce the non-abducible using their background knowledge (and generated corresponding states).

Global Abduction and Agent Execution

Intuitively, the state transfers represent the interactions between the agents, and act as the links between local abductions performed by different agents. Thus, the overall global abductive inference can be visualised as a *global abductive derivation tree*, which consists of all the local abductions by the agents (See again Figure 4.1 in Section 4.3.1):

Definition 4.10 (Global Abductive Derivation Tree). *Given a global abductive framework* $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$ *with a safe goal selection strategy* Ξ*, an agent interaction strategy* Υ *and an agent selection strategy* Ψ*, the global abductive derivation tree for a query* $\mathcal{Q}$ *is a tree where each node represents a state and each arc represents either a local inference step or a state transfer, such that:*

- *the root node is the initial state* Θ_{init} *of* $\mathcal{Q}$*;*
- *each node is obtained by the application of a local inference rule or by the application of the transfer rule.*

Definition 4.11. (Successful Global Abductive Derivation) *Given a global abductive derivation tree, a successful global abductive derivation is a finite path from the root of the tree to a solved state leaf node of the tree.*

Since each agent computation is in fact a search process where the search space is the local abductive derivation tree, different search algorithms can be used for the agent execution, such as depth-first and breath-first. Figure 4.2 is the pseudo-code for an example implementation which processes the states in a depth-first fashion. Given a query, agents executing the code can collaboratively construct a global abductive derivation tree.

```
PROC receive_query($\mathcal{Q}$) BEGIN
    $\Theta_0$ := create_initial_state($\mathcal{Q}$);
    process_state($\Theta_0$); // start local abduction
END PROC
PROC receive_state($\Theta$) BEGIN
    $\Theta_0$ := apply_trans_rule($TR$, $\Theta$);
    process_state($\Theta_0$); // start local abduction
END PROC
PROC process_state($\Theta$) BEGIN
    IF $\Theta$ is a solved state THEN
        $ANS$ := extract_answers($\Theta$);
        send $ANS$ to the global query $issuer$[4];
    ELSE
        // use agent interaction strategy
        IF $\Theta$ is transferable state AND $\Theta$ should be sent THEN
            // use agent selection strategy
            $NewAgent$ := select_recipient($\Theta$);
            IF $NewAgent \neq null$ THEN
                send $\Theta$ to $NewAgent$;
            END IF
        ELSE // local inference
            // use safe goal selection strategy
            $G$ := select_safe_goal($\Theta$);
            $LRule$ := applicable_local_rule($G$);
            $States$ := apply_local_rule($LRule$, $\Theta$);
            FOREACH $\Theta'$ IN $States$ DO
                process_state($\Theta'$);
            END FOREACH
        END IF
    END IF
END PROC
```

Figure 4.2: Pseudo-code of DAREC Agent Execution

Token-controlled Coordination Protocol

So far we have seen how the agents can interact and collaborate to search for answers for a given query. However, there are still two open issues:

- **(Completeness Concern)** In many applications the query issuer may want a reply when no answer can be found. Thus, the query issuer also needs to know when the distributed computation has finished, i.e., the global abductive derivation tree has been searched thoroughly.

- **(Performance Concern)** During a local abduction several transferable states may be derived and need to be sent out according to the adopted agent interaction strategy. If all of them are sent out straightaway, then the communication channels between agents may be quickly overloaded by the "message flood". Furthermore, since each of these states will initiate a new local abduction on the recipient agent, agents may be overloaded if they receive too many states in a short period and have to manage too many local abductions simultaneously.

To resolve these issues while allowing agents to perform concurrent computation, we propose a *token-controlled* coordination protocol for global abduction. Within this protocol, there is a *token* shared among all the agents. At any time, the token is associated with only one local abduction (of some agent). An agent is allowed to send out a transferable state only if the state belongs to one of its local abductions, and that local abduction is associated with the token. The detailed steps of the protocol are summarised as follows. We assume that each agent records three pieces of information for each local abduction it is managing – a *unique ID* for the local abduction, the *sender* of the root state, and a *foreign local abduction ID* understandable by the sender. We also assume that messages always arrive in order, i.e., if message A is sent to agent α before message B, then α will receive A and then B.

1. To start a global abduction, the query issuer sends a query along with a state to an agent in the system. This agent creates an initial state and starts a local abduction associated with the token.

2. During a local abduction of an agent's, if the agent derives a transferable state that needs to be sent out, then

   (a) if the local abduction has the token, then the agent sends out the transferable state with the token and the local abduction's ID, i.e., this local abduction will no longer be associated with the token;

   (b) otherwise, it buffers the transferable state;

   and in both cases the agent can continue remaining local abduction(s).

3. Upon receiving a state with the token and a foreign local abduction ID, the recipient agent initiates a new local abduction with the token, and *records* the foreign local abduction ID and the state sender.

4. If an agent derives a solved state, it sends the extracted answer to the query issuer regardless if it has the token or not, and continues local abduction(s), i.e., to search for more solutions.

5. If an agent finishes a local abduction (i.e., no more states to process), then

   (a) if this local abduction is associated with the token (note that this implies that there is no buffered transferable state for this local abduction), then the agent returns the token with an *End Of Search* (EOS) message, which contains the recorded foreign local abduction ID, to the sender agent of the (pre-processed) root state of this local abduction;

   (b) otherwise, the agent waits for the token;

   and in both cases the agent can continue remaining local abduction(s).

6. Upon receiving an EOS message with the token, the agent looks up the local abduction whose ID can be found in the EOS message, and

   (a) if there are buffered transferable states for the local abduction, then one of them is sent out with the token (similarly to 2(a));

   (b) else if there is no buffered state and the local abduction has finished (i.e., continued from 5(b)), then the agent returns an EOS message with the token to the root state sender of this local abduction (similarly to 5(a));

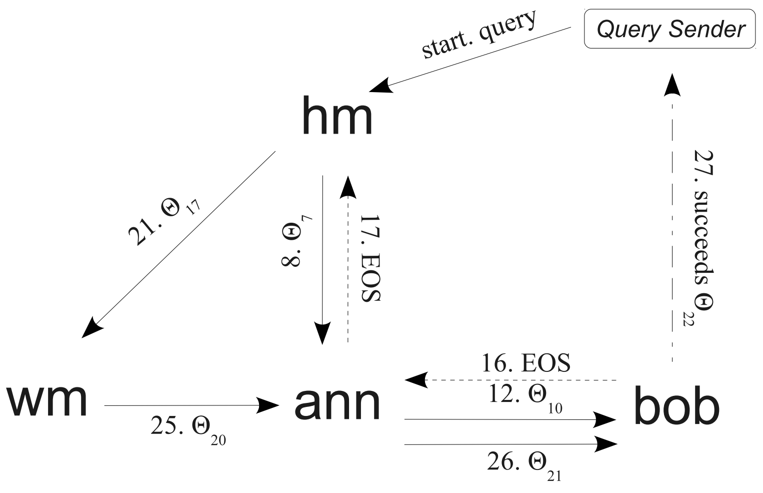
   (c) otherwise, the agent simply associates the token back to the local abduction;

   and in all cases the agent can continue remaining local abduction(s).

With this protocol and the assumption that messages arrive in order, the query issuer will know the whole global abduction has finished once it receives the EOS message from the agent it has issued the query to. In addition, if no answer is received before the EOS message, the query can be sure that there is no global abductive answer for the query.

4.3.5 Example Trace

Let's look at a (simplified) possible global abductive derivation for Example 4.1. Let $\mathcal{Q} = \{movement(cor, 12)\}$ be a global query received by **hm**, and let us assume the adoptions of a left-to-right safe goal selection strategy, the lazy agent interaction strategy and the uniform agent selection strategy. The following diagram outlines the communications between the agents:



The states generated during the global abductive derivation are described below:

1. **hm** creates the initial state: $[\Theta_0 = \langle(\{movement(cor, 12)\}, \emptyset), \mathcal{ST}^{\emptyset}\rangle]$
2. $[\Theta_1 = \langle(\{occupant(X), walkInCorridor(X, 12)\}, \emptyset), \mathcal{ST}^{\emptyset}\rangle, \Theta_2 = \langle(\{pointOfEntry(12),$
   $walkInCorridor(intruder, 12)\}, \emptyset), \mathcal{ST}^{\emptyset}\rangle]$
3. $[\Theta_3 = \langle(\{X \in \{ann, bob\}, walkInCorridor(X, 12)\}, \emptyset), \mathcal{ST}^{\emptyset}\rangle, \Theta_2]$
4. $[\Theta_4 = \langle(\{walkInCorridor(X, 12)\}, \emptyset), (\emptyset, \emptyset, \emptyset, \{X \in \{ann, bob\}\})\rangle, \Theta_2]$
5. $[\Theta_5 = \langle(\{\text{``} \leftarrow X \neq intruder, out(X, 12)\text{''}\}, \emptyset), (\{walkInCorridor(X, 12)^{\{ann,bob,wm\}}\},$
   $\emptyset, \emptyset, \mathcal{C}_4)\rangle, \Theta_2]$
6. $[\Theta_6 = \langle(\{\text{``} \leftarrow out(X, 12)\text{''}\}, \emptyset), (\Delta_5, \emptyset, \{X \neq intruder\}, \mathcal{C}_4)\rangle, \Theta_2]$
7. $[\Theta_7 = \langle(\emptyset, \emptyset), (\Delta_5, \{\text{``} \leftarrow out(X, 12)\text{''}^{\{ann,bob,wm\}}\}, \mathcal{E}_6, \mathcal{C}_4)\rangle, \Theta_2]$

8. **hm** sends Θ_7 to **ann**:

$[\Theta_8 = \langle(\{\text{“}\forall T1, T2. \leftarrow X = ann, appointment(T1, T2), T1 \leq 12, 12 \leq T2\text{”}\}, \emptyset),$

$(\{walkInCorridor(X, 12)^{\{bob,wm\}}\}, \{\text{“} \leftarrow out(X, 12)\text{”}^{\{bob,wm\}}\}, \mathcal{E}_6, \mathcal{C}_4)\rangle]$

9. $[\Theta_9 = \langle(\{\text{“}\forall T1, T2. \leftarrow appointment(T1, T2), T1 \leq 12, 12 \leq T2\text{”}\}, \emptyset),$

$(\Delta_8, \mathcal{N}_8, \mathcal{E}_6 \cup \{X = ann\}, \mathcal{C}_4)\rangle, \Theta_{10} = \langle(\emptyset, \emptyset), (\Delta_8, \mathcal{N}_8, \mathcal{E}_6 \cup \{X \neq ann\}, \mathcal{C}_4)\rangle]$

10. $[\Theta_{11} = \langle(\{\text{“} \leftarrow 11 \leq 12, 12 \leq 13\text{”}\}, \emptyset), (\Delta_8,$

$\mathcal{N}_8 \cup \{\text{“}\forall T1, T2. \leftarrow appointment(T1, T2), T1 \leq 12, 12 \leq T2)\text{”}^{\{hm,bob,wm\}}\}, \mathcal{E}_9, \mathcal{C}_4)\rangle, \Theta_{10}]$

11. **ann** eventually discards Θ_{11} as the failure goal cannot succeed: $[\Theta_{10}]$

12. **ann** passes Θ_{10} to **bob**:

$[\Theta_{12} = \langle(\{\text{“}\forall T1. \leftarrow X = bob, takenMedicine(T1), 10 \leq T1, T1 \leq 12\text{”}\}, \emptyset),$

$(\{walkInCorridor(X, 12)^{\{wm\}}\}, \{\text{“} \leftarrow out(X, 12)\text{”}^{\{wm\}}\}, \mathcal{E}_{10}, \mathcal{C}_4)\rangle]$

13. $[\Theta_{13} = \langle(\{\text{“}\forall T1. \leftarrow takenMedicine(T1), 10 \leq T1, T1 \leq 12\text{”}\}, \emptyset),$

$(\Delta_{12}, \mathcal{N}_{12}, \mathcal{E}_{10} \cup \{X = bob\}, \mathcal{C}_4)\rangle]$

14. $[\Theta_{14} = \langle(\{\text{“} \leftarrow 10 \leq 11, 11 \leq 12)\text{”}\}, \emptyset),$

$(\Delta_{12}, \mathcal{N}_{12} \cup \{\text{“}\forall T1. \leftarrow takenMedicine(T1), 10 \leq T1, T1 \leq 12\text{”}^{\{hm,ann,wm\}}\}, \mathcal{E}_{13}, \mathcal{C}_4)\rangle]$

15. **bob** eventually discards Θ_{14} as the failure goal cannot succeed.

16. **bob** has no remaining state so B sends an EOS message to **ann**.

17. **ann** has no more remaining state either so **ann** sends an EOS message to **hm**, who has one remaining state: $[\Theta_2]$

18. $[\Theta_{15} = \langle(\{walkInCorridor(intruder, 12)\}, \{pointOfEntry(12)^{\{hm\}}\}), \mathcal{ST}^{\emptyset}\rangle]$

19. $[\Theta_{16} = \langle(\{\text{“} \leftarrow intruder \neq intruder, out(intruder, 12)\text{”}\}, \mathcal{G}^d_{15}),$

$(\{walkInCorridor(intruder, 12)^{\{ann,bob,wm\}}\}, \emptyset, \emptyset, \emptyset)\rangle]$

20. $[\Theta_{17} = \langle(\emptyset, \mathcal{G}^d_{15}), (\Delta_{16}, \emptyset, \emptyset, \emptyset)\rangle]$

21. **hm** passes Θ_{17} to **wm**:

$[\Theta_{18} = \langle(\{pointOfEntry(12)^{\{hm\}}\}, \emptyset), (\{walkInCorridor(intruder, 12)^{\{ann,bob\}}\}, \emptyset, \emptyset, \emptyset)\rangle]$

22. $[\Theta_{18} = \langle(\{open(W), floor(W, 1)\}, \emptyset), (\Delta_{18}, \emptyset, \emptyset, \emptyset)\rangle]$

23. $[\Theta_{19} = \langle(\{floor(w1, 1)\}, \emptyset), (\Delta_{18}, \emptyset, \emptyset, \emptyset)\rangle]$

24. $[\Theta_{20} = \langle(\emptyset, \emptyset), (\Delta_{18}, \emptyset, \emptyset, \emptyset)\rangle]$

25. **wm** passes Θ_{20} to **ann** (to check Δ_{18}):

    $[\Theta_{21}{=}\big\langle(\emptyset,\emptyset),(\{walkInCorridor(intruder,12)^{\{bob\}}\},\emptyset,\emptyset,\emptyset)\big\rangle]$

26. **ann** succeeds the checking without expanding Δ_{21} or $\mathcal{N}_{21}$;

27. **ann** passes Θ_{21} to **bob**:

    $[\Theta_{22} = \langle(\{\text{“}\forall T1.\leftarrow X = intruder, takenMedicine(T1), 10 \le T1, T1 \le 12\text{”}\},\emptyset),$

    $(\{walkInCorridor(intruder,12)^{\emptyset}\},\emptyset,\emptyset,\emptyset)\big\rangle]$

28. **bob** succeeds the checking without expanding Δ_{22} or $\mathcal{N}_{22}$;

29. **bob** extracts answer from Θ_{22} and returns it to the query sender.

4.4 Soundness and Completeness

In this section, we give the soundness and completeness theorems of the DARE*C* distributed proof procedure, and present the proofs. First, we define the *meaning* of a DARE*C* computational state, which is a closed first-order formula representing all the elements and the tagging information of the state. Let D be a collected integrity constraint in a state. We use $ab\_con(D)$ (or $nab\_con(D)$) to denote the fact that D is an abducible constraint (or a non-abducible constraint). Let α be an agent, we use $R^{ab}_{\phi}(\mathcal{IC}_\alpha)$ to denote the set of resolvents of an abducible $\phi = a(\vec{u})$ with respect to the set of integrity constraints $\mathcal{IC}_\alpha$, i.e., $R^{ab}_{\phi}(\mathcal{IC}_\alpha) = \{\forall\vec{Y}.\leftarrow \vec{u}{=}\vec{v},\Phi^- \mid\leftarrow \Phi \in \mathcal{IC}_\alpha \texttt{ and } \Phi = \{a(\vec{v})\}\cup\Phi^- \texttt{ and } \vec{Y} = vars(\Phi)\}$, and we use $R^{nab}_{D}(\Pi_\alpha)$ to denote the set of resolvents of a non-abducible constraint $D = \forall\vec{X}.\leftarrow p(\vec{u}),\Phi$ (where $p(\vec{u})$ is its constrained non-abducible) with respect to the set of rules Π_α, i.e., $R^{nab}_{D}(\Pi_\alpha) = \{\forall\vec{X}\vec{Y}.\leftarrow \vec{u}=\vec{v},\Phi,\Gamma \mid p(\vec{v})\leftarrow\Phi'\in\Pi_\alpha \texttt{ and } \vec{Y} = vars(p(\vec{v}))\cup vars(\Phi)\}$.

Definition 4.12 (Meaning of a DARE*C* Computational State). *The meaning $\mathcal{M}(\Theta)$ of a DAREC computational state $\Theta = \big\langle(\mathcal{G},\mathcal{G}^d),(\Delta,\mathcal{N},\mathcal{E},\mathcal{C}),\tau\big\rangle$ with respect to a global abductive framework $\big\langle\Sigma,\widehat{\mathcal{F}}\big\rangle$ is a formula such that:*

$$\mathcal{M}(\Theta) = \exists.(F_{exp}\wedge F_{imp})$$

where

$$F_{exp} = \bigwedge_{L \in (\mathcal{G} \cup \mathcal{G}^d \cup \Delta \cup \mathcal{E} \cup \mathcal{C})} L \wedge \bigwedge_{D \in \mathcal{N} \wedge ab\_con(D)} D$$

and

$$F_{imp} = \bigwedge_{[F \in \bigcup_{(A \in \Delta \wedge (A,\mathcal{S}) \in \mathcal{T} \wedge \alpha \in \mathcal{S})} R^{ab}_A(\mathcal{IC}_\alpha)]} F \wedge \bigwedge_{[F \in \bigcup_{(D \in \mathcal{N} \wedge nab\_con(D) \wedge (D,\mathcal{S}) \in \mathcal{T} \wedge \alpha \in \mathcal{S})} R_{naf}(D,\Pi_\alpha)]} F$$

The meaning of a DAREC state contains two parts – the *explicit part* F_{exp}, representing the conjunction of the elements in the state, and the *implicit part* F_{imp}, represent the *meaning* of the tags for collected abducibles and non-abducible constraints. Thus, if none of the abducibles and non-abducible constraints is tagged, F_{imp} is equivalent to $\top$. Note that the tags of any delay goal in the state do not have any formal meaning for the state, i.e., it is only used for execution control (of the distributed algorithm). Note also that non-abducible constraints that are not tagged in the state do not have any formal meaning for the state either.

Example. *Let* $\Sigma = \{\alpha, \beta\}$ *with*

$$\Pi_\beta = \left\{ \; w(X,Y) \leftarrow d(X), e(Y). \; \right\}$$

and

$$\mathcal{IC}_\beta = \left\{ \; \leftarrow a(X), f(X). \; \right\}$$

where a is the only abducible predicate. Given a DAREC state in a local abduction by α

$$\Theta = \left\langle (\{q(X)\}, \{p(X)^\beta\}), (\{a(X)^\beta\}, \{\forall Y. \leftarrow a(Y), r(Y), \forall Y. \leftarrow w(X,Y)^\beta\}, \emptyset, \{X > 4\}) \right\rangle$$

then

$$\begin{aligned} \mathcal{M}(\Theta) = {} & \exists X.q(X) \wedge p(X) \wedge X > 4 \wedge \\ & \quad \forall Z. \leftarrow Z = X, f(X) \wedge \forall U,V,Y. \leftarrow U = X, V = Y, d(X), e(Y) \end{aligned}$$

Note that $\forall Z. \leftarrow Z = X, f(X)$ *is from the* β*-tagged abducible and is syntactically equivalent to*

$\forall Z.\neg(Z = X \wedge f(X))$, and $\forall U, V, Y. \leftarrow U = X, V = Y, d(X), e(Y)$ is from the β-tagged non-abducible constraint and is syntactically equivalent to $\forall U, V, Y.\neg(U = X \wedge V = Y \wedge d(X) \wedge e(Y))$. Note also that the tag on $p(X)$ by β does not result in any sub-formula of $\mathcal{M}(\Theta)$.

The definition of the meaning of an ASystem state is exactly the same as the F_{exp} in the meaning of a DAREC state, since there are no tags in ASystem states. Consequently, the following proposition holds.

Proposition 4.1. *Every DAREC state without tags for abducibles or non-abducible constraints has corresponding ASystem state with equivalent meanings.*

For example, let $\Theta_D = \langle(\{q(X)\}, \{p(X)^{\beta}\}), (\{a(X)\}, \{\forall Y. \leftarrow a(Y), r(Y), \forall Y. \leftarrow w(X,Y)\}, \emptyset, \{X > 4\})\rangle$ be a DAREC state, then its corresponding and equivalent ASystem state is $\Theta_A = \langle\{q(X), p(X)\}, (\{a(X)\}, \{\forall Y. \leftarrow a(Y), r(Y), \forall Y. \leftarrow w(X,Y)\}, \emptyset, \{X > 4\})\rangle$

4.4.1 Soundness

We will first give the *meaning* of a computed DAREC answer, and then give the soundness theorem and proof.

Definition 4.13. (Completion of Hypotheses) *Given a global abductive framework $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$, let $\langle \Delta, \theta \rangle$ be an answer computed by the DAREC algorithm for a query $\mathcal{Q}$ such that $\Delta\theta$ is ground, then the completion of the hypotheses $\Delta\theta$ is given by a formula δ, which is the conjunction of the literals $\{A \mid A \in \widehat{\mathcal{AB}} \wedge A \in \Delta\theta\} \cup \{\neg A \mid A \in \widehat{\mathcal{AB}} \wedge A \notin \Delta\theta\}$, where $\widehat{\mathcal{AB}} = \bigcup_{i \in \Sigma} \mathcal{AB}_i$.*

Theorem 4.1. (DAREC Soundness) *Given a global abductive framework $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$ and a query $\mathcal{Q}$, if there is a successful global abductive derivation for $\mathcal{Q}$ with global abductive answer $\langle \Delta, \theta \rangle$, then:*

1. *$comp(\widehat{\Pi}) \cup \{\delta\} \models_3 \mathcal{Q}\theta$;*
2. *$comp(\widehat{\Pi}) \cup \{\delta\} \models_3 I$ for every $I \in \widehat{\mathcal{IC}}$.*

where δ is the completion of $\Delta\theta$, $\widehat{\Pi} = \bigcup_{i \in \Sigma} \Pi_i$, and $\widehat{\mathcal{IC}} = \bigcup_{i \in \Sigma} \mathcal{IC}_i$.

Proof Outline

To prove the soundness, we need to show that the *global background knowledge* (i.e., the union of all the agent background knowledge) together with the completion of the hypotheses entail the query and the *global integrity constraints* (i.e., the union of all agent's integrity constraints). The proof for the first property is similar to the ASystem soundness proof [vN04], and uses the meaning of the states. Informally, given a successful global abductive derivation, we first show that (the meaning of) the initial state entails the query, and then show that every state is entailed by its successor state. The final state, which contains the hypotheses and the variable substitutions, will then entail the query by a *chain of implications*. For the second property, we need to show that each integrity constraint that contains an assumed abducible is checked at least once during the derivation.

The first part of the proof uses the following two lemmas.

Lemma 4.1. *Let Θ_0 be the initial state for a query $\mathcal{Q}$, then*

$$\models_3 \mathcal{M}(\Theta_0) \rightarrow \mathcal{Q}$$

Proof of Lemma 4.1: This is trivial. The initial state contains $\mathcal{Q}$ as the only remaining goal(s); all other stores are empty and nothing is tagged. Thus, $\mathcal{M}(\Theta_0) \equiv \mathcal{Q}$ and $\models_3 \mathcal{M}(\Theta_0) \rightarrow \mathcal{Q}$.

■ *End of Proof for Lemma 4.1 .*

Lemma 4.2. *Given a successful global abductive derivation, let Θ_{i+1} be the successor state of Θ_i, then*

$$comp(\widehat{\Pi}) \models_3 \mathcal{M}(\Theta_{i+1}) \rightarrow \mathcal{M}(\Theta_i)$$

Proof of Lemma 4.2: In a global abductive derivation, each state is obtained by the application of a local inference rule (i.e., LD1, LA1, LN1, LE1, LC1, LD2, LA2, LN2, LE2 and LC2), or the transfer rule (i.e., TR), on its predecessor state. We need to show that after each of these rules, the meaning of the new state implies the meaning of the old state under the completion of $\widehat{\Pi}$. Note that Lemma 4.2 has been proven [vN04] for ASystem derivations with ASystem states and the ASystem inference rules (i.e., D1, A1, N1, E1, C1, D2, A2, N2, E2 and C2).

The DAREC local rules LN1, LE1, LC1, LA2, LN2, LE2 and LC2 are a reformulation of the ASystem inference rules N1, E1, C1, A2, N2, E2 and C2, respectively, and they do not modify the tags in any give state, i.e., let $\Theta_i = F_{exp_i} \wedge F_{imp_i}$ and $\Theta_{i+1} = F_{exp_{i+1}} \wedge F_{imp_{i+1}}$, then $F_{imp_i} \equiv F_{imp_{i+1}}$. We first assume $F_{imp_i} = F_{imp_{i+1}} = \top$ (i.e., no abducible or non-abducible constraint is tagged in Θ_i and Θ_{i+1}), then Θ_i and Θ_{i+1} can be seen as ASystem states (by Proposition 4.1), and Θ_{i+1} is the successor of Θ_i after one of the seven ASystem inference rules. Thus, in this case Lemma 4.2 holds. In the case of $F_{imp_i} \neq \top$, Lemma 4.2 still holds according to the tautology $(A \rightarrow B) \rightarrow (A \wedge C \rightarrow B \wedge C)$, where $A = F_{exp_{i+1}}$, $B = F_{exp_i}$ and $C = F_{imp_i} = F_{imp_{i+1}}$.

Now we only need to show that Lemma 4.2 holds for LD1, LA1, LD2 and TR:

Case of LD1 : Let ϕ be a non-abducible goal selected from Θ_i for the application of **LD1**. Θ_{i+1} is obtained either by moving ϕ to the set of delayed goals or by replacing it with its resolvent of a rule in $\widehat{\Pi}$. If ϕ is delayed, then $\mathcal{M}(\Theta_{i+1}) \equiv \mathcal{M}(\Theta_i)$ (because the new tag added to ϕ has no meaning in $\mathcal{M}(\Theta_{i+1})$), and hence $\models_3 \mathcal{M}(\Theta_{i+1}) \rightarrow \mathcal{M}(\Theta_i)$. If ϕ is not delayed, then **LD1** behaves exactly the same as the ASystem inference rule D1, and hence $comp(\widehat{\Pi}) \models_3 \mathcal{M}(\Theta_{i+1}) \rightarrow \mathcal{M}(\Theta_i)$, i.e., Lemma 4.2 holds for LD1.

Case of LA1 : Let $\phi = a(\vec{t})$ be an abducible goal selected from Θ_i for the application of LA1. Θ_{i+1} is obtained either by unifying ϕ with an existing abducible, say $\varphi = a(\vec{u})$ in Δ_i (i.e., reuse assumption), or by adding ϕ to Δ_i (i.e., create new assumption) and adding new goals to $\mathcal{G}_i$. For the former case, the set of equalities $\vec{t} = \vec{u}$ generated with the unification between ϕ and φ is added to $\mathcal{E}_i$. Therefore, $\models_3 \mathcal{M}(\Theta_{i+1}) \rightarrow \mathcal{M}(\Theta_i)$ by the *Clark's Equality Theory* (CET). For the latter case, since LA1 does not remove any existing tag in the state, without loss of generality let us assume $\mathcal{M}(\Theta_i)$ does not have tags for abducibles or non-abducible constraints. Let Θ_i^A be the corresponding ASystem state of Θ_i and Θ^A_{i+1} the new ASystem state obtained by applying the ASystem inference rule A1 (for the case of collecting new abducible) on ϕ with respect to $\widehat{\Pi}$ and $\widehat{\mathcal{IC}}$, then we have

$$\mathcal{M}(\Theta^A_{i+1}) = \mathcal{M}(\Theta^A_i) \wedge F_{new}$$

where

$$F_{new} = F_{\Delta_i} \wedge F_{\mathcal{N}_i} \wedge F_{\widehat{\mathcal{IC}}}$$

and

$$F_{\Delta_i} = \bigwedge_{F \in \{\vec{u} \neq \vec{v} | a(\vec{v}) \in \Delta_i\}} F$$

and

$$F_{\mathcal{N}_i} = \bigwedge_{F \in \{\forall \vec{X}. \leftarrow \vec{u} = \vec{w}, \Phi | \forall \vec{X}. \leftarrow a(\vec{w}), \Phi \in \mathcal{N}_i\}} F$$

and

$$F_{\widehat{\mathcal{IC}}} = \bigwedge_{F \in R^{ab}_{\phi}(\widehat{\mathcal{IC}})} F$$

According to the definition of LA1 (let α be the agent applying LA1), we have

$$\mathcal{M}(\Theta_{i+1}) = \mathcal{M}(\Theta_i) \wedge F'_{new}$$

where

$$F'_{new} = F_{\Delta_i} \wedge F_{\mathcal{N}_i} \wedge F_{\mathcal{IC}_\alpha} \wedge F'_{imp}$$

and

$$F_{\mathcal{IC}_\alpha} = \bigwedge_{F \in R^{ab}_{\phi}(\mathcal{IC}_\alpha)} F$$

and (from the new tags added for ϕ after it is moved to Δ_{i+1})

$$F'_{imp} = \bigwedge_{F \in R^{ab}_{\phi}(\widehat{\mathcal{IC}} \setminus \mathcal{IC}_\alpha)} F$$

It is easy to see that $F'_{new} \equiv F_{new}$ follows from $F_{\mathcal{IC}_\alpha} \wedge F'_{imp} \equiv F_{\widehat{\mathcal{IC}}}$, and thus with the assumption $\mathcal{M}(\Theta^A_i) \equiv \mathcal{M}(\Theta_i)$, we have $\mathcal{M}(\Theta^A_{i+1}) \equiv \mathcal{M}(\Theta_{i+1})$. Since Lemma 4.2 holds for A1 (i.e., $\models_3 \mathcal{M}(\Theta^A_{i+1}) \rightarrow \mathcal{M}(\Theta^A_i)$), we have $\models_3 \mathcal{M}(\Theta_{i+1}) \rightarrow \mathcal{M}(\Theta_i)$, i.e., Lemma 4.2 also holds for LA1.

Case of LD2 : The proof is very similar to that for the case of LA1. Let $\phi = \forall \vec{X}. \leftarrow p(\vec{t}), \Gamma$ where $p(\vec{t})$ is the selected non-abducible literal to reduce. Since LD2 does not remove any existing tag in the state either, without loss of generality let us assume $\mathcal{M}(\Theta_i)$ does not have tags for abducibles or non-abducible constraints, and let Θ^A_i be the corresponding ASystem state of Θ_i. According to the definition of the ASystem inference rule D2 (with respect to $\widehat{\Pi}$ and $\widehat{\mathcal{IC}}$), the meaning of the next ASystem state Θ^A_{i+1} is obtained by replacing ϕ in $\mathcal{M}(\Theta^A_i)$ with the conjunction of resolvents of $p(\vec{u})$ with $\widehat{\Pi}$, i.e.,

$$F_{new} = \bigwedge_{F \in R^{nab}_{\phi}(\widehat{\Pi})} F$$

According to the definition of LD2, $\mathcal{M}(\Theta i + 1)$ is obtained by replacing ϕ in $\mathcal{M}(\Theta_i)$ with the following formula (let α be the agent applying LD2):

$$F'_{new} = F_{\Pi_\alpha} \wedge F'_{imp}$$

where

$$F_{\Pi_\alpha} = \bigwedge_{F \in R^{nab}_{\phi}(\Pi_\alpha)} F$$

and (from the new tags added to ϕ after it is moved to $\mathcal{N}_{i+1}$)

$$F'_{imp} = \bigwedge_{F \in R^{nab}_{\phi}(\widehat{\Pi} \setminus \Pi_\alpha)} F$$

It is easy to see that $F'_{new} \equiv F_{new}$, and hence $\mathcal{M}(\Theta^A_{i+1}) \equiv \mathcal{M}(\Theta_{i+1})$. Since Lemma 4.2 holds for D2, we have $\models_3 \mathcal{M}(\Theta^A_{i+1}) \rightarrow \mathcal{M}(\Theta^A)$, and hence $\models_3 \mathcal{M}(\Theta_{i+1}) \rightarrow \mathcal{M}(\Theta_i)$, i.e., Lemma 4.2 also holds for LD2.

Case of TR : this is trivial as the application of TR (by an agent α) effectively moves part of F_{imp} to F_{exp} in $\mathcal{M}(\Theta_i)$, by resolving the collected abducibles and non-abducible constraints tagged by α with $\mathcal{IC}_\alpha$ and Π_α, respectively, and removing all of α's tags (for the abducibles and the non-abducible constraints). Therefore, $\mathcal{M}(\Theta_{i+1}) \equiv \mathcal{M}(\Theta_i)$ and $\models_3 \mathcal{M}(\Theta_{i+1}) \rightarrow \mathcal{M}(\Theta_i)$, i.e., Lemma 4.2 holds for TR.

■ *End of Proof for Lemma 4.2 .*

Proof of Theorem 4.1: Let $\Theta_0, \ldots, \Theta_n$ be a successful global abductive derivation for $\mathcal{Q}$ where Θ_0 is the initial state and Θ_n is the final solved state.

For Property (1) of Theorem 4.1, by the definition (Definition 4.13) of the completion δ of hypotheses $\Delta\theta$ where θ is induced by $\mathcal{M}(\Theta_n)$, we have

$$\models_3 \delta \rightarrow \mathcal{M}(\Theta_n)$$

By Lemma 4.2 (i.e, $comp(\widehat{\Pi}) \models_3 \mathcal{M}(\Theta_{i+1}) \rightarrow \mathcal{M}(\Theta_i)$) and the *transitivity of implications*, we have

$$comp(\widehat{\Pi}) \models_3 \mathcal{M}(\Theta_n) \rightarrow \mathcal{M}(\Theta_0)$$

Hence, we have

$$comp(\widehat{\Pi}) \models_3 \delta \rightarrow \mathcal{M}(\Theta_0)$$

Using the transitivity of implications again with Lemma 4.1 (i.e., $\models_3 \mathcal{M}(\Theta_0) \rightarrow \mathcal{Q}$), we have

$$comp(\widehat{\Pi}) \models_3 \delta \rightarrow \mathcal{Q}\theta$$

which is equivalent to

$$comp(\widehat{\Pi}) \cup \{\delta\} \models_3 \mathcal{Q}\theta$$

For Property (2) of Theorem 4.1, without loss of generality let $I = \leftarrow a(\vec{t}), \Gamma$ be a ground instance of an integrity constraint of agent α, where $a(\vec{t})$ is an abducible [5]. If $a(\vec{t}) \notin \Delta\theta$, then the property trivially holds for I. Now assuming that $a(\vec{t}) \in \Delta\theta$, let us consider two cases: 1. if $a(\vec{t})$

[5]By definition, each integrity constraint must have at least one positive abducible.

was collected by α, then by the definition of LA1, there is a goal $F = \leftarrow \Gamma$ added by α to an intermediate state. 2. if $a(\vec{t})$ was collected by a different agent β, then it must be tagged by α after β applied LA1. Since in the final state Θ_n it is no longer tagged by α, α must have applied TR to remove the tag and add the goal F to an intermediate state. Let Θ_i $(0 < i < n)$ be such an intermediate state where F was added (in either case). Then $comp(\widehat{\Pi}) \models_3 \mathcal{M}(\Theta_i) \rightarrow F$. By Lemma 4.2 and the transitivity of implications, we have $comp(\widehat{\Pi}) \models_3 \mathcal{M}(\Theta_n) \rightarrow F$, and consequently $comp(\widehat{\Pi}) \models_3 \delta \rightarrow F$. By the assumption we also have $\models_3 \delta \rightarrow a(\vec{t})$. Hence,

$$comp(\widehat{\Pi}) \models_3 \delta \rightarrow (a(\vec{t}) \wedge F)$$

By the tautology $A \wedge \neg B \rightarrow \neg(A \wedge B)$ where A is $a(\vec{t})$ and B is F , we have

$$comp(\widehat{\Pi}) \models_3 \delta \rightarrow I$$

and finally

$$comp(\widehat{\Pi}) \cup \{\delta\} \models_3 I$$

■ *End of Proof for Theorem 4.1* .

4.4.2 Completeness

Given an abductive framework and a query, an abductive inference algorithm is said to be *complete* with respect to a logic program semantics if and only if it can compute a set of answers that is logically equivalent to the set of all the possible answers for the query with the framework under the chosen semantics. Similar to the ASystem algorithm, the DAREC (distributed abductive) algorithm uses the Fitting three-valued semantics [Fit85]. This is because both the ASystem and DAREC algorithms are based on top-down computation, and may be non-terminating during query computation due to the presence of loop (e.g., $p \leftarrow p$) in the given abductive framework. Thus, the completeness of the ASystem algorithm and the DAREC algorithm is based on a *termination condition* as summarised in Section 2.3.4. For the reader's convenience, we give it here:

Theorem 4.2. *[vN04] Given an abductive framework $\mathcal{F} = \langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$, if Π is semi-acyclic [AB91] with respect to a level mapping $|.|$ and Π is abductive non-recursive [Ver99], then for all bounded queries [AB91] $\mathcal{Q}$ with respect to $|.|$, the ASystem proof procedure is terminating with respect to $\mathcal{Q}$.*

Theorem 4.3. (DAREC Completeness) *Let $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$ be a global abductive framework. If there is a finite global abductive derivation tree T for the query $\mathcal{Q}$, and $comp(\widehat{\Pi}) \cup \widehat{\mathcal{IC}} \cup \exists \mathcal{Q}$ is satisfiable under the three-valued semantics [Fit85], then T contains a successful branch.*

Proof Outline

For the DAREC completeness proof, the idea is to show that any DAREC (global abductive) derivation tree can be *reduced* to an equivalent ASystem derivation tree, and then use the ASystem's completeness theorem. The reduction has two steps. First, we show that any DAREC derivation tree obtained with a fixed execution strategy (i.e., fixed agent interaction strategy and goal selection strategy) can be reduced to an equivalent ASystem derivation tree. Secondly, we show that any DAREC derivation tree obtained without the fixed execution strategy can be transformed to one that is obtained with the strategy.

The fixed execution strategy is defined as follows.

