DARE: A System for Distributed Abductive REasoning

J. Ma, A. Russo, K. Broda and K. L. Clark
Department of Computing, Imperial College London,
{jm103, ar3, kb, klc}@doc.ic.ac.uk

Abstract

Abductive reasoning is a well established field of Artificial Intelligence widely applied to different problem domains not least cognitive robotics and planning. It has been used to abduce high-level descriptions of the world from robot sense data, using rules that tell us what sense data would be generated by certain objects and events of the robots world, subject to certain constraints on their co-occurrence. It has also been used to abduce actions that might result in a desired goal state of the world, using descriptions of the normal effects of these actions, subject to constraints on the action combinations. We can generalise these applications to a multi-agent context. Several robots can collaboratively try to abduce an agreed higher-level description of the state of the world from their separate sense data consistent with their collective constraints on the abduced description. Similarly, multi-agent planning can be accomplished by the abduction of the actions of a collective plan where each agent uses its own description of the effect of its actions within the plan, such that the constraints on the actions of all the participating agents are satisfied.

To address this class of problems, we need to generalise the single agent abductive reasoning algorithm to a distributed abductive inference algortihm. In addition, if we want to investigate applications in which the set of collaborating robots/agents is open, we need an algorithm that allows agents to join or leave the collaborating group whilst a particular inference is under way, but which still produces sound abductive inferences. This paper describes such a distributed abductive reasoning system, which we call DARE, and its implementation in the multi-threaded Qu-Prolog variant of Prolog. We prove the soundness of the algorithm it uses and we discuss its completeness in relation to non-distributed abductive reasoning.

We illustrate the use of the algorithm with a multi-agent meeting scheduling example. The task is open in that the actual agents who need to attend is not determined in advance. Each individual agent has its own constraints on the possible meeting time and concerning which other agents must or must attend the meeting, if it attends. The algorithm selects the agents to attend and ensures that the constraints of each of the attending agents are satisfied.

Keywords: Abduction, distributed inference, multi-thread Prolog.

1 Introduction and Motivation

Abduction is a powerful inference mechanism which generates conditional proofs, the conditions being 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. It is a general knowledge based problem solving method with a wide range of applications [19]. Abductive logic programming [15] is a special case in which the knowledge base is a logic program paired with a set of consistency constraints, queries that must never succeed, that constrain the assumptions that can be abduced. It has been used in cognitive robotics [22] for abducing higher level descriptions of the world from sense data, for planning [24], where action events are abduced that will result in a desired described state of the world using a knowledge base which is an event calculus [23] formulation of the effect of actions on fluent features of the world.

The focus of this paper is distributed abduction 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. DARE has much in common with ALIAS [7], which inspired this work. Both DARE and ALIAS are distributed extensions of the Kakas and Mancarella abductive algorithm [16, 15] 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 helpers agents to be recruited on-the-fly as and when they join the agents group. In contrast, in the ALIAS systems the knowledge about agents' abilities is predefined as part of the background knowledge of an agent using the LAILA [8] 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. More details on the comparison between the two systems are given in the related work section.

Distributed abduction has been used to simulate medical diagnosis [6], where expertise is distributed over network of experts, and for knowledge base integration [3]. It has the potential to be used for multi-robot collaborative interpretation of the sense data they separately gather to arrive at an agreed higher level partial description of a shared environment, and for the essentially similar application of arriving at a mutually agreed higher level interpretation of the separate readings of a network of sensors. It can be used for multi-agent planning [18], or the scheduling of a common activity that satisfies the separate constraints, on participation, of all the agents who will participate.

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.

This paper describes and illustrates the use of the DARE algorithm and its implementation in a multi-threaded distributed Prolog, Qu-Prolog [9]. Each agent in the open group of all the agents that may participate in an abductive proof is implemented as a separate Qu-Prolog process that can be distributed over a network of hosts. Each Qu-Prolog process comprises multiple-threads allowing it to be participating in multiple abductive proofs at the same time. Communication between threads in the different agent processes uses a network demon, Pedro [20]. This supports both direct communication between remote threads using thread identifiers similar to email addresses, as well as publish and subscribe communication based on subscriptions that are restricted Prolog queries to be applied to messages posted to Pedro with no specified destination. Agents join and leave the group by posting register and unregister messages to Pedro. The registration includes the identifier of the agent and is used by other agents to send messages directly to that agent. Agents are recruited into a particular proof cluster as a result of subscriptions and advertisements posted to Pedro. The agents lodge subscriptions for predicates of conditions used in their rules for which they may need sub-proof help from other agents. The advertisements give the predicates for the conditions about which they are willing to try to find abductive proofs. These may make use of sub-proofs executed by other agents in the group, that are recruited by this agent into the current cluster. The advertised predicates are their public predicates. The inter-agent communication uses KQML performatives [11].

The structure of the rest of the paper is as follows. In Section 2 we explain the Kakas and Mancarella algorithm for single agent abductive proof. In Section 3 we illustrate our distributed DARE abductive proof algorithm using the scheduling example before formally describing the algorithm. In Section 4 we give an overview of the DARE architecture and its Qu-Prolog implementation. In section 5, we give an evaluation of the DARE system whilst in Section 6 we compare it with the ALIAS [7] and other distributed or multi-agent reasoning systems. The paper concludes with a summary and discussion of future work.

2 Background

This section summarises the basic notation and terminology used throughout the paper and recalls the notion of abduction, illustrating through an example the Kakas and Mancarella (KM) abductive proof procedure [15, 16] of which the DARE algorithm, given in Section 3, is a proper extension.

2.1 Notation and Terminology

A term is either a constant, a variable or a function $F(t_1, \ldots, t_n)$ where F is a n-ary function symbol (with $n \geq 1$) and t_i is a term. An *atomic* formula (or *atom* in brief) is a proposition or an n-ary predicate P followed by an n-tuple of terms. A *positive* literal is an atom ϕ, and a *negative* literal is a negated atom, written as `not` ϕ, where `not` is the negation as failure operator. A *literal* is either a positive or negative atom. A *clause* is either a *rule*, $\phi \leftarrow \phi_1, \ldots, \phi_n$, a *fact* (or literal) ϕ, or a *denial* $\leftarrow \phi_1, \ldots, \phi_m$, where ϕ is an atom and ϕ_i are literals. In the case of a rule ϕ is also called the *head* literal and ϕ_i are called the body literals. A clause is *definite* if all its body literals are positive. A clause is *ground* if it contains no variables. A *goal* is a set $\{\psi_1, \ldots, \psi_n\}$ of literals, denoted as $\leftarrow \psi_1, \ldots, \psi_n$ and an *empty goal* is the *empty* denial []. A *logic program* is a set of clauses. A *definite* logic program is a program in which all clauses are definite. A *normal* logic program is one in which clauses are not necessarily definite. They can be of the form $\phi \leftarrow \phi_1, \ldots, \phi_n, \mathtt{not}\psi_1, \ldots, \mathtt{not}\psi_m$ where ϕ is the *head atom*, ϕ_i are *positive* body literals and `not` ψ_j are *negative* body literals.

In general, a *model* I of a normal logic program, Π, is a set of ground atoms such that, for each ground instance G of a clause in Π, I satisfies the head of G whenever it satisfies the body. A model I is *minimal* if it does not strictly include any other model. Definite programs always have a unique minimal model. Normal programs may have instead one, none, or several minimal models. It is usual to identify a certain subset of these models, called *stable models*, as the possible meanings of a program [13]. Given a normal logic program Π, the reduct of Π with respect to I, denoted Π^I, is the program obtained from the ground instances of Π by removing all clauses with a negative literal $\mathtt{not}\phi$ in its body where $\phi \in I$ and removing all negative literals from the bodies of the remaining clauses. Clearly Π^I is a definite logic program and as such has a unique minimal (Herbrand) model. If the model of Π^I coincides with I then I is said to be a stable model of Π as formalised in Definition 1 [13].

Definition 1 *A model I of a normal logic program Π is a* stable model *if I is a minimal (Herbrand) model of Π^I, where Π^I is the definite program* $\Pi^I = \{\phi \leftarrow \phi_1, \ldots, \phi_n | \phi \leftarrow \phi_1, \ldots, \phi_n, \mathtt{not}\psi_1, \ldots, \mathtt{not}\psi_m$ *is the ground instance of a clause in Π and I does not satisfy any of the ψ_i*$\}$

A normal logic program is said to be *stratified* if no negative recursion, that is recursion "through" negation, is used in the program [2]. For example, the program $\Pi = \{p \leftarrow \mathtt{not}\ q,\ q \leftarrow r\}$ is stratified, whereas the program $\Pi = \{p \leftarrow \mathtt{not}\ p\}$ is not stratified. A program is said to be *locally stratified* if its instantiated program is stratified. Results in [13] have shown that locally stratified programs accept a unique stable model.