Definition 4.14 (Pseudo-ASystem Execution Strategy). *A pseudo-ASystem execution strategy for the DAREC distributed proof procedure is one that given a DAREC state:*

1. *if there is a tagged non-abducible goal in the set of remaining goals, then the goal will be selected and LD1 will be applied (i.e., local inference);*

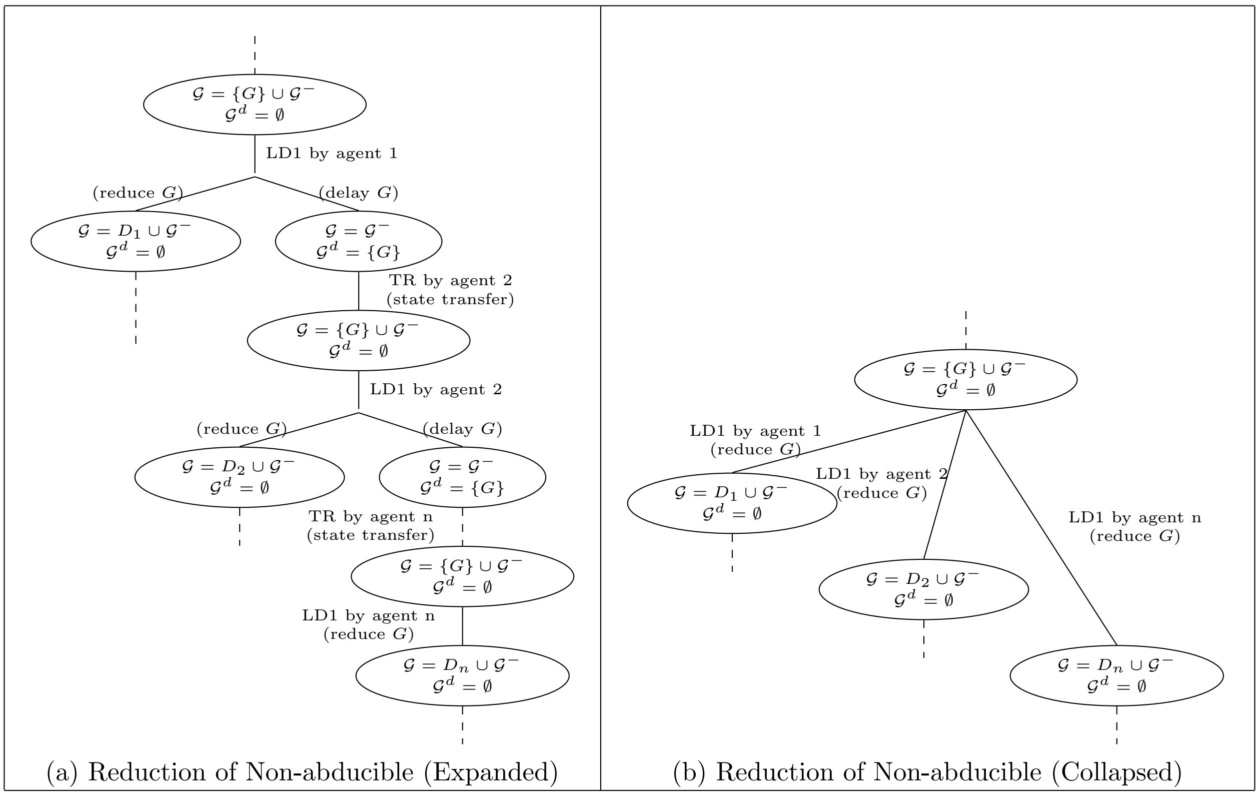
2. *if there is any (non-abducible) goal in the set of delayed goals, or if there is any tagged abducible or tagged non-abducible constraint, then this (transferable) state will be passed to another agent (i.e., state transfer).*

The first property of the pseudo-ASystem execution strategy describes a special type of goal selection strategy, such that every agent resumes the reduction of a non-abducible goal that has been delayed (by others) as soon as it can. The second property describes an *eager* agent interaction strategy, such that every agent sends out transferable states instead of performing local inference on them, so that other agents can resume the reduction of a delayed goal, or resume the check for a tagged abducible or a tagged non-abducible constraint in the state as soon as they can. Thus, the main effect of the pseudo-ASystem execution strategy is to let the agents eliminate any tag in a state as soon as possible during the collaboration.

Given a DAREC derivation tree T^D_{fix} obtained with the pseudo-ASystem execution strategy, we have the following observations:

- After the first agent applies LD1 on a non-abducible goal, a sequence of state transfers follows such that each of the remaining agents in the system will in turn (a) receive a state containing a tagged non-abducible goal, (b) select and apply LD1 to the goal (i.e., the goal will be delayed and tagged by the agent thereby creating a new transferable state), and (c) passes the new transferable state to the next agent that has not tagged the goal. The part of T^D_{fix} reflecting these derivation steps is shown in Figure 4.3a, and can be *collapsed* by removing the nodes and edges involving goal delays and state transfers (Figure 4.3b). Note that all the remaining states in the collapsed part of T^D_{fix} do not contain any tagged goal.

- After a new abducible or a new non-abducible constraint is collected by some agent, a sequence of state transfers follows such that each of the remaining agents in the system will in turn (a) receive a state containing a tagged abducible or a tagged non-abducible constraint, (b) apply TR to resolve the abducible with the local integrity constraints, or resolve the non-abducible constraint with the local background knowledge, and (c) if the abducible or the non-abducible constraint is still tagged then pass the new transferable state to the next agent that tags it. The part of T^D_{fix} reflecting these derivation steps is shown in Figure 4.4a and Figure 4.4b, and can be *collapsed* by removing the nodes and edges involving state transfers (Figure 4.4c and Figure 4.4d). Note that all the



$\mathcal{G} = \{G\} \cup \mathcal{G}^-$
$\mathcal{G}^d = \emptyset$
LD1 by agent 1
(reduce G)
(delay G)
$\mathcal{G} = D_1 \cup \mathcal{G}^-$
$\mathcal{G}^d = \emptyset$
$\mathcal{G} = \mathcal{G}^-$
$\mathcal{G}^d = \{G\}$
TR by agent 2
(state transfer)
LD1 by agent 2
$\mathcal{G} = D_2 \cup \mathcal{G}^-$
TR by agent n
(state transfer)
LD1 by agent n
$\mathcal{G} = D_n \cup \mathcal{G}^-$
(a) Reduction of Non-abducible (Expanded)
(b) Reduction of Non-abducible (Collapsed)

Figure 4.3: DAREC Derivation Tree with Pseudo-ASystem Execution Strategy (Reduction of Non-abducible Goal)

remaining states in the collapsed part of T^D_{fix} do not contain any tag to an abducible or a non-abducible constraint.

Thus, if we apply the above two types of reductions to T^D_{fix}, then we will obtain a new tree *collapse*(T^D_{fix}), whose nodes (i.e., DAREC states) do not contain any tag. By Proposition 4.1, all the nodes in *collapse*(T^D_{fix}) can be replaced by their corresponding ASystem states. In addition, all the edges (i.e., the applications of the DAREC local inference rules) in *collapse*(T^D_{fix}) can be replaced by the applications of the ASystem inference rules. Note that the edge of $LA1$ ($LD2$) connecting any two states (the *parent* and the *child*) can be replaced by that of $A1$ ($D2$) because the child state is equivalent to the one obtained by applying $A1$ ($D2$) to the parent state with respect to $\widehat{\Pi}$ and $\widehat{\mathcal{IC}}$. It is easy to see that the resulting tree is exactly an ASystem derivation tree.

Proposition 4.2. *Every DAREC global abductive derivation tree obtained with the pseudo-ASystem execution strategy can be reduced to an equivalent ASystem derivation tree.*

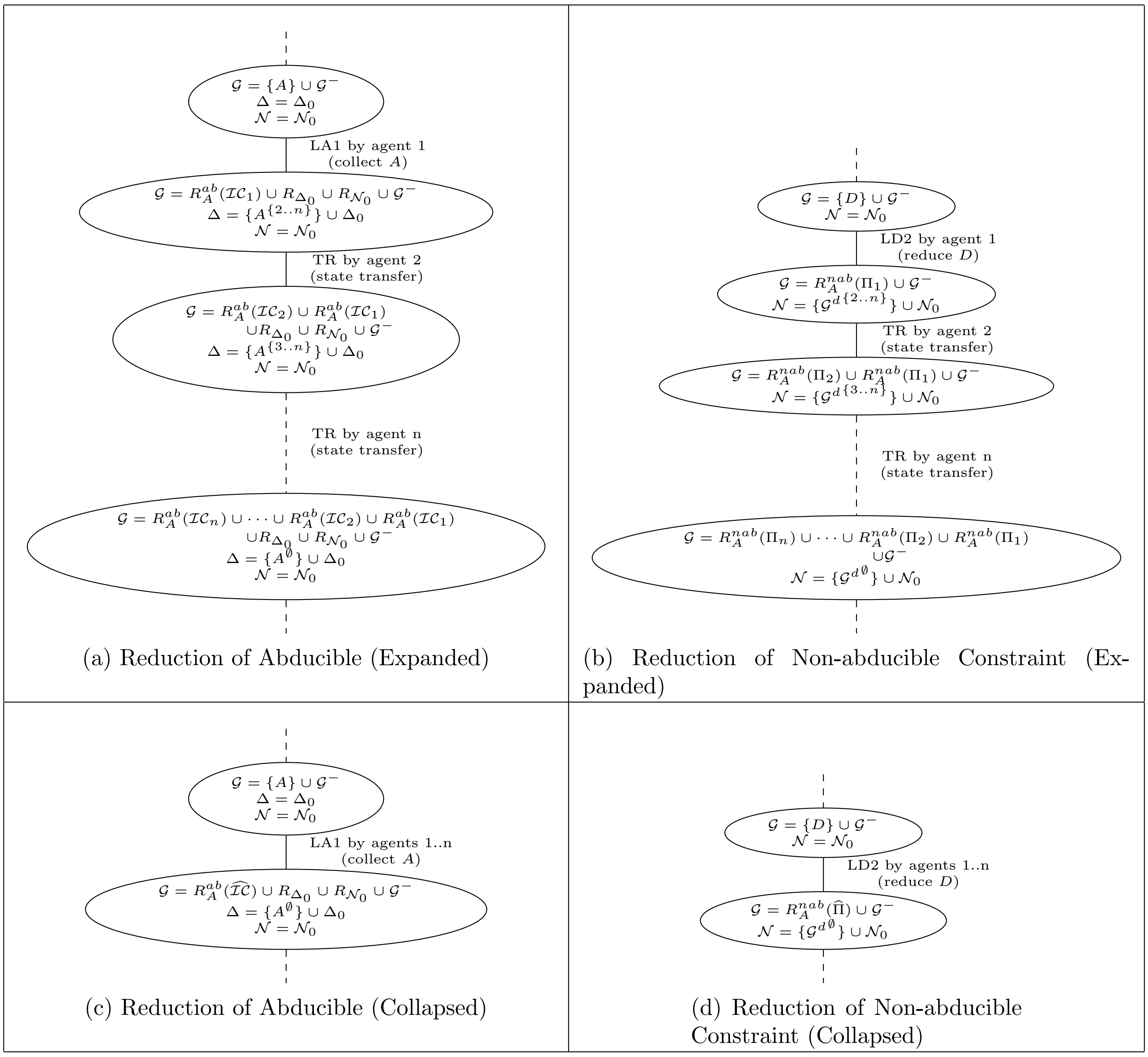

$\mathcal{G} = \{A\} \cup \mathcal{G}^-$
$\Delta = \Delta_0$
$\mathcal{N} = \mathcal{N}_0$
LA1 by agent 1
(collect A)
$\mathcal{G} = R_A^{ab}(\mathcal{IC}_1) \cup R_{\Delta_0} \cup R_{\mathcal{N}_0} \cup \mathcal{G}^-$
$\Delta = \{A^{\{2..n\}}\} \cup \Delta_0$
$\mathcal{N} = \mathcal{N}_0$
TR by agent 2
(state transfer)
$\mathcal{G} = R_A^{ab}(\mathcal{IC}_2) \cup R_A^{ab}(\mathcal{IC}_1)$
$\cup R_{\Delta_0} \cup R_{\mathcal{N}_0} \cup \mathcal{G}^-$
$\Delta = \{A^{\{3..n\}}\} \cup \Delta_0$
$\mathcal{N} = \mathcal{N}_0$
TR by agent n
(state transfer)
$\mathcal{G} = R_A^{ab}(\mathcal{IC}_n) \cup \cdots \cup R_A^{ab}(\mathcal{IC}_2) \cup R_A^{ab}(\mathcal{IC}_1)$
$\cup R_{\Delta_0} \cup R_{\mathcal{N}_0} \cup \mathcal{G}^-$
$\Delta = \{A^{\emptyset}\} \cup \Delta_0$
$\mathcal{N} = \mathcal{N}_0$
(a) Reduction of Abducible (Expanded)
$\mathcal{G} = \{D\} \cup \mathcal{G}^-$
$\mathcal{N} = \mathcal{N}_0$
LD2 by agent 1
(reduce D)
$\mathcal{G} = R_A^{nab}(\Pi_1) \cup \mathcal{G}^-$
$\mathcal{N} = \{\mathcal{G}^{d\{2..n\}}\} \cup \mathcal{N}_0$
TR by agent 2
(state transfer)
$\mathcal{G} = R_A^{nab}(\Pi_2) \cup R_A^{nab}(\Pi_1) \cup \mathcal{G}^-$
$\mathcal{N} = \{\mathcal{G}^{d\{3..n\}}\} \cup \mathcal{N}_0$
TR by agent n
(state transfer)
$\mathcal{G} = R_A^{nab}(\Pi_n) \cup \cdots \cup R_A^{nab}(\Pi_2) \cup R_A^{nab}(\Pi_1)$
$\cup \mathcal{G}^-$
$\mathcal{N} = \{\mathcal{G}^{d^{\emptyset}}\} \cup \mathcal{N}_0$
(b) Reduction of Non-abducible Constraint (Expanded)
$\mathcal{G} = \{A\} \cup \mathcal{G}^-$
$\Delta = \Delta_0$
$\mathcal{N} = \mathcal{N}_0$
LA1 by agents 1..n
(collect A)
$\mathcal{G} = R_A^{ab}(\widehat{\mathcal{IC}}) \cup R_{\Delta_0} \cup R_{\mathcal{N}_0} \cup \mathcal{G}^-$
$\Delta = \{A^{\emptyset}\} \cup \Delta_0$
$\mathcal{N} = \mathcal{N}_0$
(c) Reduction of Abducible (Collapsed)
$\mathcal{G} = \{D\} \cup \mathcal{G}^-$
$\mathcal{N} = \mathcal{N}_0$
LD2 by agents 1..n
(reduce D)
$\mathcal{G} = R_A^{nab}(\widehat{\Pi}) \cup \mathcal{G}^-$
$\mathcal{N} = \{\mathcal{G}^{d^{\emptyset}}\} \cup \mathcal{N}_0$
(d) Reduction of Non-abducible Constraint (Collapsed)

Figure 4.4: DARE*C* Derivation Tree with Pseudo-ASystem Execution Strategy (Reduction of Abducible or Non-abducible Constraint)

Now let us look at an arbitrary DAREC derivation tree T^D_{arb} obtained without the pseudo-ASystem execution strategy. Again, only the applications of LD1, LA1, LD2 and TR can add or remove tags to or from the states, so we only need to consider their derivation steps in T^D_{arb}. Let $\Theta_i \rightarrow^{Rule}_{G} \Theta_{i+1}$ denote a derivation step where G (optional) is the selected goal and $Rule$ (optional) is the applied inference rule, and let $\Theta_i \rightsquigarrow^{Rules} \Theta_n$ denote a sequence of multiple derivation steps and $Rules$ (optional) denote a list of applied inference rules:

1. **(Reduction of a non-abducible constraint)**: suppose in T^D_{arb} there is a segment of a derivation $\Theta_1 \rightarrow^{LD2}_{D} \Theta_2 \rightsquigarrow \Theta_3 \rightarrow^{TR} \Theta_4 \rightsquigarrow \Theta_5 \rightarrow^{TR} \Theta_6$ where D is a non-abducible constraint first collected in Θ_2, and the meanings of the states are as shown in Figure 4.5a. Since any application of TR does not change the meaning of a state, i.e., it merely moves part of the implicit formulas (derived from the tags for D) to the explicit part of the state meaning, we can "move forward" the applications of TR along the derivation, as shown in Figure 4.5b. Thus, the original segment of derivation in T^D_{arb} is replaced by $\Theta_1 \rightarrow^{LD2}_{D} \Theta_2 \rightsquigarrow^{sequence\ of\ TR} \Theta'_3 \rightsquigarrow \Theta'_4 \rightsquigarrow \Theta'_5$, where $\mathcal{M}(\Theta_2) = \mathcal{M}(\Theta'_3)$, $\mathcal{M}(\Theta_3) = \mathcal{M}(\Theta_4) = \mathcal{M}(\Theta'_4)$, and $\mathcal{M}(\Theta_5) = \mathcal{M}(\Theta_6) = \mathcal{M}(\Theta'_5)$. Note that the new derivation conforms to one that is obtained with the pseudo-ASystem strategy, and hence may be collapsed.

2. **(Collection of a new abducible)**: a derivation of this operation involves one application of LA1 and multiple applications of TR for any new abducible to be assumed. The transformation of such derivation is exactly the same as for the case (1).

3. **(Reduction of a non-abducible)**: suppose in T^D_{arb} there is a sub-tree (shown in Figure 4.6a) which involves the reduction of a selected non-abducible goal G by two agents with two applications of LD1 and one application of TR:

   (a) G was delayed by agent 1 resulting in a transferable state Θ_3, but agent 1 continued to process Θ_3 to Θ_4 instead of sending Θ_3 out to another agent (as with the pseudo-ASystem execution strategy);

   (b) agent 2 later received the state Θ_4 with G in it, but agent 2 processed other goals in Θ_4 instead of G (as with the pseudo-ASystem execution strategy) resulting in Θ_6;

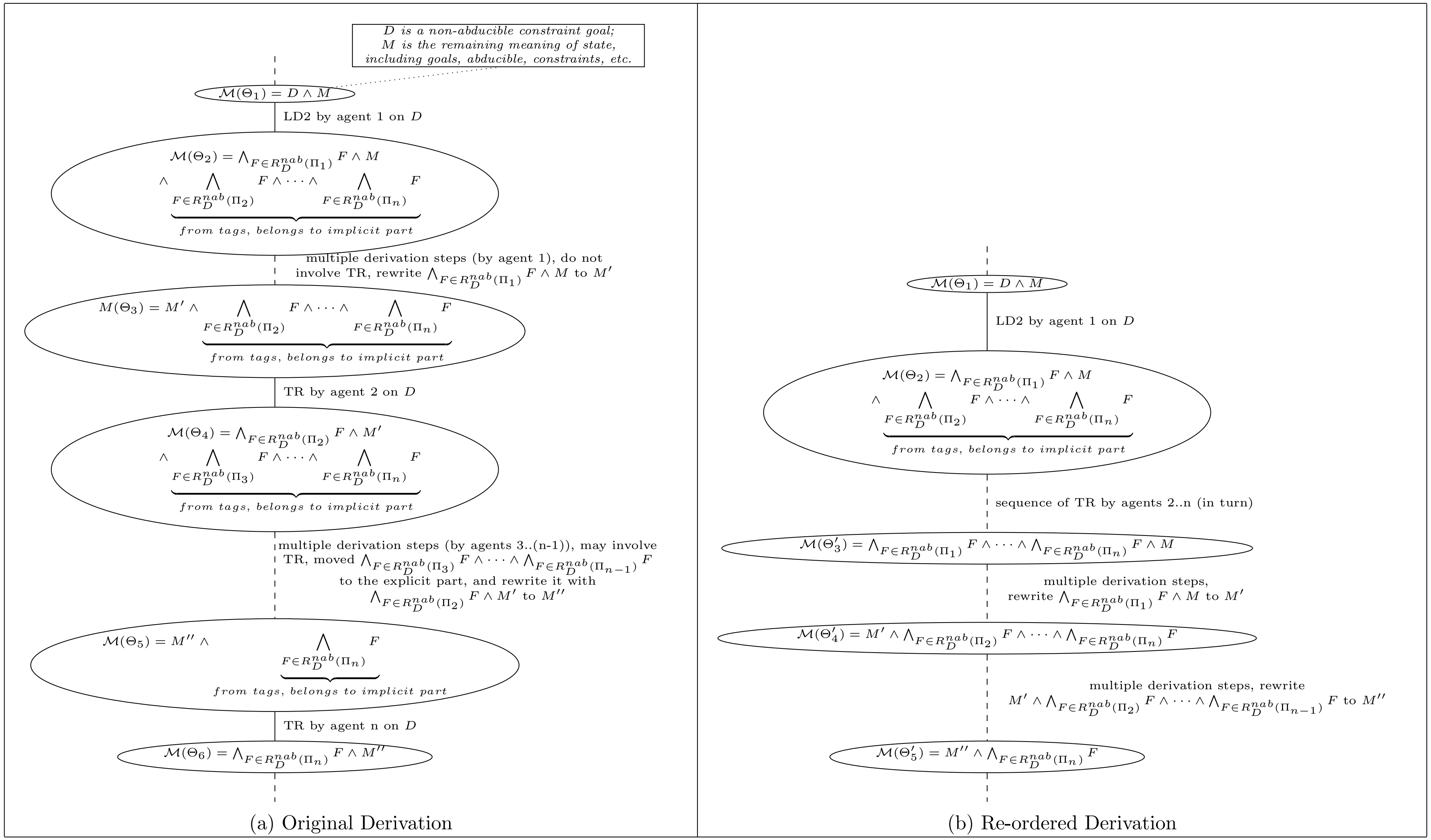
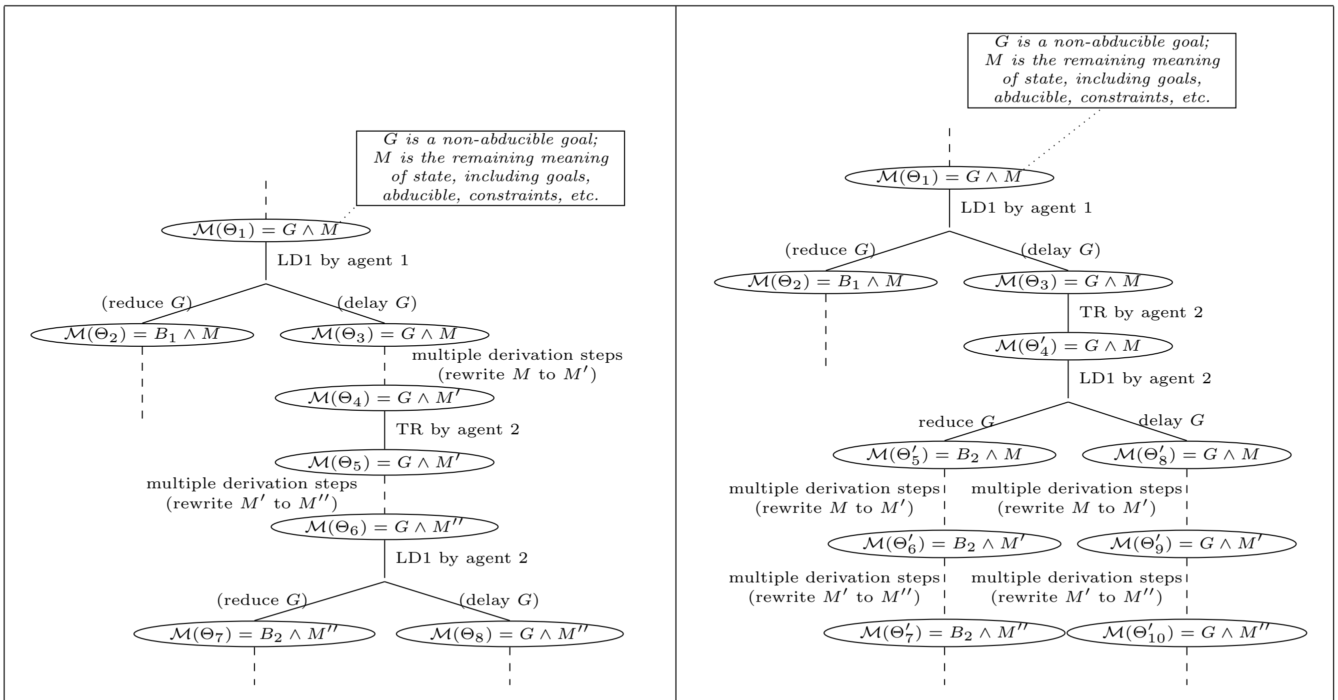

D is a non-abducible constraint goal; M is the remaining meaning of state, including goals, abducible, constraints, etc.
$\mathcal{M}(\Theta_1) = D \wedge M$
LD2 by agent 1 on D
$\mathcal{M}(\Theta_2) = \bigwedge_{F \in R_D^{nab}(\Pi_1)} F \wedge M \wedge \underbrace{\bigwedge_{F \in R_D^{nab}(\Pi_2)} F \wedge \cdots \wedge \bigwedge_{F \in R_D^{nab}(\Pi_n)} F}_{from\ tags,\ belongs\ to\ implicit\ part}$
multiple derivation steps (by agent 1), do not involve TR, rewrite $\bigwedge_{F \in R_D^{nab}(\Pi_1)} F \wedge M$ to M'
$M(\Theta_3) = M' \wedge \underbrace{\bigwedge_{F \in R_D^{nab}(\Pi_2)} F \wedge \cdots \wedge \bigwedge_{F \in R_D^{nab}(\Pi_n)} F}_{from\ tags,\ belongs\ to\ implicit\ part}$
TR by agent 2 on D
$\mathcal{M}(\Theta_4) = \bigwedge_{F \in R_D^{nab}(\Pi_2)} F \wedge M' \wedge \underbrace{\bigwedge_{F \in R_D^{nab}(\Pi_3)} F \wedge \cdots \wedge \bigwedge_{F \in R_D^{nab}(\Pi_n)} F}_{from\ tags,\ belongs\ to\ implicit\ part}$
multiple derivation steps (by agents 3..(n-1)), may involve TR, moved $\bigwedge_{F \in R_D^{nab}(\Pi_3)} F \wedge \cdots \wedge \bigwedge_{F \in R_D^{nab}(\Pi_{n-1})} F$ to the explicit part, and rewrite it with $\bigwedge_{F \in R_D^{nab}(\Pi_2)} F \wedge M'$ to M''
$\mathcal{M}(\Theta_5) = M'' \wedge \underbrace{\bigwedge_{F \in R_D^{nab}(\Pi_n)} F}_{from\ tags,\ belongs\ to\ implicit\ part}$
TR by agent n on D
$\mathcal{M}(\Theta_6) = \bigwedge_{F \in R_D^{nab}(\Pi_n)} F \wedge M''$
(a) Original Derivation
$\mathcal{M}(\Theta_1) = D \wedge M$
LD2 by agent 1 on D
$\mathcal{M}(\Theta_2) = \bigwedge_{F \in R_D^{nab}(\Pi_1)} F \wedge M \wedge \underbrace{\bigwedge_{F \in R_D^{nab}(\Pi_2)} F \wedge \cdots \wedge \bigwedge_{F \in R_D^{nab}(\Pi_n)} F}_{from\ tags,\ belongs\ to\ implicit\ part}$
sequence of TR by agents 2..n (in turn)
$\mathcal{M}(\Theta_3') = \bigwedge_{F \in R_D^{nab}(\Pi_1)} F \wedge \cdots \wedge \bigwedge_{F \in R_D^{nab}(\Pi_n)} F \wedge M$
multiple derivation steps, rewrite $\bigwedge_{F \in R_D^{nab}(\Pi_1)} F \wedge M$ to M'
$\mathcal{M}(\Theta_4') = M' \wedge \bigwedge_{F \in R_D^{nab}(\Pi_2)} F \wedge \cdots \wedge \bigwedge_{F \in R_D^{nab}(\Pi_n)} F$
multiple derivation steps, rewrite $M' \wedge \bigwedge_{F \in R_D^{nab}(\Pi_2)} F \wedge \cdots \wedge \bigwedge_{F \in R_D^{nab}(\Pi_{n-1})} F$ to M''
$\mathcal{M}(\Theta_5') = M'' \wedge \bigwedge_{F \in R_D^{nab}(\Pi_n)} F$
(b) Re-ordered Derivation

Figure 4.5: Transformation of Derivations involving LD2 and TR


G is a non-abducible goal; M is the remaining meaning of state, including goals, abducible, constraints, etc.
$\mathcal{M}(\Theta_1) = G \wedge M$
LD1 by agent 1
(reduce G)
(delay G)
$\mathcal{M}(\Theta_2) = B_1 \wedge M$
$\mathcal{M}(\Theta_3) = G \wedge M$
multiple derivation steps (rewrite M to M')
$\mathcal{M}(\Theta_4) = G \wedge M'$
TR by agent 2
$\mathcal{M}(\Theta_5) = G \wedge M'$
multiple derivation steps (rewrite M' to M'')
$\mathcal{M}(\Theta_6) = G \wedge M''$
LD1 by agent 2
(reduce G)
(delay G)
$\mathcal{M}(\Theta_7) = B_2 \wedge M''$
$\mathcal{M}(\Theta_8) = G \wedge M''$
G is a non-abducible goal; M is the remaining meaning of state, including goals, abducible, constraints, etc.
$\mathcal{M}(\Theta_1) = G \wedge M$
LD1 by agent 1
(reduce G)
(delay G)
$\mathcal{M}(\Theta_2) = B_1 \wedge M$
$\mathcal{M}(\Theta_3) = G \wedge M$
TR by agent 2
$\mathcal{M}(\Theta'_4) = G \wedge M$
LD1 by agent 2
reduce G
delay G
$\mathcal{M}(\Theta'_5) = B_2 \wedge M$
$\mathcal{M}(\Theta'_8) = G \wedge M$
multiple derivation steps (rewrite M to M')
multiple derivation steps (rewrite M to M')
$\mathcal{M}(\Theta'_6) = B_2 \wedge M'$
$\mathcal{M}(\Theta'_9) = G \wedge M'$
multiple derivation steps (rewrite M' to M'')
multiple derivation steps (rewrite M' to M'')
$\mathcal{M}(\Theta'_7) = B_2 \wedge M''$
$\mathcal{M}(\Theta'_{10}) = G \wedge M''$

(a) Original Derivation (b) Re-ordered Derivation

Figure 4.6: Transformation of Derivations involving LD1 and TR

(c) agent 2 then reduced G in Θ_6 by resolving it with a rule $G \leftarrow B_2 \in \Pi_2$ (i.e., resulting in Θ_7) and by delaying it (i.e., resulting in Θ_8).

Observe the fact that along the derivation $\Theta_3 \rightsquigarrow \Theta_4 \rightarrow \Theta_5 \rightsquigarrow \Theta_6$, the non-abducible goal G was not selected to reduce at all. Thus, this derivation could be transformed (see Figure 4.6b) in such a way that (i) Θ_3 was sent to agent 2, and G was immediately selected to reduce before M was rewritten (i.e., as with the pseudo-Asystem execution strategy), and then (ii) both the new goal D_2 (in $\Theta'_5, \ldots, \Theta'_7$) and the delayed goal G (in $\Theta'_8, \ldots, \Theta'_{10}$) were not selected to reduce until M was rewritten to M''. Finally (iii) if in the new sub-tree rooting from Θ'_8 there is another application of TR and LD1, the same transformation is applied recursively to this sub-tree. It is easy to see that the transformed derivation of the reduction of G is equivalent to the application of LD1 with the pseudo-ASystem strategy, and hence may be collapsed.

In summary, the following holds for every DAREC global abductive derivation tree.

Proposition 4.3. *Every DAREC global abductive derivation tree obtained with an arbitrary execution strategy can be transformed to an equivalent DAREC global abductive derivation tree* *obtained with the pseudo-ASystem execution strategy.*

Proof of Theorem 4.3: Let $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$ be a global abductive framework, and let T^D be a finite DAREC global abductive derivation tree for a given query $\mathcal{Q}$ obtained with an arbitrary execution strategy. By Proposition 4.3 there is a finite DAREC global abductive derivation tree T^D_{fix} transformed from and equivalent to T^D and is obtained with the pseudo-ASystem execution strategy. By Proposition 4.2, T^D_{fix} can be reduced to an equivalent ASystem derivation tree T^A. Thus, by the completeness theorem of ASystem [vN04], if $comp(\Pi) \cup \widehat{\mathcal{IC}} \cup \exists Q$ is satisfiable, then T^A must contain a successful branch, consequently T^D_{fix} and T^D must also contain a successful branch.

■ *End of Proof for Theorem 4.3 .*

4.5 Discussions

4.5.1 Usage of Agent Advertisements

In DARE, it is assumed that all agents advertise the non-abducible predicates for which they have definitions to a yellow-page directory. During collaboration, when an agent needs help to reduce a non-abducible, the agent can look up suitable helper agents from the directory and pass the query to the suitable agents. Although in DAREC, a state is passed instead and it is sent to only one other agent (because the recipient can also send it to others if it cannot reduce a delayed goal), the availability of a yellow-page directory can help to reduce communications between agents. For example, suppose that an agent α collects a non-abducible constraint "$\leftarrow p$", and the only other agent β in the system has not advertised p (i.e., β does not have any definition for p), then the non-abducible constraint does not need to be tagged by β, and α does not need to send the new state to β to check if there is no other tagged abducibles or non-abducible constraints in the new state. Similarly, if all the agents also advertise the abducible predicates regulated by their integrity constraints, then when a new abducible is collected, it does not need to be tagged by agents that do not have integrity constraints regulating it. Agent selection strategies can make use of such a yellow-page directory, whose availability is application dependent, in order to improve overall efficiency of global abduction.

4.5.2 Extension for Open Agent Systems

So far we have presented the DAREC system for *closed* MAS (i.e. a fixed set of agents). In fact, DAREC is very easy to extend to *open* MAS, where agents may join or leave during collaborative reasoning (such as in DARE). First, we *assume* that whenever an agent joins or leaves the system, all other agents will be notified. Secondly, we extend the state to be $\langle(\mathcal{G}, \mathcal{G}^d), \mathcal{ST}, \tau, S\rangle$, where S is a *set* of agent IDs (similar to the concept of *cluster* in DARE). Before a state is passed from an agent α to another β, α adds β's ID to S if S does not contain it. For example, let $A \rightsquigarrow_1 B \rightsquigarrow_2 C \rightsquigarrow_3 D \rightsquigarrow_4 B \rightsquigarrow_5 C \rightsquigarrow_6 A$ be a sequence of state transfers between four agents, then $S_0 = \{A$ at the beginning (of A's local abduction), $S_1 = \{A, B\}$ before $\rightsquigarrow_1$, $S_2 = \{A, B, C\}$ before $\rightsquigarrow_2$, and $S_3 = \{A, B, C, D\}$ before $\rightsquigarrow_3$ (and will remain as it is):

- When a new agent α joins, for every existing agent β, for any state Θ in its local abductions, β adds α's tag to all the abducibles in Δ and all the non-abducible constraints in $\mathcal{N}$ of Θ. Thus, Θ will become a transferable state if it is not one, and will need to be passed to α to check before it can be rewritten into a solved state.
- When an existing agent α leaves, the following two steps are needed:
  1. **(Clean Up Local Computations)** for every other remaining agent β:
     (a) β removes all of its local abductions that start with a root state whose S contains α, and if a token is associated with any of these local abductions, the token will be discarded;
     (b) β removes all of its buffered transferable states whose S contains α;
     (c) for every remaining state Θ in β's local abductions, β removes any tags by α in Θ.

2. **(Resume Global Computation)** If there is an existing agent γ who has sent out a transferable state Θ to α, and α was added to S of Θ by γ just before the state transfer, then γ regains the token automatically. Note that if such an agent γ exists, the global token must be lost in step 1(a) because the token must be associated with a descendent state of Θ. Note also that if γ does not exist, it implies that α has not yet been sent any state and token (i.e., it has not participated in the collaboration). Thus, the token must still be held by another agent, and there is no need to create a new one.

In the example, assuming that C leaves, then D, B and A will discard all the local abductions they have done after $\rightsquigarrow_3$, and B will regain the token so it can continue its local abductions started before $\rightsquigarrow_2$.

Any agent that has not received any state can leave any time without affecting the collaboration at all. However, if an agent joins "too late" in the collaboration, it may miss the opportunity to help others (a state with a non-abducible goal that has been delayed by everyone once may be discarded before the new agent can have a chance to process it) and hence will affect the completeness. In addition, the following situation can cause problem to the collaboration in open MAS: an agent repeatedly joins the system, modifies the state, and then leaves the system. In this case the collaboration may not terminate. But this is not an intrinsic problem of the distributed algorithm and it could be avoided by imposing special restrictions in the implementation, e.g. forbid *buggy* agents from joining and leaving too many times.

4.6 Conclusion

In this chapter, we have presented DAREC – the first general purpose distributed abductive reasoning system that guarantees *global consistency* and supports *constraint satisfaction.* The DAREC distributed algorithm is a distributed state rewriting process that is extended from the ASystem algorithm. Therefore, it can accept non-ground negative queries, compute non-ground answers and perform constructive negation. During collaborative reasoning, agents interact through state passing, which is controlled by the application dependent *agent interaction strategy* and *agent selection strategy*. This algorithm design allows concurrent computation to be performed and inter-agent communications to be reduced. We have proven the soundness and completeness of the DAREC distributed algorithm with respect to the three-valued completion semantics for abduction, and described how the system can be extended for *open agent systems*.

Although the DAREC distributed algorithm is extended from the ASystem algorithm, it is not a simple combination of the ASystem inference rules and tags handling (i.e., the extended local inference rules **LD1**, **LA1**, **LD2**, **LA2**, and the transfer rule **TR**). More importantly, the DAREC distributed algorithm focuses on the coordination of abduction which guarantees global consistency of the answers. The coordination consists of the agent selection strategy, the agent interaction strategy, and the token controlled protocol, which also allows parallel abductive search among the agents (i.e., agents can perform their local abductions concurrently and communicate asynchronously) without overloading the agents and the communication channel. Therefore, the relationship between the DAREC algorithm and the ASystem algorithm can be seen as

DAREC Algorithm = (ASystem Algorithm + Tags Handling) + (Agent Selection/Interaction Strategies + Token Controlled Protocol).

Both DARE and DAREC allow parallel computation during distributed abduction. However, the types of parallelism are different. In DARE, during agent interaction after a query is sent to several helper agents, the helper agents compute the answers for this query in parallel. The computed answers will be returned to the query sender and will be used by the query sender's local proof. In DAREC, the agent interaction is different. First, a (transferable) state (which can sufficiently describe a global abductive task) instead of a query is passed between agents. Secondly, the state is sent to one helper agent only. Thirdly, the helper agent does not need to return the computed answers to the state sender, as any computed answer is an answer for the global abductive task. The parallelism of a DAREC algorithm computation comes from the fact that the state sender agent continues its local abduction after sending out the state to the helper agent, which will start and perform a new local abduction in parallel.

DAREC has many advantages over the superseded DARE system. However, it does not consider confidentiality concerns in the distributed knowledge. In the next chapter, we will study how DAREC can be customised, through the *goal selection strategy* and the *agent interaction strategy*, to allow *confidential abductive reasoning*.

Chapter 5

Multi-agent Hypothetical Reasoning with Confidentiality (DAREC^2)

5.1 Introduction

In DAREC , the distributed agent knowledge is represented by a global abductive framework. Under such a framework, all non-abducibles are assumed to be global and shared by all agents. For example, if an agent has a rule $p \leftarrow B_1$ in its background knowledge, and another agent has $p \leftarrow B_2$, then both rules are considered to be the definitions of the same non-abducible predicate p.

However, there are potential applications that do not necessarily assume this. For example, if there are two agents a and b, such that

$$\Pi_a = \{free \leftarrow no\_lecture.\}$$

and

$$\Pi_b = \{free \leftarrow no\_appointment.\}$$

The non-abducibles $free$ and $no\_lecture$ in Π_a probably means “agent a is free” and “agent

a has no lecture to give", respectively; whereas the non-abducibles $free$ and *no_appointment* in Π_b probably means "agent b is free" and "agent b has no appointment", respectively. Thus, the predicate $free$ in Π_a is different from that in Π_b, i.e., there is a predicate naming conflict between the two agents. In these applications, there is a need to resolve predicate naming conflicts and to distinguish between each agent's *local predicates.*

One solution to this problem would be to enforce each agent's local predicate to be *renamed* in such a way that the agent's unique identifier is appended after the predicate name. For example, after the renaming process a and b will have the following new rules:

$$\Pi_a = \{free\_a \leftarrow no\_lecture\_a.\}$$

and

$$\Pi_b = \{free\_b \leftarrow no\_appointment\_b.\}$$

However, this solution has two drawbacks. First, the local predicate renaming process cannot be performed by DAREC by default, as the system does not know which predicates are local to a particular agent. Therefore, the burden of predicate renaming is left to the user while modelling the agent knowledge. Secondly, suppose that there is an agent c that needs to convene anyone who is free for a meeting, then c has to have the following (ground) rules in Π_c:

$$\Pi_c = \left\{ \begin{array}{l} schedule\_meeting\_with(a) \leftarrow free\_a. \\ schedule\_meeting\_with(b) \leftarrow free\_b. \end{array} \right\}$$

Such knowledge representation is not preferred as it is not succinct, and c must know how many other agents are available in the system. In addition, whenever there is a change to the set of agents, c has to update his background knowledge. A better representation of the problem would be

$$\begin{aligned}
\Pi_c &= \{schedule\_meeting\_with(X) \leftarrow free(X).\} \\
\Pi_a &= \{free(a) \leftarrow no\_lecture\_a.\} \\
\Pi_b &= \{free(b) \leftarrow no\_appointment\_b.\}
\end{aligned}$$

However, in this case we have the same (old) problem: we cannot decide whether $free/1$ is a shared predicate or a local predicate based on its predicate name.

There seems to be a general solution to all of the above issues – let us enforce in the language that:

- every non-abducible predicate has non-zero arity, and
- the first argument of any non-abducible atom is always an agent identifier or a variable that can only bind to agent identifiers. Thus, this argument is called the *agent identifier argument* (or *agent ID argument*);
- any atom with the agent ID argument being a variable is considered to be global, and is shared by all agents; whereas any atom with the agent ID argument being ground is consider to be local to only the agent with that ID.

For example, the background knowledge of the agents in the example would become (let us mark the agent ID argument with underline):

$$\begin{aligned}
\Pi_c &= \{schedule\_meeting\_with(\underline{c}, X) \leftarrow free(\underline{X}).\} \\
\Pi_a &= \{free(\underline{a}) \leftarrow no\_lecture(\underline{a}).\} \\
\Pi_b &= \{free(\underline{b}) \leftarrow no\_appointment(\underline{b}).\}
\end{aligned}$$

Note that the non-abducible $free(\underline{X})$ in Π_c has its agent ID argument being non-ground, so any definition of $free$ (with arity of 1) in any agent's background knowledge can be used as its definition to resolve it during inference (although this may involve agent interactions). Note also that the non-abducible $free(\underline{a})$ in Π_a (or $free(\underline{b})$ in Π_b) has its agent ID argument being ground, thus it is local to agent a (or b), i.e., only agent a (or b) can have definitions for $free(\underline{a})$ (or $free(\underline{b})$).

This solution has an additional advantage – it allows the reasoning over agent ID arguments. For example, suppose that the convener c wants to schedule meeting with only academics, then Π_c will become

$$\Pi_c = \left\{ \begin{array}{l} schedule\_meeting\_with(\underline{c}, X) \leftarrow academic(\underline{c}, X), free(\underline{X}). \\ academic(\underline{c}, a). \\ student(\underline{c}, b). \end{array} \right\}$$

Thus, given a query $schedule\_meeting\_with(\underline{c}, X)$ to c, after the collaborative reasoning there should be only one answer, with $X = a$.

With this solution, we have the following observation during the agent collaborations:

Unnecessary Interaction : Suppose there is an additional rule $busy(\underline{a}) \leftarrow \neg free(\underline{a})$ in Π_a, and during one of agent a's local abductions a needs to reduce the selected non-abducible goal $busy(\underline{a})$. According to the DAREC inference rule LD1 and LN1, the goal will be reduced to $\neg free(\underline{a})$, which will then be converted to $\leftarrow free(\underline{a})$. Subsequently, according to LD2 (applies twice), $\leftarrow free(\underline{a})$ is resolved to be $\leftarrow no\_lecture(\underline{a})$, and both denials are collected as non-abducible constraints and become tagged by all other agents, so that the agents will check them for consistency in the subsequent collaboration. Clearly, the checks by agents rather than a are unnecessary, as they will not have definitions for $free(\underline{a})$ and $no\_lecture(\underline{a})$ in their background knowledge.

Confidentiality Concern : Suppose there is an additional rule $special\_meeting(\underline{c}, X) \leftarrow student(\underline{c}, X), free(\underline{X}), bad\_attendance(\underline{c}, X).$ in Π_c, and during one of agent c's local abductions c needs to solve a denial goal $\forall X. \leftarrow special\_meeting(\underline{c})$ (e.g., "no special meeting can be scheduled"). Assuming that a left to right safe goal selection strategy is used, the denial will eventually be reduced to " $\leftarrow free(\underline{b}), bad\_attendance(\underline{c}, b)$. According to LD2, it will become tagged by b and will be collected in a transferable state, which can be passed to b so b will continue processing it. However, c may not want b to know the reason for the "special meeting" during scheduling (i.e., bad attendance), but this information can be inferred by b after it receives a state containing the collected denial.

To adopt a solution for addressing all the issues discussed above, we have customised DAREC for applications which requires the separation of shared and confidential knowledge modelling, and the support for reasoning over them. Specifically, a new type of atom, called *askable* of the form $p(\vec{u})@ID$ (similar to the *askable literal* in [SIIS00] and the *identification-based literal* in [BSW08]), is introduced. Syntactically, an askable atom is the same as a non-abducible $p(ID, \vec{u})$, but has a special characteristic such that its first argument ID can only be an agent identifier. Thus, at knowledge representation level, askable atoms and their definitions can be used to model shared knowledge, whereas other non-abducibles atoms and their definitions can be used to model confidential knowledge. In addition, the DAREC distributed algorithm is also customised, mainly through the optimisation of local inference rules and fixing special agent interaction and goal selection strategies, so that it is aware of different types of atoms and can handle them correctly. The main challenges are to make sure no confidential information of an agent's can be inferred or disclosed to others during the collaboration, and to deal with askable atoms correctly when its first argument is not ground, i.e., it can be a universal or existential variable. The new system is called DAREC^2 (e.g., DAREC with Confidentiality).

In summary, DAREC^2 differs from DAREC in the following ways:

- it allows askable atoms to be used in agent knowledge;
- it has extended inference rules to handle askable goals;
- it enforces special goal selection and agent interaction strategies to guarantee confidentiality during collaborative reasoning.

The rest of the chapter is organised as follows. Section 5.2 introduces the concept of askable atoms, and gives updated definitions to the various concepts of a global abductive framework.

Section 5.3 first describes the modifications to the local inference rules, and then describes the customisations made in the coordination. Section 5.4 proves that the algorithm can maintain confidentiality during execution, and discusses further possible optimisation and extensions. A prototype implementation of DAREC^2 in Prolog is described in Section 5.5. Related work of DAREC and DAREC^2 is discussed in Section 5.6. Finally, Section 5.7 concludes the chapter.

5.2 Distributed Framework with Confidentiality

Since DAREC^2 is a customisation of DAREC, many of the DAREC assumptions regarding the targeted multi-agent systems are inherited:

- the set of agents is fixed;
- agents agree on the same set of abducible predicates;
- the communication graph of the agents is fully connected, and the communication channels are reliable.

DAREC^2 does not require all (non-abducible) predicates to be global. It assumes that (a) every non-abducible predicate symbol has non-zero arity, and (b) the first argument of a non-abducible atom must be either an agent identifier or a variable. This special argument is called the *agent identifier argument* (or *ID argument* in brief). The set of non-abducible predicates can therefore be further categorised into two types:

- ***Public* non-abducible predicates:** They are non-abducible predicates which may appear in the local expertise (i.e., background knowledge and integrity constraints) of different agents.
- ***Local* non-abducible predicates:** They are non-abducible predicates which can only appear in one agent's local expertise. The agent whose local expertise containing a local non-abducible predicate is called the *owner* of the predicate. Without loss of generality,

it is further assumed that every local non-abducible predicate name is always suffixed with the owner agent's ID, and the first argument of an atom with local non-abducible predicate is always the owner agent's ID, i.e., $p\_id(id, \vec{u})$ where id is the owner agent's ID, and $\vec{u}$ is a vector of the atom's remaining arguments.

In order to distinguish between these two types of predicates and to simplify notation, we introduce two syntactic sugars into to the DAREC^2 knowledge representation language:

Definition 5.1 (Askable Atom). *An askable atom is an atom with public non-abducible predicate, i.e., $p(ID, \vec{u})$, and is often written as $p(\vec{u})@ID$.*

Definition 5.2 (Private Atom). *A private atom of an agent ag is an atom with a local non-abducible predicate of the agent, i.e., $p\_ag(ag, \vec{u})$, and is often written as $p(\vec{u})$ in Π_{ag} or $\mathcal{IC}_{ag}$.*

An askable literal is either an askable atom, or the negation of an askable atom, i.e., $\neg(p(\vec{u})@ID)$, or simply $\neg p(\vec{u})@ID$. Similarly, a private literal is either a private atom or the negation of a private atom.

An atom is defined to be *askable* to indicate that it has an operational meaning: during collaborative reasoning, if an askable atom is a goal to be reduced and its ID argument is ground, i.e., equal to the ID of some agent α, then this goal should only be resolvable with the background knowledge of agent α (Π_α). In words, only α can (be asked to) reduce this goal. This implies that only Π_α should contain definitions of the askable atom. Thus, the following *restrictions on the appearance of askable atoms* in different agent local expertise are assumed in DAREC^2 :

- if $p(\vec{u})@ID$ appears as the head of a rule in the background knowledge of an agent α (Π_α), then ID must be ground and equal to α;
- if $p(\vec{u})@ID$ (or $\neg p(\vec{u})@ID$) appears as a body literal of a rule or integrity constraint in the local expertise of an agent α ($\Pi_\alpha \cup \mathcal{IC}_\alpha$), then ID can be either a variable or an agent identifier, which may or may not be equal to α.

An atom is defined to be *private* to indicate that during collaborative reasoning it cannot be revealed to any other agents through state transfers. Private atoms of an agent, and their definitions in the agent's background knowledge, can be used to model the part of private or confidential knowledge of the agent. How these private atoms are defined, or how they are used to define askable atoms by the agent, are not disclosed during or after the collaboration.

For the rest of this chapter, a *(DAREC^2) global abductive framework* simply refers to a DAREC global abductive framework that allows the usage of askable and private atoms. For example, a rule is $A \leftarrow L_1, \ldots, L_n$ $(n \geq 0)$ where A is either an askable atom or a private atom, and each L_i is a literal. A non-abducible (atom) constraint is called an *askable (atom) constraint* if the constrained atom is an askable atom, i.e., of the form $\forall \vec{X}. \leftarrow p(\vec{u})@ID, \Phi$, and is called a *private (atom) constraint* if the constrained atom is a private atom, i.e., of the form $\forall \vec{X}. \leftarrow p(\vec{u}), \Phi$.

Now let us look at how Example 4.1 (simple ambient intelligent system) in Chapter 4 can be represented in DAREC^2. For the convenience of the reader, we include the original example description here.

Example 5.1. *Ann and Bob live in the same care home, where a number of sensing devices are installed. For example, a corridor sensor (***cor***) detects movements along the corridor, and a window monitor (***wm***) can check which window(s) of the house are open/closed. There is also a home controller (***hm***) that can respond to events taking place inside the house, such as setting off an alarm if an intruder is detected, or notifying a nurse when a resident is in difficulty. Bob has a mental condition. Unless taking regular medication, he tends to wander around the house instead of staying in his room. So Bob is always carrying a personal device (***bob***) that logs his medication intakes. Ann is in good health and can leave the house when necessary, e.g. going to a dental appointment. Ann also carries a personal device (***ann***) which keeps her calendar and appointments. All the sensing and personal devices (except the base sensors, like the corridor sensor, which merely generate detected event notifications to* **hm***) have reasoning capability. About 12pm on Monday,* **cor** *detects movement and informs* **hm***.* **hm** *then needs to collaborate with various devices to explain the event before taking appropriate action.*

The global abductive framework $\langle\{hm, wm, ann, bob\}, \{\mathcal{F}_{hm}, \mathcal{F}_{wm}, \mathcal{F}_{ann}, \mathcal{F}_{bob}\}\rangle$ with $\mathcal{AB}_{hm} = \mathcal{AB}_{wm} = \mathcal{AB}_{ann} = \mathcal{AB}_{bob} = \{walkInCorridor\}$ can be reformulated as follows (with confidentiality concerns):

$\mathcal{F}_{bob}$: Bob cannot be walking in the corridor if he has taken medicine in the past 2 hours, and his most recent intake is at 11am. However, Bob does not want others to know this effect of the medicine to him, and he wants to keep his log of medicine intake private. Thus, the non-abducible predicate *takenMedicine* is considered local.

$$\left[\begin{array}{l} \Pi_{bob} = \left\{\ takenMedicine(11).\ \right\} \\ \mathcal{IC}_{bob} = \left\{\ \leftarrow walkInCorridor(bob, T), takenMedicine(T1), T-2 \leq T1, T1 \leq T.\ \right\} \end{array}\right]$$

$\mathcal{F}_{ann}$: Ann has a dental appointment from 11am to 1pm. She does not mind others knowing she is away, but she does not want others to know the reason, and she definitely wants to keep all of her doctor appointments confidential. Thus, *out* is considered public and *appointment* is considered local.

$$\left[\begin{array}{l} \Pi_{ann} = \left\{\begin{array}{l} appointment(dental, 11, 13). \\ out(T)@ann \leftarrow appointment(A, T1, T2), T1 \leq T, T \leq T2. \end{array}\right\} \\ \mathcal{IC}_{ann} = \emptyset \end{array}\right]$$

$\mathcal{F}_{hm}$: The home controller has knowledge about possible causes to known events. For example, movement in the corridor can be of either an occupant or an intruder. Let us assume that only the home controller has information about the current occupants and other devices of the house, and does not need to disclose this information to other devices. However, in order to prove that the movement can be caused by an intruder, it needs to infer that there is a possible point of entry, which can only be proven by the window monitor. Thus, *occupant* and *monitor* are considered local; whereas *pointOfEntry* is considered public.

$$\left[\begin{array}{l} \Pi_{hm} = \left\{\begin{array}{l} movement(cor, T)@hm \leftarrow \\ \qquad occupant(X), walkInCorridor(X, T). \\ movement(cor, T)@hm \leftarrow \\ \qquad monitor(M), pointOfEntry(T)@M, walkInCorridor(intruder, T). \\ occupant(X) \leftarrow X \in \{ann, bob\}. \end{array}\right\} \\ \mathcal{IC}_{hm} = \left\{ \leftarrow walkInCorridor(X, T), X \neq intruder, out(T)@X. \right\} \end{array}\right]$$

$\mathcal{F}_{wm}$: The window monitor has the status information of the windows on different floors. Any open window on the 1st floor is a possible point of entry for a potential intruder. Since the window monitor only needs to give feedback to the home controller regarding point of entry, the windows status information can remain private. Thus, *floor* and *open* are considered local.

$$\left[\begin{array}{l} \Pi_{wm} = \left\{\begin{array}{l} pointOfEntry(T)@wm \leftarrow open(W), floor(W, 1). \\ open(w1). \\ floor(w1, 1). \\ floor(w2, 2). \end{array}\right\} \\ \mathcal{IC}_{wm} = \emptyset \end{array}\right]$$

5.3 Distributed Abduction with Confidentiality

The main objective of the DAREC^2 distributed algorithm (which is a customisation of the DAREC distributed algorithm) is to provide support for askable and private atoms, and maintain confidentiality during collaboration without degrading the system performance. Here we define what we mean by *confidentiality*.

Property 5.1 (Confidential Reasoning). *Given a global abductive task, a DAREC distributed algorithm (or its customisation) guarantees confidential reasoning if and only if no private predicate or atom of any agent can be seen by another agent during or after the computation.*

In this section, we present the DAREC^2 distributed algorithm. We will prove that it guarantees confidential reasoning in Section 5.4.1.

5.3.1 Customisation of the Local Inference Rules

Let us first consider the following observations from the application of the existing DAREC local inference rules with selected goals involving private or askable atoms during an agent's (α's) local abduction:

1. Any private atom of α's is defined only in Π_α. Thus, it should not be delayed when it is selected to reduce, as no one else can be expected to help in resolving it.
2. If an askable goal is selected and its ID argument is equal to α, then α should not delay it either as only Π_α may have definitions for the askable atom, i.e., no one else is able to help.
3. If a denial goal $\forall\vec{X}. \leftarrow \phi, \Phi$ is selected where ϕ is either a private atom of α's or an askable atom whose ID argument is equal to α, then after new denial goals are generated by resolving ϕ with Π_α, the denial should not need to be collected into the set of dynamic integrity constraints or be tagged by all other agents, as no other agent can generate new denial goals from it by resolving ϕ.
4. If a denial goal $\forall\vec{X}. \leftarrow p(\vec{u})@\beta, \Phi$ is collected as a dynamic integrity constraint, where β is an agent identifier different from α, then the denial does not need to be tagged by any agent other than β. This is because everyone except β does not have any definition for $p(\vec{u})@\beta$ in its background knowledge, and trivially satisfies the denial. Furthermore, the denial does not need to be tagged by β, as the ID argument of its constrained atom $p(\vec{u})@\beta$ clearly indicates that it has to be checked by β. Thus, such denial goals can be treated as delayed goals rather than global integrity constraints.
5. Recall that in DAREC, tags of delayed (non-abducible) goals have no semantic meanings and they are only used to avoid indefinite state transfers due to repeated delay of some

goal by all agents. Suppose now only askable goals can be delayed (i.e., as follows from observation (1)), then the tags of delayed goals can be replaced by the finite domain constraints of the ID argument generated from the delayed goals. For example,

Example 5.2. *Consider a system with only two agents α and β, where*

$$\Pi_\alpha = \left\{ \begin{array}{l} p(X)@\alpha \leftarrow q(X). \\ q(1). \end{array} \right\} \textit{ and } \Pi_\beta = \left\{ \begin{array}{l} p(X)@\beta \leftarrow r(X). \\ r(2). \end{array} \right\}$$

Let Θ_1 be a state in a local abduction by α containing only one goal $p(X)@ID$ where ID is a variable. By LD1 of DAREC, two states Θ_2 and Θ_3 could be generated: in Θ_2 the goal is replaced with $ID = a, q(X)$, and in Θ_3 the goal $p(X)@ID$ is delayed and is tagged by α. Then after Θ_3 is sent to β, by LD1 again two states Θ_4 and Θ_5 could be generated: in Θ_4 the goal is replaced with $ID = b, r(X)$, and in Θ_5 the goal $p(X)@ID$ is delayed again and is tagged by both α and β. Given any agent selection strategy that is aware of the tags (e.g., the uniform agent selection strategy (Definition 4.9)) Θ_5 will not be sent to α again (and will be discarded as there is no suitable candidate recipient), and hence the algorithm will terminate. On the other hand, even if the adopted agent selection strategy is not aware of the tags so that Θ_5 is sent to α, α will not generate any new state from Θ_5 by reducing or delaying $p(X)@ID$ again, since it has already been tagged by α. Thus, the algorithm will still terminate.

In fact, instead of tagging the delayed goal $p(X)@ID$ with α in Θ_3, we could try to add an inequality $ID \neq \alpha$ to the set of collected inequalities (i.e., $\mathcal{E}_3$). Similarly, instead of tagging the same goal with β in Θ_5, we can also try to add an inequality $ID \neq \beta$ to $\mathcal{E}_5$. However, $ID \in \{\alpha, \beta\}$ [1] and $ID \neq \alpha$ are inconsistent with $ID \neq \beta$. Thus, Θ_5 will not be generated as a child state of Θ_3, and the algorithm will terminate without invoking the agent selection strategy at all.

Using ID argument constraints instead of tags has another advantage – it gives delayed (askable) goals some semantic meaning, i.e., $p(\vec{u})@ID$ with $ID \in \Sigma$ and $ID \neq \alpha$ is

[1]This constraint comes from the definition of ID argument and can be enforced by the DAREC^2 distributed algorithm (see description later).

equivalent to $\bigvee_{[ID \in (\Sigma \setminus \{\alpha\})]} p(\vec{u})@ID$, *or* $\underbrace{p(ag_1, \vec{u}) \vee \cdots \vee p(ag_n, \vec{u})}_{\{ag_1,\ldots,ag_n\} = \Sigma \setminus \{\alpha\}}$.

In order to support a reasoning process that is aware of askable and private literals, and optimise it according to the above observations, the DAREC local inference rules are extended. In particularly, LD1 and LD2 are extended to be LDP1, LDP2, LDA1 and LDA2, where LDP1 and LDP2 are for positive and denial goals of private atoms, respectively, and LDA1 and LDA2 are for positive and denial goals of askable atoms, respectively. We will present the new rules in order. But we first give the definition of a DAREC^2 state, which extends that of a DAREC state.

Definition 5.3 (**DAREC^2 Computational State).** *A DAREC^2 computational state (or state in brief)* Θ *is* $\langle (\mathcal{G}, \mathcal{G}^d), \mathcal{ST}, \tau \rangle$, *where*

- $\mathcal{G}$ *is a set of remaining goals;*
- $\mathcal{G}^d$ *is a set of delayed goals, each of which is either an askable atom or an askable atom constraint;*
- $\mathcal{ST} = (\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C})$ *is a tuple of four stores, where*
  - Δ *is a set of abducibles;*
  - $\mathcal{N}$ *is a set of abducible constraints;*
  - $\mathcal{E}$ *is a set of (in-)equalities;*
  - $\mathcal{C}$ *is a set of CLP constraints;*
- τ *is a set of abducible tags.*

All free variables appearing in the state Θ *are existentially quantified within the scope of the whole state.*

A DAREC^2 state differs from a DAREC state in the following ways:

- askable atom constraints are treated as delayed goals;

- the set of collected dynamic integrity constraints contains only abducible constraints;
- only abducibles can be tagged.

Consequently, a DAREC^2 *solved state* is a state in which there is no delayed goal or abducible tag, and the four stores are consistent. A DAREC^2 *transferable state* is a state that contains a delayed goal or an abducible tag.

Given a (DAREC^2) global abductive framework $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$, let $\Theta_i = \langle (\mathcal{G}_i, \mathcal{G}_i^d), (\Delta_i, \mathcal{N}_i, \mathcal{E}_i, \mathcal{C}_i), \tau_i \rangle$ be a state yet to be processed in the local abduction by agent $\alpha \in \Sigma$, where $\mathcal{G}_i \neq \emptyset$. Suppose a goal ϕ is selected from $\mathcal{G}_i$ according to a safe goal selection strategy Ξ, and let $\mathcal{G}_i^- = \mathcal{G}_i \setminus \{\phi\}$.

Inference Rule (LDP1). *If $\phi = p(\vec{u})$ is a private atom, let $p(\vec{v}_j) \leftarrow \Phi_j$ $(j = 1, \ldots, n)$ be n rules in Π_α, then:*

– *(local reduction)* $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_1\} \cup \Phi_1 \cup \mathcal{G}_i^-$

`OR` $\vdots$

`OR` *(local reduction)* $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_n\} \cup \Phi_n \cup \mathcal{G}_i^-$

Inference Rule (LDP2). *If $\phi = \forall \vec{X}.\leftarrow \varphi, \Gamma^-$ is a denial goal and $\varphi = p(\vec{u})$ is a selected private atom, then:*

– $\mathcal{G}_{i+1} = \{\forall \vec{Y}. \leftarrow \Gamma^+ \mid p(\vec{v}) \leftarrow \Phi \in \Pi_\alpha$ `and` $\vec{Y} = \vec{X} \cup vars(p(\vec{v})) \cup vars(\Phi)$ `and` $\Gamma^+ = \{\vec{u} = \vec{v}\} \cup \Phi \cup \Gamma^-\} \cup \mathcal{G}_i^-$.

These two rules are very similar to LD1 and LD2, except that:

- in LDP1, private goals cannot be delayed, and hence cannot be tagged (i.e., no need to modify τ_i);
- in LDP2, private atom constraints do not need to be collected or tagged (i.e., no need to modify $\mathcal{N}_i$ or τ_i).

In contrast, LDA1 and LDA2 are less similar to LD1 and LD2, because they need to deal with the generation of CLP constraints over ID arguments (i.e., to replace the tags of delayed goals), and more importantly they have to correctly handle non-ground ID arguments according to their quantifiers while processing askable constraints.

Inference Rule (LDA1). *Let α be the current agent processing Θ_i. If $\phi = p(\vec{u})@ID$ is an askable atom, one of the following cases applies:*

- *ID is ground:*

  * *(local reduction) if $ID = \alpha$, let $p(\vec{v}_j)@\alpha \leftarrow \Phi_j$ $(j = 1, \ldots, n)$ be n rules in Π_α, then*

  – $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_1\} \cup \Phi_1 \cup \mathcal{G}_i^-$

  `OR` $\vdots$

  `OR` $\mathcal{G}_{i+1} = \{\vec{u} = \vec{v}_n\} \cup \Phi_n \cup \mathcal{G}_i^-$

  * *(delay goal) if $ID \neq \alpha$, then $\mathcal{G}_{i+1} = \mathcal{G}_i^-$ and $\mathcal{G}_{i+1}^d = \{\phi\} \cup \mathcal{G}_i^d$*

- *ID is a variable:*

  – *(accept task) if $\{ID = \alpha\} \cup \mathcal{E}_i \cup \mathcal{C}_i$ is satisfiable, then $\mathcal{G}_{i+1} = \{p(\vec{u})@\alpha\} \cup \mathcal{G}_i^-$;*

  `OR` *(delegate task) if $\{ID \neq \alpha\} \cup \mathcal{E}_i \cup \{ID \in \Sigma\} \cup \mathcal{C}_i$ is satisfiable, then $\mathcal{G}_{i+1} = \mathcal{G}_i^-$, $\mathcal{G}_{i+1}^d = \{\phi\} \cup \mathcal{G}_i^d$, and $\mathcal{E}_{i+1} = \{ID \neq \alpha\} \cup \mathcal{E}_i$*

Note that the ID variable of any positive askable goal is existentially quantified. The *accept task* in LDA1 is to bind the ID argument of an askable goal to the current agent's identifier, and to use the current agent's background knowledge to reduce this goal (e.g., Θ_2 and Θ_4 in Example 5.2 are generated in this way). The *delegate task* in LDA1 delays the askable goal so it can be (later) reduced by another agent (e.g., Θ_3 in Example 5.2 is generated in this way).

Inference Rule (LDA2). *Let α be the current agent processing Θ_i. If $\phi = \forall \vec{X}. \leftarrow \varphi, \Gamma^-$ is a denial goal and $\varphi = p(\vec{u})@ID$ is a selected askable atom in the body of ϕ, one of the following cases applies:*

- *ID is ground:*

    * *(reduce local constraint) if* $ID = \alpha$, *then* $\mathcal{G}_{i+1} = \{\forall \vec{Y}. \leftarrow \Gamma^+ \mid p(\vec{v})@\alpha \leftarrow \Phi \in \Pi_\alpha$ and $\vec{Y} = \vec{X} \cup vars(p(\vec{v})) \cup vars(\Phi)$ and $\Gamma^+ = \{\vec{u} = \vec{v}\} \cup \Phi \cup \Gamma^-\} \cup \mathcal{G}_i^-$;
    * *(collect remote constraint) if* $ID \neq \alpha$, *then* $\mathcal{G}_{i+1} = \mathcal{G}_i^-$ *and* $\mathcal{G}_{i+1}^d = \{\phi\} \cup \mathcal{G}_i^d$;

- *ID is an existential variable (i.e.,* $ID \notin \vec{X}$*), then*

  – *(accept constraint) if* $\{ID = \alpha\} \cup \mathcal{E}_i \cup \mathcal{C}_i$ *is satisfiable, then* $\mathcal{G}_{i+1} = \{\forall \vec{X}. \leftarrow p(\vec{u})@\alpha, \Gamma^-\} \cup \mathcal{G}_i^-$;

  OR *(delegate constraint) if* $\{ID \neq \alpha\} \cup \mathcal{E}_i \cup \{ID \in \Sigma\} \cup \mathcal{C}_i$ *is satisfiable, then* $\mathcal{G}_{i+1} = \mathcal{G}_i$, $\mathcal{E}_{i+1} = \{ID \neq \alpha\} \cup \mathcal{E}_i$ *and* $\mathcal{G}_{i+1}^d = \{\phi\} \cup \mathcal{G}_i^d$;

- *ID is a universal variable (i.e.,* $ID \in \vec{X}$*), then*

  – *(instantiate constraint) let* $\Sigma^- = \{\beta \mid \beta \in \Sigma$ and $\{ID = \beta\} \cup \mathcal{E}_i \cup \mathcal{C}_i\ is\ satisfiable\}$, *and* $\mathcal{G}^{new} = \{\forall \vec{X}^-. \leftarrow p(\vec{u})@\beta, \Gamma^- \mid \beta \in \Sigma^-$ and $\vec{X}^- = \vec{X} - \{ID\}\}$, *then* $\mathcal{G}_{i+1} = \mathcal{G}^{new} \cup \mathcal{G}_i^-$

LDA2 reduces a denial goal by reducing the selected askable atom in the denial. If the ID argument of the askable is ground, then the denial only needs to be checked by the agent with matching identifier. If this agent is the current agent, the denial must be reduced by resolving the askable atom with the current agent's local background knowledge (i.e., the *reduce local constraint* case); otherwise, the denial is considered as a remote (integrity) constraint and can only be delayed by the current agent (i.e., the *collect remote constraint* case). If the ID argument is an existential variable, then the current agent can either take the responsibility to satisfy the denial goal by binding its identifier to the existential variable (i.e., the *accept constraint* case), or declare that it will not be responsible for satisfying the denial goal by adding an inequality to the state ensuring that the existential variable can never be bound to its identifier (i.e., the *delegate constraint* case). Finally, if the ID argument is a universal variable, then all the agents must satisfy the denial goal. This is enforced by generating one instance of the denial goal for each agent in the system (i.e., by binding the universal variable to all the agent identifiers in turn).

With respect to the transfer rule, since askable atom constraints are treated as delayed goals and are not tagged in a DAREC^2 state, a minor simplification to TR is needed (i.e., there is no need to process tagged non-abducible constraints):

Transfer Rule (TR'). *Let* $\Theta_t = \big\langle (\mathcal{G}, \mathcal{G}^d), (\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C}), \tau \big\rangle$ *be the state received by an agent* α *after a state transfer, then the root state for the new local abduction by* α *is* $\Theta_0 = \langle (\mathcal{F} \cup \mathcal{G}^d \cup \mathcal{G}, \emptyset),$ $(\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C}), \tau' \rangle$, *where* $\mathcal{F}$ *and* τ' *are obtained as follows:*

1. *Let* Δ^{new} *be the set of* α*-tagged abducibles, i.e.,* $\{A \mid A \in \Delta \texttt{ and } (A, \mathcal{S}) \in \tau \texttt{ and } \alpha \in \mathcal{S}\}$*;*

2. *Given an abducible* $A = a(\vec{u})$*, the set of resolvents of* A *with* $\mathcal{IC}_\alpha$ *is* $R_A(\mathcal{IC}_\alpha) = \{\forall \vec{Y}. \leftarrow \vec{u}{=}\vec{v}, \Phi^- \mid \leftarrow \Phi \in \mathcal{IC}_\alpha \texttt{ and } \Phi = \{a(\vec{v})\} \cup \Phi^- \texttt{ and } \vec{Y} = vars(\Phi)\}$*;*

3. $\mathcal{F} = \bigcup_{A \in \Delta^{new}} R_A(\mathcal{IC}_\alpha)$*;*

4. τ' *is obtained by removing* α *from the tags* τ*, i.e.,* $\{(L, \mathcal{S}) \mid (L, \mathcal{S}) \in \tau \texttt{ and } L \notin \Delta^{new}\} \cup \{(L, \mathcal{S}') \mid (L, \mathcal{S}) \in \tau \texttt{ and } L \in \Delta^{new} \texttt{ and } \mathcal{S}' = \mathcal{S} - \{\alpha\}\}$.

The rest of the DAREC local inference rules LA1, LN1, LE1, LC1, LA2, LN2, LE2 and LC2 can be applied to a DAREC state exactly in the same way as for a DAREC state. Hence, the set of local inference and transfer rules for DAREC^2 is given by LA1, LDP1, LDA1, LN1, LE1, LC1, LA2, LDP2, LDA2, LN2, LE2, LC2 and TR'.

5.3.2 Customisation of the Coordination

The general DAREC distributed algorithm does not fix any agent interaction strategy or fix any goal selection strategy (as long as it is safe). However, such *flexibility* of the algorithm cannot guarantee the confidentiality property. This is illustrated by the following cases:

1. During a local abduction by an agent α, after a goal (e.g., an askable $p(\vec{u})@\beta$ s.t. $\beta \neq \alpha$) is delayed, the state becomes a transferable state and may be sent to another agent, if the adopted agent interaction strategy decides to do so (e.g., the eager agent interaction

strategy). However, if the set of remaining goals is not empty and contains private (atom) goals of α, then these private goals will be sent along with the state, and hence can be revealed to the state recipient agent.

2. Let $F = \forall \vec{X}. \leftarrow a(\vec{u}), \Gamma$ be a denial goal selected to reduce on the abducible $a(\vec{u})$. During the application of LA2 F is collected as a dynamic integrity constraint (i.e., abducible constraint) for the new state. However, if Γ contains private atoms of α, then these atoms can be seen by all other agents, as the collected denial F will remain in (the descendants of) the new state.

3. Similarly to case (2), if a denial goal $F = \forall \vec{X}. \leftarrow p(\vec{u})@\beta, \Gamma$ $(\beta \neq \alpha)$ is delayed, and if Γ contains private atoms of α, then these atoms can at least be seen by agent β when β receives a state and processes the delayed goal.

To avoid cases (2) and (3), we need to make sure no abducible constraint can be collected and no askable constraint can be delayed if they contain some private atoms as body literals. This can be controlled by the *secure* goal selection strategy given in Definition 5.4. To avoid case (1), we need to make sure no transferable state containing private goals can be sent out by an agent. This can be controlled by the *lazy* agent interaction strategy given in Definition 5.5. This behaves as the opposite of the eager agent interaction strategy – it allows a transferable state to be sent out only if the state does not contain any remaining goal, i.e., the agent tries to process a state with its maximum efforts.

Definition 5.4 (***Secure*** **Goal Selection Strategy).** *A safe goal selection strategy Ξ adopted by an agent α is secure if and only if for a given denial goal $\forall \vec{X}. \leftarrow \Phi$ selected to be reduced, Ξ never selects from Φ an abducible or an askable whose ID argument is not equal to α, if Φ contains a private literal of α's.*

Definition 5.5 (**Lazy Agent Interaction Strategy).** *Given a transferable state $\Theta = \langle(\mathcal{G}, \mathcal{G}^d), \mathcal{ST}, \tau\rangle$ such that either $\mathcal{G}^d \neq \emptyset$ or $\tau \neq \emptyset$, an agent with the lazy agent interaction sends out Θ if and only if $\mathcal{G} = \emptyset$.*

Finally, since in DAREC^2 , delayed goals and askable constraints are no longer tagged, the *uniform agent selection strategy* (Definition 4.9) will also need to be modified to make use of the constraint reasoning over ID arguments:

Definition 5.6 (Uniform Agent Selection Strategy for DAREC^2). *Given a global abductive framework $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$, suppose that the agent identifiers Σ can be sorted lexicographically. Let $\Theta = \langle (\emptyset, \mathcal{G}^d), (\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C}), \tau \rangle$ be a transferable state, the uniform agent selection strategy for DAREC^2 selects an agent $\beta \in \Sigma$, such that*

- request help for delayed goals: *If $\mathcal{G}^d \neq \emptyset$, let $\widehat{\Omega}$ be a set of agent identifiers such that $\widehat{\Omega} = \{id \mid Q@ID \in \mathcal{G}^d \text{ and } id \in \Sigma \text{ and } \{ID = id\} \cup \mathcal{E} \cup \mathcal{C} \ is\ satisfiable\} \cup \{id \mid \forall \vec{X}. \leftarrow Q@ID, \Gamma \in \mathcal{G}^d \text{ and } ID \notin \vec{X} \text{ and } id \in \Sigma \text{ and } \{ID = id\} \cup \mathcal{E} \cup \mathcal{C} \ is\ satisfiable\}$. If $\emptyset \notin \widehat{\Omega}$ (i.e., each delayed goal must be have at least one candidate helper agent), then let $\mathcal{S} = \{id \mid \Omega \in \widehat{\Omega} \text{ and } id \in \Omega\}$. β is the first agent in the list obtained by lexicographically sorting $\mathcal{S}$.*

- request consistency check for abducibles and non-abducible constraints (remain the same): *if $\mathcal{G}^d = \emptyset$, and $\Omega = \bigcup_{\langle \phi, \mathcal{S} \rangle \in \tau} \mathcal{S}$ is not empty, then β is the first agent in the list obtained by lexicographically sorting Ω.*

5.3.3 Sample Execution Trace of DAREC^2

The execution trace of DAREC^2 for Example 5.1 given the query $\mathcal{Q} = \{movement(cor, 12)@hm\}$ is very similar to that for DAREC shown in Section 4.3.5, except that some states will have different contents in their stores. This is because the two executions both assume the lazy agent interaction strategy and the uniform agent selection strategy. Here we present the DAREC^2 execution trace and give extra comments on the places that are different from the DAREC execution trace.

1. **hm** created the initial state $\Theta_0 = \langle (\{movement(cor, 12)@hm\}, \emptyset), \mathcal{ST}^{\emptyset} \rangle$. After the application of LDP1, Θ_0 was rewritten into two states: $\Theta_1 = \langle (\{occupant(X), walkInCorridor(X, 12)\}, \emptyset), \mathcal{ST}^{\emptyset} \rangle$ and $\Theta_2 = \langle (\{monitor(M), pointOfEntry(T)@M, walkInCorridor(intruder, 12)\}, \emptyset), \mathcal{ST}^{\emptyset} \rangle$:

(a) For Θ_1, after the applications of LDP1, LC1 and LA1, Θ_1 became a transferable state $\Theta_3 = \langle(\{\leftarrow X \neq \mathit{intruder}, \mathit{out}(12)@X\}, \emptyset), (\{\mathit{walkInCorridor}(X, 12)^{\{ann,bob,wm\}}\}, \emptyset, \emptyset, \{X \in \{ann, bob\}\})\rangle$. However, since the set of remaining goals is not empty, Θ_3 is not allowed to be sent out yet. After the applications of LN2, LE1 and LDA2, the remaining denial goal became $\leftarrow \mathit{out}(12)@X$ and was moved to the set of delayed goals, and the additional constraints $\mathcal{E} = \{X \neq \mathit{intruder}, X \neq \mathit{hm}\}$ were added to the set of collected inequalities. The resulting state $\Theta_4 = \langle(\emptyset, \{\leftarrow \mathit{out}(12)@X\}), (\Delta_3, \emptyset, \mathcal{E}, \mathcal{C}_3)\rangle$ could not be further processed by **hm**.

(b) For Θ_2, after the applications of LDP1, LDA1, LA1, LN2 and LE1, Θ_2 became a transferable state $\Theta_5 = \langle(\emptyset, \{\mathit{pointOfEntry}(12)@\mathit{wm}\}), (\{\mathit{walkInCorridor}(\mathit{intruder}, 12)^{\{ann,bob,wm\}}\}, \emptyset, \emptyset, \emptyset)\rangle$, which could not be further processed by **hm**.