Within the context of this paper, agents are represented as normal logic programs that include a (non-empty) set of rules, called *knowledge base* of the agent, and a (possibly empty) set of denials, called *integrity constraints* of the agent. Agents that collaborate to prove a given goal form a cluster. For example, if agent $A1$ asks agent $A2$ to help with proving a sub-goal G_1 of a given goal G, then $A1$ and $A2$ are within the same cluster. Note that, in principle, a cluster may well be the entire group of agents currently present in the system, but in the applications that we have in mind it is often a proper subset of the group.

2.2 Abductive Reasoning

Abduction is the process of reasoning from effects to possible causes. In essence, it is concerned with the construction of explanations, Δ, for a set of goals, G, with respect to a *knowledge base* T and *integrity constraints* IC. Relative to the given knowledge base, the set of explanation Δ must *cover* the goals G, while being *consistent* with the integrity constraints IC. Abductive explanations are usually restricted to ground literals of some predefined set of predicates, $\mathcal{A}$, called *abducible* predicates. When negative conditions are used in the knowledge base T and/or in the integrity constraints IC, negative literals of non-abducible predicates are also hypothesized as part of an abductive explanation to maintain the consistency of the explanation with the IC during its computation. For clarity, positive (or negative) literals with *abducible predicates* are referred to as *base abducibles* and negative literals with non-abducible predicates as *non-base abducibles*. In the case of theories expressed as normal logic programs with negative conditions, abductive explanations are usually composed of ground instances of based and/or non-base abducibles [15, 16]. Throughout the paper, the term *abducible* refers to both a base and non-base abducible. The abductive notion of entailment is typically not classical but based on some *canonical model* semantics such as the *stable model* semantics. To provide a formal definition of an abductive task we need to define the concepts of coverage and consistency used above. In general, given a normal logic program Π, called a *knowledge base*, a set of ground literals G, a set of denial clauses IC, and a set of predicates $\mathcal{A}$, an abductive reasoning task is to find a set of ground literals Δ, of abducibles, such that G and IC are satisfied in the canonical model of $\Pi \cup \Delta$. In this paper, we assume Π to be a locally stratified program for which the canonical model is the unique stable model of $\Pi \cup \Delta$. So, coverage and consistency simply mean that $\Pi \cup \Delta \models G$ and $\Pi \cup \Delta \models IC$, where $\models$ is the notion of entailment under the stable model semantics. The idea is that the given knowledge base Π represents an incomplete knowledge where the abducible predicates are not defined but are assumable; and the goal G denotes an existentially quantified (or ground) query which must be made to succeed by adding a set Δ of abducibles to the knowledge base Π subject to the constraints in IC. Intuitively, abducible predicates are incomplete information in the knowledge base and can be used at any time (when a goal is provided) to partially complete the knowledge base. The abductive reasoning setting is formalised in Definition 2 below, which defines the notions of *abductive context* and *abductive explanation*.

Definition 2 (Abductive context, Abductive explanation)
An abductive context *is a four-tuple $AC = \langle \Pi, G, IC, \mathcal{A} \rangle$ where Π is a normal logic program, G is a set of ground literals, IC is a set of denial clauses, and $\mathcal{A}$ is a set of predicates, such that $\Pi \models IC$. Now, let $L_{\mathcal{A}}$ denote the set of all ground literals with predicates in $\mathcal{A}$ and negative ground literals of predicates not in $\mathcal{A}$ but that appear negated in Π and IC. Then an* abductive explanation *of AC is a set $\Delta \subseteq L_{\mathcal{A}}$ of ground literals such that $\Pi \cup \Delta \models G$ and $\Pi \cup \Delta \models IC$.*

A number of abductive proof procedures and their extensions have been proposed over the years. One of the most influential is the KM procedure [15, 16]. This is reviewed in the rest of this section and illustrated with an example as it provides the basis for the DARE distributed abductive algorithm given in Section 3. The KM proof procedure is composed of two phases, an *abductive derivation* and a *consistency derivation* that interleave with each other. Each abducible generated during the first phase is temporarily added to a set of abducibles that have already been generated during the proof. But this addition is only made permanent if the second phase confirms that the entire new set of abducibles is consistent, relative to the program Π with the given integrity constraints IC. Let G be a set of literals that we want to prove, which we write as $\leftarrow G_1 \ldots, G_n$. The abductive proof procedure succeeds if its top-down procedure can prove G, i.e. reduces it to $\Box$ (empty list of literals) via a set of rules, and with a set Δ of abducibles collected along the procedure. To illustrate how the KM abductive proof procedure works, a simple single-agent scheduling problem example is considered.

Example 2.1 This simple *single-agent* 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 $AC = \langle \Pi, G, IC, \mathcal{A} \rangle$ be the abductive context defined below. The knowledge base contains

$$\Pi = \begin{cases} \texttt{conveneMeeting(X)} \leftarrow \texttt{studentCanAttend(X)}, \texttt{tutorCanAttend(X)}, \texttt{not weekend(X)} \\ \texttt{studentCanAttend(X)} \leftarrow \texttt{studentName(dan)}, \texttt{free(dan,X)} \\ \texttt{studentCanAttend(X)} \leftarrow \texttt{studentName(ben)}, \texttt{free(ben,X)} \\ \texttt{tutorCanAttend(X)} \leftarrow \texttt{tutorName(pat)}, \texttt{free(pat,X)} \\ \texttt{free(dan,monday)} \\ \texttt{free(ben,tuesday)} \\ \texttt{free(pat,monday)} \\ \texttt{free(pat,tuesday)} \\ \texttt{weekend(saturday)} \\ \texttt{weekend(sunday)} \end{cases}$$

$$G = \{ \ \texttt{conveneMeeting(X)} \ \}$$

$$IC = \{ \ \leftarrow \texttt{tutorName(pat), studentName(dan)} \ \}$$

$$\mathcal{A} = \{ \ \texttt{tutorName, studentName} \ \}$$

The program 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 goal is for X bound to Tuesday and both Pat and Ben to attend the meeting.

The abductive algorithm takes in input the above abductive context and starts an abductive derivation process to prove the goal

$G_0 = \leftarrow \texttt{conveneMeeting(X)}$

with initially an empty set $\Delta_0 = \emptyset$ of abduced assumptions.
This derivation is essentially a standard SLDNF-resolution, i.e. reasoning backwards for a refutation, but collecting any required abductive assumption along the process. Given G_0, the SLDNF resolution process unifies it with the head of the first rule in Π and collects the body literals of this rule as new queries to prove. The new goal is then

$G_1 = \leftarrow \texttt{studentCanAttend(X)}, \texttt{tutorCanAttend(X)}, \texttt{not weekend(X)}$

and $\Delta_1 = \Delta_0$.

The first literal `studentCanAttend(X)` is then removed from G_1 and the SLDNF resolution process starts again on the new single-literal query $\leftarrow$`studentCanAttend(X)`. This literal unifies with the head of the second rule in Π, and so the body literals of this second rule are added in front of current list of remaining literals to proved so giving the new goal

$G_2 = \leftarrow \texttt{studentName(dan)}, \texttt{free(dan,X)}, \texttt{tutorCanAttend(X)}, \texttt{not weekend(X)}$

and $\Delta_2 = \Delta_1$.

At this point the next single-literal query is $\leftarrow$ `studentName(dan)` for which the knowledge base Π is incomplete (does not have any information about this predicate). But in the given abductive context this is an abducible predicate, so the abductive derivation instead of failing to prove this query, which is what standard SLDNF resolution would do in this case, adds the literal `studentName(dan)` temporarily to Δ_3 and a consistency derivation is activated with the temporary set $\Delta^{'} = \Delta_3 \cup \{$`studentName(dan)`$\}$ of abduced predicates. If the consistency derivation is successful then the abductive derivation process continues its proof on the remaining goal $G_4 = \leftarrow$ `free(dan,X)`, `tutorCanAttend(X)`, `not weekend(X)` and the set Δ_4 of abducibles given by $\Delta^{'}$ possibly extended with assumptions accumulated during the consistency derivation[1].