2. **hm** sent Θ_4 to **ann** after invoking the uniform agent selection strategy. After the application of TR, a new state $\Theta_6 = \langle(\{\leftarrow \mathit{out}(12)@X\}, \emptyset), (\{\mathit{walkInCorridor}(X, 12)^{bob,wm}\}, \emptyset, \mathcal{E}_4, \mathcal{C}_4)\rangle$ was obtained. Subsequently, after the application of LDA2, two new states were obtained as the ID argument X of the askable literal $\mathit{out}(12)@X$ in the denial goal was existentially quantified: $\Theta_7 = \langle(\{\forall T1, T2. \leftarrow \mathit{appointment}(T1, T2), T1 \leq 12, 12 \leq T2\}, \emptyset), (\Delta_6, \emptyset, \{X = \mathit{ann}\} \cup \mathcal{E}_4, \mathcal{C}_4)\rangle$ and $\Theta_8 = \langle(\emptyset, \{\leftarrow \mathit{out}(12)@X\}), (\Delta_6, \emptyset, \{X \neq \mathit{ann}\} \cup \mathcal{E}_4, \mathcal{C}_4)\rangle$.

   (a) for Θ_7, the remaining denial goal could not be satisfied, and was hence discarded (after a few local inference steps);

   (b) for Θ_8, it could not be further processed by **ann** so Θ_8 was passed to the next helper agent;

3. **ann** sent Θ_8 to **bob** after invoking the uniform agent selection strategy. After the application of TR, a new state $\Theta_9 = \langle(\{\forall T1. \leftarrow X = \mathit{bob}, \mathit{takenMedicine}(T1), 10 \leq T1, T1 \leq 12, \leftarrow \mathit{out}(12)@X\}, \emptyset), (\{\mathit{walkInCorridor}(X, 12)^{wm}\}, \emptyset, \mathcal{E}_8, \mathcal{C}_4)\rangle$. Subsequently, after the application of LE2 on the first remaining denial goal, two new states are obtained: $\Theta_{10} = \langle(\{\forall T1. \leftarrow \mathit{takenMedicine}(T1), 10 \leq T1, T1 \leq 12, \leftarrow \mathit{out}(12)@X\}, \emptyset), (\{\mathit{walkInCorridor}(X, 12)^{wm}\}, \emptyset, \{X = \mathit{bob}\} \cup \mathcal{E}_8, \mathcal{C}_4)\rangle$ and $\Theta_{11} = \langle(\{\leftarrow \mathit{out}(12)@X\}, \emptyset), (\{\mathit{walkInCorridor}(X, 12)^{wm}\}, \emptyset, \{X \neq \mathit{bob}\} \cup \mathcal{E}_8, \mathcal{C}_4)\rangle$:

   (a) for Θ_{10}, the first remaining denial could not be satisfied, so Θ_{10} was discarded (after a few local inference steps);

   (b) for Θ_{11}, the union of the inequality store and the constraint store contains $\{X \neq \mathit{bob}, X \neq$

$ann, X \in \{ann, bob\}\}$, and hence became inconsistent. Thus, the state Θ_{11} was also discarded.

4. Recall that **hm** still had a transferable state Θ_5. **hm** sent Θ_5 to **wm** after invoking the uniform agent selection strategy. After the application of TR, a new state $\Theta_{12} = \langle(\{pointOfEntry(12)@wm\}, \emptyset), (\{walkInCorridor(intruder, 12)^{\{ann,bob\}}\}, \emptyset, \emptyset, \emptyset)\rangle$ is generated by **wm**. Subsequently, after the applications of LDA1 and LDP1 by **wm**, the remaining goal was reduced successfully giving the transferable state $\Theta_{13} = \langle(\emptyset, \emptyset), \mathcal{ST}_{12}\rangle$. This state was passed to **ann** and **bob** in turn so that they could check the collected abducible. The subsequent derivation succeeded without adding any new element to Θ_{13}, and finally generated the solved state $\Theta_{13} = \langle(\emptyset, \emptyset), (\{walkInCorridor(intruder, 12)\}, \emptyset, \emptyset, \emptyset)\rangle$.

5.4 Discussions

5.4.1 Confidential Reasoning by DAREC^2

The main difference between DAREC^2 and the generic DAREC distributed algorithm includes the changes to a subset of the inference rules, and fixing a goal selection strategy and an agent interaction strategy. In this section, we prove that the DAREC distributed algorithm can guarantee *confidential reasoning* (Property 5.1).

Proposition 5.1. *Private atoms of an agent are never contained in any transferable state exchanged between agents during the execution of the DAREC^2 distributed algorithm, and thus DAREC^2 guarantees confidential reasoning.*

Proof of Proposition 5.1: To prove this proposition, we will use proof by contradiction. First, we assume that *during the agent collaboration, a transferable state* $\Theta = \big\langle(\mathcal{G}, \mathcal{G}^d), (\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C}), \tau\big\rangle$ *containing a private atom P of an agent α is passed between two agents.* By definition, P can only appear in $\Pi_\alpha \cup \mathcal{IC}_\alpha$, thus only α could add P to a state. Without loss of generality, we may further assume that Θ was generated by α, and that the state transfer took place between α and another agent, say β.

By the definition of lazy goal selection strategy, the set of remaining goals $\mathcal{G}$ of Θ must be empty. Thus, P must appear either in $\mathcal{G}^d$ or in $\mathcal{N}$.

Case 1: assume that P appears in $\mathcal{G}^d$. By the definition of LDP1, P is not allowed to be delayed if it is a positive private atom goal, so P must appear as a body literal of some askable constraint in $\mathcal{G}^d$. Let this askable constraint be $F =$ "$\forall\vec{X}. \leftarrow Q@ID, \Gamma$" where $Q@ID$ is an askable and $P \in \Gamma$. F could only have been added to $\mathcal{G}^d$ by the application of LDA2 when $Q@ID$ was selected by a secure goal selection strategy Ξ. By the definition of LDA2, ID must either be ground and not equal to α, or be an existentially quantified variable that cannot be bound to α. In either case, since $Q@ID$ was selected by Ξ, Γ must not have contained any private atom of α's, which contradicts the assumption that $P \in \Gamma$. Hence, P cannot appear in $\mathcal{G}^d$.

Case 2: assume that P appears in $\mathcal{N}$, i.e., P appears as a body literal of an abducible constraint in $\mathcal{N}$. Such an abducible constraint must have been collected into $\mathcal{N}$ by the application of LA2 when an abducible was selected by the secure goal selection strategy Ξ. However, according to Ξ, when the abducible was selected, the denial should not have contained any private atom of α. But this contradicts to the assumption of Case 2. Hence, P cannot appear in $\mathcal{N}$ either. ■
End of Proof for Proposition 5.1 .

5.4.2 Impact of the Usage of Agent Advertisements on Confidential Reasoning

As we discussed in Section 4.5.1, the usage of agent advertisements in DAREC can help reduce unnecessary agent interactions during the collaboration. This is also true in DAREC^2 . However, in the case of DAREC^2, agents should only advertise the set of *askable* atoms that they have definitions for and the set of *abducible* atoms that they have integrity constraints for. For example, in the execution trace of DAREC^2 in Section 5.3.3, if the agent advertisements were available, then the abducible $walkInCorridor(X, 12)$ would only be tagged by **bob**, and in Step 4 the state Θ_{12} would not need to be sent to **ann**. The usage of such agent advertisements does not affect the confidential reasoning property guaranteed by DAREC^2 , as the private atoms are not advertised.

5.4.3 Soundness and Completeness of DAREC^2

DAREC^2 is a customisation of DAREC. The askable and private atoms in DAREC^2 are equivalent to the non-abducible atoms in DAREC. Same as the DAREC algorithm, the DAREC^2 algorithm is sound with respect to the three-valued semantics. However, the DAREC algorithm is not complete with respect to the three-valued semantics. This is due to the *secure safe* goal selection strategy. Consider the following situation with two agents a and b, such that

$\mathbf{\Pi_a}$	$\mathbf{\Pi_b}$
$p(X) \leftarrow \neg q(X), r(X)@b.$	$r(1)@b.$

where p and q are local predicates of a's, and r is a public predicate of b's. Suppose a is given a query $p(X)$. There is an answer $X = 1$ for the query with respect to $\Pi_a \cup \Pi_b$. However, the DAREC^2 algorithm cannot compute it. There is because during a's local abduction, a denial goal $\forall X. \leftarrow \neg q(X), r(X)@b$ is obtained. When it is selected to reduce, the goal selection strategy must be *safe* so that the literal $\neg q(X)$ cannot be selected, as the literal contains universally quantified variable. However, the goal selection strategy must also be *secure* so that $r(X)@b$ cannot be selected, as there is a private literal in the denial goal. Therefore, the denial goal cannot be reduced and local abduction flounders. This incompleteness of the DAREC^2 is caused by the *secure safe* goal selection strategy. One mean to avoid this situation is to impose an *allowedness* condition to the agent background knowledge – *all variables occurring in a clause must also occur in a positive private body literal.*

5.5 Implementation of DAREC^2

In this section, we describe a prototype implementation of DAREC^2. A DAREC^2 system consists of a set of abductive agents, each of which can be implemented as a reasoning module, that can respond to queries from outside the system, or to internal collaboration requests from other agents within the system, and return answers once they are found. This modular implementation design has two advantages. First, these agents can be run as standalone programs, in which case the whole distributed system can execute as a distributed theorem prover for decentralised knowledge. Alternatively, each agent as a module can be integrated into the architecture (e.g., BDI) of an agent of a bigger multi-agent system, in order to aid their tasks, such as distributed cognitive perception and collaborative planning, by providing the hypothetical reasoning capabilities.

DAREC^2 could be implemented in different programming languages. Our current prototype uses Prolog as the main implementation language. This is because Prolog not only provides the best integration between Constraint Programming and Logic Programming, but also has a very efficient mechanism for unification, making it an excellent tool for implementing theorem provers. Among many existing Prolog systems, we have chosen YAP-6 [2] as the development system, due to its following advantages:

- It is considered the fastest open source Prolog implementation, and has been actively supported.
- It provides many important features and libraries essential to the implementation of DAREC^2 . For example, it has constraint solvers for finite domain constraints (CLP($\mathcal{FD}$)) and for Real domain constraints (CLP($\mathcal{R}$)) ported from the SWI-Prolog, and the *socket programming* library for implementing TCP communications between agents.
- It has multi-threading support, which is essential for the development of distributed software.

Before presenting the details of the implementation, we discuss some general properties and features used. As aforementioned, the availability of a "yellow-page" like directory, that records information about the non-abducible (or askable in the case of DAREC^2) predicates that are defined in the various agents and information about the integrity constraints for abducibles

[2] http://www.dcc.fc.up.pt/~vsc/Yap/

defined in the various agents, can be used to implement application specific agent selection strategies that will reduce unnecessary agent interactions. Such information can be automatically generated by the agents from their local expertise, and the disclosure of such information does not violate the *confidential reasoning* property that is guaranteed by the DAREC^2 distributed algorithm. Our current DAREC^2 implementation also makes use of such a directory and its relevant facilities.

Both DAREC and DAREC^2 assume a fixed set of agents during collaborative reasoning. However, in practice the set of the agents in the system may change before or after the collaborations, e.g., the set may expand during initial system set up. In our implementation, we enforce that there is always a *leader* agent in the system. Such a leader may either be appointed (e.g., the first agent running in the system) or be elected by the existing agents (e.g., after the ex-leader leaves) using some leader election algorithms. The leader acts as the portal of the system. Thus, any new agent wishing to join the system or any agent wishing to leave the system must report to the leader, and external queries should be sent to the leader.

The rest of this section is organised as follows. Section 5.5.1 describes the agent knowledge specification files as the input to DAREC^2 agents. Section 5.5.2 gives an overview of the internal architecture of a DAREC^2 agent. Section 5.5.3 and Section 5.5.4 describe the communications between agents and the protocols for which the agent system is set up (e.g., agent joining, leaving and knowledge update). Finally, Section 5.5.5 describes the execution of each agent for their collaboration in a global abductive task.

5.5.1 Agent Knowledge Specifications

In our implementation, we allow an agent to change its local expertise (i.e., background knowledge and integrity constraints) at runtime. However, the set of abducible predicates is fixed before runtime (i.e., it is agreed by all the agents). At the beginning an agent takes two files – a *topic file* containing the declarations of abducible predicates, and a *theory file* containing the initial local expertise of the agent. Both files are in Prolog syntax.

Within the topic file, an abducible predicate is declared using `abducible/1`. For example, to declare the abducible predicate symbol *walkInCorridor* with arity of two, the following fact is included in the topic file.

```
abducible(walkInCorridor(_,_)).
```

The theory file for an agent is a Prolog program. The rules in the background knowledge are written as Prolog clauses, e.g.,

```
free(D)@alice :- day(D), \+ appointment(D).
day(D) :- D in 1..7.
appointment(3).
```

Each integrity constraint is also written as a Prolog clause, but always with the head being `ic`, e.g.,

```
ic :- happens(A1, T), happens(A2, T), A1 =/= A2.
```

Note that in the theory file, negation $\neg$ is written as `\+` and inequality is written as `=/=`. CLP atoms are written using the convention of $CLP(\mathcal{FD})$ and $CLP(\mathcal{R})$.

5.5.2 Overview of the Agent Architecture

Our DAREC^2 agent implementation is a multi-threading implementation, which allows each agent to perform multiple global abductive tasks simultaneously. The key components of the agent implementation are shown in Figure 5.1.

There are two types of threads: a persistent *server thread* (ST) and one or more *worker threads* (WT). The main functionality of the ST is to respond to incoming messages (e.g., queries, agent advertisements, control signals for collaborations), manage WTs (e.g., create a WT for a new task), and maintain local storages. Its execution will be described in detail in Section 5.5.5. The main functionality of a WT is to perform local abduction in search for solutions (i.e., the

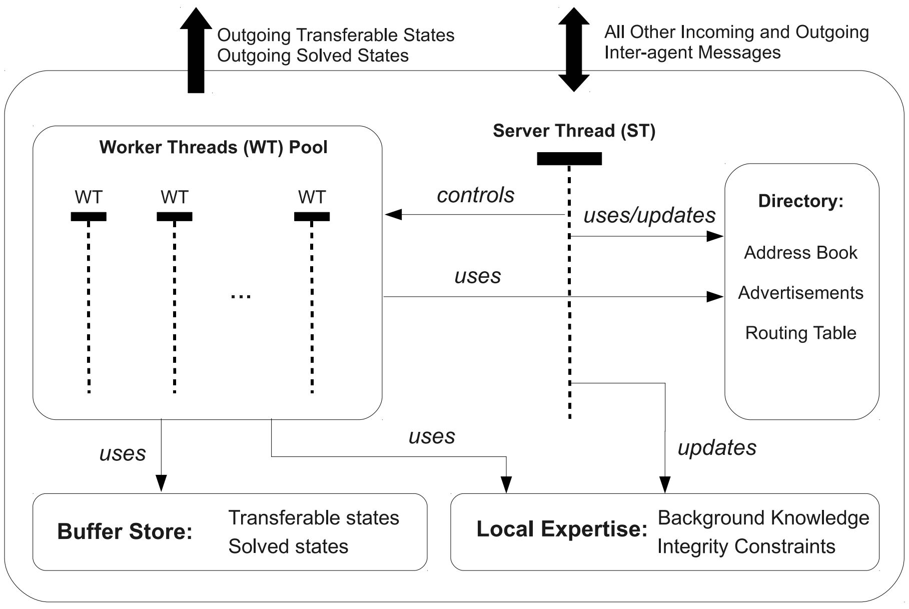


Figure 5.1: DAREC^2 Agent Internal Architecture

solved states). It can also send out collaborative requests (containing information such as the transferable state) to other agents, and return solutions to the query issuer(s). Its execution will be described in detail in Section 5.5.5.

There are three storage-like components: the *Local Expertise (store)*, the *State Buffer (store)* and the *Directory*. The Local Expertise store contains information of predicate types, agent background knowledge and integrity constraints. It is updated by ST, and is used by the WTs. The State Buffer temporarily stores transferable states and solved states that cannot be sent immediately, e.g., due to waiting for control token (described in Section 4.3.4). We would like to point out that in our current implementation, there is an option for the system to perform either the "push" or the "pull" style of query answering. In the "push" answering, any solved state found by a WT can be sent back to the query issuer regardless of whether the owning WT has a control token or not. This is good for applications where all solutions are required as soon as possible. However, if there are many solutions for a query, then the communication channel may be overloaded. In contrast, in the "pull" answering, a solved state can be sent back only if the owning WT has a control token. This is good for applications that want to control how many answers are needed and when to receive them. The usage of the State Buffer will be described in more detail in Section 5.5.5.

The Directory is mainly used by the WTs for selecting (helper) agents during collaboration. It has two sub-components:

Address Book : Agents in the system communicate through TCP messages. Each agent has two types of identifier: a unique *alias* name and a unique network address. An alias is a string constant (e.g., "ann" , "hm", "a1", "a2") that is mainly used for knowledge representation, i.e., in the agent knowledge specification. A network address is mainly used for low level TCP communication and consists of an IP address and a port number. Note that there may be several agents running on the same host machine on a network, thus the port number is used to further distinguish between these agents, i.e., each pair of IP address and port number uniquely identified an agent on the network. The Address Book records the mapping between agent aliases and agent network addresses. During the collaboration, if an agent wishes to send a message (e.g., containing transferable state) to another agent, say with alias "helper", then the sender agent first looks up the network address of "helper" from the Address Book, and then sends out the message using TCP.

Yellow Pages : Agents can advertise two types of information to all others: the (defined) askable atoms for which they have definitions, and the abducibles for which they have integrity constraints. The Yellow Pages records the agent advertisements, each of which is a pair $\langle Alias, Atom\rangle$ where $Alias$ is the alias of the advertising agent and $Atom$ is either an askable or abducible. Note that the askable atom in each advertisement always has the form $P@A$ where A must be ground and be an agent alias (because it is the head of a rule in that agent's background knowledge). Thus, an askable (abducible) advertisement is one that contains an askable (abducible) atom. In addition, the atom in each advertisement is in its *canonical* form, i.e., each variable in it is replaced by a new constant. By default, skolem constants start with "\$k" and have a number as suffix, e.g.,

"\$k1", "\$k2", ... Below are some examples of atoms and their canonical forms:

Original Atom	**Canonical Form**
$free(mon)@alice$	$free(mon)@alice$
$hapens(pickup, T)$	$happens(pickup, \$k1)$
$before(T1, T2)$	$before(\$k1, \$k2)$
$equal(X, X)$	$equal(\$k1, \$k1)$

Each agent's advertisements can be automatically generated from its local expertise. For the agent with alias "ag":

1. collect all the askable heads of the rules in the background knowledge;
2. collect all the (positive) abducible body literals of the integrity constraints;
3. transform the collected atoms to their canonical form and remove duplicates;
4. for each remaining atom SA, the pair $\langle ag, SA \rangle$ is an advertisement by "ag".

In addition, the Directory also records the alias of the current leader in the system.

5.5.3 Agent Communications

Agents communicate through peer-to-peer messages only. Each message has the following format:

```
p2p(FromAddr, ToAddr, MsgType, Payload)
```

where `FromAddr` and `ToAddr` are the network addresses of the sender and the recipient, respectively, and `MsgType` is a constant indicating the *type* of the message so that the recipient can understand how to parse the content `Payload` of the message upon receiving it. Note that a simple *broadcast* communication from an agent to a set of recipients can be implemented as the sender passing a peer-to-peer message to each of the recipients (in turn).

5.5.4 Protocols for Agent Joining, Leaving and Knowledge Update

Although the collaborative reasoning assumes the set of agents to be fixed, during runtime agents may join or leave the system, e.g., during initial system set up or changes to the larger MAS that uses DAREC^2 to aid its functionality. In addition, in a dynamic MAS environment, agents must be able to update their knowledge. In order to guarantee the collaboration assumptions and maintain the consistency during collaborations, the changes to the set of agents and the agent knowledge must not occur when the system is engaged in any global abductive reasoning tasks. Thus, agent joining/leaving and knowledge update must follow a set of pre-defined protocols, which are described next.

System Leader

As briefly discussed earlier in the Chapter, our implementation ensures that there is always a leader (agent) in the system. The leader behaves in the same way as other agents during collaboration, but has extra responsibilities, such as handling the changes to the system and receiving external queries (i.e., initiating a global abductive task for each received query). In order to make sure that the set of agents and the agent knowledge do not change when the system is engaged in any collaborative reasoning, the leader maintains a numeric variable `OngoingTasks` for counting the number of global abductive tasks being performed by the system. This variable, initially being zero, is incremented whenever the leader initiates a global task, and is decremented whenever a task is finished (see Section 5.5.5). Thus, an agent is allowed to join, leave or update its knowledge only when `OngoingTasks` is equal to zero.

The leader in a system can be either appointed or elected. Leader appointment occurs in two cases. At the initial system set up, the first agent to come alive in the system will be assumed (to be appointed) the role of being the leader. At the system runtime, the administrator can send a control message to the current leader, asking it to abdicate the role and to pass it to a specified agent in the system. Leader election often occurs when the current leader has to leave the system. The current implementation adopts a very simple algorithm: the agent aliases are assumed to be lexicographically sortable, and the agent (other than the leaving leader) with the smallest alias is "elected" as the next leader. Note that the changing of leader can only take place when the system is not engaged in any global reasoning tasks. The process is illustrated in Figure 5.2.

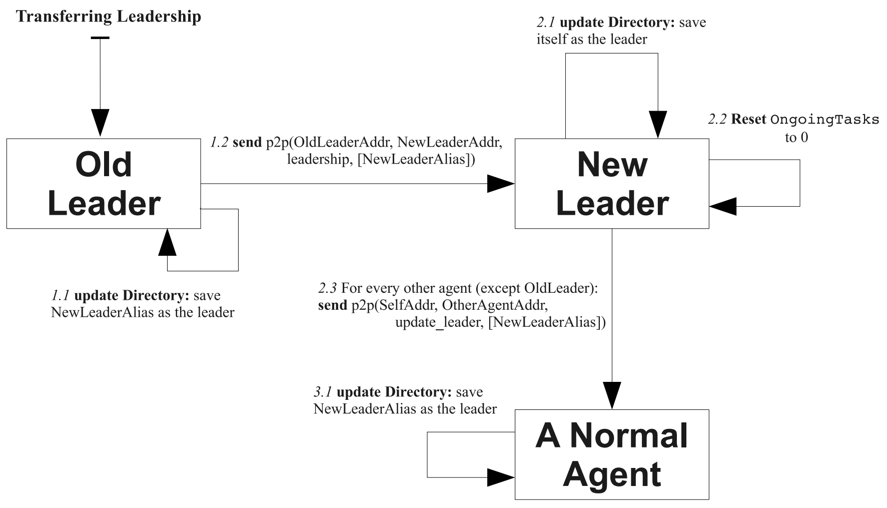

Transferring Leadership
Old Leader
1.2 send p2p(OldLeaderAddr, NewLeaderAddr, leadership, [NewLeaderAlias])
1.1 update Directory: save NewLeaderAlias as the leader
2.1 update Directory: save itself as the leader
New Leader
2.2 Reset OngoingTasks to 0
2.3 For every other agent (except OldLeader): send p2p(SelfAddr, OtherAgentAddr, update_leader, [NewLeaderAlias])
3.1 update Directory: save NewLeaderAlias as the leader
A Normal Agent

Figure 5.2: Changing of Leader

The main responsibilities of a leader include:

- deciding whether a new agent can join the existing system (see Agent-Joining Protocol)
- handling the departure of an existing agent (see Agent-Leaving Protocol)
- deciding whether an existing agent can commit knowledge update (see Knowledge-Update Protocol)
- coordinating agent collaboration in a Global Abductive Task, e.g., receiving queries and aggregating solutions (see ST and WT executions).

Agent-Joining Protocol

When a new agent wishes to join the system, it first needs to report to the system leader, and then synchronise its advertisements with others if it is allowed to join the system. Upon receiving a joining request, the leader makes two checks: it makes sure that the new agent alias does not clash with any existing agent's alias, and that the system is not too busy (i.e., `OngoingTasks > 0`) to accept the new agent. If the two checks are passed, the leader will inform the new agent, and initiates the Directory synchronisation between all the agents. The detailed protocol is shown in Figure 5.3

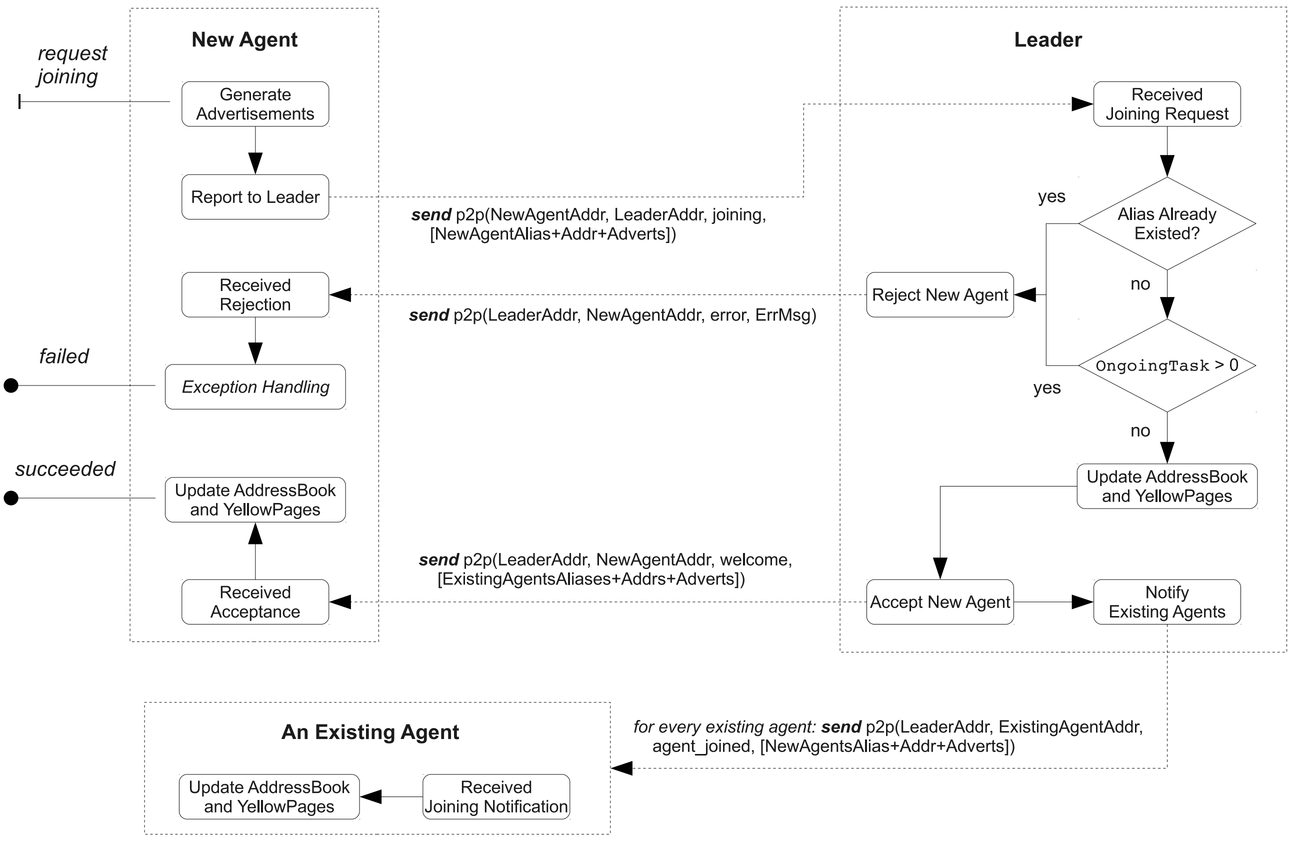
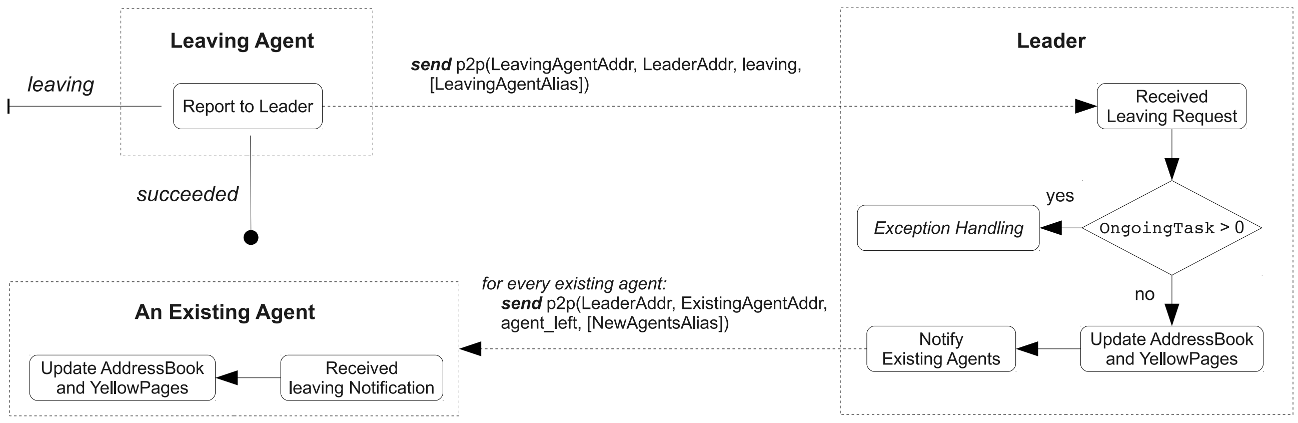

New Agent
request joining
Generate Advertisements
Report to Leader
Received Rejection
failed
Exception Handling
succeeded
Update AddressBook and YellowPages
Received Acceptance
Leader
Received Joining Request
yes
Alias Already Existed?
no
Reject New Agent
OngoingTask > 0
yes
no
Update AddressBook and YellowPages
Accept New Agent
Notify Existing Agents
send p2p(NewAgentAddr, LeaderAddr, joining, [NewAgentAlias+Addr+Adverts])
send p2p(LeaderAddr, NewAgentAddr, error, ErrMsg)
send p2p(LeaderAddr, NewAgentAddr, welcome, [ExistingAgentsAliases+Addrs+Adverts])
An Existing Agent
Update AddressBook and YellowPages
Received Joining Notification
for every existing agent: send p2p(LeaderAddr, ExistingAgentAddr, agent_joined, [NewAgentsAlias+Addr+Adverts])

Figure 5.3: New Agent Joining the System

Agent-Leaving Protocol

Similar to the agent-joining protocol, any existing agent wishing to leave the system needs to report to the leader. However, the leaving agent does not need approval from the leader, as its departure may not be controllable by the leader (or any other agent) in the system. Upon receiving the agent departure signal message, the leader needs to check whether the system is engaged in any global reasoning task. If this is the case, in order to guarantee the consistency of the answers (for the tasks), an exception handling procedure must be invoked to cancel any computation involving the leaving agent. In DAREC^2 a global abductive answer must be checked by all the agents in order to guarantee consistency, therefore all the agents will be involved at one time or another. Any computation of a global abductive task before the leaving agent actually gets involved does not need to be cancelled. However, identifying these computations can introduce computational overhead although it is easy to implement. In the current implementation, this procedure simply terminates any ongoing tasks, and informs the query issuer(s) about the computation failure due to changes to the system. The query issuer(s) can then decide whether to query again. The detailed protocol is shown in Figure 5.4.



Leaving Agent
leaving
Report to Leader
succeeded
send p2p(LeavingAgentAddr, LeaderAddr, leaving, [LeavingAgentAlias])
Leader
Received Leaving Request
yes
Exception Handling
OngoingTask > 0
no
Notify Existing Agents
Update AddressBook and YellowPages
for every existing agent: send p2p(LeaderAddr, ExistingAgentAddr, agent_left, [NewAgentsAlias])
An Existing Agent
Update AddressBook and YellowPages
Received leaving Notification

Figure 5.4: Existing Agent Leaving the System

It is worth pointing out that such an agent-leaving protocol can be used for implementing an agent failure (due to crashing or loss of communication connection) handling protocol:

- Each agent regularly *pings* others.
- If a no longer reachable non-leader agent is detected, the leader will be informed and will execute the agent-leaving protocol for the non-reachable agent.
- If the leader is no longer reachable (this is more serious), the leader election algorithm will be invoked (by the first agent who detects the leader failure), and the elected agent will execute the leaving protocol for the ex-leader.

Knowledge Update Protocol

Agents may update their local expertise at system runtime. However, such knowledge update must not occur when the system is engaged in any global task, in order to guarantee consistency.

Thus, each agent wishing to update its local expertise must first report to the leader, and the leader will check whether the system is busy. If there is no ongoing global task, then the leader will inform the requesting agent of the approval, so that it can commit the update. Note that the knowledge update request message may also contain the changes of the agent advertisements (e.g., what to add or delete). Thus, if the update request is approved, the leader will also inform all the agents to synchronise their Directory about the advertisement changes. The detailed protocol is shown in Figure 5.5.

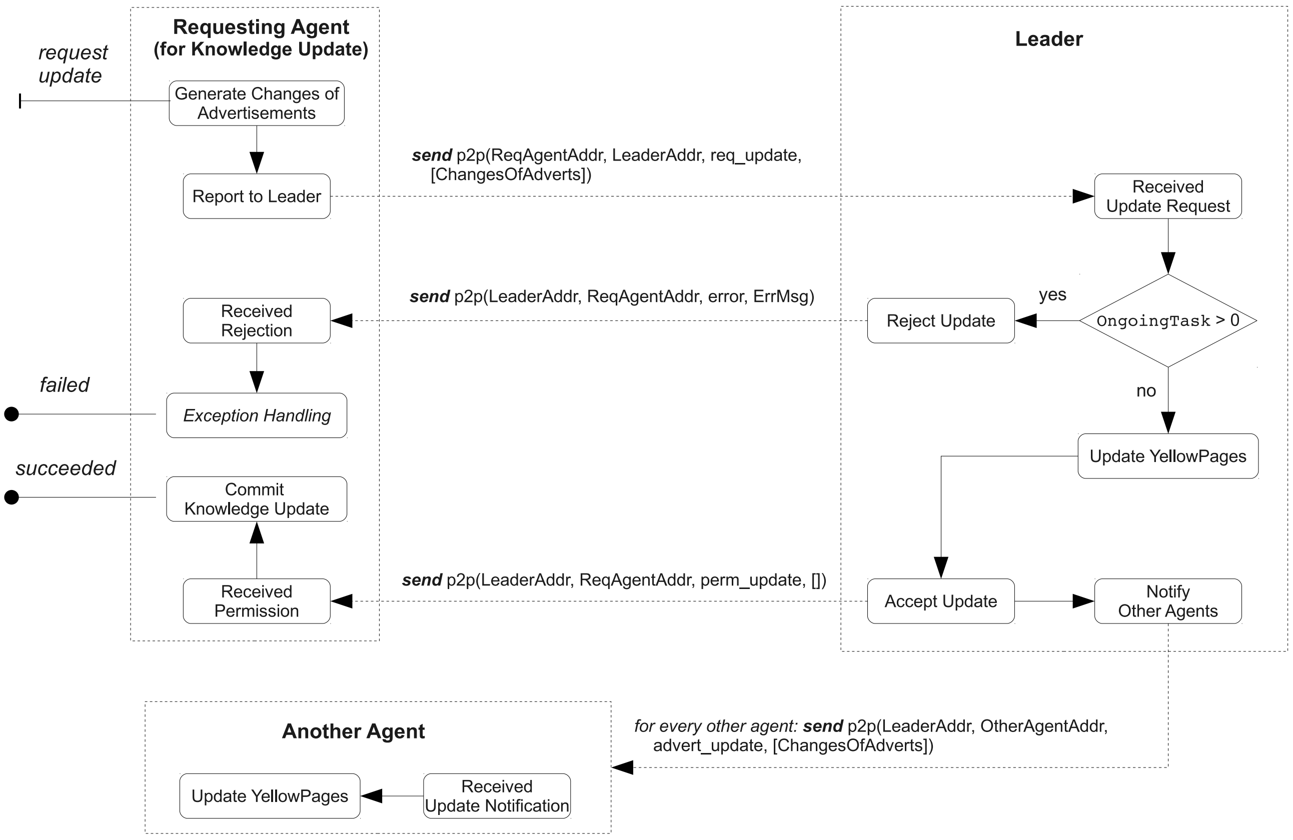

Requesting Agent (for Knowledge Update)
request update
Generate Changes of Advertisements
Report to Leader
send p2p(ReqAgentAddr, LeaderAddr, req_update, [ChangesOfAdverts])
Leader
Received Update Request
send p2p(LeaderAddr, ReqAgentAddr, error, ErrMsg)
Received Rejection
Reject Update
yes
OngoingTask > 0
no
failed
Exception Handling
succeeded
Commit Knowledge Update
Update YellowPages
send p2p(LeaderAddr, ReqAgentAddr, perm_update, [])
Received Permission
Accept Update
Notify Other Agents
Another Agent
for every other agent: send p2p(LeaderAddr, OtherAgentAddr, advert_update, [ChangesOfAdverts])
Update YellowPages
Received Update Notification

Figure 5.5: Update of Agent Knowledge

5.5.5 Executions of the Server Thread and the Worker Threads

Each agent has a single server thread (ST) and zero or more worker threads (WT). The ST is persistent along the lifetime of the agent, whereas each WT lives only during the collaboration of a global (abductive) task for a given query.

The ST has the following main responsibilities:

- During agent start up, it creates the initial local expertise of the agent (i.e., loading the topic and theory files), and reports to the (ST of the) leader agent to join the system.
- During agent shut down, it performs clean up (i.e., terminating any existing local abductive tasks), and reports to the (ST of the) leader agent to leave the system.
- At system runtime, it responds to different inter-agent messages from the joining, leaving and knowledge update protocols. Its behaviour depends on whether the agent is assuming the role of the leader.
- During global abductive tasks, it receives global queries (if the agent is the leader) and collaboration requests, and it creates and coordinates WTs for local computations.

The WT's responsibility is simply to perform local (abductive) tasks. Note that each global (abductive) task may consist of multiple local tasks performed by each agent, and the total number of such local tasks of all agents could be very big depending on the query and the agent local expertise. If each WT of an agent is responsible for a single local task, e.g., like in the implementation of DARE for maximising local concurrent computation, the agent may soon face a problem of running out of WTs (as each agent has limited computational resources). To address this problem, in the implementation of DAREC^2 each WT of an agent is allowed to engage in multiple local tasks at the same time, but at any time only one of them is active and all of them must belong to the same global task of a global query. Thus, instead of simply running an abductive meta-interpreter, each WT also needs to manage the different local tasks it owns in an efficient way, and to implement the token-controlled collaboration (with other WTs) described in Section 4.3.4.

Table 5.1 gives a summary of the inter-agent communication messages that are sent or received by the ST of a DAREC^2 agent.

The Life and Reactive Behaviour of the Server Thread (ST)

The ST of an agent maintains the following information, which is used for fulfilling its responsibilities:

Type	Used In	Sender/Receiver	Description
`leadership`	*Changing Leader*	from old leader (ST) to new leader (ST)	inform the agent to accept leadership
`update_leader`		from new leader (ST) to non-leader agents (ST)	inform other agents about the change of leader
`req_joining`	*Joining Protocol*	from new agent (ST) to leader (ST)	request for joining the system
`den_joining`		from leader (ST) to new agent (ST)	joining request denied
`perm_joining`		from leader (ST) to new agent (ST)	joining request permitted
`agent_joined`		from leader (ST) to existing agents (ST)	inform existing agents about the new agent
`leaving`	*Leaving Protocol*	from leaving agent (ST) to leader (ST)	inform the leader about its departure
`agent_left`		from leader (ST) to remaining agents (ST)	inform remaining agents about the gone agent
`req_update`	*Update Protocol*	from knowledge update agent (ST) to leader (ST)	request for knowledge update
`den_update`		from leader (ST) to knowledge update agent (ST)	update request denied
`perm_update`		from leader (ST) to knowledge update agent (ST)	update request permitted
`query`	*Collaboration*	from (external) query issuer to leader (ST)	send a query to the leader
`trans_state`		from one agent (WT) to another agent (ST)	state transfer between two agents
`solved_state`		from one agent (WT) to another agent (ST)	centralise solved states to the leader
`eos`		from one agent (WT) to another agent (ST)	token-control signal
`answer`		from leader (ST) to query issuer	forward solved states to the query issuer
`next`		from (external) query issuer to leader (ST) or from leader (ST) to an agent (ST) that previously sent a solved state	demand more answers
`discard`		from (external) query issuer to leader (ST) or from leader (ST) to other agents (ST)	abandon all remaining computations for the query
`abort`		from leader (ST) to another agent (ST)	terminate all local computations

Table 5.1: Summary of the Inter-agent Communication Message Types

- If the agent is assuming the role of the leader, then its ST maintains the counter `OngoingTasks`, which is accessible by the ST only.

- If the agent is assuming the role of the leader, then the agent is responsible for receiving global queries and creating the global tasks. Our implementation allows the system (or the leader) to accept queries even when the system is engaging in some ongoing global tasks. Thus, being able to track local computations for different global tasks is necessary for the collaboration and coordination. In our implementation, for each query received, the leader will generate a unique (numerical) ID for it and its global task. This query ID is also associated to every agent's local computations that belong to the global task. Thus, the ST of the leader maintains the mapping of received queries and the query IDs, as a set of tuples $\langle QueryID, Query, IssuerAddr \rangle$.

- If the agent is assuming the role of the leader, then the agent is also responsible for forwarding the solved states found by any agent in the system. If the query issuer demands more solutions, then the leader has to inform the agent who has returned the last solved state to continue the search (i.e., returning a token to that agent). Thus, the leader also maintains the tuples $\langle QueryID, TokenWaitingAgentAddr \rangle$.

- Regardless of whether the agent is the leader or not, its ST is responsible for controlling a set of WTs that perform local computations. Each WT has its unique thread ID, and owns one or more local tasks for the same global task. Thus, the ST records a set of tuples $\langle WTID, QueryID \rangle$ that can be used for keeping track of local computations within the agent.

The execution of the ST can be seen as a reactive process, which can be described as the flow-chart given in Figure 5.6.

The Life and Operational Behaviour of a Worker Thread

A WT is created (by the ST) when the agent starts to participate in the global task for a given query, and is terminated either when it has done all of its computation, or when the

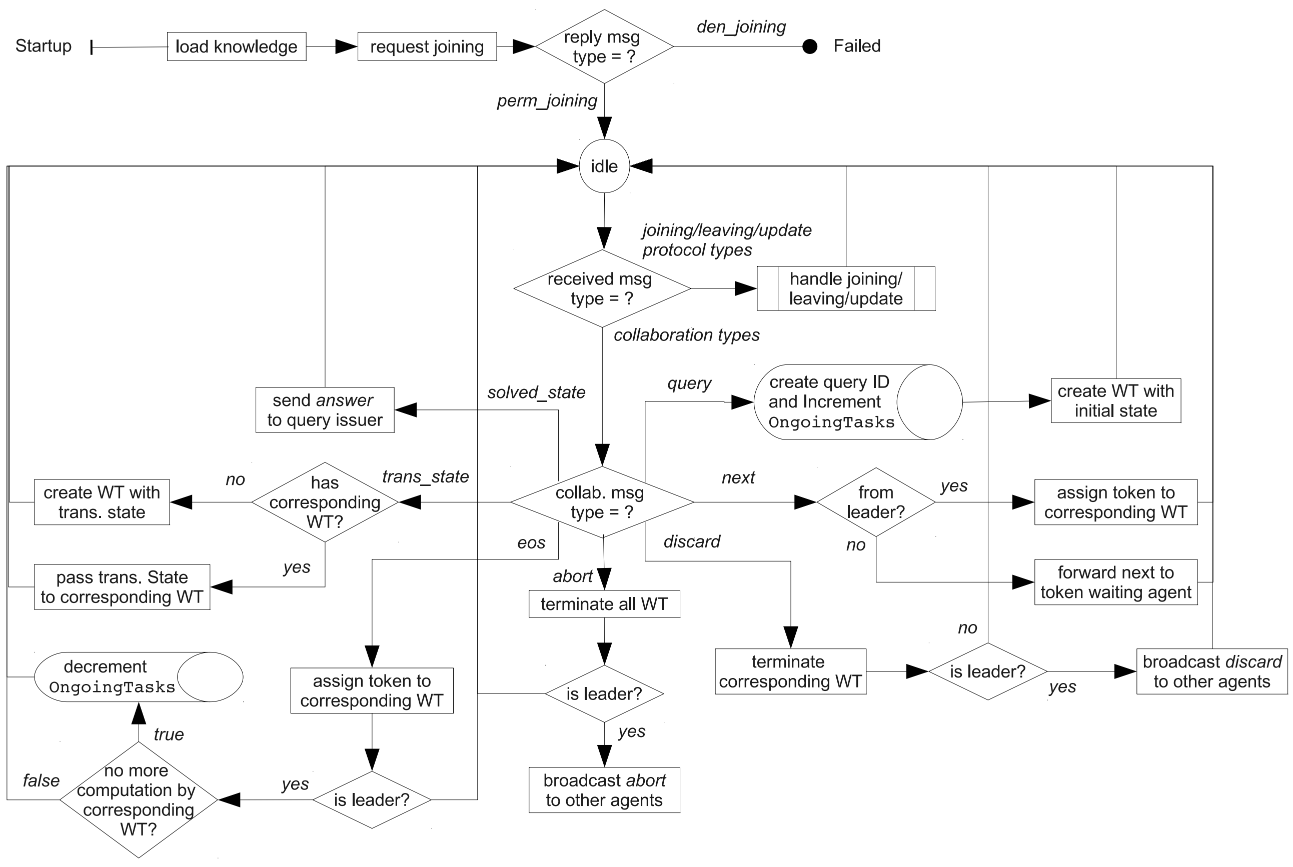

Startup
load knowledge
request joining
reply msg type = ?
den_joining
Failed
perm_joining
idle
joining/leaving/update protocol types
received msg type = ?
handle joining/ leaving/update
collaboration types
send *answer* to query issuer
solved_state
query
create query ID and Increment OngoingTasks
create WT with initial state
no
has corresponding WT?
trans_state
collab. msg type = ?
next
from leader?
yes
assign token to corresponding WT
create WT with trans. state
eos
discard
no
forward next to token waiting agent
yes
pass trans. State to corresponding WT
abort
terminate all WT
no
decrement OngoingTasks
assign token to corresponding WT
terminate corresponding WT
is leader?
yes
broadcast *discard* to other agents
is leader?
true
yes
false
no more computation by corresponding WT?
yes
is leader?
broadcast *abort* to other agents

Figure 5.6: Execution Flowchart of ST. *Note that all the collaboration messages except* `query` *contain the query ID, and the ST of each agent can use it to identify its corresponding WT.*

global task ends (i.e., the ST receives `discard` or `abort` for the query). The execution of a WT can be seen as the running of an abductive meta-interpreter, which is interruptible by the ST. The interactions between a WT and its ST are through inter-thread messages of the format $(Type, Content)$, which are summarised in Table 5.2. In addition, every WT can send three types of collaboration inter-agent messages (i.e., `trans_state`, `solved_state` and `eos`) to the ST of other agent, during collaboration to implement the token-control protocol.

As aforementioned, in order to avoid the explosion of WT creation for (the global task of) a query, each WT in an agent is responsible for all the local tasks for a query, i.e., it can be engaged in several local tasks at the same time, but only one of the local tasks can be active at any time. During the active local task, it is possible that a generated outgoing transferable or solved state cannot be sent immediately, i.e., due to waiting of the token. Even when the active local task is finished, the reply message `eos` may not be sent immediately for the same

Type	**Content**	**Description**
`init`	*State*	signal a newly created WT to start a local task with the given state
`push`	*State*	interrupt the corresponding WT and urge it to start a new local task with the given state
`token`	*none*	assign the token to the corresponding WT, and hence signal it to send the next buffered state or reply, if there is any
`kill`	*none*	signal the WT to free up resources and terminate

Table 5.2: Summary of the Inter-thread Communication Message Types from ST to its WT

reason. Thus, these states and the reply need to be stored in the BufferStore component of the agent. Furthermore, when the computation of the currently active local task is finished, a *proactive* WT (as currently implemented) continues to work on the next local task even though there are still some buffered outgoing states and/or reply of the current task. In addition, a *simple* WT (as currently implemented) without clever task scheduling algorithms always treats the new local task as the most urgent, i.e., it will suspend the currently active local task and start to work on the new task. Therefore, the main challenge of implementing the WT, apart from the interruptible meta-interpreter, is to manage resources (e.g., memory) correctly and efficiently between the switching of local tasks.

The Meta-Interpreter: Prolog has a powerful backtracking mechanism, which allows logic inference to be performed in a very efficient depth-first search way. Thus, we have implemented an abductive meta-interpreter that deploys a depth-first search for the computation of a local abductive task. The pseudo-code of the meta-interpreter is very close to the one described in Figure 4.2 (the `process_state` procedure), with only the following differences:

- The solved states are sent to the leader instead of the query issuer. If the WT does not own a token, then the state will be buffered first (i.e., implementing the *pull* style answering mechanism).
- It uses the *lazy* agent interaction strategy by default.
- It uses the *uniform* agent selection strategy for DAREC^2 by default.

- It uses a simple left-to-right secure goal selection strategy.
- It uses the set of DAREC^2 local inference rules.
- In addition to the standard components of a DAREC^2 computational state, the actual *state* processed by the meta-interpreter contains two more pieces of information: the *original query* and the *query ID*. The former is used for answer extraction in the case of a solved state (i.e., bindings for the variables appearing in the original query), and the latter is used to identify which query (or its global task) the current state belongs to.

Note that the *eager* agent interaction strategy and a *heuristic* agent selection strategy that is based on the *helpfulness* of candidate agents[3] are also implemented for the purposes of benchmarking. There are options to switch between these strategies for the system. Note also that during a local computation, the meta-interpreter uses the existing Prolog libraries $CLP(\mathcal{FD})$ and $CLP(\mathcal{R})$ for solving CLP constraints, and uses our implemented *inequality solver* for solving collected inequalities. The source code of the inequality solver is given in the Appendix A.1.

Usage of the State Buffer Each WT has a reserved space in the State Buffer for keeping its buffered transferable/solved states and the buffered EOS replies for its finished local tasks. While the transferable states and the EOS replies must be sent in the order they were generated during a local task, the solved states (from any local task) should be sent as soon as the WT receives a token, in order to reduce the time taken for the query issuer to wait for an answer. When the WT receives a new local task, its current computation will be suspended – intermediate execution data (e.g., backtracking data used by the depth-first meta-interpreter) and the currently buffered transferable states and EOS replies are backed up – and it will start another computation of the local task. After this local task is finished, a previously suspended computation will be restored so that the WT can continue to work on it. Thus, the space used

[3]The *helpfulness* of a candidate state recipient agent is the sum of the number of abducibles in the state that the agent needs to check, and number of the delayed goals in the state that the agent can potentially help to solve. Thus, the heuristic agent selection strategy will select one of the candidate agents that have the biggest helpfulness, instead of depending on the lexicographical ordering of the agent aliases.

by each WT in the State Buffer has three components (see Figure 5.7): the *Solutions Store* (SS), the *Current Workbench* (CW) and the *Suspended Workbenches* (SW).

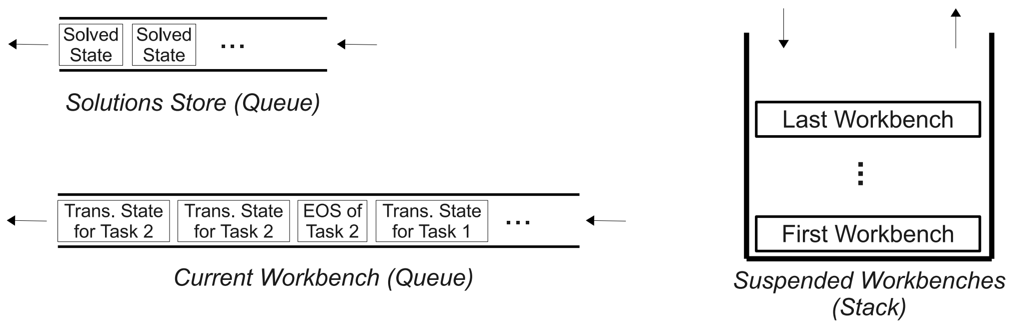


Figure 5.7: Usage of the State Buffer by a WT

The SS is a simple first-in-first-out (FIFO) queue, buffering the solved states that are computed by the local tasks performed by the current WT and are waiting to be sent back to the leader (upon receiving the `token` message by the current WT from the ST). The CW is also a FIFO queue, but it stores two things – transferable states and EOS replies. It is possible that these states and replies are generated by different local tasks, but those belonging to the same local task are always buffered as a sequence in the queue, with the EOS reply being the last element of the sequence. The SW is a stack (i.e., first-in-last-out) of suspended workbenches. Intuitively, while a WT is performing a local task, any generated transferable state is first buffered on the CW. If the WT receives a new local task, then it will save the CW to the SW, and use a new workbench to buffer any new transferable state. If the current task is finished, a EOS will be added to the CW (e.g., the third element in the current workbench queue in Figure 5.7), and if there are some suspended workbenches in SW, the last one will be popped from the SW and merged with the CW (i.e., appending all its elements to the CW. E.g., the fourth element in the current workbench queue in Figure 5.7), and the WT can then resume its previously suspended local task. The state chart diagram of the life cycle of a WT is given in Figure 5.8, and the flow charts of the execution of each state are given in Figures 5.9.

Note that the *previous agent* (which is sent an EOS reply in Figure 5.9c) of a local task being owned by the *current agent*, is the owner agent of the transferable state which has been received by current agent and used as the root state of the local task. Note also that (in Figure 5.9c) after a WT in the leader agent informs its ST about the end of a global task, the ST will

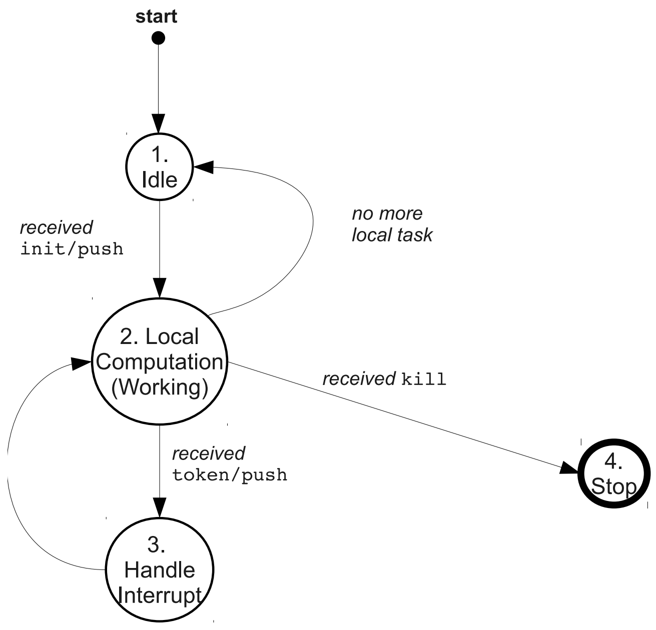


Figure 5.8: State Chart Diagram of WT's Life Cycle

decrement the `OngoingTasks` counter.

5.6 Related Work

5.6.1 Centralised Abductive Systems

In addition to the Kakas-Mancarella proof procedure (KM) [KM90b] and ASystem [ANDB01, KvND01, NK01], other well-known top-down logic programming based abductive proof procedures and systems include SLDNFA [DS92, DS98], IFF [FK97], ACLP [KM95, KM97, KMM00], CIFF [EMS+04b, EMS+04a, MTS+09] and $\mathcal{S}$CIFF [AGL+05, AC05, ACG+08, GAL09].

We chose to extend ASystem into the DAREC (and DAREC^2) distributed abductive algorithm due to the following considerations. Both KM and ACLP rely on the interleaving of *abductive* and *consistency* derivations, which would make the co-operation strategy between agents less flexible as discussed in our presentation of DARE (see Chapter 3). IFF, CIFF and $\mathcal{S}$CIFF, on the other hand, use a special IFF-theory for representing the abductive logic framework – the program consists of the if-and-only-if definitions (i.e., the completion of the rules [Cla78]) for predicates. In a multi-agent system, since the rules cannot be centralised, it is not possible to complete them to obtain such an IFF-theory. Although ASystem extends SLDNFA and incorporates several ideas from IFF and ACLP, such as performing the abductive inference as a state rewriting process and using an independent solver for CLP constraints during the inference, it does not inherit the above mentioned shortcomings.

Abductive reasoning can also be performed using bottom-up *answer set programming* (ASP) based algorithms such as SModels [NS97, SN01], DLV [LPF+06] and Clasp [GKNS07]. These systems represent abductive problems as *disjunctive logic programs* (i.e., in which rules can have disjunction of atoms as the head). The computation of abductive answers is done in two steps: first, the logic program is grounded (with some preprocessor), and then the stable models of the ground program are computed. Abductive answers, i.e. sets of ground abducibles, are extracted from the computed stable models.

Although bottom-up algorithms are generally accepted to be more efficient than top-down algorithms, and do not suffer from "loops" caused by cyclic logic programs, they cannot work with logic programs with unbound domains as the logic program cannot always be grounded before the model computation. Furthermore, in a multi-agent system since the rules cannot be centralised, the overall program cannot be grounded. These issues present new challenges in extending centralised bottom-up algorithms for multi-agent abduction, and no solution has been proposed yet. Note that the distributed ASP system in [EGG+09] focuses on distributing the model computation (i.e., parallel computation) of centralised knowledge to improve performance instead of the computation over distributed knowledge, which is the main focus of this thesis.

5.6.2 Distributed Abductive Systems

ALIAS [CLM+03] is the only other multi-agent abductive system that is closely related to DAREC (and DAREC^2). However, the two systems have many significant differences. First, ALIAS extends KM to the multi-agent context, and, because of the limiting features of KM, it cannot handle non-ground (negative) queries, non-ground abducibles and CLP constraints. Secondly, in ALIAS the knowledge base of agents uses a special language called *LAILA* [CLMT01] for specifying statically and a priori, the communications with other agents. Finally, consistency of the abduced assumptions is only required locally (i.e., with respect to each agent's knowledge base). On the other hand, DAREC (and DAREC^2) can accept non-ground (negative) queries, compute non-ground answers and support reasoning over CLP constraints. In DAREC (and DAREC^2) the collaboration among agents is dynamically defined, by means of the *yellow page* directory that allows an agent to dynamically identify other helper agents. The notion of consistency of the abduced assumptions is global with respect to the overall knowledge base of all the agents (i.e., *global consistency*) in DAREC (and DAREC^2). In the case of DAREC^2, shared and private knowledge can be distinguished in knowledge specification, and confidentiality is maintained. Also, the communications between agents may be guided by the reasoning results of the ID arguments in askable goals during distributed inference.

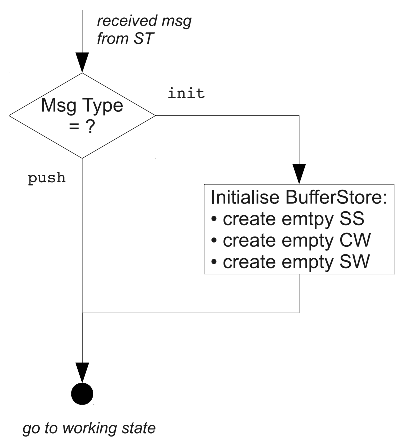
Another (less) related multi-agent abductive system is MARS proposed by Bourgne et. al. [BIM10]. In MARS, each agent has a local abductive task and it needs to refine its local hypotheses to be consistent with respect to all other agents' knowledge. Different to DAREC (and DAREC^2), agents in MARS may not be fully connected and they can only communicate with their neighbours. In addition, each agent's background knowledge is presented as a causal theory, and local abduction is done through inverse entailment with a consequence finding algorithm [NII03]. Agents refine their hypotheses through learner-critic style interactions with their neighbours: 1. The learner agent computes all the new consequences (called the *context*) entailed by its background knowledge and a proposed hypothesis, and passes them to all of its neighbours (the critics). 2. Each critic agent then checks the received context for consistency with respect to its own background knowledge. If an inconsistency is found, the learner agent will be informed and the negation of the proposed hypothesis will be assumed (by the learner agent). Otherwise, the critic agent will compute all the new consequences entailed by the received context and its background knowledge, and propagate them (with the received context) to its neighbours, who then act as the new critics. 3. Such a recursive chain of interactions terminates (successfully) when no new consequences can be found by all the critics, in which case the learner agent can formally assume the proposed hypothesis.

5.6.3 Speculative Multi-agent Reasoning Systems

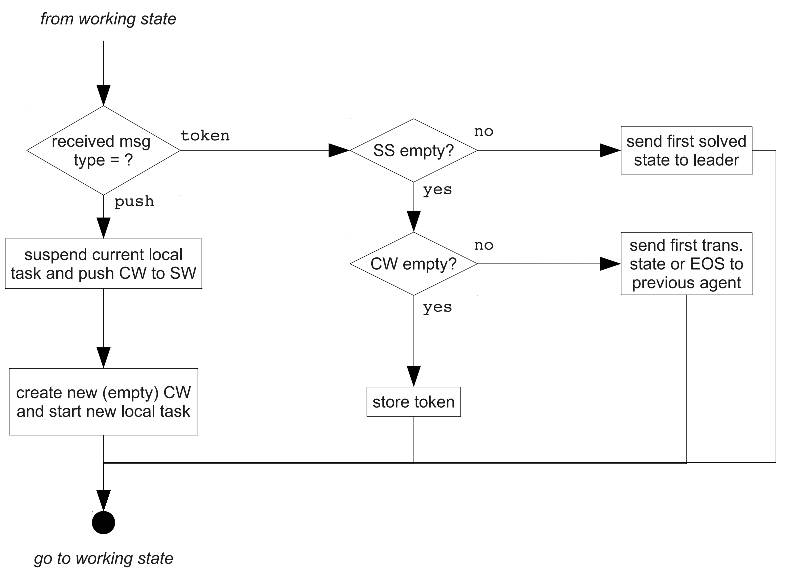
Speculative reasoning is another example of abduction-based multi-agent reasoning systems. It was first proposed by Satoh [SIIS00] to address two important issues arising during the query-answer interactions between logic programming based agents (each of which has its knowledge as a logic program and computes answers for received queries with a top-down inference algorithm). First, answers may not be returned in timely fashion due to either the physical communication channel delays or the long computation process by the queried agent. Secondly, the queried agent may not return all the answers at once (e.g., an answer is returned as soon as it is found), and it may also need to revise its previous answer that has already been sent to the querying agent during the collaboration. In a speculative computation system, after a goal is sent out as an external query, the sender agent may use some *default answers* of the goal to continue its local inference, and hence prevent the agent from wasting idle CPU cycles while waiting for a returned answer. When a new or revised answer is returned, the query sender agent may reuse previous computation as much as possible by discarding only the part of the computation that was done using old and inconsistent answers for the query, and by revising the part of the computation that was done using old but consistent answers to use the returned answer. Abduction is used during local inference to support the *answer revision* process, e.g., answers used for sent goals are treated as assumptions. The first such system [SIIS00] was developed for master-slave structured systems (i.e., there is only one agent that can send queries) where an answer to a (ground) query is simply either *yes* or *no*. Since then it has been extended for hierarchical MAS [SY02] (i.e., each agent can have only one parent and can send queries to its children only), and for constraint processing [SCH03, HSM$^+$10] where the answers can be a set of constraints over the variables in the (non-ground) query, and finally for hierarchical MAS to support negation and non-ground answers with constraints [MBG$^+$10]. In all these system, goals that can be sent out as queries are called *askable goals* and have the form $A@S$ where A is an atom and S is an agent identifier. Their operational meaning is very similar to that of the askable atoms in DAREC^2. However, whereas in speculative computation when an askable goal is selected, the agent identifier must be ground, in DAREC^2 it can also be a variable with quantifier. Furthermore, in speculative computation (and in DARE) agent interactions are query answering, whereas in DAREC (and DAREC^2) they are state passing.

5.7 Conclusion

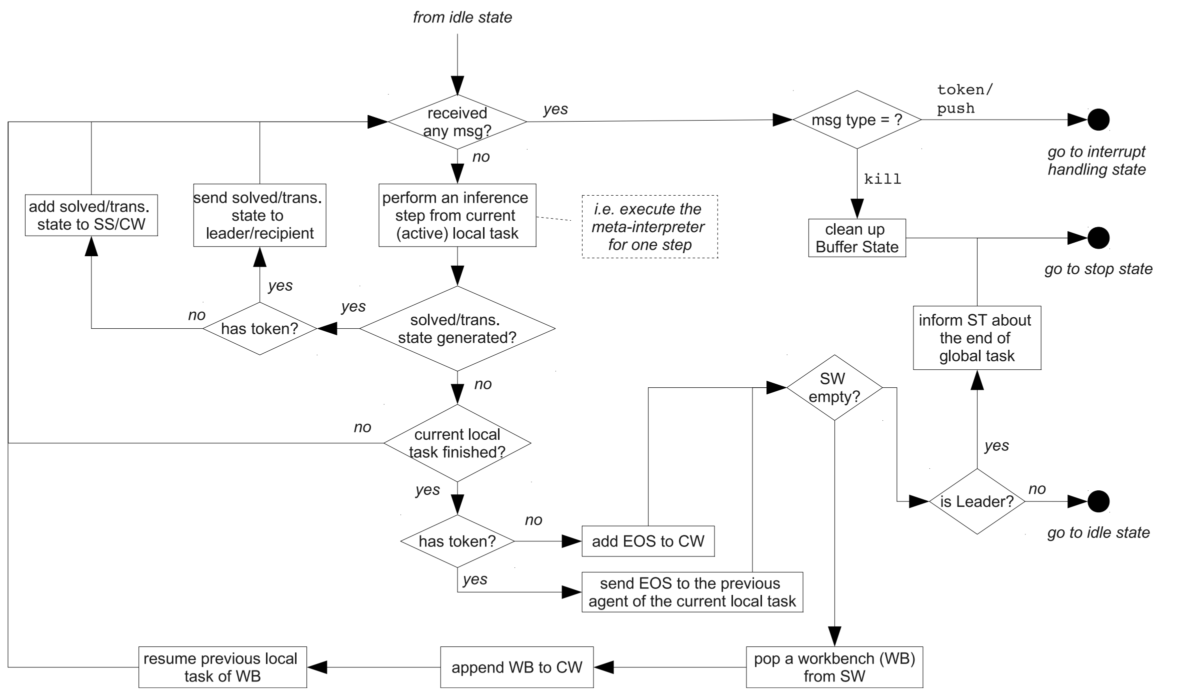
In this chapter we have presented the DAREC^2 system for multi-agent abductive reasoning with confidentiality. In DAREC^2 , non-abducible predicates are divided into two sets: *public* and *local*. Thus, agents can specify their confidential knowledge with *private atoms* (with public predicate) and their shared knowledge with *askable atoms* (with local predicate). The DAREC^2 distributed abductive algorithm is built on from that of DAREC. Optimisations and small modifications are made to the local inference rules in order to handle askable (atom) goals and private (atom) goals correctly and efficiently. The new algorithm can guarantee *confidential reasoning* through the use of the *secure safe goal selection strategy* and the *lazy agent interaction strategy*. A prototype of the system has been implemented in Prolog. In Chapter 6 we will describe our experimental results of the system prototype, and in Chapter 7 we will provide a case study of using DAREC^2 for a real world application – distributed policy analysis.



(a) Execution Flow Chart for the Idle State of WT


from working state
received msg type = ?
token
push
SS empty?
no
yes
send first solved state to leader
suspend current local task and push CW to SW
CW empty?
no
yes
send first trans. state or EOS to previous agent
create new (empty) CW and start new local task
store token
go to working state

(b) Execution Flow Chart for the Interrupt Handling State of WT


from idle state
received any msg?
yes
no
msg type = ?
token/ push
go to interrupt handling state
kill
clean up Buffer State
go to stop state
add solved/trans. state to SS/CW
send solved/trans. state to leader/recipient
perform an inference step from current (active) local task
i.e. execute the meta-interpreter for one step
yes
no
has token?
yes
solved/trans. state generated?
no
inform ST about the end of global task
SW empty?
no
current local task finished?
yes
yes
has token?
no
is Leader?
no
go to idle state
add EOS to CW
yes
send EOS to the previous agent of the current local task
resume previous local task of WB
append WB to CW
pop a workbench (WB) from SW

(c) Execution Flow Chart for the Working State of WT

Figure 5.9: Execution Flow Charts of WT

Chapter 6

Experiments and Benchmarking

In order to test our DAREC^2 system prototype and study its performance, collections of distributed logic programs (LPs) of various sizes and structures are needed. Each collection represents the agents' knowledge of a DAREC^2 framework. However, it is always difficult to find testing LPs required for specific experiments. Therefore, we have developed an experimental environment for DAREC^2. This environment can randomly generate distributed abductive LPs according to a list of parameters such as the number of agents, and the size and structure of their LPs. It also provides a facility that supports the auto-execution of the system for each experiment. Each auto-execution consists of generating input knowledge files, launching DAREC^2 agents, submitting queries and collecting answers. During the execution of DAREC^2, profiling information, such as execution time and communication cost, are collected to allow us to investigate the scalability of the system. It is worth pointing out that our environment is general enough for generating centralised LPs. Therefore, it can also be used to provide testing data sets for systems such as ACLP and ASystem.

The rest of this chapter is organised as follows. Section 6.1 describes how the distributed LP generator works and how different types of LPs can be produced. Section 6.2 describes the auto-testing structure of our environment and discusses some of the DAREC^2 experimental results. Section 6.3 concludes this chapter.

6.1 A Random Generator for Distributed Abductive Constraint Logic Programs (GenDACLP)

A general-purpose generator for distributed abductive constraint logic programs, called GenDACLP, has been developed in Java. Although arbitrary logic programs (LPs) can be randomly produced by the generator according to a set of tunable parameters, the LPs that are useful for our experiments are those where the distributed computation (by the DAREC^2 agents) always terminates and never flounders.

In order to guarantee flounder-free for a DAREC^2 computation, the union of the distributed LP generated for each agent must satisfy the *allowedness condition* [DC89] – any variable appearing in a rule must also appear either in the head or in a positive body literal. On the other hand, in order to guarantee termination for a distributed abductive task, both the query and the union of all agents' LPs need to satisfy the *abductive non-recursive* property [Ver99]. However, since the LPs are generated independently from the queries, in practice it is not feasible to check this property for arbitrarily generated LPs with respect to all the possible queries. In fact, as shown in [Ver99] the sole cause of non-terminating derivations is repeated reductions of the same abducible constraint. For example, suppose $p(X) \leftarrow a(f(X))$ is a rule in a LP where $a/1$ is the only abducible predicate, and suppose that during the inference (by SLDFNA, ASystem or DAREC^2) for some query, an abducible constraint $\forall X. \leftarrow a(X), \neg p(X)$ is collected. Then if at some point later in its derivation an abducible $a(X)$ is assumed, the checking of the abducible constraint will generate a new goal $p(X)$, whose further reduction will cause $a(f(X))$ to be assumed and the same abducible constraint be checked again, i.e., the inference will not terminate and keep assuming new abducibles $a(X), a(f(X)), a(f(f(X)), \ldots$.

An easy and practical way to prevent such situation from arising is to enforce that in the Herbrand Base of the LP the set of all abducible atoms is always finite. The current implementation of GenDACLP simply ensures that in the generated LPs:

- all constants are integers, and

- there is no function symbol.

Thus, every generated LP will satisfy the abductive non-recursive property and DAREC^2 will terminate given any (bounded) query [Ver99] (so will SLDNFA and ASystem). Another advantage of generating LPs with such conditions is that the experiment environment can also be used for other theorem provers employing top-down or bottom up algorithms.

6.1.1 The Input and Output of the GenDACLP

The set of tunable parameters that the generator can accept is summarised in Table 6.1 (the *overall program* is the union of all the agent's LPs):

Table 6.1: List of Input Parameters for GenDACLP

Key Name	**Value Type**	**Description**
`randomSeed`	positive `int`	a seed to the random number generator (i.e., same seed will guarantee same generated logic programs, provided that other parameters remain unchanged).
`progStruct`	`String`	currently the possible values are: • `stratified`: the overall program is stratified; • `acyclic`: the overall program is acyclic; • `arbitrary`: the overall program is randomly generated and does not necessarily satisfy either of the above properties.

`numAbduciblePredicates`	non-negative `int`	number of abducible predicates in the overall program
`numPublicPredicates`	non-negative `int`	number of public predicates (i.e., used for askable atoms) in the overall program
`numLocalPredicates`	non-negative `int`	number of local predicates (i.e., used for private atoms) per agent
`numConstants`	positive `int`	number of constants (enumerated as natural numbers) in the overall program (i.e., they constitute the Herbrand Universe)
`numAgents`	positive `int`	number of agents
`numAskableRules`	non-negative `int`	number of rules defining askable atoms
`numPrivateRules`	non-negative `int`	number of rules defining private atoms
`numICRules`	non-negative `int`	number of integrity constraints (i.e., written as rules whose heads are `ic`)
`abducibleICs`	`boolean`	`true` if each integrity constraint to be generated contains at least one positive abducible
`avgRuleBodySize`	positive `int`	average number of body literals in each rule (applies also to the integrity constraints)
`avgArgumentSize`	positive `int`	average number of arguments of each predicate (applies to abducible, public and local predicates)
`negationBias`	`float` within $[0,1]$	average $\frac{numNegLits}{totalBodyLits}$ of all the rules: with the value 0 (1), all the body literals will be positive (negative).
`abducibleBias`	`float` within $[0,1]$	average $\frac{numAbducibleLits}{totalBodyLits}$ of all the rules

`askableBias`	`float` within $[0,1]$	average $\frac{numAskableLits}{totalBodyLits - numAbducibleLits}$ of all the rules
`constraintBias`	`float` within $[0,1]$	average $\frac{numClpConstraints}{numPrivateLits + numClpConstraints}$ of all the rules
`variableBias`	`float` within $[0,1]$	average $\frac{numVariables}{numArguments}$ of all the atoms
`variableCoupling`	`float` within $[0,1]$	average $1 - \frac{totalVariables}{totalPostivePrivateLitArguments}$ of all the rules

With these parameters, different type of LPs with specific structure (e.g., stratified and acyclic) can be generated, including and not limited to the following:

- *ground* LPs can be generated by setting `variableBias = 0`;
- *definite* LPs can be generated by setting `negationBias = 0`;
- *constraint* LPs can be generated by setting `constraintBias` to some positive value;
- *abductive* LPs can be generated by setting `abduciblePredicate` to some positive value;
- *distributed* LPs can be generated by setting `numAgents` to some positive value greater than 1.

All the generated LPs satisfy the allowedness conditions. In order to guarantee that the generated distributed LPs are suitable for the testing and benchmarking of our DAREC^2 system prototype, some of the parameters must satisfy the following conditions:

- `progStruct` is `acyclic` (in order to guarantee termination);
- `abducibleICs` is `true` (as this is one of the requirements of DAREC^2).

Running GenDACLP

The generator is implemented in Java. The values for the tunable parameters are specified in a *knowledge configuration* file, where each line has the format of `Key=Value`, as the input for the generator. An example is given in Figure 6.1.

```
randomSeed = 23421
progStruct = acyclic
numAbduciblePredicates = 2
numPublicPredicates = 3
numLocalPredicates = 5
numConstants = 20
numAgents = 3
numAskableRules = 18
numPrivateRules = 30
numICRules = 6
abducibleICs = true
avgRuleBodySize = 4
avgArgumentSize = 2
negationBias = 0.3
abducibleBias = 0.2
askableBias = 0.2
constraintBias = 0.2
variableBias = 0.7
variableCoupling = 0.7
```

Figure 6.1: Example Configuration File for GenDACLP (`sample.config`)

Given a knowledge configuration (e.g., `sample.config`), the initial output of GenDACLP is a Prolog file (e.g., `total.pl` in Figure 6.2) containing a set of abducible declarations (e.g., lines 2–3 in `total.pl`) and all the clauses for the specified number of agents (e.g., the rest of `total.pl`). In the output file, agent identifiers are integers starting from 0, and the clauses are partitioned for the agents. For example, `sample.config` specifies three agents, and hence in `total.pl` the clauses are divided into three parts: lines 5–21, lines 23–39 and lines 41–60. Each part contains the randomly generated integrity constraints (e.g., lines 5–6), private rules (e.g., lines 7–15) and askable rules (e.g., 16–21) in order. Note that each private atom has a predicate name starting with l and ending with $\_n$ where n is the owner agent's identifier. To produce input files for DAREC^2, GenDACLP can further "slice" `total.pl` into four files, where the first file (i.e., the topic file) contains only the abducible declarations, and each of the other three files contains the clauses for an agent (i.e., the theory file).