The consistency derivation takes the new temporary set $\Delta^{'}$ of abduced predicates and considers all the integrity constraints that include the newly abduced predicate `studentname(dan)`. In our example, there is only one such constraint. The resolution with the abduced predicate gives the literal `tutorName(pat)`, which **has to fail** for the constraint to be satisfied. The literal `tutorName(pat)` is itself abducible but not included in $\Delta^{'}$, so it cannot be proved and the consistency derivation succeeds.

The abductive derivation can then continue its proof with

$\Delta_4 = \{$`studentName(dan)`$\}$ and $G_4 = \leftarrow$ `free(dan,X)`, `tutorCanAttend(X)`, `not weekend(X)`.

Continuing the SLDNF resolution proof, the variable X gets unified with monday, the proof of `tutorCanAttend(monday)` generates a new set of temporary abduced predicates $\Delta^{'} = \Delta_4 \cup \{$`tutorName(pat)`$\}$. But, this time the consistency derivation on the newly abduced predicate `tutorName(pat)` **fails** because together with the predicate `studentName(dan)` previously abduced would violate the integrity constraint. At this point, the failure of the consistency derivation causes the current abductive derivation to fail, and 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 goal was

$G_1 = \leftarrow$ `studentCanAttend(X)`, `tutorCanAttend(X)`, `not 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 predicate and start the abductive derivation again. Continuing in a similar way, it is easy to see that the answer to the initial goal $G_0 = \leftarrow$ `conveneMeeting(X)` is

$\Delta = \{$`studentName(ben)`, `tutorName(pat)`$\}$ with unification $X = tuesday$. •

The above example run covers, however, only a few of the possible cases that can arise during the abductive and consistency derivations. The full KM proof procedure can be found in [15, 16].

Example 2.1 shows a very simple snapshot of a scheduling problem. In a real context, the knowledge base would be much bigger and the computation cost much higher if expressed as a single agent 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. for the literal `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, the KM abductive proof procedure would need to be extended to allow for abductive derivations over distributed knowledge and consistency derivations over distributed integrity constraints. This is where this paper provides a novel contribution. In the next section, a distributed abductive inference algorithm is presented and illustrated in detail using a multi-agent re-formulation of the above example. The soundness of the algorithm is also proved.

[1]This is because one of the main features of KM proof procedure is that the abductive and consistency derivation can interleave with each other.

3 Collaborative Abductive Reasoning

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 sub-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 abductive proof procedure illustrated in the previous section so to allow abductive derivations over multiple agents (A_i), equipped with individual knowledge base 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. Within the context of this paper, the existence of a *shared predicate ontology* among agents is assumed, which includes a given set $\mathcal{A}$ 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 $\mathcal{A}$ of *abducible predicates*. As in the KM proof procedure, these include positive (resp. negative) literals with abducible predicates (i.e. base abducibles) and negative literals with non-abducible predicates (i.e. non-base abducibles). The knowledge of each agent (A_i) is represented as a stratified logic program (Π_i).

3.1 An overview

Within the DARE system, each agent is capable of performing *local* abductive reasoning to explain (sub-)goals using its own knowledge base 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 knowledge base 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 assumption 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 of an agent needs necessarily to be public. To preserve a certain level of encapsulation of information, an agent has the ability, via the KQML *advertise* performative, to announce the information (or predicates) that it will provide proof for. Advertised predicates must be part of the pre-defined shared ontology, and not necessarily all predicates in the shared ontology have to be advertised. Together with the knowledge base, an agent has also 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 system guarantee consistency of the abductive answer with respect to the 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 distributed abductive context and distributed explanation as a generalisation of the notions of abductive context and abductive explanation given in the previous section to the case of cluster of agents.

Definition 3 (Distributed Abductive Context)
Let $A_1, \ldots, A_n$ be a group of agents and let Π_i and 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 $1 \leq i \leq n$. A distributed abductive context *is the tuple* $\texttt{DAC} = \langle \{\Pi_i\}, G, \{IC_i\}, A_{init}, \mathcal{A} \rangle$

where G is a conjunction of literals, A_{init} is the agent in the group that receives the top level goal G, i ranges from 1 to n, and $\mathcal{A}$ is a set of predicates, such that $\Pi_i \models IC_i$, for each $1 \leq i \leq n$

As explained in Section 4, 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 4 (Evolved Abductive Context)
Let $\mathtt{DAC} = \langle \{\Pi_i\}, G, \{IC_i\}, A_{init}, \mathcal{A}\rangle$ *be a distributed abductive context with respect to the group* $A_1, \ldots, A_n$ *of agents. An* evolved abductive context *is the tuple* $\mathtt{DAC}^{'} = \langle \{\Pi_j^{'}\}, G, \{IC_j^{'}\}, A_{init}, \mathcal{A}\rangle$ *with respect to a group of agents* $A_1^{'}, \ldots, A_m^{'}$*, with* A_{init} *belonging to the group and* $1 \leq j \leq m$.

In an evolved abductive context $\mathtt{DAC}^{'}$, the group of agents does not need to be the same as that of the starting context DAC, with the exception of the initial agent A_{init}. Within the scope of this paper it is also assumed that the program Π and the integrity constraints IC of an agent do not evolve.

Definition 5 (Distributed Explanation)
Let DAC *be a distributed abductive context, let* $C = \{A_1, \ldots, A_k\}$ *be a cluster of agents and let* $\mathcal{A}_c$ *be the abducible predicates that appear in the agents in* C*. Now, let* $L_{\mathcal{A}}$ *denote the set of all ground literals with predicates in* $\mathcal{A}_c$ *and negative ground literals with non-abducible predicates that appear in* $\bigcup_{1\leq j\leq k}(\Pi_j \cup IC_j)$*. Then an* abductive explanation of DAC with respect to the cluster C *is a set of ground literals* $\Delta \subseteq L_{\mathcal{A}}$ *for which there exists a ground instance* $G\theta$ *of the goal* G *such that*

$(\bigcup_{1\leq j\leq k}\Pi_j) \cup \Delta \models G\theta$ *and*
$(\bigcup_{1\leq j\leq k}\Pi_j) \cup \Delta \models \bigcup_{1\leq j\leq k} IC_i^{\Delta}$

where IC_i^{Δ} *is the set of integrity constraints in* A_i *whose abducibles unify with a literal in* Δ.

The DARE algorithm uses two main phases, called respectively *global abductive derivation* (GAD) and *global consistency derivation* (GCD) to compute an abductive explanation.

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 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 standard SLDNF resolution (step (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 (step (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 (step (1) of GAD). If L is a ground base abducible and L^c (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 (step (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 step (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 (step (6) of GAD). This is the point where in the standard KM proof procedure 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 consistency 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 $\{L\} \cup \Delta$ (step (7) of GAD). The overall process of the distributed abductive derivation is diagrammatically represented in Figure 1.

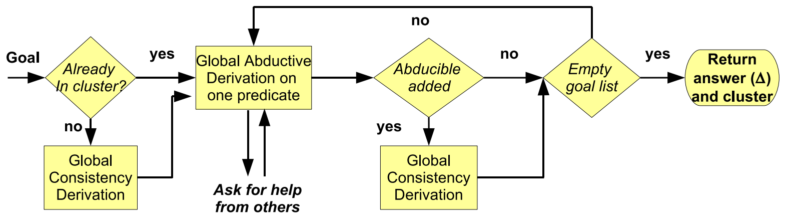
Goal
Already In cluster?
yes
no
Global Abductive Derivation on one predicate
Global Consistency Derivation
Ask for help from others
Abducible added
no
yes
Global Consistency Derivation
Empty goal list
no
yes
Return answer (Δ) and cluster

Figure 1: Global Abductive Derivation

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 in 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 resolve L with the integrity constraints in A, if at least one of these integrity constraints fails, 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, a *local abductive derivation* and a *local consistency derivation*. 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 local abductive derivation but with SLDNF resolution applied only between L and its integrity costraints instead of L and its rules (see step (1) of `GCD` and setp (4) of the local abductive derivation). While checking the consistency of L with its 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 local abductive derivations 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 2.