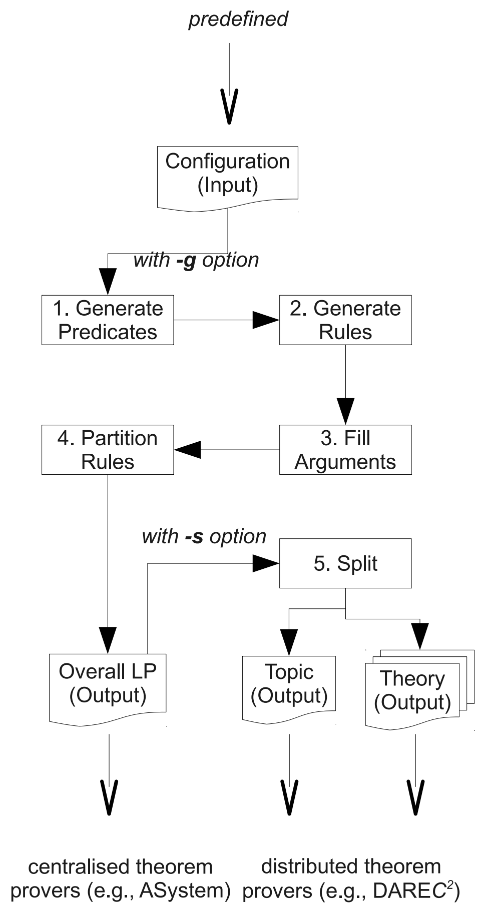
It is easy to see that `total.pl` is the union of all agents' background knowledge and integrity constraints. Therefore, it can be used by some centralised theorem prover (e.g., ASystem), while the topic file and the theory files are used by the agents of DAREC^2. The answers computed by the two systems are expected to be the same.

```
% Topic File: 2
abducible(a1(_, _, _)).
abducible(a2(_, _)).
% Agent 0: 17
ic :- a2(15, 15).
ic :- a2(X0, X0), l3_0(X0, 7, X0), l1_0(X0).
l1_0(0) :- a2(1, 13), a2(14, 8).
l1_0(3) :- a2(10, 17), p3(18, 19)@2, \+ a2(17, 18).
l2_0 :- l4_0(X1, 15, X1), l4_0(X1, 15, 2), \+ l4_0(18, X1, X1), l4_0(X1, X1, X1).
l2_0 :- l4_0(X2, 4, X0), l4_0(4, 4, 0), l4_0(0, X1, 6), l4_0(X0, X2, X2), X2 #=< X0.
l3_0(6, 0, 14).
l4_0(14, 4, 19).
l5_0 :- \+ a1(11, X0, X0), l4_0(X0, X0, X0), \+ l4_0(13, 1, X0).
l5_0 :- a1(11, 19, 6), l2_0, \+ l4_0(7, 7, 1).
l5_0 :- l4_0(X0, X0, X0), a1(X0, X0, X0), p3(X0, X0)@X0.
p1(0, 16)@0 :- l2_0, a1(15, 1, 1), a2(10, 3), a1(17, 4, 0).
p1(X0, X0)@0 :- X0 in 9..18, l2_0.
p1(X1, X1)@0 :- l4_0(X0, X1, X0), l4_0(4, X0, X1).
p2(11)@0 :- p1(16, 2)@0, l2_0, p1(13, 14)@0, a1(8, 15, 6).
p3(18, X0)@0 :- a1(X0, X0, 9), a1(X0, X0, 3), l4_0(X0, 15, 12), X0 #=< X0, 11 #=< X0.
p3(3, 17)@0 :- \+ a1(15, 12, 8).
% Agent 1: 17
ic :- a1(16, 1, 6).
ic :- a1(X0, 9, 14), \+ l3_1(X0, 2, 0), a1(X0, 17, 1), a1(14, X0, 3), X0 #< X0, l3_1(1, X0, X0), \+ l1_1(X0).
ic :- a1(X0, X0, 16), p3(2, X0)@X0, l5_1, l1_1(X0), X0 in 1..2.
l1_1(7) :- l5_1, 10 #< 0, a2(16, 15), \+ l2_1.
l2_1 :- \+ l4_1(X1, X1, X1), l4_1(X0, X0, X1), l4_1(X0, X1, X1), a1(13, 14, X0), l4_1(X1, X0, X1), X1 #=< X0.
l2_1.
l3_1(16, 2, 14) :- l2_1, 5 #< 14, 9 #=< 4, \+ p1(9, 9)@1, 16 #< 9, 18 #=< 0, 8 #=< 18.
l4_1(3, 0, 1).
l4_1(6, 15, 5).
l5_1 :- X0 in 2..2, a1(3, X0, X0), p3(4, X0)@X0, l4_1(X0, 6, 1).
l5_1 :- \+ l2_1, l4_1(13, X0, 13), a1(X0, 3, X0), l2_1, l4_1(X0, X0, X1), p1(4, X1)@X1.
l5_1 :- l2_1, l4_1(X0, X0, X0), \+ l4_1(X0, 8, 1).
l5_1 :- p3(14, 1)@2, \+ l4_1(12, 8, 14).
p1(6, 5)@1 :- a2(7, 5), \+ l4_1(7, 1, 19).
p1(8, 8)@1 :- \+ a1(5, 1, 7).
p1(X0, X0)@1 :- a2(15, 9), l4_1(X0, X0, X0), l2_1, a2(14, X0).
p2(2)@1 :- l4_1(14, X1, 11), \+ l1_1(X1), \+ l2_1, X1 in 11..15, l4_1(11, X1, 15), a2(X1, 11), \+ a2(X1, X1).
% Agent 2: 20
ic :- a2(X0, X0), X0 #< 17, l1_2(X0), l2_2.
l1_2(11) :- \+ p1(0, X0)@X0, \+ a1(5, X0, X0), l4_2(X0, X0, X0), l2_2, a2(12, X0), p3(X0, X0)@X0, l5_2.
l1_2(13) :- l2_2, \+ l5_2, l5_2.
l1_2(4) :- l5_2, a2(1, 8), p1(2, 15)@0, a2(0, 6).
l1_2(9) :- p3(16, 0)@1, l5_2.
l2_2 :- \+ l4_2(9, 17, 17), \+ l4_2(3, 3, 9), \+ a1(8, 18, 5).
l2_2 :- l4_2(15, X0, 17), l4_2(X1, X1, X1).
l2_2 :- l4_2(4, X0, X0).
l3_2(11, 11, 15) :- \+ l2_2.
l3_2(15, 2, 17).
l3_2(6, 19, 3) :- l1_2(18), 14 #=< 3, \+ l4_2(4, 6, 11).
l5_2 :- 4 #=< 12, l4_2(12, 15, 18), \+ l4_2(3, 1, 2), \+ p3(4, 15)@1, l2_2.
p1(16, 11)@2 :- \+ l2_2, l4_2(X0, 9, X0), X0 in 1..5, l4_2(X1, 9, X1), a2(X0, X1).
p1(17, 17)@2 :- l4_2(X0, X0, X0), a2(X0, X0).
p1(3, 11)@2 :- 15 #< 13, \+ a1(10, 14, 10), \+ l4_2(16, 1, 16), a1(15, 15, 14), l2_2, a1(11, 9, 11).
p1(X0, X0)@2 :- X0 in 1..10, \+ l2_2, X0 #=< 17, l2_2.
p1(X0, X0)@2 :- a2(X0, X0), l2_2, a2(1, 3), \+ a1(X0, X0, 9), l4_2(X0, 15, X0), a1(X0, X0, 17), \+ a2(8, X0).
p2(6)@2.
p2(X0)@2 :- l1_2(X1), l4_2(X0, X1, X1), l1_2(X0), p1(X1, X0)@X1.
p3(X0, X0)@2 :- l4_2(X0, 15, X0).
```

Figure 6.2: Example Output File for GenDACLP (`total.pl`)

6.1.2 Implementation of GenDACLP

The process for generating the distributed LPs from a set of given parameters consists of the following five steps (see Figure 6.3):



Step	Relevant Parameters
1.	`numAbduciblePredicates,` `numPublicPredicates,` `numLocalPredicates`
2.	`numAskableRules,` `numPrivateRules,` `numICRules,` `abducibleICs,` `avgRuleBodySize,` `avgArgumentSize,` `negationBias,` `abducibleBias,` `askableBias`
3.	`numConstants,` `variableBias,` `variableCoupling`
4.	`numAgents`

Figure 6.3: Key Steps of the GenDACLP

1. **Generate Predicates**: during this step, the specified numbers of abducible predicates, public predicates and local predicates are generated. For example, if N abducible (public or local) predicates with average argument size of M are required, predicate names $a1, \ldots, aN$ ($p1, \ldots, pN$ or $l1, \ldots, lN$) will be created, and the arity of each of them is an integer randomly selected from $[0, 2 \times M]$. Note that there are only three fixed constraint predicates for $CLP(\mathcal{FD})$ atoms, they are #</2 (*less than*), #=</2 (*less than or equal to*), and $domain/3$ (e.g, $domain(X, 1, 10)$). The predicates *greater than* and *greater than or equal to* are not needed as they are symmetric to *less than* and *less than or equal to*, respectively.

2. **Generate Rules:** during this step, specified numbers of rules defining askable atoms or private atoms or as integrity constraints are randomly created. However, all the literals appearing in these rules have empty arguments, which will be *filled* in at the next step.

Moreover, different algorithms need to be used in order to guarantee that the rules created at this step constitute an overall LP with a specified program structure, e.g. stratified and acyclic. These algorithms will be described shortly.

3. **Fill Arguments:** during this step, the arguments of all the literals appearing in the overall LP are defined, so that the *allowedness* condition is satisfied. The pseudo-code of this algorithm is shown in Figure 6.4.

```
PROC fill_arguments BEGIN
  $setOfConstants$ = [0, numConstants];
  FOREACH $rule$ in the overall logic program DO
    $numPosArgs$ = total number of arguments in the positive non-constraint body literals in $rule$;
    $maxNumVars$ = $ceiling(numPosArgs \times$ variableBias $\times$ (1 − variableCoupling));
    $setOfVariables = \{X_0, \ldots, X_{maxNumVars}\}$;
// 1. fill arguments for positive body literals, and collect variables that
// can be used for filling arguments of negative or constraint literals
    $setOfSafeVars$ = empty;
    FOREACH $posBodyLiteral$ in $rule$ DO
      FOREACH $argument$ of $posBodyLiteral$ DO
        IF nextRandomFloat() < variableBias THEN
          $argument$ = a variable randomly selected from $setOfVariables$;
          add $argument$ to $setOfSafeVars$;
        ELSE
          $argument$ = a constant randomly selected from $setOfConstants$;
        END IF
      END FOREACH
    END FOREACH
// 2. fill arguments for remaining literals
    FOREACH $remainingLiteral$ in $rule$ DO
      FOREACH $argument$ of $remaingLiteral$ DO
        IF $remainingLiteral$ is an askable head AND $argument$ is the ID argument THEN
          $argument$ is left unassigned;
        ELSE IF nextRandomFloat() < variableBias THEN
          $argument$ = a variable randomly selected from $setOfSafeVars$;
        ELSE
          IF $argument$ is an ID argument THEN
            $argument$ = (a constant randomly selected from $setOfConstants$) mod numAgents;
          ELSE
            $argument$ = a constant randomly selected from $setOfConstants$;
          END IF
        END IF
      END FOREACH
    END FOREACH
  END FOREACH
END PROC
```

Figure 6.4: Pseudo-code for Filling Arguments

4. **Partition Rules:** this step has three sub-steps: (a) randomly distribute all the rules to the specified number of agents; (b) instantiate the agent ID argument of the head atom of every askable rule (i.e., bind it to the alias of the rule's owner agent); (c) resolve naming conflicts of the local predicate (i.e., for every private atom appearing in a rule of an agent say ag, rename its local predicate by appending $\_ag$). Note that the last sub-step has no effect on the testing of DAREC^2. It is useful if the overall LP is tested by a centralised theorem prover.

5. **Split Overall LP:** this step does not involve any computation and simply splits the overall LP into one topic file shared by all the agents and one theory file for each agent.

Generating Acyclic LPs

For a generated logic program P to be acyclic, there must exist a (partial) ordering of all the predicates appearing in P, such that for every rule in P, the predicate of the head must be greater in the ordering than all the predicates in the body. Checking the satisfiability of this condition for a given LP is not straightforward, as it requires the construction of the directed graph(s) representing the dependency relationships between all the predicates, and loop detection over the graphs. However, to generate a LP that satisfies the condition is easier – we first order all the predicates, and generate the rules one by one ensuring that for each rule only the predicates smaller than that of the head can be selected to construct the body literals. The key steps of the algorithm used by GenDACLP for generating an acyclic LP are as follows (note that each integrity constraint is in fact a rule with a special atom ic as the head, and every query to the program will have the implicit sub-goal $\neg ic$ in it):

1. **Order Predicates**: in this step every predicate is assigned an integer value indicating its *rank*. Let MIN be zero and let MAX be the total number of predicates to be used for program generation. The rank assignment is done as follows:

   (a) all abducible and constraint predicates have the rank of MIN;

   (b) ic has the rank of MAX+1;

   (c) let L be a shuffled list of all the public and local predicates; each predicate in L is assigned a rank equal to its position in L, i.e, from 1 to MAX.

2. **Generate Askable Rules**: in this step, the specified number of askable rules are generated as follows:

   (a) a public predicate P_H is randomly selected to construct the head atom;

   (b) the number of body literals is a random number in $[0, 2 \times$ `avgRuleBodySize`$]$, and each body literal L is generated as follows:

      i. let $N1$ be the next random float number in $[0.0, 1.0)$; L is positive if and only if $N1 >$ `negationBias`, and

      ii. let $N2$ be the next random float number in $[0.0, 1.0)$; if $N2 <$ `abducibleBias` then an abducible predicate is randomly selected to construct L; otherwise

      iii. let $N3$ be the next random float number in $[0.0, 1.0)$; if $N3 <$ `askableBias` then an askable predicate *whose rank is smaller than that of* P_H is randomly selected to construct L; otherwise,

      iv. let $N4$ be the next random float number in $[0.0, 1.0)$; if $N4 <$ `constraintBias` then a constraint predicate is randomly selected to construct L; otherwise

      v. a local predicate *whose rank is smaller than that of* P_H is randomly selected to construct L.

3. **Generate Private Rules**: in this step, the specified number of private rules are generated as in step (2), except that a local predicate is randomly selected to construct the head atom.

4. **Generate Integrity Constraints**: in this step the specified number of integrity constraints are generated as in step (2), except that the head atom is always *ic*.

Generating Stratified and Arbitrary LPs

The algorithm used for generating acyclic LPs can be easily modified to generate LPs with other structures. For example:

- to generate an *arbitrary* LP, the predicates do not need to be ordered, and the selection of public or local predicate for constructing a body literal can be completely random (i.e., without comparing its rank with that of the head);

- to generate a *stratified* LP, two small modifications are needed. First, in step (1) instead of giving a total order to the (public and local) predicates, we can give a partial order to them, e.g., randomly select an integer from $[1, MAX]$ and assign it to a predicate. By doing so, it is possible for two predicates to have the same rank, in which case they are in the same stratum. Secondly, in step (2)(b)(iii) and step (2)(b)(iv) while selecting a public or local predicate to construct the body literal, the rank of the selected predicate must be smaller than or equal to (strictly smaller than) that of head atom if the literal is positive (negative).

6.2 Experiments and Discussions

6.2.1 Environmental Setup

Experiments of the DAREC^2 system implementation can be automated with GenDACLP. Each experiment consists of a series of test cycles, each of which has the following step:

1. **generation of distributed LPs:** this is done by providing a knowledge configuration file and running GenDACLP;

2. **launching of agents:** this is done by starting an agent on a network for each theory file generated (the first agent is assigned the role of leader);

3. **query computation:** this asks the system to compute all solutions for all the possible askable goals of the leader (i.e, each advertisement made by the leader is a query);

4. **data collection:** this involves the logging of system profiling information, such as the time spent and message exchanges;

5. **system shut down**.

A network of up to 30 machines inter-connected through Ethernet was used for the experiments. The network was reliable, and the latency was small, e.g., with on average 0.15 milliseconds ping time between two machines. Each machine had 4 GB of RAM and an Intel Core2 Duo processor running at 3.00 GHz. All the machines ran Ubuntu 10.04, and the DAREC^2 code was executed with YAP Prolog 6.2.1. The profiling information collected included the *total time* spent on each test, and the *total number* and the *total size* of messages exchanged. Note that for the tests where there were more than 45 agents, each machine may host more than one agent. Note that also in order to guarantee termination, only acyclic LPs were used.

6.2.2 Experiments

The objectives of our experiments are two-fold: to check for correctness of the system implementation, and to study how the system performance is affected by distributed LPs with different sizes or structures.

For the first objective, we can first run DAREC^2 with the generated distributed LPs, then run ASystem with the union of the LPs and compare the answers computed by the two systems for some given queries. Any inconsistency between these answers would indicate an (implementation) *bug* of DAREC^2. Indeed, we performed this step for every conducted experiment, and we discovered and fixed several bugs at the early stage.

For the second objective, we can generate distributed LPs by fixing a subset of the tunable parameters (such as the total number of abducible predicates and the variable bias), and gradually changing the rest of parameters (such as the number of agents or the number of rules per agent). We can then run DAREC^2 with these sets of generated LPs and compare their execution time and communication costs. However, there are 16 numeric parameters that we can investigate, and the designs of experiments (i.e., deciding what and how parameters should be fixed, and how the others should be adjusted) could be infinite. In this chapter, we only describe and discuss two of the experiments that we have conducted and found most interesting.

Basic Settings

In these two experiments, we fixed the following parameters:

- `avgRuleBodySize = 4`: on average each rule has four body literals;
- `avgArgumentSize = 2`: on average each predicate has two arguments;
- `variableBias = 0.7`: each predicate argument has a 0.7 chance being a variable;
- `variableCoupling = 0.5`: on average each variable appears in a rule twice;
- `negationBias = 0.25`: on average one in four body literals is negative;
- `abducibleBias = 0.2`: on average one in five body literals is abducible;
- `askableBias = 0.2`;
- `constraintBias = 0.1`;
- `numICRules = 0`: i.e., no integrity constraints
- `numConstants = 100`;
- `numAbducibles = 5`;
- `numPublicPredicates = 10`.

Note that in the experiments we did not test DAREC^2 with integrity constraints. It allowed us to focus more on the structure of the agent background knowledge, and would not affect the plausibility of the experiments due to the following reason. The integrity constraints of an agent ag's abductive framework can be "compiled" into its background knowledge Π_{ag}. For example, for each integrity constraint $\leftarrow L_1, \ldots, L_n$ of ag's we can transform it as a rule $ic \leftarrow L_1, \ldots, L_n$ into Π_{ag} where ic is a (new) private atom, as long as we also add a single rule $consistent@ag \leftarrow \neg ic$ to Π_{ag} and append the goal $consistent@ag$ to the global query. This means that the computation for checking integrity constraints can be "converted" into the computation of checking a negative non-abducible (i.e., $\neg ic$). Therefore, during the experiments we could compensate the computation for integrity constraint checking by increasing the value of the `variableBias` parameter, without generating the integrity constraints.

In each experiment, we increased the number of agents gradually, and the remaining parameters (i.e., `numLocalPredicates`, `numAskableRules` and `numPrivateRules`) were either fixed or computed according to some heuristics (see experiment descriptions). For each GenDA-CLP configuration file of the experiment, we randomly generated three sets of distributed LPs (i.e., using different random seeds). Each set was tested by both DAREC^2 and our YAP-implementation of ASystem. We call such a run a *test*.

Experiment A: fixes the size of each agent's LP, and increases the number of agents

In this experiment, each agent had 10 local predicates, 30 private rules (i.e., each private atom of an agent had about three definitions in that agent) and 30 askable rules (i.e., each askable atom had about three definitions in each agent). Thus, each agent had the same size of LP (with 60 rules). We increased the number of agents from $1, 3, 5, \ldots$ up to 27, and hence increased the size of the overall LP. For each test, up to 50 queries were randomly generated, each of which had the form of $p(\vec{X})@id$ (i.e., all arguments except the ID were variables). The agents were asked to compute all the solutions. The aim of this experiment was to study how well the system can scale with respect to the number of agents.

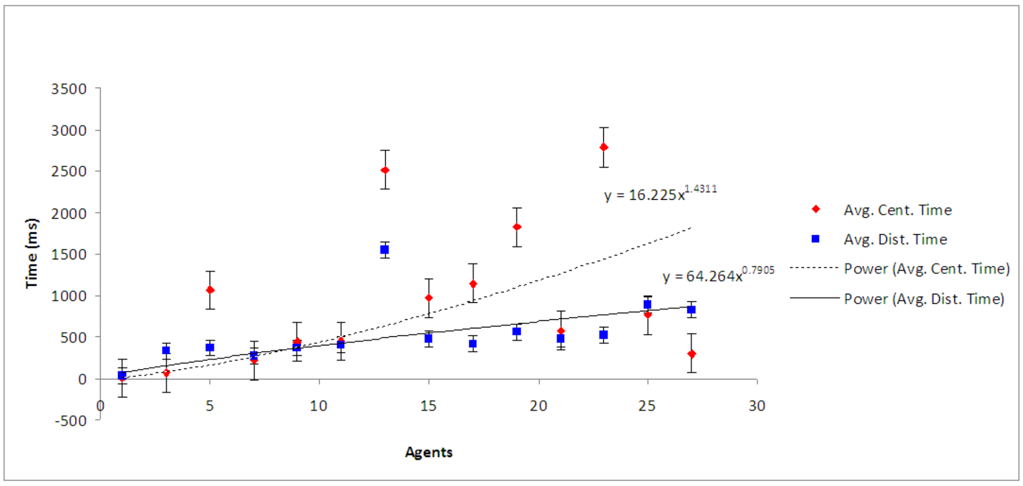


Figure 6.5: Experiment A: Average Centralised/Distributed Computation Time vs. Number of Agents (Size of the Overall Logic Program)

Tests	Number of Agents	Number of Queries	Number of Answers	Cent. Time (ASystem) [milliseconds]	Dist. Time (DAREC^2) [milliseconds]	Number of Messages	Total Traffic (bytes)
test1/set1	1	10	7	10	24	24	1795
test1/set2	1	9	11	3	42	31	2552
test1/set3	1	10	6	2	19	22	1561
test10/set1	19	50	102	3790	839	414	46698
test10/set2	19	50	39	1483	419	230	22686
test10/set3	19	50	44	191	435	250	25756
test11/set1	21	50	47	410	512	262	26698
test11/set2	21	50	45	625	441	238	24116
test11/set3	21	50	59	687	470	274	26983
test12/set1	23	50	35	2104	486	226	24311
test12/set2	23	50	41	1791	502	244	23055
test12/set3	23	50	70	4465	570	362	39839
test13/set1	25	50	42	809	561	238	24743
test13/set2	25	50	42	991	571	262	34033
test13/set3	25	50	55	478	1534	332	45254
test14/set2	27	50	66	202	600	360	44189
test14/set3	27	50	53	391	1045	268	27236
test2/set1	3	29	24	115	734	137	14205
test2/set2	3	30	19	68	122	110	10341
test2/set3	3	29	32	13	122	135	13093
test3/set1	5	47	40	92	226	213	20018
test3/set2	5	47	76	2963	634	351	36956
test3/set3	5	50	50	145	257	240	23349
test4/set1	7	50	31	189	262	206	20482
test4/set2	7	50	53	341	289	256	26094
test4/set3	7	50	52	132	267	244	24261
test5/set1	9	50	36	350	335	228	23023
test5/set2	9	50	49	343	334	240	23701
test5/set3	9	50	60	638	449	272	27708
test6/set1	11	50	40	991	404	224	21559
test6/set2	11	50	37	107	452	226	23783
test6/set3	11	50	47	260	367	252	26626
test7/set1	13	50	36	2197	3503	222	22884
test7/set2	13	50	51	1260	740	288	31786
test7/set3	13	50	63	4087	407	302	33604
test8/set1	15	50	78	226	378	306	30987
test8/set2	15	50	40	2025	612	232	24326
test8/set3	15	50	48	655	432	290	34935
test9/set1	17	50	33	466	362	224	22639
test9/set2	17	50	36	1010	384	230	24145
test9/set3	17	50	47	1945	484	252	25309

Table 6.2: Experiment A (Collected Data(: all the *sets* within a *test* used the same configuration except the randomly number generator seed for the GenDACLP

The data collected during this experiment is shown in Table 6.2. The execution time of all the tests (taking the average of all the sets' in each test) in this experiment are given in Figure 6.5. We used *power regression*[1] for the trend lines of the data series, and the lines fit both the average centralised computation time series (dotted curve) and the average distributed computation time series (solid curve) better than those using *linear regression*. We can observe that the increase in time for distributed abduction over increasing number of agents (i.e., increasing size of the overall LP) actually *decelerates* (i.e., the exponent of x is less than 1 in Figure 6.5), which is opposite to the case for centralised abduction (i.e., abduction is known to have big complexity). We conjecture that this is due to the parallel computation (i.e., agents perform local abductions concurrently) during distributed abduction. There is another observation from this figure: the average distributed computation time series increases steadily, whereas the average centralised computation time series fluctuates noticeably. We conjecture that this is

[1]Power regression can be used to compare measurements that increase at a specific rate.

due to the fact that centralised abduction was affected more by other activities (such as garbage collection in Prolog) in the host machine, as the resources (CPU and memory) of the host were limited and fixed while the size of problem increased. In contrast, in distributed abduction agents performed local abductions with equal-size LPs that were much smaller than the overall LP. Thus the problem size of each local abduction remained more or less the same and the agents had less stress for the usage of resources (i.e., fewer garbage collections took place). In addition, since both the centralised and distributed abductive algorithms were implemented as interpreters in Prolog, the smaller the logic program was, the less overhead would incur (e.g., due to scanning through the rules) during their executions.

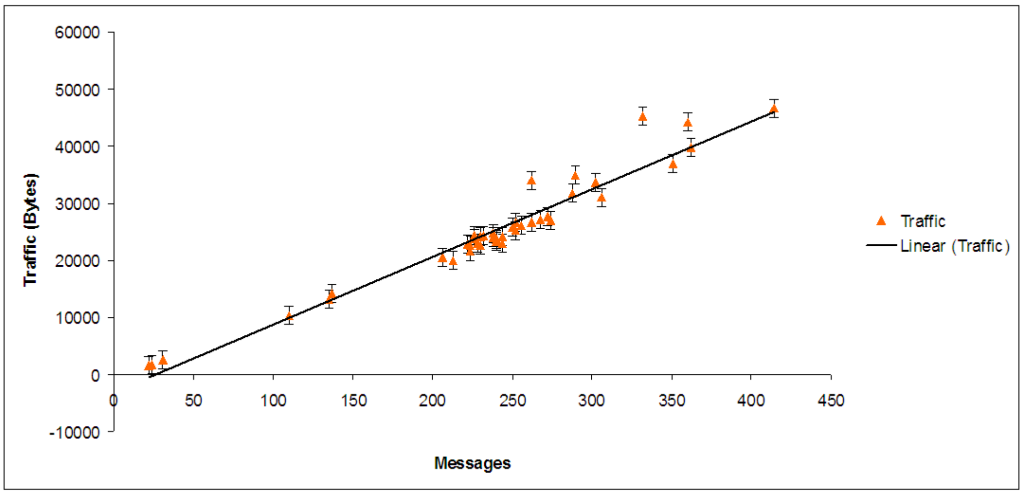


Figure 6.6: Experiment A: Communication Cost vs. Messages Exchanged

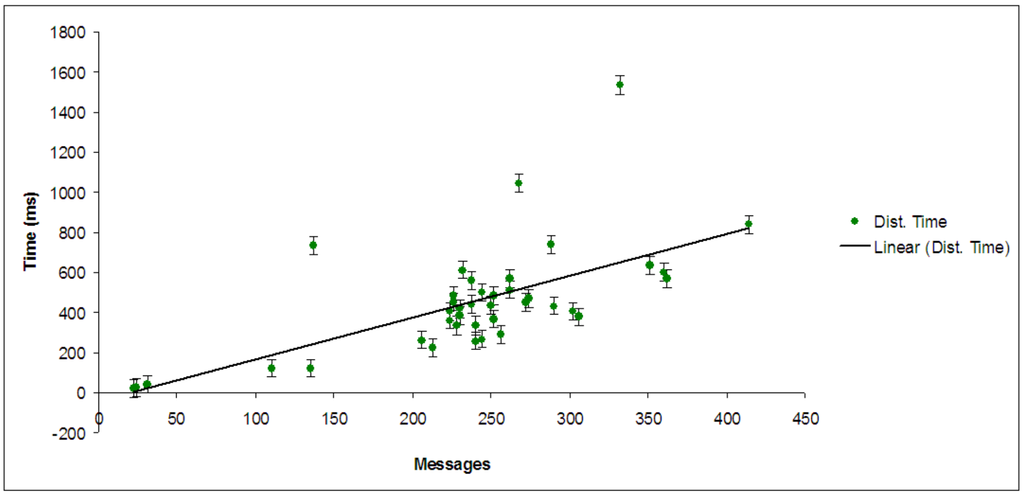


Figure 6.7: Experiment A: Average Distributed Computation Time vs. Messages Exchanged

Figure 6.6 and Figure 6.7 show the relationship between the total communication cost (in bytes) and number of messages exchanged during distributed abduction, and the relationship between the total computation time (in milliseconds) and the numbers of message exchanged, respectively. As expected, both the communication cost and the computation time are proportional to the number of messages exchanged. This indicates that in distributed abduction, the performance was affected by the number of inter-agent communications (i.e., sending and receiving messages) significantly.

Experiment B: fixes the size of the overall LP, and increases the number of agents while fixing the number of askable rules in each agent

In this experiment, the size of the overall LP was fixed to be 1380 rules, and the number of agents increased from $5, 7, \ldots$ up to 23. For each test, each agent's LP had $n = 1380/\texttt{numAgents}$ number of rules, where 30 of them were askable rules. The number of local predicates for each agent was calculated by $(n - 30)/3$. Thus, on average each agent had three definitions for each askable or private atom. Note that between the tests, the number of askable rules in an agent's LP was fixed, whereas the number of (local predicate and) private rules decreased as the number of agents increased. By fixing the number of askable rules in each agent, we tried to ensure a certain amount of agent interaction during the distributed computation, as the aim of this experiment was to study the importance of parallel computation in distributed abduction.

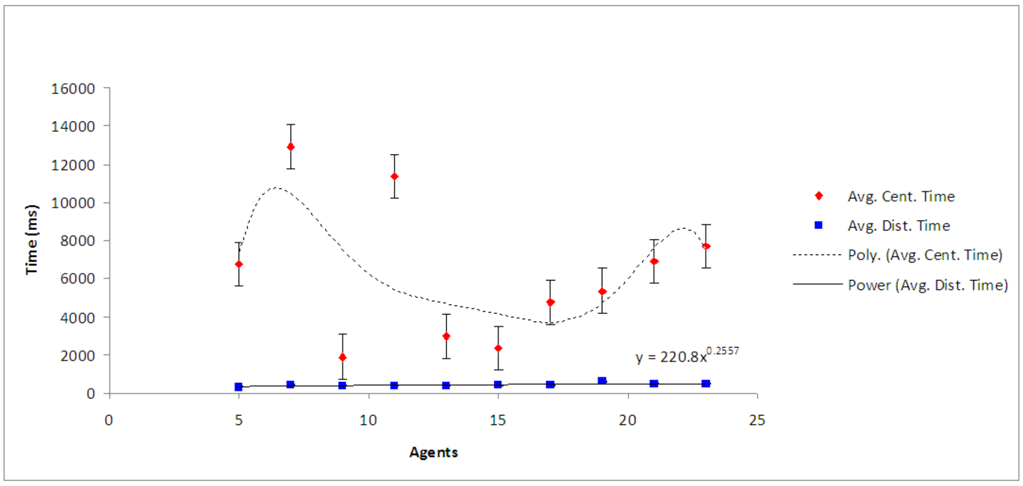


Figure 6.8: Experiment B: Average Centalised/Distributed Computation Time vs. Number of Agents

Tests	**Number of Agents**	**Number of Queries**	**Number of Answers**	**Cent. Time (ASystem) [milliseconds]**	**Dist. Time (DAREC^2) [milliseconds]**	**Number of Messages**	**Total Traffic (bytes)**
test01/set1	5	47	35	3232	373	211	22023
test01/set2	5	45	31	16395	332	199	20571
test01/set3	5	45	48	641	249	237	24342
test02/set1	7	50	62	25588	603	292	33206
test02/set2	7	50	48	209	262	242	24440
test03/set1	9	50	36	1176	325	240	23937
test03/set2	9	50	46	2080	472	258	26465
test03/set3	9	50	42	2455	345	240	25448
test04/set1	11	50	38	31005	398	254	26904
test04/set2	11	50	65	1056	379	294	33370
test04/set3	11	50	69	1948	375	284	29668
test05/set1	13	50	46	1300	414	276	31376
test05/set2	13	50	58	1995	354	256	26123
test05/set3	13	50	33	5668	364	220	21781
test06/set1	15	50	26	1781	387	214	23651
test06/set2	15	50	57	2521	435	288	32119
test06/set3	15	50	50	2800	427	258	24454
test07/set1	17	50	44	11067	410	242	25614
test07/set2	17	50	35	1010	408	216	22898
test07/set3	17	50	40	2248	409	238	24751
test08/set1	19	50	45	13972	521	298	32441
test08/set2	19	50	44	494	472	278	32389
test08/set3	19	50	39	1601	839	232	23153
test09/set2	21	50	55	3585	451	262	26903
test09/set3	21	50	66	10183	507	288	29656
test10/set2	23	50	40	13161	501	248	27248
test10/set3	23	50	37	2194	460	222	23949

Table 6.3: Experiment B (Collected Data(: all the *sets* within a *test* used the same configuration except the randomly number generator seed for the GenDACLP

The data collected during this experiment is shown in Table 6.3. Figure 6.8 shows the comparison of the average computation time between centralised abduction and distributed abduction over the number of agents, and Figure 6.9 gives a clearer relationship between the average distributed computation time and the number agents. As we can see in Figure 6.9, the increase in time for distributed abduction over increasing number of agents still decelerates (e.g., like in Experiment A, where the exponent of x is less than 1). In Figure 6.8 we can see that the average computation time for centralised abduction fluctuates without clear increasing or decreasing trend. We believe this is because the size of the overall LPs remains the same and the fluctuation in time is caused by variation of the overall LPs (which were generated randomly). However, in Figure 6.8 distributed abduction always performed better than centralised abduction. This again shows that parallel computation in DAREC^2 helps it scale better than ASystem.

Figure 6.10 and Figure 6.11 give the total communication cost, the total computation time, and the number of message exchanged during distributed abduction. Their relationships are consistent with those in Experiment A.

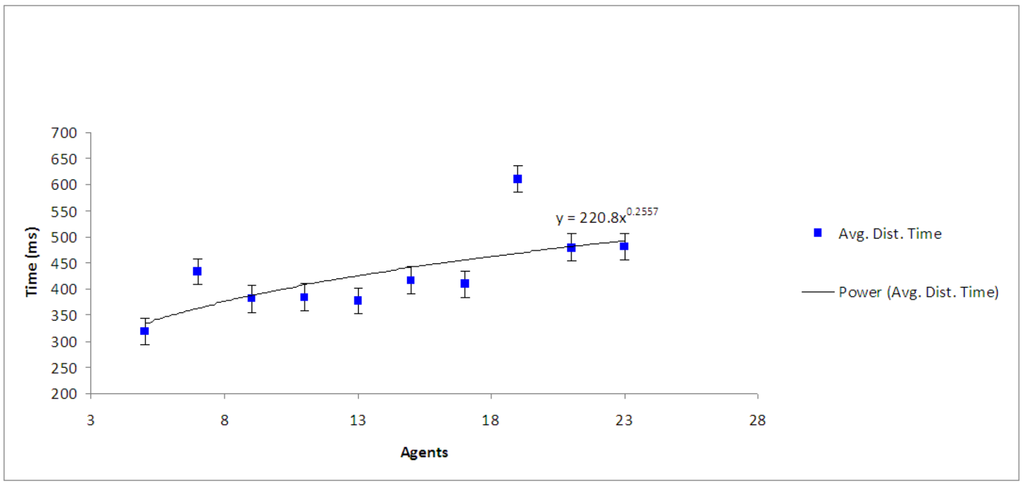


Figure 6.9: Experiment B: Average Distributed Computation Time vs. Number of Agents

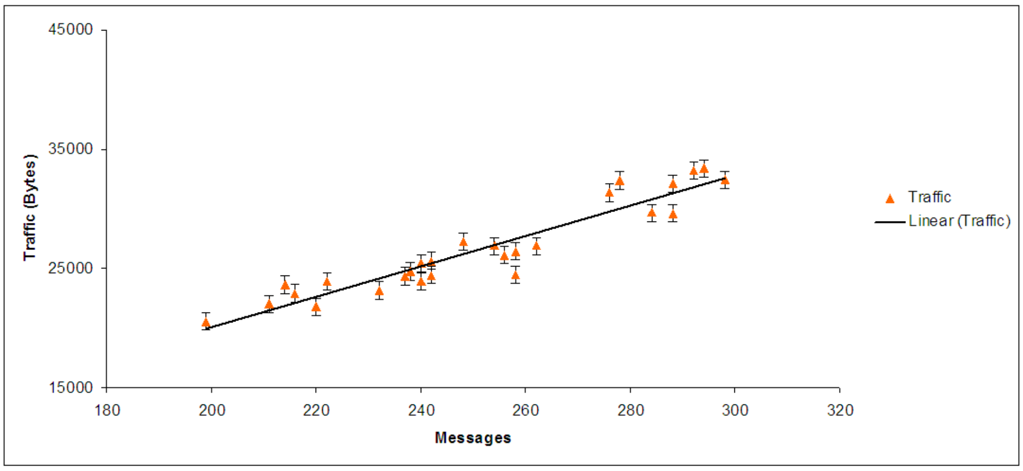


Figure 6.10: Experiment B: Communication Cost vs. Messages Exchanged

6.2.3 Other Experiments:

In addition to Experiment **A** and Experiment **B**, we have also conducted other similar experiments by adjusting some of the fixed parameters. However, they are not described here either because their data gives the same conclusions as those given by Experiment **A** and Experiment **B**, or because their data contains too much *noise* (i.e., some measurements regarding the computational time are too far away from the trend lines). The noise is most likely caused by the external activities occurring in the hosts while the hosts were running the DAREC^2 agents. Since the network used for conducting the experiments was part of the departmental undergraduate teaching lab, other users might have logged into the hosts and have been running

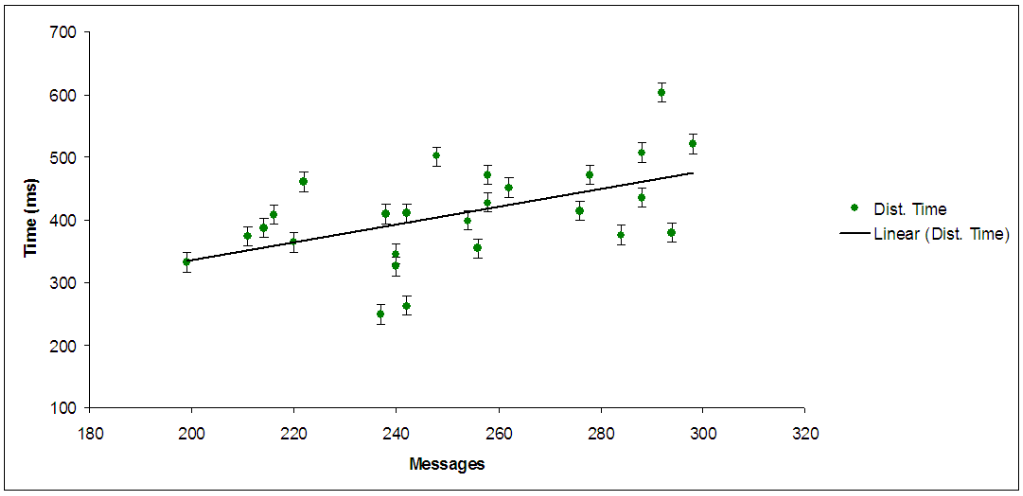


Figure 6.11: Experiment B: Average Distributed Computation Time vs. Messages Exchanged

their (computationally intense) processes during our experiments. As a future work, we need to find a better network reserved specifically for our testing and run these experiments again in order to reduce noise.

We have also conducted experiments of comparing DAREC with different agent interaction strategies. In these experiments, we modified the DAREC^2 system to adopt the *eager* agent interaction strategy and a *safe* goal selection strategy. For each set of the generated distributed logic programs, we first ran the original DAREC^2 (i.e., the one with the *lazy* agent interaction strategy and the *secure safe* goal selection strategy), and then ran the modified DAREC^2. The overall computational time and the total message number were recorded during the experiments. The collected data showed that the modified DAREC^2 was always slower than the original DAREC^2 – the time taken by the former was between 1.3 and 29.96 times that taken by the latter. The data also showed that there were always more messages exchanged during the modified DAREC^2 computation than the original DAREC^2 computation – the number of messages exchanged by the former was between 1.14 and 28.71 times that by the latter. However, these experiments are not described in detail here because the modified DAREC^2 cannot guarantee *confidential reasoning* (i.e., it is just another customisation of DAREC).

6.3 Conclusion

In this chapter, we have described our experiments with the DAREC^2 system prototypes. In particular, we have presented a flexible random generator (GenDACLP) for (distributed) logic programs, which is used for generating testing data sets (i.e., distributed agent knowledge) for the testing of DAREC^2. As aforementioned, by configuring the input parameters this generator can produce logic programs with different structures and properties, which can also be used for the testing or benchmarking of other reasoning systems that are based on logic programming. Large numbers of different experiments with DAREC^2 can be easily designed and conducted by the auto-testing environment we have developed. However, we have only discussed in this chapter two of them. In addition to the main objectives of DAREC^2 , which are to support correct distributed reasoning and to maintain confidentiality during reasoning, all of our experiments have shown that the performance of distributed abduction benefits a lot from the concurrent computation of the DAREC^2 algorithm execution.

Chapter 7

Distributed Policy Analysis

7.1 Introduction

Policy-based management is a popular and promising paradigm for managing distributed networks and systems [Str03]. It separates the system specification and the system implementation. Thus, administrators can control the system through its exposed high level functionalities without needing to worry about its low level implementation. Policies [SL02] are rules governing the system behaviour, and they can help the target system adapt to new environments without further human interventions. There are two main types of policies: *authorisation* and *obligation*. Authorisation polices define what operations are allowed with certain conditions, and obligation policies define what operations need to be performed after being triggered by events or the satisfaction of certain conditions.

A typical policy-based management system (PBMS) has a policy enforcement point (PEP) and a policy decision point (PDP) [YPG00] (see Figure 7.1). Any action to be executed by the system must be initiated with a *request*, which is intercepted by the PEP. The decision of either approving or declining the request is computed by the PDP, which has access to a *policy repository* and relevant system domain information. Finally, the action is performed by the PEP in the case of approval.

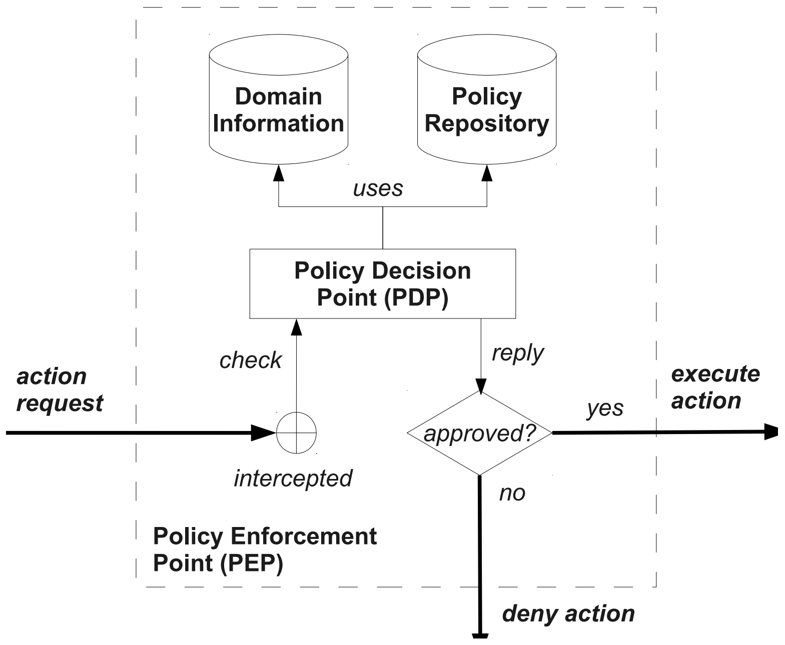


Figure 7.1: A Policy Enforcement Point

Since most policies are specified by human administrators, it is desired to check whether there are errors in the policies or whether the system will behave as expected given the policies. In a complex PBMS where the size of policies or the size of the system domain is large, this becomes a difficult task that requires automated support. In this chapter, we will look at one such formal policy framework described in [CLM+09]. Within this framework, both authorisation and obligation policies specified in a policy language, for example Ponder or XACML, can be reformulated in a first-order logic language. The system domain is dynamic (i.e., subjects to changes upon events) and is formulated with Event Calculus [KS86] (EC). Various analysis tasks over the policies and the system domain can be solved using abductive reasoning. These analysis tasks include:

- modality conflict: checks if there is a request approved and declined at the same time, or if there is an obligation for an action lacking permission to fulfil it;
- separation of duty: checks if two conflicting actions (in terms of roles) can be permitted at the same time;
- coverage gaps: checks if there is a request neither approved nor declined by the policies;

- policy comparison: checks whether two sets of policies are equivalent, or one subsumes the other;
- behaviour simulation: provides a system execution trace and determines which policy decisions arise at the final state.

The existing framework [CLM+09] assumes a central repository of all the policies and the system domain (i.e., a centralised PEP/PDP). Analysis tasks are modelled as abductive queries, and abductive reasoning is used, either to prove the queries (in the case of the last two types of tasks), or, in the other cases, to find possible system execution traces (in terms of action requests) that would lead the system from a given initial state to the target (violating) state described by the analysis queries.

However, in many systems where confidentiality becomes a primary concern we cannot assume a centralised repository. For example, during an international joint-rescue operation for an earthquake-hit zone, a (temporarily formed) coalition network may involve a US sensor network control server and a set of personal assistant devices (PDAs) for a team of UK paramedics. The US control server contains private authorisation policies regulating the access of sensing data, and the UK PDAs have private obligation policies to retrieve data from the US fabric under certain situations. Any potential (modality) conflict in these two private sets of policies must be resolved before the operation in order to guarantee seamless collaboration. In this case, confidentiality concerns preclude the possibility of centralising all the policies (and any private domain information on which the policy evaluation may depend), and the existing policy framework becomes inadequate for policy analysis over these systems, as non-distributed abductive reasoning algorithms are not applicable. Let us be clear: we are dealing here not just confidentiality at runtime of the actual agents, but also confidentiality restrictions at analysis time.

Distributed policy analysis is needed. To address the aforementioned issues, the existing policy framework needs to be extended, so that

- the operational model can cope with multiple PEPs/PDPs, each of which may operate independently with respect to its own set of policies and its own local domain;

- the policy specification and system domain description language can allow the specification of distributed knowledge, and the separation between shared and private knowledge.

In this chapter, we describe how the DAREC^2 language can help to extend the existing framework, and how the DAREC^2 algorithm can be used to perform distributed confidential policy analysis.

The rest of the chapter is organised as follows. Section 7.2 gives an introduction to the existing policy framework [CLM+09] for centralised policy analysis. In Section 7.3, we first describe how the operational model and language are extended from the existing framework, and then show how distributed policy analysis can be performed using DAREC^2 using a running example. Section 7.4 discusses two further possible extensions to the new framework, and Section 7.5 gives final conclusions.

7.2 A Formal Framework for Centralised Policy Analysis

In this section, we briefly introduce the formal policy framework developed in [CLM+09]. We will first describe the operational model of the framework, and then present the policy specification language and the domain description language, and finally give policy analysis examples.

7.2.1 Operational Model

As aforementioned, the operational model [CLM+09] (Figure 7.2) of a policy managed system (or simply the *regulated* system) assumes a centralised PEP/PDP. A system execution trace is a sequence of system states, each of which is associated with a time point. System domain properties may persist over states by default, or be affected by occurred events. There are two types of events: *non-regulatory events* that are exogenous to the system (e.g., power failure), and *regulatory events* that are the result of a *PEP/PDP* step. The PEP/PDP step at a system state consists of the following sub-steps:

1. an *action request*, called the *regulatory input*, arises (e.g., as the fulfilment of an existing obligation) and is intercepted by the PEP;

2. the PEP invokes policy evaluation at the PDP;

3. the PDP has access to the *policy repository* and the current *system domain*. The evaluation of the policies relevant to the request depends on the existing policies and the system state information. The latter can be divided into two parts: the state of the domain called the *non-regulatory state*, and the state of the PEP/PDP (e.g., whether the conditions of a policy are satisfied or whether an existing obligation is fulfilled) called the *regulatory state*.

4. after policies are evaluated by the PDP, the PEP acts accordingly, either by executing the action or by denying the action (i.e., two types of regulatory events).

In addition to the centralised PEP/PDP assumption, the current operational model does not allow concurrent actions, i.e., at any time only one of regulatory and non-regulatory events can occur.

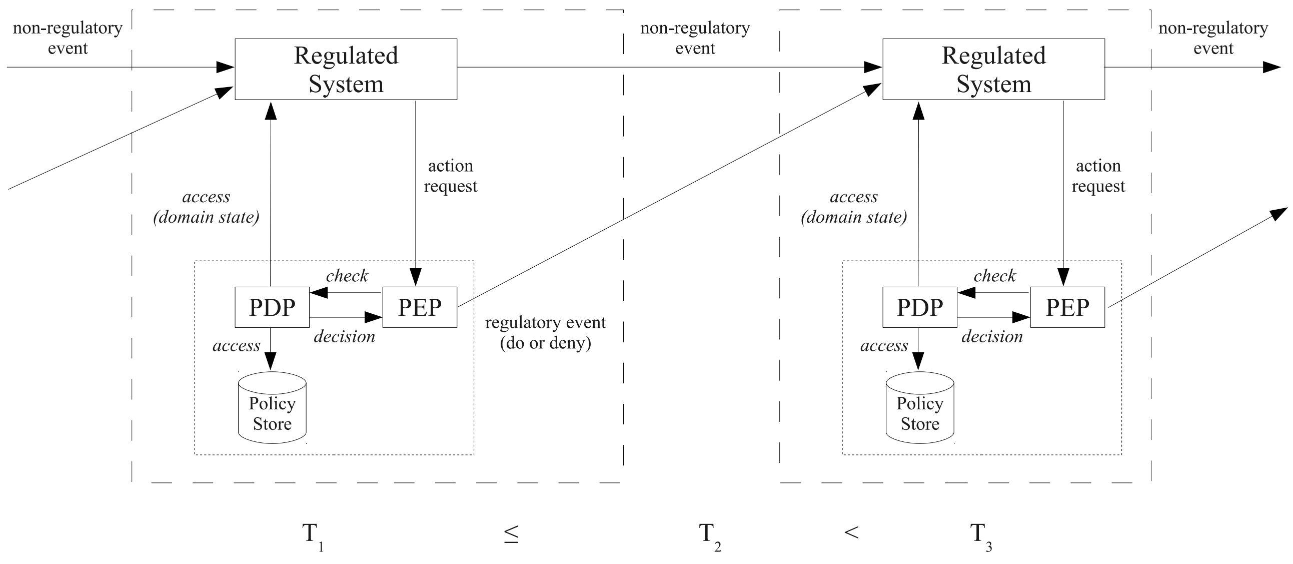


Figure 7.2: Operational Model with Centralised PEP/PDP

7.2.2 System Specification

The language [CLM+09] used for specifying the system policies and system domain is a first-order logic language $\mathcal{L}$. All terms in $\mathcal{L}$ are divided according to sorts `subject`, `target`, `action`, `time`, `fluent` and `event`. The predicates for $\mathcal{L}$ are summarised in Table 7.1.

Policy Specification

There are two types of policies: *authorisation* and *obligation*. Policies are expressed as logical rules. An authorisation (policy) rule describes a permission, e.g., what *subject* can perform what *action* on *what target* at what *time* and under what certain *conditions*. Such a permission can be either *positive* or *negative*, and is modelled as:

$$permitted(Su, Ta, Ac, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.$$

$$denied(Su, Ta, Ac, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.$$

where each L_i is a literal representing a condition and each C_j is an (arithmetic) constraint over time variables (i.e., variables of the sort `time`). The predicates allowed in the policy conditions are those in $\{req\} \cup \mathcal{L}_S^{\Pi} \cup \{holds, happens, clipped\} \cup \mathcal{L}_{stat}$. Note that each of these condition literals, except those formed from a predicate in $\mathcal{L}_{stat}$, has its associated time variable (as the last argument) constrained by $C_1, \ldots, C_n$. Thus, it is required that every variable appearing in $C_1, \ldots, C_n$ must also appear in $L_1, \ldots, L_m$. In addition, the evaluation of a policy rule should not depend on a future condition. Therefore, it is required that for each time argument T_i of the condition L_i, it holds that $C_1, \ldots, C_n \models T_i \leq T$, and if L_i's predicate is *happens*, then L_i, $C_1, \ldots, C_n \models T_i < T$.

Example 7.1. *"If a file was declassified at least 3 days ago then it is allowed to be deleted now."*

$$permitted(X, file(Y), delete, T) \leftarrow$$
$$do(Z, file(Y), declassify, T1), T1 \leq T - 3days.$$

Type	**Predicate**	**Description**
Regulatory Input ($\mathcal{L}_I^\Pi$)	$req(Su, Ta, Ac, T)$	the request of an action Ac by a subject Su on the target Ta at time T
Regulatory Output ($\mathcal{L}_O^\Pi$)	$do(Su, Ta, Ac, T)$	the execution of a system action
	$deny(Su, Ta, Ac, T)$	the denial of a system action
Regulatory State ($\mathcal{L}_S^\Pi$)	$permitted(Su, Ta, Ac, T)$	a positive permission
	$denied(Su, Ta, Ac, T)$	a negative permission
	$obl(Su, Ta, Ac, T_s, T_e, T)$	an obligation is initiated at time T for the subject Su to perform an action Ac on the target Ta between T_s and T_e (exclusive)
	$fulfilled(Su, Ta, Ac, T_s, T_e, T)$	the obligation has been fulfilled at time T (e.g., the action has been performed within the time frame)
	$violated(Su, Ta, Ac, T_s, T_e, T)$	the obligation has been violated at time T (e.g., the action has not been performed after the expiration time)
	$cease\_obl(Su, Ta, Ac, T_{init}, T_2, T_e, T)$	the obligation has been ceased at time T (e.g., it has been either fulfilled or violated). T_{init} is the time when the obligation instance was initiated.
Non-Regulatory Event ($\mathcal{L}_E^D$)	$occurred(E, T)$	an exogenous event to the system
Non-Regulatory State ($\mathcal{L}_S^D$)	$holds(F, T)$	a fluent F holds at time T
	$initially(F)$	the fluent F initially holds (i.e., at time 0)
	$happens(E, T)$	an event E takes place at time T
	$clipped(T1, F, T)$	the fluent F has been "clipped" (e.g., ceased by an event) between $T1$ and T
	$initiates(E, F, T)$	the event E initiates a fluent F at time T
	$terminates(E, F, T)$	the event E terminates a fluent F at time T
Static ($\mathcal{L}_{Stat}$)		user defined predicates used for describing static properties/relationships that do not change between states.

Table 7.1: Predicates in $\mathcal{L} = \mathcal{L}_I^\Pi \cup \mathcal{L}_O^\Pi \cup \mathcal{L}_S^\Pi \cup \mathcal{L}_E^{\mathcal{G}^d} \cup \mathcal{L}_S^D$: The arguments Su, Ta, Ac, F and E are of the sorts `subject`, `target`, `action`, `fluent` and `event`, respectively, and the arguments T, $T1$, T_s, T_e, T_{init} are of the sort `time`.

Example 7.2. *"If a node has already broadcasted one message within the past 5 seconds, then it is not allowed to broadcast any message now."*

$$\begin{aligned}&\mathit{denied}(\mathit{node}(X), \mathit{message}(Y), \mathit{broadcast}, T) \leftarrow\\ &\quad \mathit{do}(\mathit{node}(X), \mathit{message}(Z), \mathit{broadcast}, T1), T1 \geq T - 5\mathit{seconds}, T1 \leq T.\end{aligned}$$

An obligation (policy) rule describes what action (by whom and to whom) must be performed within a specific time frame, once the specified conditions are satisfied. It is modelled similarly to the authorisation rules but with *obl* as the head:

$$\mathit{obl}(Su, Ta, Ac, T_s, T_e, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.$$

Example 7.3. *"A client node must provide a digital certificate within 5 seconds of establishing a connection to the server."*

$$\begin{aligned}&\mathit{obl}(\mathit{client}(X), \mathit{server}(Y), \mathit{send}(\mathit{certificate}(C)), T_s, T_e, T) \leftarrow\\ &\quad \mathit{do}(\mathit{client}(X), \mathit{server}(Y), \mathit{connect}, T1),\\ &\quad T = T1 + 0.01\mathit{second}, T_s = T, T_e = T_s + 5\mathit{seconds}.\end{aligned}$$

There are three rules describing the different status for an obligation:

$$\begin{aligned}&fulfilled(Su, Ta, Ac, T_s, T_e, T) \leftarrow\\ &\quad \mathit{obl}(Su, Ta, Ac, T_s, T_e, T_{init}),\\ &\quad \mathit{do}(Sub, Tar, Act, T'),\\ &\quad \neg \mathit{cease\_obl}(Su, Ta, Ac, T_{init}, T_s, T_e, T'),\\ &\quad T_{init} \leq T_s \leq T' < T_e, T' < T.\end{aligned} \tag{7.1}$$

$$\begin{aligned}
&\mathit{violated}(Su, Ta, Ac, T_s, T_e, T) \leftarrow \\
&\quad \mathit{obl}(Su, Ta, Ac, T_s, T_e, T_{init}), \\
&\quad \neg \mathit{cease\_obl}(Su, Ta, Ac, T_{init}, T_s, T_e, T_e), \\
&\quad T_{init} \leq T_s < T_e \leq T.
\end{aligned} \tag{7.2}$$

$$\begin{aligned}
&\mathit{cease\_obl}(Su, Ta, Ac, T_{init}, T_s, T_e, T) \leftarrow \\
&\quad \mathit{do}(Su, Ta, Ac, T'), T_s \leq T' < T \leq T_e.
\end{aligned} \tag{7.3}$$

$$\begin{aligned}
&\mathit{cease\_obl}(Su, Ta, Ac, T_{init}, T_s, T_e, T) \leftarrow \\
&\quad \mathit{do}(Admin, Su, \mathit{revoke}(Su, Ta, Ac, T_s, T_e), T'), T_s \leq T' < T \leq T_e.
\end{aligned} \tag{7.4}$$

Rule 7.3 and Rule 7.4 say that for any obligation instance created at T_{init}, if its regulated action (e.g., Ac by Su on Ta) is performed at T', or it is revoked at T', then it is considered to be ceased within time interval $(T', T_e]$, where $T' \in [T_s, T_e)$. Rule 7.1 says that if the regulated action of a "not-yet-ceased" obligation instance (created at T_{init}) is performed at time T' within the specified time interval $[T_s, T_e]$, then the obligation instance is considered to be fulfilled after T'. On the other hand, Rule 7.2 says the obligation instance is considered to be violated from T_e if its regulated action has not been performed between T_s and T_e (exclusive).

The behaviour of the PEP may vary from system to system, and can be modelled using *policy regulatory rules*, which are the same as authorisation rules but with either *do* or *deny* as the head. For example, the default *basic availability* policy regulatory rule says *"an action will be executed if it is permitted by a policy rule"* can be specified as:

$$\begin{aligned}
&\mathit{do}(Su, Ta, Ac, T) \leftarrow \\
&\quad \mathit{req}(Su, Ta, Ac, T), \mathit{permitted}(Su, Ta, Ac, T).
\end{aligned} \tag{7.5}$$

and the default *positive/negative availability* policy regulatory rules say *"an action will be* *executed as long as there is no policy rule forbidding it"* are specified as:

$$do(Su, Ta, Ac, T) \leftarrow \\ \quad req(Su, Ta, Ac, T), \neg denied(Su, Ta, Ac, T). \tag{7.6}$$

$$deny(Su, Ta, Ac, T) \leftarrow \\ \quad req(Su, Ta, Ac, T), denied(Su, Ta, Ac, T). \tag{7.7}$$

Note that sometimes these rules are called *blanket rules*, which can be used for specifying *default permissions* for those actions without explicit authorisation policy rules or for resolving permission conflicts.

Domain Description

The system domain is dynamic, in the sense that the values of its properties may persist by default and may change due to the occurrence of relevant events. The dynamicity of the domain is modelled using a simplified version of the Event Calculus (SEC) [Kow92, Sha99]:

$$holds(F, T) \leftarrow \\ \quad initially(F), \\ \quad \neg clipped(0, F, T), T > 0. \tag{7.8}$$

$$holds(F, T) \leftarrow \\ \quad happens(E1, T1), initiates(E1, F, T1), \\ \quad \neg clipped(T1, F, T), T1 < T. \tag{7.9}$$

$$\begin{aligned} &clipped(T1, F, T) \leftarrow \\ &\quad happens(E2, T2), terminates(E2, F, T2), \\ &\quad T1 < T2 < T. \end{aligned} \tag{7.10}$$

There are two types of events that can affect domain properties: non-regulatory (or exogenous, represented with *occurred*), and regulatory output (represented with *do* and *deny*). The following three rules generalise them:

$$\begin{aligned} &happens(occ(Ev), T) \leftarrow \\ &\quad occurred(Ev, T). \end{aligned} \tag{7.11}$$

$$\begin{aligned} &happens(do(Su, Ta, Ac), T) \leftarrow \\ &\quad do(Su, Ta, Ac, T). \end{aligned} \tag{7.12}$$

$$\begin{aligned} &happens(deny(Su, Ta, Ac), T) \leftarrow \\ &\quad deny(Su, Ta, Ac, T). \end{aligned} \tag{7.13}$$

Finally, the effect of the events on domain properties can be described using the domain dependent rules:

$$initiates(E, F, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.$$

$$terminates(E, F, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_n.$$

where $L_1, \ldots, L_m, C_1, \ldots, C_n$ are the same as those in an authorisation rule, except that each L_i must have a predicate in $\{holds\} \cup \mathcal{L}_{stat}$.

7.2.3 Example Policy Analysis Tasks

Given a target system, the *initial state* and history of exogenous events can be defined as a set $\mathcal{H}$ of facts *initially* and *occurred*. Thus, the total system specification Π is $\mathcal{P} \cup \mathcal{P}_{obl\_aux} \cup \mathcal{P}_{reg} \cup$

$\mathcal{EC} \cup \mathcal{D} \cup \mathcal{H} \cup \mathcal{S}$, where $\mathcal{P}$ is the set of authorisation policy and obligation policy rules, $\mathcal{P}_{obl\_aux}$ is the fixed auxiliary rules for obligation (i.e., Rules 7.1–7.4), $\mathcal{P}_{reg}$ is the set of policy regulatory rules describing the behaviour of PEP/PDP (e.g., Rule 7.5 or Rule 7.6 and Rule 7.7), $\mathcal{EC}$ is the set of domain independent EC rules (i.e., Rules 7.8–7.10 and Rules 7.11–7.13), $\mathcal{D}$ is the set of event effect rules and $\mathcal{S}$ is the set of static state properties expressed in $\mathcal{L}_{stat}$. The existing framework requires that Π is *locally stratified*.

The operational model of the system does not allow concurrent actions. This is expressed as an integrity constraint:

$$\leftarrow happens(E1, T), happens(E2, T), E1 \neq E2. \qquad (7.14)$$

A policy analysis task can be informally defined as follows. Given a system specification, the property (of the set of policies) is expressed as a query, which is to be proven or explained. In the case of explaining the query, a possible explanation is a sequence of system actions that can lead the system from its initial state to a goal state that satisfies the property (described by the query). A sequence of system actions can be represented as a sequence of action requests. The corresponding *system trace*, which is a series of system states, can be derived from the sequence of action requests and the system specification.

For example, a modality conflict *"is there a case where a subject is obliged to perform some action but does not have the permission?"* can be expressed as the following query (note that the first two goals in the query ensure that the obligation instance has not yet been fulfilled or revoked):

$$\begin{aligned}
&\exists Su, Ta, Ac, T_s, T_e, T_i, T.[\\
&\quad obl(Su, Ta, Ac, T_s, T_e, T_i) \wedge\\
&\quad \neg cease\_obl(Su, Ta, Ac, T_i, T_s, T_e, T) \wedge\\
&\quad denied(Su, Ta, Ac, T) \wedge T_s < T\\
&]
\end{aligned}$$

A static SoD analysis between two roles ($role_1$ and $role_2$) can be expressed as:

$$\begin{aligned}
&\exists Su, Ta, T.[\\
&\quad permitted(Su, Ta, assign(role\_1), T) \wedge\\
&\quad permitted(Su, Ta, assign(role\_2), T)\\
&]
\end{aligned}$$

If any system trace can be found for the above queries, it is an indication of a *flaw* in the policies.

In the existing formal policy framework [CLM+09], policy analysis tasks are solved by abduction. For example, let the abductive framework be $\langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$, where Π is the system specification, $\mathcal{AB} = \{req\}$, and $\mathcal{IC}$ contains only the integrity constraint Rule 7.14. Then given a policy property query $\mathcal{Q}$, the abductive (policy analysis) task is to find a set of system requests Δ, such that $\Pi \cup \Delta \models \mathcal{Q}$ and $\Pi \cup \Delta \models \mathcal{IC}$, where $\models$ is the logical entailment under a selected semantics. Initially, the framework proposed the use of *stable model* semantics. However, since in practice ASystem is used for solving the actual analysis tasks, the (Fitting's) *3-valued* semantics is used instead.

7.3 Extended Framework for Distributed Policy Analysis

As briefly described in Section 7.1, a coalition network (e.g. Figure 7.3) may consist of entities belonging to different parties (e.g., the US sensor fabric control server and the UK paramedic team PDAs). Each entity abstracted as a node has its own PEP/PDP and its private policies regulating operations performed within it (e.g., authorisation policies of the US control server regulating the access of sensing data), and may have private knowledge of its local domain state (e.g., live intelligence in the area collected by the UK PDAs). In this case, the overall network system consists of multiple PEPs/PDPs, and the policies and private knowledge cannot be centralised. Consequently, policy analysis tasks cannot be performed using (centralised) abductive reasoning.

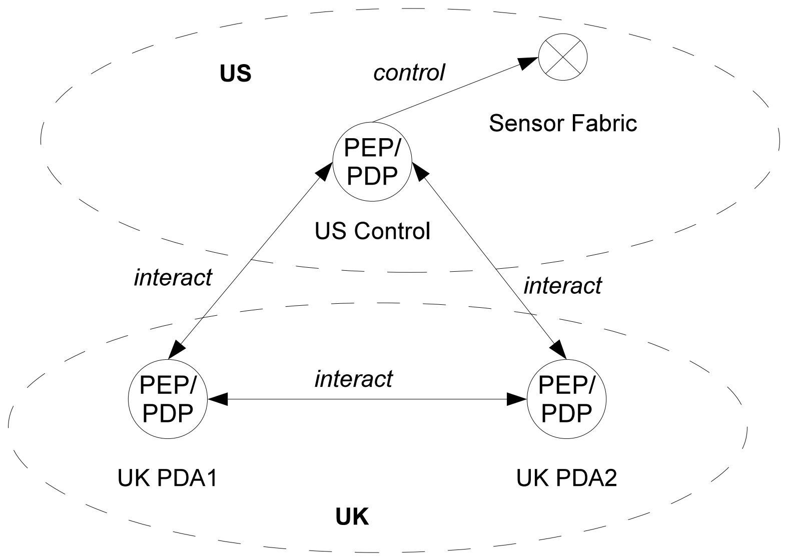


Figure 7.3: Coalition Network with Multiple PEPs/PDPs

7.3.1 Extending the Operational Model and the Language

Despite the existence of multiple PEPs/PDPs, if we only consider the serialised execution [1] of the overall system, i.e., there are no concurrent events and only one PEP/PDP can be active

[1]For any execution of the overall system with concurrent events, a serialised version of execution can be obtained by adopting logical clock synchronous protocols (e.g., Lamport Timestamps [Lam78]).

at any time, the operational model (Figure 7.4) of the overall system is very similar to that in Figure 7.2 for a centralised PDP/PEP.

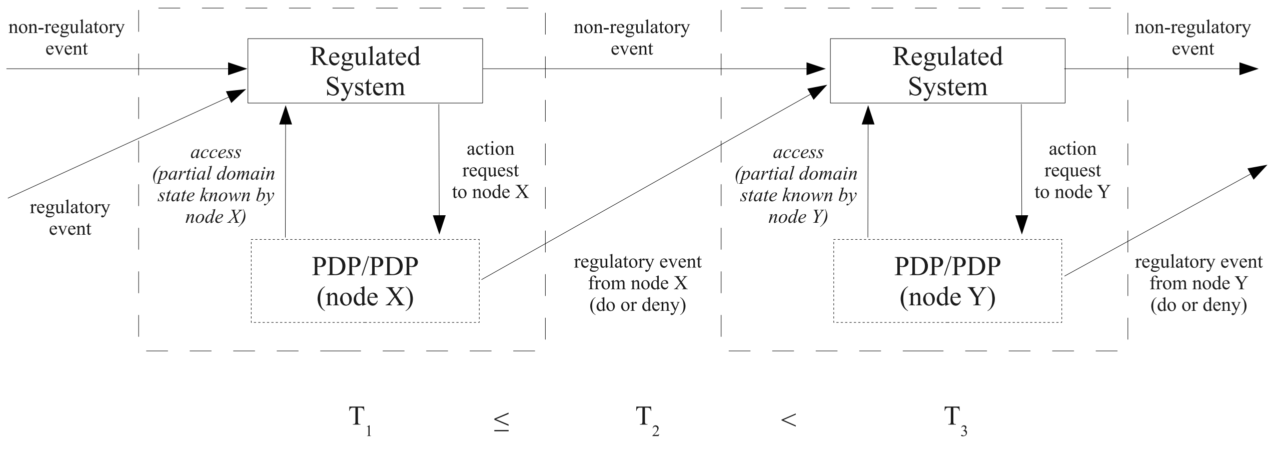


Figure 7.4: Operational Model with Multiple PEPs/PDPs

The main difference between the two operational models is that in the former model, at each PEP/PDP step by a node, say X, only the request at X (i.e., the action that should be performed at X if permitted) can be intercepted by X's PEP, and the policy evaluation by X's PDP can only use the policies and local domain knowledge of X, plus some *shared knowledge*. There are two types of shared knowledge. One is the common static knowledge known to everyone (e.g., number of days in a week, the domain independent EC rules), and the other is a subset of local knowledge belonging to some node Y (i.e., its *originator*) that Y is willing to share with others (e.g., the US control center may disclose the types of sensor data available). The common static knowledge can be duplicated at each node to form part of its local domain knowledge. Shared knowledge cannot be duplicated, as it may be dynamic and depend on the local state of its originator (e.g., the types of sensor data available may change over time). Thus, the existing logic language also has to be extended for the new operational model with two requirements:

1. *location awareness*, i.e., to allow the indication of where an event takes place and the originator of shared knowledge, and

2. the distinction between shared and local (i.e., private) knowledge.

The DAREC^2 language has all the necessary features to achieve these requirements: private knowledge is expressed using local predicates (and private atoms), whereas shared knowledge is expressed using public predicates (and askable atoms). Instead of being modelled as a single abductive framework, a system with the new operational model can be modelled as a DAREC^2 global abductive framework $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$, where each $\mathcal{F}_i \in \widehat{\mathcal{F}}$ is the abductive framework for the node $i \in \Sigma$.

The background knowledge Π_i of a node i's abductive framework is a specification of the sub-system for i defined as follows. First, policies and behaviour of the PEP/PDP are private to i. Therefore, each of the policy rules $\mathcal{P}_i$ and the policy regulatory rules $\mathcal{P}_{reg_i}$ has the form (i.e., its head is a private atom):

$$[\mathit{permitted}/\mathit{denied}/\mathit{do}/\mathit{deny}](Su, Ta, Ac, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_m.$$

or

$$\mathit{obl}(Su, Ta, Ac, T_s, T_e, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_m.$$

but each condition L_i can be either a private literal (i.e., positive or negative private atom) like in the old framework or an askable literal (i.e., positive or negative askable atom) of the form $\mathit{holds}(F, T)@ID$ or $\mathit{happens}(E, T)@ID$.

Secondly, the effects of an event on the local domain state of i need to be known only by i. Thus, each rule in $\mathcal{D}_i$ has the form (i.e., its head is a private atom):

$$[\mathit{initiates}/\mathit{terminates}](E, F, T) \leftarrow L_1, \ldots, L_m, C_1, \ldots, C_m.$$

and similar to the policy rules each L_i can be a private literal or an askable literal of the form $\mathit{holds}(F, T)@ID$. The initial state of i also needs to be known by i only, and thus it is described by a set of private atoms $\mathit{initially}(F)$.

Thirdly, the occurrence of a (non-)regulatory event at node i is expressed with a private atom of the form $occured(E,T)$ or $do(Su,Ta,Ac,T)$ or $deny(Su,Ta,Ac,T)$. However, it is possible that an event of i's may affect the local domain state of another node (e.g., the action of getting sensor data executed at the US control center may cause the requesting UK PDA to have sensor data). Therefore, it is necessary that i "exposes" this event as shared knowledge so that other nodes can reason about it with their local domain states. This can be done through the *knowledge exposure* rules, each of which has an askable atom as the head and a private atom as the body (i.e., turning a piece of private knowledge to be shared knowledge), e.g.,:

$$\begin{aligned}&happens(do(Su, video\_sensor, get\_data), T)@us\_control \leftarrow\\&\quad happens(do(Su, video\_sensor, get\_data, T).\end{aligned}$$

Similarly, a local domain property (i.e., fluent) that i wishes to expose as shared knowledge can be declared using a similar knowledge exposure rule:

$$\begin{aligned}&holds(sensor\_status(Sensor, Status), T)@us\_control \leftarrow\\&\quad holds(sensor\_status(Sensor, Status), T).\end{aligned}$$

Note that these "exposed" atoms can be used as conditions in a private policy rule or an event effect rule in any other node.

Finally, to allow node i to reason about (the effects of) events exposed by other nodes to its local domain state, the following EC rules are added to $\mathcal{EC}$ (Rules 7.8–7.10 and Rules 7.11–7.13) resulting in $\mathcal{EC}'$:

$$\begin{aligned}&holds(F,T) \leftarrow\\&\quad happens(E1,T1)@ID, ID \neq i,\\&\quad initiates(E1,F,T1),\\&\quad \neg clipped(T1,F,T), T1 < T.\end{aligned} \tag{7.15}$$

$$
\begin{aligned}
& \mathit{clipped}(T1, F, T) \leftarrow \\
& \quad \mathit{happens}(E2, T2)@ID, ID \neq i, \\
& \quad \mathit{terminates}(E2, F, T2), \\
& \quad T1 < T2 < T.
\end{aligned}
\tag{7.16}
$$

The set $\mathcal{EC}'$ of EC domain independent rules, and the set of auxiliary obligation status rules (Rules 7.1–7.4) are the same in all the nodes, and are assumed to be duplicated as the local (or private) knowledge at i and at other nodes (for any other node j, replace $ID \neq i$ with $ID \neq j$ for Rule 7.15 and Rule 7.16).

In summary, the background knowledge of node i is $\Pi_i = \Pi_i^{private} \cup \Pi_i^{exposure}$, where $\Pi_i^{private} = \mathcal{P}_i \cup \mathcal{P}_{reg_i} \cup \mathcal{D}_i \cup \mathcal{H}_i \cup \mathcal{S}_i \cup \mathcal{P}_{obl\_aux} \cup \mathcal{EC}'$, and $\Pi_i^{exposure}$ is the set of knowledge exposure rules for i. Let us remark that every rule in $\Pi_i^{private}$ has a private atom as the head, whereas every rule in $\Pi_i^{exposure}$ has an askable atom as the head.

In the old framework, there is only one abducible predicate req and it is agreed by all the nodes. Hence, $\mathcal{AB}_i = \{req\}$. In addition, no concurrent event is allowed. Therefore, $\mathcal{IC}_i$ contains the following

$$\leftarrow \mathit{req}(Su, Ta, Ac, T), \mathit{happens}(E, T)@ID, ID \neq i. \tag{7.17}$$

$$\leftarrow \mathit{req}(Su, Ta, Ac, T), \mathit{happens}(E, T), \neg \mathit{regulated}(Su{:}Ta{:}Ac, E). \tag{7.18}$$

where $\mathit{regulated}$ is defined in $\mathcal{S}_i$ as:

$$\mathit{regulated}(Su{:}Ta{:}Ac, \mathit{do}(Su1, Ta1, Ac1)) \leftarrow Su{:}Ta{:}Ac = Su1{:}Ta1{:}Ac1.$$

$$\mathit{regulated}(Su{:}Ta{:}Ac, \mathit{deny}(Su1, Ta1, Ac1)) \leftarrow Su{:}Ta{:}Ac = Su1{:}Ta1{:}Ac1.$$

7.3.2 Distributed Policy Analysis

Given a system with multiple PEPs/PDPs as a global abductive framework $\left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$, the same set of policy analysis tasks in [CLM+09] can be performed in a distributed fashion by using DAREC^2. That is, given the same query $\mathcal{Q}$ as for centralised policy analysis, we use DAREC^2 to find a sequence of system action requests Δ such that $\bigcup_{i \in \Sigma} \mathcal{F}_i \cup \Delta \models \mathcal{Q}$ and $\bigcup_{i \in \Sigma} \mathcal{F}_i \cup \Delta \models \bigcup_{i \in \Sigma} \mathcal{IC}_i$ under the 3-valued semantics.

To illustrate distributed policy specification and distributed policy analysis, let us consider the following example scenario involving Role-Based Access Control (RBAC) and Separation of Duties (SoD).

Example 7.4. *In a US-UK joint operation, a bomb squad and two commanders (one from each country) are involved. To be able to use highly dangerous explosive to destroy sensitive targets, there must be a* bomb setter *(i.e., a squad member with the* setter *role) and a* bomb detonator *(i.e., a squad member with the* detonator *role). Assignment of roles can only be performed by the commanders, and is performed dynamically during the operation. SoD is implemented in two levels: 1. no one is allowed to arm and then detonate a bomb; 2. no commander can manage (e.g., (un)assign) both roles. The second SoD can be implemented explicitly be forcing each commander to manage one role throughout the operation. Then the first SoD must be guaranteed by the policies of role assignments by the two commanders, which may not be disclosed to each other, as their conditions may depend on private information (metric). Before the operation starts, it is required to check that with the two commanders (and their policies), the squad team can always fulfil their objectives by using bombs (i.e., modality conflict and behaviour simulation), and the separation of duties property is ensured.*

In this example, there are three entities (nodes), each of which has its own PEP/PDP, i.e., the commanders (*us_cmd* and *uk_cmd*) regulate role assignment actions with their policies, and *squad* employs RBAC for controlling the usage of explosive.

Distributed System Specification

Let us elaborate the example further. For simplicity, we only describe the policies and domain properties relevant to the aforementioned two analysis tasks.