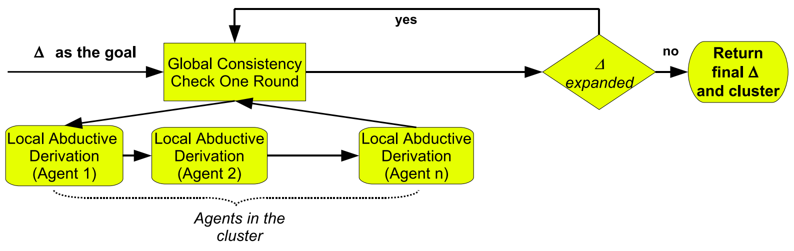
Δ as the goal
Global Consistency Check One Round
yes
Δ expanded
no
Return final Δ and cluster
Local Abductive Derivation (Agent 1)
Local Abductive Derivation (Agent 2)
Local Abductive Derivation (Agent n)
Agents in the cluster

Figure 2: Global Consistency Derivation

At the beginning of a GCD, an agent A from the cluster is chosen which starts a local abductive derivation taking as goal 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 local consistency derivation in A. These become *must-fail* goals. The local consistency derivation proceeds in a similar way as described in Section 2 for the KM proof procedure. The main difference in this case is step (5). When the literal L to fail is a not assumed non-base ground abducible, a global abductive derivation is called with goal L^c. In this case, although the global abductive derivation 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 expland to include agents with reasoning capabiities 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 to 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" is, in a sense, similar to the task of succeeding a positive non-abducible predicate[2]. The local consistency derivation behaves in a similar way as the global abductive derivation case. If the current agent cannot fail a non-base abducible, it can ask agents outside the cluster to help 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 C means that all the agents in C are not able to prove its complement. Negation as failure has therefore, in this case, the same meaning as in [10] 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 predicate means finding an agent in the system who is able to prove the non abducible predicate. This reflects the standard concept that failing to prove a negated literal is in essence equivalent to finding a proof for its complement.

3.1.1 The DARE algorithm

This section first illustrates our distributed abductive inference algorithm via the example of meeting scheduler and then gives its formal definition together with an informal proof of its soundness. Building upon the initial example given in Section 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 responsable for organising the meeting. A second lecturer (A_8 below) may join during the execution. 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 `studentName`, `lecturerName` and `tutorName` (as it was also the case in the previous single-agent example).

The distributed abductive context of our example is the tuple $DAC = \langle \{\Pi_i\}, G, \{IC_i\}, A_1, \mathcal{A} \rangle$ where the knowledge base Π_i and integrity constraints IC_i of the agent A_i, for $1 \leq i \leq 8$, are as defined below, the goal G is $\leftarrow$`conveneMeeting(T)`, and the set of abducible predicates is given by $\mathcal{A} = \{$`studentName`, `lecturerName`, `tutorName`$\}$.

A_1 **(Convener)**

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

A_2 **(Tutor)**

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

$$IC_2 = \left\{ \begin{array}{l} \leftarrow \texttt{tutorName(pat)}, \texttt{studentName(dan)} \\ \leftarrow \texttt{tutorName(pat)}, \texttt{lecturerName(joe)} \end{array} \right\}$$

[2]Recall that a non-base abducible is a negated non-abducible predicate.

A_3 **(Student)**

$$\Pi_3 = \begin{cases} \texttt{studentCanAttend(T)} \leftarrow \texttt{studentName(ben),free3(T).} \\ \texttt{free3(monday). free3(thursday). free3(friday).} \end{cases}$$

A_4 **(Student)**

$$\Pi_4 = \begin{cases} \texttt{studentCanAttend(T)} \leftarrow \texttt{studentName(dan),free4(T).} \\ \texttt{free4(monday). free4(wednesday).} \end{cases}$$

A_5 **(Nursery)**

$$\Pi_5 = \{ \texttt{nursery(wednesday). nursery(friday).}$$

A_6 **(Lecturer)**

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

A_7 **(Timetabler)**

$$\Pi_7 = \begin{cases} \texttt{freeFromTeaching(T,X)} \leftarrow \texttt{not teaching7(T,X).} \\ \texttt{teaching7(T,X)} \leftarrow \texttt{teachingJuniors7(T,X).} \\ \texttt{teaching7(T,X)} \leftarrow \texttt{teachingSeniors7(T,X).} \\ \texttt{teachingSeniors7(thursday,joe). teachingJuniors7(wednesday,rob).} \end{cases}$$

A_8 **(Lecturer)**

$$\Pi_8 = \begin{cases} \texttt{lecturerCanAttend(T)} \leftarrow \texttt{lecturerName(rob),not tired8(T),freeFromTeaching(T,rob).} \\ \texttt{tired8(thursday).} \end{cases}$$

Typically, the convener is the agent that makes the initial query $\leftarrow$`conveneMeeting(T)` with an empty set Δ of assumptions and a cluster consisting of only himself (i.e $C_1 = \{A_1\}$). 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 C. If the only available agents are $A_1 \dots A_7$, 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 $C = \{A_1, A_2, A_3, A_5, A_7, A_8\}$. 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 calculated 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 atoms, so step 1 in each of these agents succeeds trivially, and two different clusters $C_1 = \{A1, A3\}$ and $C_2 = \{A1, A4\}$ are formed.

2. Student agent A_3 (resp. A_4) starts its GAD process, matching `studentCanAttend(tuesday)` to the head of one of its program clauses to derive the subgoal `studentName(ben)` (resp. `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, C_1, to check that the abduced atom does not violate any integrity constraint of the agents in C_1 (i.e. A_1 and A_3). This is clearly the case as neither agent in 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 request 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 subgoal `tutorCanAttend(wednesday)`. A_2 temporarily abduces `tutorName(pat)` and activates a GCD which fails because of the first IC of A_2.

5. Agent 1 again backtracks and solves `day1(thursday)`, which leads to A_3 succeeding with $\Delta = \{$`studentName(ben)`$\}$. A similar computation as before follows, but using this time A_2. This results in the new abducible `not lecturerName(joe)` being added to Δ, so far given by $\{$`studentName(ben)`, `tutorName(pat)`$\}$, for the second IC of A_2 to be satisfied. Unfortunately, the `free2(thursday)` sub-goal of A_2 will fail.

6. Agent 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)`, `not lecturerName(joe)`$\}$ of abducibles and cluster $C = \{A1, A2, A3, A5\}$.

7. Agent 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 `not 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 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 atom `lecturerName(rob)` giving $\Delta = \{$ `lecturerName(rob)`, `studentName(ben)`, `tutorName(pat)`, `not 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 `not tired8(friday)`. This requires adding `not 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 8.

9. Agent 8 next solves the subgoal `freefromTeaching(friday,rob)` by recruiting agent A_7. This gives then set of assumptions
   $\Delta = \{$`studentName(ben)`, `tutorName(pat)`, `not lecturerName(joe)`, `lecturerName(rob)`, `not tired8(friday)`$\}$ 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), not lecturerName(rob)`. When `studentName(dan)` is abduced as part of solving the sub-goal `studentCanAttend(wednesday)` in A_4, the GCD will require `not 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), not 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.

The full algorithm is defined below. In the following, Δ is the set of ground abducibles that are collected during the proof and 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 C of the final distributed abductive context $\langle\{\Pi_i\}, G, \{IC_i\}, A_{init}, \mathcal{A}\rangle$. The algorithm starts with the agent A_{init} performing a global abductive derivation for the query G with $\Delta = \emptyset$ and $C = \emptyset$. The first step in this abductive derivation will add agent A_{init} to the cluster.

Global Abductive Derivation

Let A be the current agent, Δ be the current set of abduced literals, C be the current cluster and G the current goal ($\leftarrow L_1, \ldots, L_k$). If G is (reduced to) the empty goal [] then the global abductive derivation **succeeds** and Δ and 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 $\mathcal{C}'$.

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 $B \cup G^{'}$[3] with $\Delta' = \Delta$ and $\mathcal{C}' = \mathcal{C}$.

3. If L is a non-abducible and the previous rule does not apply, then if there exists a global abductive derivation on $\leftarrow L$ with Δ and $\mathcal{C}$ by a helper agent H (whether in the group or in the current cluster), and Δ' and $\mathcal{C}'$ (the first rule 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 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 L^c is in Δ, if there exists a global consistency derivation on $\{L\} \cup \Delta$ with $\mathcal{C}$, and Δ' and $C^{'}$ 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 $\{\leftarrow L^c\}$ 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 $C^{'''}$ are obtained after the local consistency derivation, then A continues the global abductive derivation on G' with Δ' and $\mathcal{C}^{'}$, where Δ' and $C^{'}$ are obtained after the global consistency derivation.

Global Consistency Derivation

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

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 on $G = \Delta_{k-1}$ and $\Delta = []$ by A_k, and Δ_k and C_k are obtained after the derivation:
   - If $k < n$, then Δ_k is passed to A_{k+1} with cluster $C = C_k$
   - 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** and Δ_n. If one consistency check round succeeds but $\Delta_0 \subset \Delta_n$, then start another consistency check round on Δ_n.

[3]Note that the concepts of goal as a set of literals and goal as its denial representation are used interchangeably here.

Local Abductive Derivation

The local abductive derivation of an agent A is to make sure that the passed in goal (a set of abducibles) does not violate any integrity constraint of A. Since the goal 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 $L^c \in \Delta$, 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 $L^c \notin \Delta$. Let $\mathcal{I}$ be the set of integrity constraints containing L, and $\mathcal{I}'$ be the set obtained by removing L from each constraint in $\mathcal{I}$. If there exists a successful local consistency derivation on $\mathcal{I}'$ with $\Delta \cup \{L\}$ and $\mathcal{C}$, and Δ' and $C^{'}$ are obtained after the local consistency derivation, then A continues the local abductive derivation on G' with Δ' and $C^{'}$.

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 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 []. Otherwise, let $F' \cup G = F$ and G' is obtained by removing a literal L from G, Δ' and $C^{'}$ are obtained while applying one of the following rules:

1. If L is a non-abducible: Let $\mathcal{S}$ be the set of all the instantiated non-empty bodies of the rules in the (local) agent whose heads can match with L, A continues the local consistency derivation on $\mathcal{S} \cup F'$ with $\Delta' = L^c \cup \Delta$ 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 $L^c \in \Delta$, 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 $L^c \notin \Delta$, then A continues the consistency derivation with F' and $\Delta' = \Delta \cup L^C$ and $C^{'} = C$.
5. If L is non-base ground abducible and $L \notin \Delta$, if there exists a successful global abductive derivation on $\leftarrow L^c$ 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.

The soundness of the algorithm requires showing that given a distributed abductive context DAC=$\langle\{\Pi_i\}, G, \{IC_i\}, A_{init}, \mathcal{A}\rangle$, and given a successful global abductive derivation, by A_{init}, of G with a final set Δ of abduced assumptions and a final cluster C of agents of an evolved abductive context $\mathtt{DAC}^{'}$, then Δ is an abductive explanation of $\mathtt{DAC}^{'}$ with respect to the cluster C for the goal G. A proof of the soundness property is given below.

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

Proof: The proof is by induction on the number k of agents in $\mathcal{C}$. Without loss of generality it is assumed that $A_1 = A_{init}$.

Base Case: $(k = 1)$
In this case the final cluster C returned by the global abductive derivation is composed of only one agent $C = \{A_1\}$. It is easy to see that in this circumstance the DARE algorithm behaves as the KM abductive proof procedure as all computations are local to the single agent in the cluster. A DARE abductive derivation coincides with a KM abductive derivation and the definition of abductive explanation of the DAC with respect to a singleton cluster for a given goal is implied to the notion of KM abductive explanation given in Definition 2. Hence, by the soundness of KM proof procedure [16], we can conclude that Δ is an abductive explanation of DAC with respect to the cluster $C = \{A_1\}$.

Induction Hypothesis: We assume that for any global abductive derivation that computes Δ, as abductive solution for an instance $G\theta$ of the goal G, with a cluster $C = \{A_1, \ldots A_k\}$, for $k \geq 1$, of agents in a corresponding evolved abductive context $\mathtt{DAC}_{fin}$, Δ is an abductive explanation of $\mathtt{DAC}_{fin}$with respect to the cluster C.

Inductive Step: $(k + 1)$
Let us assume that Δ has been computed by a GAD with a cluster $C = \{A_1, \ldots, A_{k+1}\}$. We need to show that Δ is an abductive explanation of $\mathtt{DAC}_{fin}$with respect to this cluster C of $k + 1$ agents. The underlying idea is to show that any cluster $\mathcal{C}$ of $k + 1$ agents, satisfying the condition of $\mathtt{DAC}_{fin}$, can be reduced to a cluster $\mathcal{C}^+ = \{A_1^+, \ldots, A_k^+\}$ of k agents such that their respective background knowledge $\Pi_1^+ = \Pi_1 \cup \{\Pi_{k+1}$, and $\Pi_j^+ = \Pi_j$, for $1 < j \leq k$, and integrity constraint $IC_1^+ = IC_1 \cup IC_{k+1}$ and $IC_j^+ = IC_j$, for $1 < j \leq k$. Of course Δ is still an abductive solution for the given goal instance $G\theta$. So by inductive hypothesis we could say that Δ is an abductive explanation of the abductive context $\mathtt{DAC}_{fin}$with respect to the cluster C^+, and since the cluster C^+ is equivalent to the cluster $C = \{A_1, \ldots, A_{k+1}\}$, Δ is an abductive explanation of $\mathtt{DAC}_{fin}$with respect to the cluster C. The knowledge bases Π_j^+, for $2 \leq j \leq k$, are consistent with the integrity constraints IC_j^+ as agents A_j^+ are the same as agents A_j, for $2 \leq j \leq k$. This is also the case for agent A_1^+, namely $\Pi_1 \cup \Pi_{k+1} \models IC_1^\Delta \cup IC_{k+1}^\Delta$, since A_{k+1} belongs to the cluster and step (1) of the global abductive derivation guarantees consistency of the integrity constraints $IC_1^\Delta \cup IC_{k+1}^\Delta$ with respect to $\Pi_1 \cup \Pi_{k+1}$. Hence, given the GAD that has computed Δ with cluster $C = \{A_1, \ldots, A_{k+1}\}$, there exists an exactly equal GAD that computes Δ but with respect to the cluster $C^+ = \{A_1^+, A_2, \ldots, A_k\}$, and Δ can be computed again as abductive solution for the goal instance $G\theta$ but using a cluster of only k agents.

4 DARE Implementation Architecture

This section describes the architecture and main features of the DARE system. This is an *open distributed abductive inference* system that supports collaborative proof of a query given to or internally generated by any agent in the system. Each agent is equipped with its own logic program knowledge base with associated integrity constraints. The agents in the system can dynamically change during a particular inference process. This can result in the agents involved in the proof, the current proof cluster, to change. DARE is therefore an *open* distributed abductive inference system. As shown in Section 3, its distributed abductive algorithm is flexible enough to guarantee the soundness of the proof process despite this key dynamic feature. The agents can have overlapping or disjoint knowledge base. Two knowledge bases are disjoint if they do not share the same definition for a predicate. The system assumes the communication between agents to be safe and reliable, namely 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 currently not been considered. In its current form, the system does not allow for the possibility of malicious agents interfering in the collaboration between other agents.

4.1 Architecture Overview

As mentioned already, the DARE system comprises of an open *group* of agents. A query can be submitted to any of the agents, A_i say, in the group, 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 (i.e logic program Π_i). 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 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. These two features are explained and illustrated below.

4.2 Inter-Agent Communication

Inter-agent communication is via asynchronous message-passing, using KQML [11] performatives. The messages are communicated between the agents using a communications demon, Pedro[20]. This will route messages with a specified agent destination to that agent. An agent identity has the form `agentName@host`[4]. A thread named `Th` within an agent has an identity of the form `Th:agentName@host`. Messages can be addressed either to the agent or to a specific thread within the agent. If the former, the message is sent to the initial agent thread, else it will be routed directly to the named thread within the agent. All threads have their own message buffer of received unread messages. Pedro will also forward messages posted to it without a specified agent destination using lodged message pattern subscriptions. We use this to give some of the functionality of a KQML matchmaker[17]. When an agent is launched it first connects to the Pedro demon. Its name `agentName` and host name `host` are recorded by Pedro so that Pedro can route to the agent all messages addressed to `agentName@host`, or to a thread `Th:agentName@host`[5]. The agent then posts a `register` message to Pedro containing its identity to inform the existing DARE agents of its arrival. This is its entry into the DARE group of agents. This message will be forwarded to all DARE agents because they will have subscribed for such `register` messages. The agent then lodges a subscription with Pedro for both `register` and `unregister` messages, so that it will become aware of new agents that join afterwards, and when an agent leaves. It then posts `advertise` messages to Pedro announcing the predicates for which it is willing to offer abductive proofs and it lodges subscriptions for advertisements posted by other (usually new) agents that mention predicates for which it may require proof help.

Notice there is one key difference between Pedro and a standard KQML matchmaker. The latter will remember avertisements as well as subscriptions for advertisements, whereas Pedro only remembers the subscriptions. This means that whenever an agent receives a `register` message it must send its advertisements directly to the new agent. Each agent maintains its own "yellow pages" directory of other agents in the DARE group. For each such agent it stores the agent identity and the predicates it has advertised (and not yet unadvertised).

[4]Only one agent with a given name can connect to Pedro from each host but agents with the same name can connect from different hosts.