Node *squad* has local knowledge about its members and the commanders, e.g., $\mathcal{H}_{squad}$ contains:

$$initially(member(X)) \leftarrow X \in \{alpha, beta\}. \tag{7.19}$$

$$initially(commander(X)) \leftarrow X \in \{us\_cmd, uk\_cmd\}. \tag{7.20}$$

By enforcing RBAC, a member can set up/detonate a bomb if and only if he is assigned the corresponding role, e.g., $\mathcal{P}_{squad}$ contains:

$$\begin{aligned} &permitted(X, bomb, arm, T) \leftarrow \\ &\quad holds(member(X), T), holds(hasRole(X, setter), T). \end{aligned} \tag{7.21}$$

$$\begin{aligned} &permitted(X, bomb, detonate, T) \leftarrow \\ &\quad holds(member(X), T), holds(hasRole(X, detonator), T). \end{aligned} \tag{7.22}$$

The role assignment actions (performed by a commander) can affect its local domain state, e.g., $\mathcal{D}_{squad}$ contains:

$$\begin{aligned} &initiates(do(Su, Ta, assign(Role)), hasRole(Ta, Role), T) \leftarrow \\ &\quad holds(commander(Su), T), holds(member(Ta), T). \end{aligned} \tag{7.23}$$

Suppose the UK commander, *uk_cmd*, can assign the *detonator* role to soldiers. It uses the basic availability policy regulatory rule (7.5), i.e., *"a soldier can be assigned a role if he is permitted by at least one local policy"*. In addition, it maintains a local *credential level* database of the soldiers, and has a positive authorisation policy *"the detonator role can be assigned to a* *soldier with credential level greater than 4"*. Thus, $\mathcal{P}_{reg_{uk\_cmd}}$ contains

$$do(Su, Ta, Ac, T) \leftarrow \\ req(Su, Ta, Ac, T), permitted(Su, Ta, Ac, T). \tag{7.24}$$

and $\mathcal{P}_{uk\_cmd}$ contains

$$permitted(uk\_cmd, Ta, assign(detonator), T) \leftarrow \\ holds(credential\_level(Ta, L), T), L > 4. \tag{7.25}$$

and $\mathcal{H}_{uk\_cmd}$ contains

$$initially(credential\_level(alpha, 3)). \tag{7.26}$$

$$initially(credential\_level(beta, 5)). \tag{7.27}$$

and $\Pi^{exposure}_{uk\_cmd}$ contains

$$do(uk\_cmd, Ta, assign(Role), T)@uk\_cmd \leftarrow \\ do(uk\_cmd, Ta, assign(Role), T). \tag{7.28}$$

Now suppose the US commander, *us_cmd*, can assign the *setter* role to soldiers. It uses the positive availability rule (7.6), i.e., *"a soldier can be assigned a role if it is not prohibited by any local policy"*. In addition, it *tries* to implement SoD by having a local negative authorisation policy *"a soldier cannot be assigned to the setter role if it has been assigned the conflicting detonator role"*. Thus, $\mathcal{P}_{reg_{us\_cmd}}$ contains

$$do(Su, Ta, Ac, T) \leftarrow \\ req(Su, Ta, Ac, T), \neg denied(Su, Ta, Ac, T). \tag{7.29}$$

and $\mathcal{P}_{us\_cmd}$ contains

$$\begin{aligned}
&denied(us\_cmd, Ta, assign(setter), T) \leftarrow \\
&\quad conflicting(setter, Role), \\
&\quad do(Cmd, Ta, assign(Role), T1)@Cmd, T1 < T.
\end{aligned} \tag{7.30}$$

and $\mathcal{S}_{us\_cmd}$ contains

$$conflicting(setter, detonator). \tag{7.31}$$

and $\Pi^{exposure}_{us\_cmd}$ contains

$$\begin{aligned}
&do(us\_cmd, Ta, assign(Role), T)@us\_cmd \leftarrow \\
&\quad do(us\_cmd, Ta, assign(Role), T).
\end{aligned} \tag{7.32}$$

Finally, the global abductive framework for the whole system is $\mathcal{F}^{dis} = \left\langle \Sigma, \widehat{\mathcal{F}} \right\rangle$, where $\Sigma = \{squad, uk\_cmd, us\_cmd\}$, $\widehat{\mathcal{F}} = \{\mathcal{F}_{squad}, \mathcal{F}_{uk\_cmd}, \mathcal{F}_{us\_cmd}\}$ and $\mathcal{AB}_{squad} = \mathcal{AB}_{uk\_cmd} = \mathcal{AB}_{us\_cmd} = \{req\}$.

Example Distributed Analysis Tasks

In this scenario, there are two policy analysis tasks: to check if the squad team can use bombs, and to check if the SOD constraint can be violated.

The first task can be seen as a type of modality conflict check or behaviour simulation, and can be initiated by submitting a query (*"Can two different members of the squad be allowed to arm and detonate bombs respectively?"*):

$$
\begin{array}{l}
\exists X, Y, T.[\\
\quad permitted(X, bomb, arm, T) \wedge\\
\quad permitted(Y, bomb, detonate, T) \wedge\\
\quad X \neq Y\\
]
\end{array}
$$

to node *squad*.

The second task is SoD analysis, and can be initiated by submitting the query (*"Can any member of the squad be allowed to arm and detonate bombs at the same time?"*):

$$
\begin{array}{l}
\exists X, T.[\\
\quad permitted(X, bomb, arm, T) \wedge\\
\quad permitted(X, bomb, detonate, T)\\
]
\end{array}
$$

to node *squad*.

Let us fix the sort of `time` to be integers ranging between 0 and 5. Below is an abstracted DAREC^2 derivation for the first query[2]:

1. *squad* has the initial state Θ_0, which has

$$
\mathcal{G}_0 = \left\{ \begin{array}{l} permitted(X, bomb, arm, T),\\ permitted(Y, bomb, detonate, T),\\ X \neq Y \end{array} \right\}
$$

[2]Note that in the actual DAREC^2 system specification, constants *alpha*, *beta*, *uk_cmd* and *us_cmd* are replaced with their unique integer identifiers, e.g., $1, 2, 3, 4$, respectively, since the built-in finite domain constraint solver can deal with integers only.

as the goals, and all other state stores are empty.

2. After the local computation by *squad* (each private atom goal is resolved first with $\mathcal{P}_{squad}$, and then with $\mathcal{EC}$, and then with $\mathcal{H}_{squad} \cup \mathcal{D}_{squad}$), a new state Θ_1 is obtained, which contains the delayed goals of

$$\mathcal{G}_1^d = \left\{ \begin{array}{l} do(Su1, X, assign(setter), T1)@Su1, \\ do(Su2, Y, assign(detonator), T2)@Su2 \end{array} \right\}$$

and collected the inequalities and constraints of

$$\mathcal{E}_1 \cup \mathcal{C}_1 = \left\{ \begin{array}{l} T \in [0..5], T1 \in [0..5], T2 \in [0..5], T1 < T, T2 < T, \\ X \in \{alpha, beta\}, Y \in \{alpha, beta\}, X \neq Y \\ Su1 \in \{us\_cmd, uk\_cmd\}, Su2 \in \{us\_cmd, uk\_cmd\} \end{array} \right\}$$

3. *squad* passes Θ_1 to *uk_cmd*, and *uk_cmd* performs local computation on Θ_1. *uk_cmd* first tries to bind $Su1$ to itself for the first goal, and then resolves the goal (first with $\mathcal{P}_{reg_{uk\_cmd}}$, then with $\mathcal{P}_{uk\_cmd}$, then with $\mathcal{H}_{uk\_cmd}$, then with $\mathcal{IC}_{uk\_cmd}$). This binds X to *beta*, as *alpha* does not have the right credential level. Note that this binding causes the binding of Y to *alpha*, due to the inequality $X \neq Y$ and the constraint $Y \in \{alpha, beta\}$. A new state Θ_2 contains the remaining goals

$$\mathcal{G}_2 = \left\{ \begin{array}{l} do(Su2, alpha, assign(setter), T2)@Su2, \\ \forall E. \leftarrow happens(E, T1)@squad \\ \forall E. \leftarrow happens(E, T1)@us\_cmd \end{array} \right\}$$

and with the collected abducible

$$\Delta_2 = \left\{ \; req(uk\_cmd, beta, assign(detonator), T1) \; \right\}$$

and the collected abducible denials

$$\mathcal{N}_2 = \left\{ \begin{array}{l} \forall Su, Ta, Ac, T. \leftarrow req(Su, Ta, Ac, T1), \\ \qquad Su{:}Ta{:}Ac \neq \mathit{uk\_cmd}{:}\mathit{beta}{:}\mathit{assign}(\mathit{detonator}) \end{array} \right\}$$

uk_cmd then tries to bind $Su2$ to itself for the first goal in Θ_2, but eventually fails to resolve it. Therefore, a new inequality $Su2 \neq \mathit{uk\_cmd}$ is added to Θ_2 (this results in binding $Su2$ to *us_cmd* as there is already a constraint $Su2 \in \{\mathit{uk\_cmd}, \mathit{us\_cmd}\}$ in Θ_2), which is then passed to *us_cmd*.

4. *us_cmd* performs local computation for the received state. It tries to resolve the goal $do(\mathit{us\_cmd}, \mathit{alpha}, \mathit{assign}(\mathit{setter}), T2)$, and succeeds by abducing $req(\mathit{us\_cmd}, \mathit{alpha},$ $\mathit{assign}(\mathit{setter}), T2$ and collects $T2 < T1$. It then tries to reduce the denial goal $\forall E. \leftarrow$ $\mathit{happens}(E, T1)@\mathit{us\_cmd}$, and collects $T2 \neq T1$. The new state Θ_3 has remaining goals of

$$\mathcal{G}_3 = \left\{ \begin{array}{l} \forall E. \leftarrow \mathit{happens}(E, T2)@\mathit{squad} \\ \forall E. \leftarrow \mathit{happens}(E, T2)@\mathit{uk\_cmd} \\ \forall E. \leftarrow \mathit{happens}(E, T1)@\mathit{squad} \end{array} \right\}$$

and abducible denials of

$$\mathcal{N}_3 = \left\{ \begin{array}{l} \forall Su, Ta, Ac, T. \leftarrow req(Su, Ta, Ac, T1), \\ \qquad Su{:}Ta{:}Ac \neq \mathit{uk\_cmd}{:}\mathit{beta}{:}\mathit{assign}(\mathit{detonator}) \\ \forall Su, Ta, Ac, T. \leftarrow req(Su, Ta, Ac, T2), \\ \qquad Su{:}Ta{:}Ac \neq \mathit{us\_cmd}{:}\mathit{alpha}{:}\mathit{assign}(\mathit{setter}) \end{array} \right\}$$

and assumed abducibles of

$$\Delta_3 = \left\{ \begin{array}{l} req(\mathit{uk\_cmd}, \mathit{beta}, \mathit{assign}(\mathit{detonator}), T1), \\ req(\mathit{us\_cmd}, \mathit{alpha}, \mathit{assign}(\mathit{setter}), T2) \end{array} \right\}$$

with the collected constraints and inequalities of

$$\mathcal{E}_3 \cup \mathcal{C}_3 = \left\{ \begin{array}{l} T \in [0..5], T1 \in [0..5], T2 \in [0..5], \\ T1 < T, T2 < T, T1 \neq T2 \end{array} \right\}$$

.

5. Θ_3 is passed to *uk_cmd* and *squad* for the reduction of $\mathcal{G}_3$. This succeeds without changing the rest of Θ_3.

Finally, for this query DAREC^2 can return one answer:

$$\begin{array}{l} (\exists T1, T2, T.) \\ \quad req(uk\_cmd, beta, assign(detonator), T1), \\ \quad req(us\_cmd, alpha, assign(setter), T2), \\ \quad T1 < T, T2 < T, T1 \neq T2. \end{array}$$

Note that during the distributed policy analysis computation, the *uk_cmd*'s credential level database and each party's policy rules as well as the policy regulatory rules are never disclosed to others. This conforms to the DAREC^2's confidential reasoning property.

For the second query, DAREC^2 can also compute one answer:

$$\begin{array}{l} \exists T1, T2, T.[\\ \quad req(uk\_cmd, beta, assign(detonator), T1) \wedge \\ \quad req(us\_cmd, beta, assign(setter), T2) \wedge \\ \quad T1 < T \wedge T2 < T, T2 < T1 \\ ] \end{array}$$

Note that $T2 < T1$ is collected after *us_cmd* resolves its goal (see Step 4) with $\mathcal{P}_{us\_cmd}$ (which allows *us_cmd* to assign the setter role as long as the target has not been assigned the detonator role). This indicates a violation of the SoD constraint, which could occur if *beta* first requests the *setter* role from *us_cmd*, and then requests the *detonator* role from *uk_cmd* (i.e, $T2 < T1$). Therefore, the two commanders should be informed and need to revise their local policies.

7.4 Discussions

Interaction between PDPs

Figure 7.4 reflects that during a PEP/PDP step by some node, its PDP cannot interact with the PDPs of other nodes. In practice, it is possible that the decision for a permission by one node depends on the decision for another permission by another node. For example, suppose there is a policy rule in *us_control* such that *"A UK PDA can get sensor data from the sensor fabric if the PDA allows the control center to listen to all of its communication."*. This could be expressed as follows if the language were extended to allow a regulatory state predicate to be shared (e.g., *permitted* and *denied*):

$$\begin{aligned}&permitted(X, sensor\_fabric, get\_data, T)@us\_control \leftarrow \\ &\quad holds(owner(X, uk), T), type(X, pda), \\ &\quad permitted(us\_control, X, listen\_communication, T)@X.\end{aligned}$$

DAREC^2 can still be used to analyse such policies. However, there is a potential problem due to the fact that different PEPs/PDPs may have their own policy regulatory rules describing their behaviour. Suppose *pda*1 is a UK PDA and its PEP is described by the positive availability rule:

$$\begin{aligned}&do(Su, Ta, Ac, T) \leftarrow \\ &\quad req(Su, Ta, Ac, T), \neg denied(Su, Ta, Ac, T).\end{aligned}$$

i.e., *an action is executed as long as no local policy rule prohibits it.* Then the action $do(us\_control, pda1, listen\_communication, T)$ should be allowed to be executed at $pda1$, but the evaluation of $permitted(us\_control, pda1, listen\_communication, T)@pda1$ would give answer `no`, if a simple knowledge exposure rule:

$$permitted(Su, Ta, Ac, T)@pda1 \leftarrow permitted(Su, Ta, Ac, T).$$

was used by $pda1$. In this particular example, the correct knowledge exposure rule that should be used instead is:

$$permitted(Su, Ta, Ac, T)@pda1 \leftarrow \neg denied(Su, Ta, Ac, T).$$

Therefore, if we allow PDPs of different nodes to interact with each other, it is the user's responsibility to ensure that the correct knowledge exposure rules are specified for each node.

Confidentiality of Local Requests

There are some situations in which, during the analysis tasks, the nodes may want to keep some or all of their internal actions and action requests (i.e., assumed abducibles) confidential. For example, granting permission for a UK PDA to get sensor data by the US control center may trigger an internal logging action at the US control center, and the US control center may not wish to expose such information during distributed analysis. To cope with this situation, the language can be extended to allow the distinction between shared and private abducibles. This can be done easily by extending the abducible predicates like we did for the non-abducible predicates. However, as a safe condition of the system specification, an action can be considered private to a node only if it has effects on the node's local state only. In order to guarantee that none of the assumed private requests can be disclosed during the distributed reasoning computation, the DAREC^2 algorithm will have to be extended too. Since the DAREC^2 algorithm is a collaborative state rewriting process by a given set of nodes, one possible approach could be to let each node keep track of the set of assumed requests for each state it has processed (rewritten). This is similar to the idea proposed in [MRBL09]. However, this approach may introduce a big space overhead, as most of the states will have the same set of assumed requests. A different approach could be to let each node record and encrypt its assumed private requests in the states, and decrypt them only when the states need to be processed again by the same node. Since the assumed requests (and their actions) by a node cannot have effect to other nodes' local domains, other nodes do not need to decrypt and reasoning about them in order to process the states containing them. The implementation of these extensions will be part of our future work.

7.5 Conclusion

In this chapter we have presented an application for DAREC^2 – distributed security policy analysis. This work is an extension to an existing formal policy framework [CLM+09] for centralised policy analysis. First, we have extended the operational model and policy language of the existing framework for systems with multiple PEPs/PDPs, each of which operates independently with its local policies and domain knowledge. As we have shown, the new policy specification language is based on the DAREC^2 language, which supports location awareness of distributed knowledge and separation of shared and private knowledge of each node. This is important for applications in distributed systems, where confidentiality is often one of the major concerns. Secondly, we have applied DAREC^2 to perform policy tasks in a distributed manner. The DAREC^2 algorithm can be applied to distributed policy analysis in the same way a centralised abductive system (e.g., ASystem) is applied to centralised policy analysis, but in addition can guarantee that the private knowledge of each node remains confidential to the node during the computation, i.e., during the analysis. There is a need to keep local system requests private to the node performing them. This is not yet supported by DAREC^2, but a further extension for the separation of shared and private abducibles has been discussed.

Knowledge exposure rules (i.e., rules with askable atom as the head) are the "linkages" between knowledge bases, and they need to be specified carefully in order to guarantee the correct system model in which PDPs can interact with each other. Some of the analysis tasks require the completeness of the solving algorithm. Since the completeness of DAREC^2 depends on termination, it is necessary that the overall system (i.e., policy and domain) specification satisfies the allowedness and abductive non-recursive properties. However, the latter property is difficult to check as the overall system specification is assumed to be distributed. As future work we will try to identify conditions for knowledge exposure rules that can guarantee termination of DAREC^2 execution without analysing the overall system specification.

Chapter 8

Conclusion and Future Work

Abductive logic programming is a powerful inference tool that has been used in a wide range of applications. However, most existing work of abductive algorithms is based on centralised computation over a single repository of knowledge. Little work has been done for abductive problems in a distributed setting, where communication overheads or confidentiality concerns preclude the possibility of centralised knowledge and computation. Although the ALIAS system has been developed to address issues in the cooperation and competition between logic-based abductive reasoning agents, its distributed algorithm has many limitations and it does not consider confidentiality aspects. The main contribution of this thesis is the development of a general purpose distributed abductive reasoning system called DAREC. DAREC is the first computational logic-based multi-agent system that supports collaborative abductive reasoning over distributed constraint logic programs and guarantees global consistency. The distributed algorithm of DAREC extends ASystem, and its soundness and completeness (upon termination) with respect to the (Fitting's) three-valued semantics have been proved in this thesis. An extension of DAREC to open agent systems (i.e., where agents may leave or join the system during collaborative reasoning) has been described. DAREC^2, a customisation of DAREC with special goal selection and agent interaction strategies, is the first distributed abductive system that considers confidentiality. DAREC^2 allows shared and private knowledge of agents to be distinguished in the knowledge specification, and guarantees that no private knowledge is disclosed during collaborative reasoning. A prototype implementation of DAREC^2 has been given in this thesis. As for the evaluation of the system, we have presented its experimental results obtained from a specifically developed auto-testing environment, and a case study of its application in distributed security policy analysis. Therefore, our work also contributes to the research of policy based management in distributed networks. In addition, during the implementation and experiments of DAREC^2, an efficient Prolog-based inequality solver and a general purpose generator for distributed or centralised (abductive) (constraint) logic programs have also been developed, which can be used independently of DAREC^2, e.g., for the implementation of other logic programming based theorem provers and the generation of their test cases.

The work presented in this thesis leaves a number of possible future research issues in knowledge representation, algorithm optimisation and potential applications.

First of all, since we have developed an auto-testing environment for DAREC^2, we would like to conduct extensive experiments to study how the performance of distributed abduction is affected by different classes of distributed programs with different structures and properties. In particular, we will implement various goal selection strategies, agent interaction strategies and agent selection strategies for DAREC^2, and provide options for the user to choose between these strategies during the execution of DAREC^2.

There are application domains where there is a need for distinguishing between shared and private abducibles. Take distributed policy analysis as an example. During a distributed analysis task, the nodes may want to keep some or all of their internal actions and action requests (i.e., expressed as assumed abducibles) of a system execution trace confidential. This has already been mentioned in Section 7.4 of Chapter 7, and a solution through cryptography (e.g., agents may exchange states containing encrypted private abducible or non-abducible atoms) has been proposed. As future work, we would like to implement such extension, and study its practical impact to the performance of distributed abduction.

The current system prototype of DAREC^2 is a proof of concept for distributed abduction with confidentiality. Hence, it was not implemented with low level optimisations in mind. To improve its execution efficiency and make it practical for real world applications, many optimisations such as efficient data structures (e.g., internal representations of integrity constraints and state) studied in Nuffelen's ASystem implementation [vN04], predicate tabling techniques implemented in XSB [SW10], and heuristics or strategies for solution search adopted by high performance classical planning systems (e.g., GraphPlan [BF97] and FastForward [HN01]), may be adopted. As future work, we would like to investigate the feasibility of these optimisations and incorporate them into the DAREC^2 implementation. In order to provide portability of the system, we would also like to implement DAREC^2 in a cross-platform language such as Java.

In addition to distributed policy analysis, we have identified two other potential applications of DAREC^2. The first application is in the aforementioned ambient intelligence domain. In [BCR09], Broda et. al. proposed a logical agent-based environment monitoring and control system called SAGE. In this system, forward chaining deductive inference is used to map low level sensor data to high level events, and then multi-agent abductive reasoning is needed to provide possible explanations for these events. DAREC^2 fits this purpose and can provide all the necessary features that are required. The other application is in the domain of *declarative networking* [LCG+09, AMC+09], in which network protocols are specified as distributed logic programs, and are executed through distributed query processing. Similar to policy analysis, DAREC^2 can be used in network protocol analysis where confidentiality is considered and the overall knowledge cannot be centralised. For example, the Internet infrastructure may consist of a set of inter-connected routers belonging to different organisations. Each of these routers is running some routing protocol such as BGP [RLH06] for selecting the best path to forward a received packet to its destination. Different policies, such as local preferences on paths or neighbouring routers, could be manually added to a router and could affect the running protocol's best path selection algorithm. These policies are often private to the router or to the organisation it belongs to. We would like to investigate how DAREC^2 can be applied to perform confidential network protocol analysis, such as checking the *convergence property* [GS05, LCG+09] or *path oscillation situation* during protocol execution.

Appendix A

Example Source Code

A.1 The Inequality Solver

```
/**
 * @author  Jiefei Ma
 * @date    March 2010
 * Department of Computing, Imperial College London
 *
 * NOTE: this version can be run on SICStus4 or YAP6
 */




:- if(current_prolog_flag(dialect, yap)).
%{
:- module(inequalities, [op(700, xfx, =/=), '=/='/2, inequalities/2]).
:- use_module(library(lists), [member/2, append/3]).
%}
:- elif(current_prolog_flag(dialect, sicstus)).
%{
:- module(inequalities, ['=/='/2, inequalities/2]).
:- op(700, xfx, =/=).
%}
:- endif.

:- use_module(library(atts)).
:- use_module(library(ordsets), [
    ord_union/3,
    list_to_ord_set/2
  ]).

:- attribute aliens/1.

% public
X =/= Y :-
  (var(X) ; var(Y)), !,
  X \== Y,
  reinforce_neq(X, Y),
  reinforce_neq(Y, X).
X =/= Y :-
  (unifiable(X, Y, Eqs) ->
    (Eqs = [A = B] ->
      A =/= B % no choice point
    ;
      member(A = B, Eqs), % backtrackable
      A =/= B
    )
  ;
    true
  ).

unifiable(X, Y, [X = Y]) :-
  (var(X) ; var(Y)), !.
unifiable(X, Y, []) :-
  atomic(X), atomic(Y), !, X == Y.
unifiable(X, Y, Eqs) :-
  functor(X, F, A),
  functor(Y, F, A),
  X =.. [F|ArgsX],
  Y =.. [F|ArgsY],
  all_unifiable(ArgsX, ArgsY, Eqs).
all_unifiable([], [], []).
all_unifiable([X|TX], [Y|TY], AllEqs) :-
  unifiable(X, Y, Eqs),
  all_unifiable(TX, TY, RestEqs),
  append(Eqs, RestEqs, AllEqs).

reinforce_neq(A, B) :-
  var(A), !,
  (get_atts(A, aliens(S)) ->
    (\+ strictmember(S, B) -> NewS = [B|S] ; NewS = S),
    put_atts(A, aliens(NewS))
  ;
    put_atts(A, aliens([B]))
  ).
reinforce_neq(_, _).

strictmember([H|T], X) :-
  (X == H ->
    true
  ;
    strictmember(T, X)
  ).

% hook
verify_attributes(Var, Val, Goals) :-
  get_atts(Var, aliens(S1)), !, % are we involved?
  \+ strictmember(S1, Val), % is it an alien?
  ((var(Val), get_atts(Val, aliens(S2))) ->
  % thanks Domenico Corapi for helping with fixing the bug, 2010/03/31
  %(var(Val) ->
    %get_atts(Val, aliens(S2)),
    % \+ strictmember(S2, Var) % this should be implied by the previous test
    list_to_ord_set(S2, NewS2),
    list_to_ord_set(S1, NewS1),
    ord_union(NewS2, NewS1, S3), % copy forward aliens
    put_atts(Val, aliens(S3)),
    Goals = []
  ;
    generate_goals(S1, Val, Goals)
  ).
verify_attributes(_, _, []).

generate_goals([], _, []).
generate_goals([H|T], Val, Gs) :-
  generate_goals(T, Val, Gs1),
  (var(H) ->
    Gs = Gs1
  ;
    Gs = [(Val =/= H)|Gs1]
  ).

% hook
attribute_goal(Var, Goal) :-
  get_atts(Var, aliens(S)),
  list_to_ord_set(S, S1),
  construct_body(S1, Var, Goal).

construct_body([H|T], Var, Goal) :-
  (T = [] ->
    Goal = (Var =/= H)
  ;
    construct_body(T, Var, G),
    Goal = ((Var =/= H),G)
  ).

% public
```

```
inequalities(Var, Ineqs) :-
  get_atts(Var, aliens(S)), !,
  list_to_ord_set(S, S1),
  collect_inequalities(S1, Var, Ineqs).
inequalities(_, []).

collect_inequalities([], _, []).
collect_inequalities([H|T], Var, [N|Rest]) :-
  (Var @=< H ->
    N = (Var =/= H)
  ;
    N = (H =/= Var)
  ),
  collect_inequalities(T, Var, Rest).
```

Listing A.1: Source code of the Inequality Solver `inequalities.pl`

Bibliography

[AB91] Krzysztof R. Apt and Marc Bezem. Acyclic programs. *New Generation Computing*, 9(3/4):335–364, 1991.

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. *Towards a Theory of Declarative Knowledge*, pages 89–148. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[AC05] Marco Alberti and Federico Chesani. The computational behaviour of the *S*CIFF abductive proof procedure and the SOCS-SI system. *Intelligenza Artificiale*, 2(3):45–51, 2005.

[ACG$^+$06] Philippe Adjiman, Philippe Chatalic, François Goasdoué, Marie-Christine Rousset, and Laurent Simon. Distributed reasoning in a peer-to-peer setting: Application to the semantic web. *Journal of Artificial Intelligence Research*, 25:269–314, February 2006.

[ACG$^+$08] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Verifiable agent interaction in abductive logic programming: the *S*CIFF framework. *ACM Transactions on Computational Logic*, 9(4), 2008.

[AGL$^+$05] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. The *S*CIFF abductive proof-procedure. In *9th Congress of the Italian Association for Artificial Intelligence*, pages 135–147, 2005.

[AMC$^+$09] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell C Sears. Dedalus: Datalog in time and space. Technical Report

UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[ANDB01] Ofer Arieli, Bert Van Nuffelen, Marc Denecker, and Maurice Bruynooghe. Coherent composition of distributed knowledge-bases through abduction. In *Proceedings of the Artificial Intelligence on Logic for Programming*, Logic for Programming, Artificial Intelligence, and Reasoning 2001, pages 624–638, London, UK, 2001. Springer-Verlag.

[AvE82] Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29:841–862, July 1982.

[Bar99] Roman Bartak. Constraint programming: In pursuit of the holy grail. In *Proceedings of the 8th Annual Conference of Doctoral Students (Invited Lecture)*, pages 555–564, 1999.

[Bar03] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003.

[BCR09] Krysia Broda, Keith Clark, Rob Miller 0002, and Alessandra Russo. Sage: A logical agent-based environment monitoring and control system. In *Ambient Intelligence, European Conference*, pages 112–117, 2009.

[BF97] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artifical Intelligence*, 90(1-2):281–300, 1997.

[BIM10] Gauvain Bourgne, Katsumi Inoue, and Nicolas Maudet. Abduction of distributed theories through local interactions. In *19th European Conference on Artificial Intelligence*, pages 901–906, 2010.

[BN08] Moritz Y. Becker and Sebastian Nanz. The role of abduction in declarative authorization policies. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages*, PADL'08, pages 84–99, Berlin, Heidelberg, 2008. Springer-Verlag.

[BSW08] Steve Barker, Marek J. Sergot, and Duminda Wijesekera. Status-based access control. *ACM Transactions on Information System Security*, 12:1:1–1:47, October 2008.

[CDT91] Luca Console, Daniele Theseider Dupré, and Pietro Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[Cia02] Ann Ciampolini. A distributed medical diagnosis with abductive logic agents. In *Proceedings of Agents Applied in Health Care Workshop, 15th European Conference on Aritificial Intelligence*, 2002.

[Cla78] Keith L. Clark. Negation as failure. *Logic and Data Bases*, pages 293–322, 1978.

[CLM$^+$03] Anna Ciampolini, Evelina Lamma, Paola Mello, Francesca Toni, and Paolo Torroni. Cooperation and competition in ALIAS: A logic framework for agents that negotiate. *Annals of Mathematics and Artificial Intelligence*, 37:65–91, 2003.

[CLM$^+$09] Robert Craven, Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil Lupu, and Arosha Bandara. Expressive policy analysis with enhanced system dynamicity. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 239–250, New York, NY, USA, 2009. ACM.

[CLMT01] Anna Ciampolini, Evelina Lamma, Paola Mello, and Paolo Torroni. LAILA: a language for coordinating abductive reasoning among logic agents. *Computer Languages, Systems and Structures*, 27:137–161, 2001.

[CRZA05] Keith L. Clark, Peter J. Robinson, and Silvana Zappacosta-Amboldi. Multi-threaded communicating agents in qu-prolog (tutorial paper). In *Computational Logic in Multi-Agent Systems, 6th International Workshop*, pages 186–205, 2005.

[CT04] Anna Ciampolini and Paolo Torroni. Using abductive logic agents for modeling the judicial evaluation of criminal evidence. *Applied Artificial Intelligence*, 18(3-4):251–275, 2004.

[DC89] Hendrik Decker and Lawrence Cavedon. Generalizing allowedness while retaining completeness of sldnf-resolution. In *Proceedings of 3rd Workshop on Computer Science Logic*, pages 98–115, 1989.

[DCV93] Daniel Diaz, Philippe Codognet, and Domaine De Voluceau. A minimal extension of the wam for clp(fd). In *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790. MIT Press, 1993.

[DS92] Marc Denecker and Danny De Schreye. Sldnfa: An abductive procedure for normal abductive programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 686–700, 1992.

[DS98] Marc Denecker and Danny De Schreye. Sldnfa: An abductive procedure for abductive logic programs. *Journal of Logic Programming*, 34(2):111–167, 1998.

[Dur01] Edmund H. Durfee. Mutli-agents systems and applications. chapter Distributed Problem Solving and Planning, pages 118–149. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[EGG$^+$09] Enrico Ellguth, Martin Gebser, Markus Gusowski, Benjamin Kaufmann, Roland Kaminski, Stefan Liske, Torsten Schaub, Lars Schneidenbach, and Bettina Schnor. A simple distributed conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning, 10th International Conference*, pages 490–495, 2009.

[EK89] Kave Eshghi and Robert A. Kowalski. Abduction compared with negation by failure. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 234–254, 1989.

[EMS$^+$04a] Ulrich Endriss, Paolo Mancarella, Fariba Sadri, Giacomo Terreni, and Francesca Toni. Abductive logic programming with ciff: System description. In *JELIA*, pages 680–684, 2004.

[EMS$^+$04b] Ulrich Endriss, Paolo Mancarella, Fariba Sadri, Giacomo Terreni, and Francesca Toni. The CIFF proof procedure for abductive logic programming with constraints.

In *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence*, pages 31–43, 2004.

[Esh88] Kave Eshghi. Abductive planning with event calculus. In *International Conference on Logic Programming/Symposium on Logic Programming*, pages 562–579, 1988.

[Fit85] Melvin Fitting. A kripke-kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[FK97] Tzee Ho Fung and Robert A. Kowalski. The iff proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.

[GAL09] Marco Gavanelli, Marco Alberti, and Evelina Lamma. Integration of abductive reasoning and constraint optimization in *S*CIFF. In *Logic Programming, 25th International Conference*, pages 387–401, 2009.

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 386–392, 2007.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Joint Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[GRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[GS05] Timothy G. Griffin and João L. Sobrinho. Metarouting. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 1–12, 2005.

[HN01] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal Artificial Intelligence Research*, 14:253–302, 2001.

[HSC07] Hiroshi Hosobe, Ken Satoh, and Philippe Codognet. Agent-based speculative constraint processing. *IEICE - Transactions on Information and Systems*, E90-D:1354–1362, September 2007.

[HSM$^+$10] Hiroshi Hosobe, Ken Satoh, Jiefei Ma, Alessandra Russo, and Krysia Broda. Speculative constraint processing for hierarchical agents. *AI Communications*, 23(4):373–388, 2010.

[II04] Katsumi Inoue and Koji Iwanuma. Speculative computation through consequence-finding in multi-agent environments. *Annals of Mathematics and Artificial Intelligence*, 42(1-3):255–291, 2004.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The clp(r) language and system. *ACM Transaction on Programming Languages and Systems*, 14:339–395, May 1992.

[KKT92] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North-Holland, 1952.

[KM90a] Antonis C. Kakas and Paolo Mancarella. Abductive logic programming. In *Proceedings of the Workshop Logic Programming and Non-Monotonic Logic*, pages 49–61, 1990.

[KM90b] Antonis C. Kakas and Paolo Mancarella. Database updates through abduction. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *Proceedings of 16th International Conference on Very Large Databases*, pages 650–661. Morgan Kaufmann, 1990.

[KM90c] Antonis C. Kakas and Paolo Mancarella. Generalised stable models: A semantics for abduction. In Luigia C. Aiello, editor, *Proceedings of the Ninth European Conference on Artificial Intelligence*, pages 385–391, Stockholm, Sweden, 1990. Pitman Publishing.

[KM95] Antonis C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 399–413, 1995.

[KM97] Antonis C. Kakas and Costas Mourlas. Aclp: Flexible solutions to complex problems. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference*, pages 388–399, 1997.

[KMM00] Antonis C. Kakas, A. Michael, and Costas Mourlas. Aclp: Abductive constraint logic programming. *Journal of Logic Programming*, 44(1-3):129–177, 2000.

[Kow92] Robert Kowalski. Database updates in the event calculus. *Journal of Logic Programing*, 12:121–146, January 1992.

[KS86] Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, January 1986.

[Kum92] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13:32–44, April 1992.

[Kun87] Kenneth Kunen. Negation in logic programming. *Journal Logic Programming*, 4(4):289–308, 1987.

[KvND01] Antonis C. Kakas, Bert van Nuffelen, and Marc Denecker. A-System: Problem solving through abduction. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, pages 591–596, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[Lau78] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[LCG$^+$09] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 52:87–95, November 2009.

[Lec09] Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms.* Wiley-IEEE Press, 2009.

[LPF$^+$06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV sstem for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, July 2006.

[MBCR08] Jiefei Ma, Krysia Broda, Keith L. Clark, and Alessandra Russo. A dynamic system for distributed reasoning. In *Proceedings of the AAAI Spring Symposium*, SS-08-02, 2008.

[MBD94] Lode Missiaen, Maurice Bruynooghe, and Marc Denecker. Chica, an abductive planning system based on event calculus. *Journal of Logic and Computation*, 5:579–602, 1994.

[MBG$^+$10] Jiefei Ma, Krysia Broda, Randy Goebel, Hiroshi Hosobe, Alessandra Russo, and Ken Satoh. Speculative abductive reasoning for hierarchical agent systems. In *Computational Logic in Multi-Agent Systems, 11th International Workshop*, pages 49–64, 2010.

[MRBC08] Jiefei Ma, Alessandra Russo, Krysia Broda, and Keith Clark. Dare: A system for distributed abductive reasoning. *Autonomous Agents and Multi-Agent Systems*, 16:271–297, June 2008.

[MRBL09] Jiefei Ma, Alessandra Russo, Krysia Broda, and Emil Lupu. Multi-agent planning with confidentiality. In *Proceedings of The 8th International Conference on*

Autonomous Agents and Multiagent Systems - Volume 2, AAMAS '09, pages 1275–1276, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[MTS$^+$09] Paolo Mancarella, Giacomo Terreni, Fariba Sadri, Francesca Toni, and Ulle Endriss. The CIFF proof procedure for abductive logic programming with constraints: Theory, implementation and experiments. *Theory and Practice of Logic Programming*, 9(6):691–750, 2009.

[NII03] Hidetomo Nabeshima, Koji Iwanuma, and Katsumi Inoue. Solar: A consequence finding system for advanced reasoning. In *Analytic Tableaux and Related Methods*, pages 257–263, 2003.

[NK01] Bert Van Nuffelen and Antonis C. Kakas. A-system: Declarative programming with abduction. In *Logic Programming and Non-Monitonic Reasoning*, pages 393–396, 2001.

[NS97] Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference*, pages 421–430, 1997.

[Pea87] Judea Pearl. Embracing causality in formal reasoning. In *Proceedings of the sixth National conference on Artificial intelligence - Volume 1*, AAAI'87, pages 369–373. AAAI Press, 1987.

[Pei31] Charles S. Peirce. *Collected Papers of Charles Sanders Peirce.* Harvard University Press, 1931.

[Poo88] David Poole. Representing knowledge for logic-based diagnosis. In *Fifth Generation Computer Systems*, pages 1282–1290, 1988.

[RC10] Peter J. Robinson and Keith L. Clark. Pedro: a publish/subscribe server using prolog technology. *Software - Practice and Experience*, 40(4):313–329, 2010.

[RG95] Anand S. Rao and Michael P. Georgeff. Bdi agents: From theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, pages 312–319, 1995.

[RLH06] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). IETF RFC 4271, January 2006.

[RMNK02] Alessandra Russo, Rob Miller, Bashar Nuseibeh, and Jeff Kramer. An abductive approach for analysing event-based requirements specifications. In *Proceedings of the 18th International Conference on Logic Programming*, ICLP '02, pages 22–37, London, UK, 2002. Springer-Verlag.

[SCH03] Ken Satoh, Philippe Codognet, and Hiroshi Hosobe. Speculative constraint processing in multi-agent systems. In *Intelligent Agents and Multi-Agent Systems, 6th Pacific Rim International Workshop on Multi-Agents*, pages 133–144, 2003.

[SDDM09] M.P. Sindlar, M.M. Dastani, F. Dignum, and J.C. Meyer. Mental state abduction of bdi-based agents. pages 161–178, 2009.

[Sha89] Murray Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2*, pages 1055–1060, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[Sha99] Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today*, pages 409–430. 1999.

[Sha00] Murray Shanahan. An abductive event calculus planner. *Journal of Logic Programming*, 44(1-3):207–240, 2000.

[Sha05] Murray Shanahan. Perception as abduction: Turning sensor data into meaningful representation. *Cognitive Science*, 29(1):103–134, 2005.

[SIIS00] Ken Satoh, Katsumi Inoue, Koji Iwanuma, and Chiaki Sakama. Speculative computation by abduction under incomplete communication environments. In *4th International Conference on Multi-Agent Systems*, pages 263–270, 2000.

[SL02] Morris Sloman and Emil Lupu. Security and management policy specification. *IEEE Network*, 16(2):10–19, March 2002.

[SN01] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference*, pages 434–438, 2001.

[Str03] John Strassner. *Policy-Based Network Management: Solutions for the Next Generation (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[SW10] Terrance Swift and David Scott Warren. XSB: Extending prolog with tabled logic programming. *The Computing Research Repository*, abs/1012.5123, 2010.

[SY02] Ken Satoh and Keiji Yamamoto. Speculative computation with multi-agent belief revision. In *The First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 897–904, 2002.

[Teu96] Frank Teusink. Three-valued completion for abductive logic programs. *Theoretical Computer Science*, 165:171–200, September 1996.

[Ton95] Francesca Toni. A semantics for the kakas-mancarella procedure for abductive logic programming. In *Proceedings of the Logic Programming Workshop GULP'95*, pages 231–244, 1995.

[Top87] Rodney W. Topor. Domain-independent formulas and databases. *Theoretical Computer Science*, 52:281–306, June 1987.

[vEK76] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, October 1976.

[Ver99] Sofie Verbaeten. Termination analysis for abductive general logic programs. In *Proceedings of the 1999 International Conference on Logic Programming*, pages 365–379, Cambridge, MA, USA, 1999. Massachusetts Institute of Technology.

[vN04] Bert van Nuffelen. *Abductive Constraint Logic Programming: Implementation and Applications*. PhD, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June 2004.

[YPG00] R. Yavatkar, D. Pendarakis, and R. Guerin. A framework for policy-based admission control. IETF RFC 2753, 2000.