[5]The connection to Pedro opens a TCP/IP connection for communication between Pedro and the Qu-Prolog process which is the agent, and is time-slice executing all of the agent's threads. Messages for the agent, or a thread within the agent, are sent down this TCP/IP connection and Qu-Prolog automatically inserts it into the appropriate thread message buffer.

4.3 Internal Architecture of Agents

Each agent has its own knowledge base, which is a normal logic program and a set of integrity constraints as defined in Section 3. Each is also equipped with the abductive algorithm described in Section 3. The agents are linked in an acquaintance network by means of the local "yellow pages" directories within each agent. Using its directory an agent can find suitable helper agents and request them to provide it with subproofs, as the need arises. When asked for such help, the helper agent will return the computed abductive answers (if any) to the requesting agent, one at a time.

Each agent can be involved in several proofs at the same time. They are *multi-threaded* and *multi-tasking*. Specifically, each agent has: a *Coordinator Thread* (CT) for handling proof requests, a *Directory Thread* (DT) for maintaining the "yellow pages" directory, a number of *Worker Threads* (WT) managing different proof tasks concurrently and independently, each of which is linked to a *Reasoning thread* (RT). In addition, there can be several *Broker threads* (BT) spawned by the RTs to outsource the finding of abductive proofs of certain conditions.

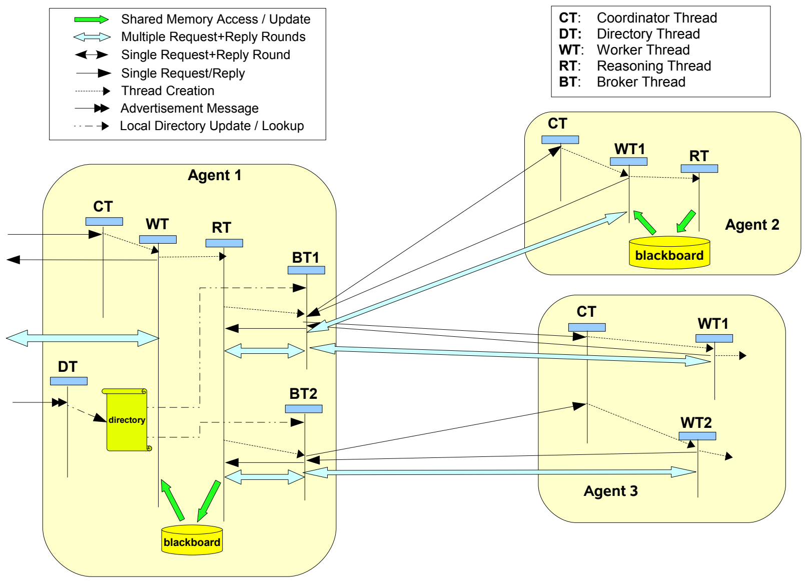

Figure 3: Agent Internal Architecture

In summary, the architecture of each agent supports three main functionalities: maintaining the directory, handling incoming proof requests and requesting help from other agents. These are described in detail in what follows.

Maintaining the Local Directory

The DT is a persistent thread responsible for maintaining the agent's "yellow pages" directory. The following tasks are performed by the DT:

1. It starts by performing the initialisation interaction with the Pedro server described above: connection, and the posting of the agent's `register`, `subscription` and `advertisement` messages. Messages forwarded by Pedro because of one of the subscriptions will be sent to the DT thread.

2. Whenever DT receives a `register` message from another agent, it sends its advertisements

to that agent. If it receives an `unregister` message from another agent, it removes all the advertisements for that agent from its directory.

3. It updates its local directory in response to `advertise` or `unadvertise` messages from another agent, whenever they are received.

Handling Incoming Requests

As mentioned above, the incoming proof requests are handled by the co-ordinator thread, CT. To allow multiple reasoning tasks to run simultaneously, for each *accepted* incoming request CT spawns a worker thread WT. An incoming request includes the specific goal to be proven, the identities of all agents in the current cluster, and the current Δ of abduced conditions. If the agent is in the current cluster, the request is always accepted. If not in the current cluster, *and* the given Δ satisfies its consistency constraints (more precisely, if a global consistency check on Δ starting with that agent succeeds - case 1 of the global abductive derivation algorithm), the request is accepted. If a new agent cannot accept a proof request, its CT sends an `sorry` message in response to indicate that it cannot join the current cluster, given the current Δ.

A spawned WT immediately spawns a reasoning thread RT to perform the abductive reasoning task using the DARE abductive meta-interpreter. It then sends a `ready` message to the client agent. More specifically, it sends the message to the thread within the client agent that sent the proof request. As explained below, this will be a broker thread within the client agent, and all communication regarding the proof task is actually between the WT thread and this broker thread.

An RT may itself seek sub-proof help from any number of other agents via *Broker Threads* (BT) that it creates. It also eagerly generates all the abductive answers for the condition it is proving and stores them in order of generation on a *blackboard* internal to the agent (see Figure 3) The WT waits for and services `next` requests from its client agent thread. On receipt, it removes the next result from its RT from the blackboard and sends it back to the client thread. Of course, the WT thread must wait if the next answer has not yet been found by its RT, until it is found and placed on the backboard. The RT informs its WT when it has explored all proof paths, and no more answers will be found. At this point the WT will respond with an `eos` message to its client when it receives a `next` request.

The rationale for having a producer-consumer relationship between the WT and its RT using a blackboard as a buffer is to isolate the reasoning process from the client agent. The RT just searches for *all* possible answers for its given reasoning task, adding each to the blackboard as it is found, independently of its use by the client. There is one exception. If the WT receives a `discard` message from the client thread, it erases any unused answers placed on the blackboard by its RT and terminates the RT if it is still running. It is worth noting that after a reasoning thread is asked to terminate, it will send a `discard` message to all its active broker threads (see next section), which will in turn forward this to the WTs in the server agents that are finding sub-proof answers for them.

Getting Help From Other Agents

As previously explained, each agent keeps a local "yellow pages" directory regarding sub-proof capabilities that have been advertised by other agents. This section describes in detail the interactions when an agent asks for sub-proof help.

Whenever an RT decides to ask for help, it creates a BT to handle all the request-response communications with helper agents. BT is given the goal to outsource, the identities of the agents in the current cluster, and the current Δ. Each RT maintains the cluster set of agent identities for the current state of its proof.

BT checks the advertisements in the agent's "yellow pages" directory to create a helper list. It then sends a request to the CTs of all the agents in the helper list giving the goal, the current Δ, and the identities of all agents in the current cluster. An implicit proof contract with a helper agent is established as soon as the BT receives from that agent a `ready` message, sent by the WT within the agent that will have been created for the proof task. BT then sends that WT a `next` message to ask for its first answer. Usually it will be able to send `next` messages to several helper agents. It then waits for the first proof to be returned by any of these helpers. When the helpers are on different hosts, this is an or-parallel search for alternative proofs.

Whenever the RT wants an alternative result for an outsourced sub-goal `G`, (either for the first time or during backtracking), it sends a `next` message to the BT handling answers for `G`. The BT searches its message buffer for an answer from any helper agent. These answers will have been inserted into its message buffer as and when they arrive, so the answers from each of the different helper agents will be interspersed in the buffer. There are three cases:

- A `tell` message is found and removed containing an answer. BT extracts the answer from the message and forwards it to the RT. BT then sends a `next` message to the helper agent WT thread which sent the message, so that if there is another answer from that agent, it can be returned and buffered in BT's message queue.
- An `eos` message is found. BT removes the sender from its list of helpers. When this list becomes empty, BT sends an `eos` message to its RT.
- There is no message in the buffer, but the helper list is non-empty. BT suspends until a message arrives.

Each `tell` message will contain the possibly instantiated sub-goal, a possibly augmented Δ, and a list of agent identities that should be added to the cluster set if this answer is used. This list will be non-empty if the helper agent is a new agent not in the cluster set passed to it in the sub-proof request, and if it has itself requested sub-proofs from agents not in the given cluster set.

Finally, each time the BT services a `next` request from its RT it checks to see if the local "yellow pages" directory contains the identity of any agent not on the current helper list that has since advertised ability to help with the sub-goal BT is handling. If so, this agent is added to the helper list and sent a sub-goal proof request allowing this late entrant to contribute candidate proofs.

The rationale for having a separate BT to handle the communications is to isolate the RT from the helper agents. BT forwards answers to its RT in the order that they arrive from the different helper agents and merges the answers. From the RT's point of view, a BT is an internal thread that can provide it with alternative proofs for a sub-goal as they are needed. RT can have several different BTs active and buffering results for different sub-goals. Whenever the RT backtracks to a sub-goal with an associated BT, it asks the BT for the next available result. If no more results are available (i.e. the `eos` message is returned by the BT), RT fails that sub-goal and continues the backtracking.

Each BT exits automatically after sending the `eos` to the RT. It is worth noting that a BT can have in its buffer a maximum of one answer from each helper agent since it sends the `next` request to a helper agent only after it as forwarded that helper's previous answer to its RT. Of course, independently the RT threads within each helper agent are finding all the answers and caching them on their internal blackboards. In this way the DARE system keeps network traffic to a minimum whilst still maintaining the reasoning performance since the RT can continue its reasoning when the BT is fetching the next result. Finally, whenever an RT of an agent is terminated by its linked WT, RT itself sends a `discard` message to all the BTs it has created. Upon receiving such a message, each of these BTs will forward the `discard` message to all the current helper agents.

5 Evaluation

In Section 3, the proof of the soundness of the distributed abduction algorithm was shown. Note that the openness of the DARE system does not effect the soundness. We can allow agents to join or leave the group and the proof cluster without effecting the soundness of the final proof. If an agent in the current cluster leaves before the proof is complete, the DARE system will discard any sub-proof of a condition C provided by that agent. It will backtrack to the point where the sub-proof of C from that agent was used to find an alternative proof. Alternative proofs may already be waiting, as the broker will have sent requests to all agents who have advertised that they can offer proofs for the predicate of C. The alternative proof of C may even be from an agent that has only recently advertised it can offer proofs for the predicate of C. When an agent A joins whilst a proof is in progress it immediately becomes available to provide extra candidate proofs for each outsourced goal of the current partial proof that uses one of its advertised predicates. The brokers in charge of collecting candidate proofs for these goals will immediately send out requests for proofs to the new agent A. However, its services cannot be used for a goal C' that has already been discarded on backtracking. That is, a possible overall proof that uses the knowledge of this newly arrived agent to provide an alternative proof of C', will not be in the DARE search space of possible proofs because of the late arrival of A. So, late arrival may mean a possible proof is missed, but not that a proof that is found will be unsound.

What about the "completeness" of the DARE system? The use of depth first backtracking search within each agent means that a proof may not be found if there is a non-terminating proof path in the search space of the local proofs being explored within that agent. But this is also true of a single agent abductive proof system that uses depth first backtracking search for proofs. In fact, because sub-proofs of outsourced conditions are pursued concurrently, DARE combines depth first backtracking search within each agent with occasional or-parallel invocation of search for alternative proofs. Thus, suppose that a DARE agent A invokes concurrent search for alternative proofs of a condition C by asking two helper agents $A1$ and $A2$ to try to find proofs. These agents will internally use depth first backtracking search. Suppose that $A1$ has a non-terminating branch in its search space, and because of this never returns a candidate proof. Proofs from $A2$ will still be sent to A. If the knowledge embedded in $A1$ and $A2$ was instead incorporated into A then its sequential backtracking search would not have found proofs offered by $A2$ if it had started by using the knowledge of $A1$.

But are there proofs that we would find, even using backtracking search, if we consolidated the knowledge of all the agents, rules are well as constraints, into one agent and applied a non-distributed abductive proof procedure? First we must decide which agents' knowledge we will consolidate, since DARE handles constant change in the group of available agents. In the discussion above, we have already seen that late arrival can effect ability to find a proof, so we will not be able to claim that DARE will find any proof that would be found by a single agent using the combined knowledge of the agents in the final proof cluster. We must insist that all such agents have been available, for each of their advertised predicates, throughout the entire proof, in order to be able to compare DARE with non-distributed abductive proof.

We have argued that the DARE system will even be able to find some proofs that a single agent using depth first backtracking search would not because of DARE's weak or-parallelism. However, this outsourcing of sub-goals also has the potential to generate a non-terminating loop of outsourcing requests. Consider, for instance, a group of three agents with the following knowledge:

A1	**A2**	**A3**
`p` $\leftarrow$ `a.`	`p` $\leftarrow$ `b.`	`p` $\leftarrow$ `c.`

where all the predicates are non-abducible and all the agents have advertised the predicate p. Imagine that *A1* is asked about p. It cannot prove it so it may outsource the proof of p to *A2* and *A3*. Clearly, *A2* will not prove it either. If the selection of helpers is purely based on the advertisement, *A2* would then ask *A1* and *A3* about p. This is similar for *A3*, who would seek help from *A1* and *A2* for the same goal. A loop occurs and therefore the distributed abductive algorithm will not terminate.

Our implementation of the algorithm overcomes this problem by having an agent pass a *DontAsk* list 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 *A1* outsources the proof of p to *A2* and *A3*, the list *DontAsk* $= \{A1, A2, A3\}$ is also passed to the two helpers. When *A2* fails to prove p, it cannot forward the goal to *A1* and *A3* even though they have made the appropriate advertisements. This is the same for *A3*, who cannot forward the same goal to *A1* and *A2*. Thus, the situation of looping due to *mutually asking for help* is avoided. However, the *DontAsk* technique cannot avoid the following looping situation:

A4	**A5**
`p` $\leftarrow$ `q.`	`q` $\leftarrow$ `p.`

where *A4* advertises p and *A5* advertise q. If *A4* is asked to prove p, it resolves it and sends q with *DontAsk* $= \{A1, A4\}$ to *A5* (Δ is always empty in this scenario). *A5* resolves q to p. But since the *DontAsk* list only restricts the agents from whom *A5* may seek help regarding its given goal q, *A5* can ask *A4* for a the proof of p just with the constraint that it should not forward the sub-goal p to the requesting agent, *A5*. But, *A4* is again free to ask *A5* for help with q, and looping occurs. But the pair of rules, {`p` $\leftarrow$ `q`, `p` $\leftarrow$ `q`} will also result in a non-terminating proof within a single agent system. In DARE, one can modify the distributed algorithm to perform iterative deepening search with respect to outsourced proofs (e.g. by limiting the number of times the outsourcing of sub-proofs is allowed, or by setting a timeout limit for the brokers).

The above discussion leads us to our completeness result for DARE that compares it with the KM abductive system within a single agent.

Theorem 5.1 Let $A_1, \ldots, A_n$ be a group of agents and let $\texttt{DAC} = \langle \{\Pi_i\}, G, \{IC_i\}, A_{init}, \mathcal{A} \rangle$, with $1 \leq i \leq n$, be a distributed abductive context. Let Δ be an abductive explanation of `DAC`, with respect to a cluster $C = \{A_1, \ldots, A_k\}$, for $k \leq n$, where all the agents in C have been in `DAC`, throughout the whole proof process.

If the KM abductive proof procedure can compute such a Δ with respect to the abductive context $\langle \Pi, G, IC, A \rangle$, where $\Pi = \bigcup \Pi_i$ and $IC = \bigcup IC_i$, for each $1 \leq i \leq k$, then Δ is the output of a global abductive derivation for a goal instance $G\theta$, computed by the cluster $C = \{A_1, \ldots, A_k\}$, using the same search strategy as KM within each agent.

6 Related work

Distributed abduction

ALIAS [4, 25] inspired our work and is the only other distributed abductive system of which we are aware. As far as we understand the ALIAS system, and its extension [5], which is coupled with the LAILA [8] language for co-ordinating abductive reasoning amongst a group of agents, and expressing the knowledge of each agent, these are some key differences:

- The acquaintance relation between ALIAS agents is specified in each agents' knowledge base by explicit annotations of sub-goals in the logic program rules which specify which other agent or agents should be queried for proofs of that sub-goal. In DARE, the agents that can be asked for a proof of a sub-goal are determined as and when that sub-goal needs to be solved using a local directory of agents who have advertised the willingness and capability to help.

- In DARE, any agent who cannot join the current proof cluster because of incompatiblity of its knowledge base with agents already in the cluster, concerning the current Δ, is ignored. Moreover, we only require that a proof cluster to be mutually consistent *with the* Δ *for the proof.* With respect to other assumptions they can have incompatible knowledge. We believe this is not the case with ALIAS. We believe that it assumes that all the agents who might contribute to a proof have mutually consistent logic knowledge bases.

- When an ALIAS agent abduces a new hypothesis, it has to ask all the agents in the bunch of co-operating agents (in our terms the cluster) to check for consistency of the augmented Δ. This is similar to the global consistency check in DARE, and may result in expansion of the set Δ. However, in checking that each of its consistency constraints is satisfied (that as a query it fails), an ALIAS agent A can only ask a known other agent A' for help in trying to show that a negated condition `not C` fails, where C is non-abducible, by trying to prove `C`. It can do this by using a rule for `C` in its knowledge base that explicitly queries A'. In DARE, a new agent can be dynamically recruited into the cluster in order to prove `C`, and it may do so by expanding the Δ.

- We believe that an ALIAS bunch can only be actively pursuing one distributed inference at a time, whereas in DARE there can be several proofs being pursued concurrently inside each agent, which can also be simultaneously involved in quite different proof clusters.

- ALIAS does have the concept of rambling agent that can join an existing bunch. The new agent does not join to help with a sub-goal of a proof. On joining, the new agent offers a set of abducibles Δ' to be added to the Δ of the terminated proof of the bunch. This Δ' can be viewed as a different kind of query being posed to the bunch. The new agent is asking if its assumptions Δ' are compatible with the current shared assumptions Δ and knowledge bases of the bunch. The adding of Δ' to Δ will trigger a global consistency check over the enlarged bunch. This in turn may lead to an expansion of Δ' to Δ", in order to satisfy all the consistency constraints. Δ" is interpreted as an 'answer' to the query: "are my assumptions Δ' ok", of the rambling agent. This is an variant of the DARE abductive proof process which it would be interesting to investigate.

More general distributed or multi-agent inference

Our DARE implementation language Qu-Prolog has the occurs check in its unification algorithm and was developed explicitly for implementing sound first order inference systems. The use of its features to implement agent based co-operative inference systems for full first order predicate logic is illustrated in [21] and [26]. The former has each agent using a tableaux style inference system, the latter uses an algebraic approach to resolution inference using knowledge expressed as clauses.

[14] is a co-operative agent based system for program verification which makes use of broker agents to find agents with appropriate expertise for specialised sub-proofs. [12] is a similar approach to linking agent fronted theorem proving expertise. It links hybrid theorem provers together using KQML messaging.

7 Discussion and Future work

This paper describes a distributed abductive reasoning (DARE) system proposing a new distributed abductive algorithm and the architecture of its multi-threaded distributed Q-Prolog implementation. The (DARE) algorithm extends the (KM) abductive proof procedure by allowing the knowledge base 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. Each DARE agent is multi-threaded, allowing the system to handle different queries and perform different reasoning tasks concurrently. A soundness proof of the DARE algorithm has also be given, completeness and termination discussed.

The reasoning agent of the system and the inference meta-interpreter for the proposed algorithm have been implemented with Qu-Prolog 8. They have been tested on several Linux PC with $3.0GHz$ processor and $1GB$ RAM. In order to test the system in a harsh environment, we have also ported Qu-Prolog 8 to the *Gumstix computers*[6], which have the size of a chewing gum. It has a $400MHz$ processor, $64MB$ RAM and $16MB$ ROM, and it is running ARM Linux. The Gumstix computers can connect to a 802.11 Wireless LAN with suitable expansion boards. This allows us to explore applications of DARE whereby some agents are on Gumstix computers, with perhaps quite simple knowledge bases. Others with larger knowledge bases and the Pedro server can be hosted on PCs on the same Wireless LAN. This type of configuration is particularly suitable for applications such as multi-sensor or multi-robot co-operative sense data interpretation, and multi-robot planning.

The current meta-interpreter implementation of the DARE algorithm presupposes that an agent in a given cluster does local abductive reasoning whenever it has the appropriate reasoning capability. Other agents are asked for help only when the local abductive derivation has failed. A possible extension/variation of this meta-interpreter is to allow for *lazy* agents. These are agents that even though they have appropriate reasoning capability for answering a given query, opt to ask for help to other agents instead of undertaking their local abductive derivation. Of course appropriate heuristics will be needed as to when an agent should/could be lazy and when not, to guarantee progress of the computation as well as not network overloading with large number of messages between agents.

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. Future extension of this work includes the development of specialised meta-interpreters tailored to the particular domains of application. We are currently developing a distributed abductive planner, based in Event Calculus, 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. Other applications will include the use of distributed abductive reasoning for analysis and verification of protocols in distributed Web-services and Web-service composition. This will provide extensions to existing non-distributed abductive reasoning application such as [1].

Acknoledgement

This research is continuing through participation in the International Technology Alliance sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence.

References

[1] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Specification and verification of interaction protocols: a computational logic approach based on abduction. Technical report, Dipartimento di Ingegneria di Ferrara, 2003.

[2] Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[3] Ofer Arieli, Bert Van Nuffelen, Marc Denecker, and Maurice Bruynooghe. Coherent composition of distributed knowledge-bases through abduction. *Lecture Notes in Computer Science*, 2250:624+, 2001.

[4] A. Ciampolini, E. Lamma, P. Mello, C. Stefanelli, and P. Torroni. An implementation for abductive logic agents. *Lecture Notes in Computer Science*, 1792:61+, 2000.

[5] A. Ciampolini, E. Lamma, P. Mello, and P. Torroni. Rambling abductive agents in alias. In *Proc. ICLP Workshop on Multi-Agent Sytems in Logic Programming (MAS'99)*, 1999.

[6] A. Ciampolini, P. Mello, and S. Storari. Distributed medical diagnosis with abductive logic agents. In *Proceedings of BIXMAS 2002 workshop*, 2002.

[6]http://www.gumstix.org

[7] 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(1–2):65–91, 2003.

[8] Anna Ciampolini, Evelina Lamma, Paola Mello, and Paolo Torroni. LAILA: a language for coordinating abductive reasoning among logic agents. *Computer Language*, 27(4):137–161, dec 2001.

[9] K. L. Clark, P. J. Robinson, and S. Zappacosta-Amboldi. Multi-threaded communicating agents in Qu-Prolog. In F Toni and P. Torroni, editors, *Computational Logic in Multi-agent systems.* LNAI 3900, Springer, 1998.

[10] K.L. Clark. Negation as failure. *Logic and Data Bases*, pages 293–322, 1978.

[11] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proceedings 3rd International Conference on Information and Knowledge Management*, 1994.

[12] A. Franke, S. Hess, C. Jung, M. Kohlhase, and V. Sorge. Agent-oriented integration of distributed mathematical services. *Universal Computer Science*, 5(3):156–187, 1999.

[13] Michael Gelfond and Vladimir Lifschitz. The stable model sematics for logic programming. In *Proceedings of International Conference of Logic Programming*, pages 1070–1080, 1988.

[14] C. Hunter, P. Robinson, and P. Strouper. Agent-based distributed software verification. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science*, volume 24 of *ACM International Conference Proceeding Series*, pages 159–164, 2005.

[15] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.

[16] Antonis C. Kakas and Paolo Mancarella. Database updates through abduction. In *VLDB '90: Proceedings of the 16th International Conference on Very Large Data Bases*, pages 650–661, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[17] D. Kuokka and L. Harada. On using KQML for Matchmaking. In *1st International Joint Conf. on Multi-agent Systems*, pages 239–245. MIT Press, 1995.

[18] Jiefei Ma. Distributed abductive reasoning system and abduction in the small. Technical report, Department of Computing, Imperial College London, 2007.

[19] T. Menzies. Applications of abduction: Knowledge level modeling. *International Journal of Human Computer Studies*, 45:305–355, 1996.

[20] P. J. Robinson. Pedro Reference Manual. Technical report, http://www.itee.uq.edu.au/~pjr, 2007.

[21] P. J. Robinson, M. Hinchley, and K. L. Clark. QProlog: An Implementation Language with Advanced Reasoning Capabilities. In M. Hinchley et al, editor, *Formal Appraches to Agent Based systems, LNAI 2699.* Springer, 2003.

[22] M. Shanahan. Perception as Abduction. *Cognitive Science*, 29:103–134, 2005.

[23] Murray Shanahan. *Solving the frame problem: a mathematical investigation of the common sense law of inertia.* MIT Press, Cambridge, MA, USA, 1997.

[24] Murray Shanahan. An abductive event calculus planner. *Journal of Logic Programming*, 44(1-3):207–240, 2000.

[25] Paolo Torroni. *Reasoning and Interaction in Logic-Based Multi-Agent Systems.* PhD thesis, Department of Electronics, Computer Science and Systems, University of Bologna, Italy, 2002.

[26] S. Zappacosta. Distributed implementation of a connection graph based on cylindric set algebra operators. In M. Hinchley et al, editor, *Formal Appraches to Agent Based systems, LNAI 2699.* Springer, 2003.