Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Distributed Logic-based Learning in Multi-agent Systems

Author:
Paweł Gomoluch

Supervisor:
Alessandra Russo

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

September 2015

Abstract

In this thesis a distributed ILP (Inductive Logic Programming) learning system is introduced, called Distributed Top-directed Abductive Learning (DTAL). DTAL is based on existing solutions such as TAL (Top-directed Abductive Learning - a centralised ILP system, introduced by Domenico Corapi) and DAREC (Distributed Abductive Reasoning with Constraints - a distributed abductive system, developed by Jiefei Ma). A definition of a distributed learning task is stated. Soundness and completeness of DTAL in finding solutions of a distributed ILP task are proven. Furthermore, an experimental implementation of DTAL is described, developed as a meta-rule layer that can be put on top of existing DAREC implementation.

Contents

Chapter 1

Introduction

Distributed learning is one of the most interesting challenges in both machine learning and multi-agent systems areas. While a lot of work has been done in the field of multi-agent reinforcement learning (see Panait and Luke (2005) for a general survey), less effort has been devoted to other types of learning. In particular, the number of publications dealing with distributed Inductive Logic Programming (ILP) is very limited.

In this thesis, *Distributed Top-directed Abductive Learning* (DTAL) is introduced – a general framework for performing distributed ILP learning in a multi-agent system, building upon existing solutions such as TAL (*Top-directed Abductive Learning*, Corapi et al. (2010) and Corapi (2011)) and DAREC (*Distributed Abductive Reasoning with Constraints*, Ma (2011)). The system uses the idea of translating a learning task into a corresponding abductive task, employed by TAL, and relies on DAREC to perform distributed abduction. Theorems about soundness and completeness of DTAL are stated and proven. An experimental implementation of the system is also developed in Prolog.

This thesis is organised as follows. In Chapter 2, the basic concepts of Inductive and Abductive Logic Programming are introduced. TAL and DAREC systems are presented, forming the basis for the newly developed distributed system. In Chapter 3, the distributed learning task is defined. Then the DTAL system is defined, capable of solving the task. Soundness and completeness of DTAL are proven. Chapter 4 describes experimental implementation of DTAL, created in Prolog, and its current limitations. In Chapter 5, two other logic-based machine learning algorithms for distributed environments are presented. Finally, Chapter 6 summarises the work and indicates directions for its future development.

Chapter 2

Background

2.1 General setting

Distributed or *multi-agent* learning can be understood and approached in many ways. For example, should a setting in which many independently learning agents coexist, and possibly cooperate, be considered an instance of *multi-agent* learning? Weiß and Dillenbourg (1999) state some fundamental questions about the nature of multi-agent learning and distinguish various types of multi-agent learning, based on the concepts of *multiplication*, *division* and *interaction*.

Multiplied learning corresponds to the simplest scenario in which every agent learns on its own. While interaction between agents may occur, it is not directly involved in the learning process.

Divided learning occurs when each of the agents is responsible for a separate part of the learning task. This may mean agents exploring separate regions of the environment or separate areas of the hypothesis search space. Another example would be a heterogenic system composed of specialised agents, capable of acquiring different types of data. In divided learning, the interaction between agents would typically be largely specified by the design of the system and intended problem decomposition.

Interactive learning occurs when the interaction between agent is used directly and dynamically in the learning process. Rather than performing separate tasks and aggregating results afterwards, the agents communicate and perform reasoning together. Typically, the interactions are much more dynamic than in *divided learning*, which means that they cannot be precisely specified at design level, and occur as they are necessary to solve a particular problem. Obviously, the three classes of multi-agent learning are not disjoint and the aim is not draw precise boundaries between them. Nevertheless, they are very helpful in understanding what multi-agent learning can possibly mean.

This classification is later brought up by Kazakov and Kudenko (2001) who state the general setting for ILP-based machine learning in multi-agent systems. After explaining basic concepts of machine learning, multi-agent systems and inductive logic programming, they start investigating the possibility of applying ILP in multi-agent environments. They indicate some publications in the area at the time of writing (2001). Those works, however, focus on various ways of applying ILP to individually reasoning agents, rather than attempt to distribute the ILP process itself. In the classification defined by Weiß and Dillenbourg, they would largely fall into the category of *multiplied learning*.

2.2 Inductive Logic Programming

Inductive Logic Programming (ILP, Lavrač and Džeorski (1994)) is an approach to machine learning in which background knowledge, examples and hypotheses are expressed using a subset of predicate logic. The Definition 2.1 states what is an ILP programming task, and what it means that a given hypothesis is a solution of the task.

Definition 2.1. *An Inductive Logic Programming task is a tuple $\langle E, B, M \rangle$, where E is the set of examples, B is the background knowledge and M is mode bias, defining the language $\mathcal{L}_H$ of the hypothesis space. A hypothesis $H \in \mathcal{L}_H$ is a solution of the learning task if and only if:*

1. $B \cup H \models_s E$,
2. $B \cup H$ *is consistent.*

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

In this thesis, it is assumed that the background knowledge B is a normal logic program.

Many ILP algorithms exist. Among them, one of the most robust is *Top-directed Abductive Learning* (TAL), described in detail in section 2.4. Advantages of TAL include capability to perform non-monotonic learning (using negation as failure) and non-observational learning (learning of concepts other than those by which the examples are defined). The basic idea behind TAL is to map an ILP task into an Abductive Logic Programming task.

2.3 Abductive Logic Programming

Abductive Logic Programming (ALP, Kakas et al. (1992)) is an extension of logic programming, enabling abduction, that is searching for possible explanations of observed facts rather than just attempting to prove them using existing knowledge. For example, given a knowledge base B:

$$p \leftarrow q$$
$$q \leftarrow a$$
$$q \leftarrow b$$

set of abducibles $\{a, b\}$ and observation (query) $Q = \{p\}$, an ALP system could be expected to find two possible explanations of the observation, $\{a\}$ and $\{b\}$.

It is easy to note some similarity between the aims of ILP and ALP. Given some background knowledge and observations, both are trying to construct an explanation.

The Definitions 2.2 and 2.3 cited here from Ma (2011) introduce the formal setting for ALP.

Definition 2.2. *(Ma (2011)) An abductive (logic) framework is a tuple* $\langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ *where*

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

The set of abducible predicates $\mathcal{AB}$ is disjoint from the set of predicates definied in Π.

Definition 2.3. *(Ma (2011)) Given an abductive logic framework* $\langle \Pi, \mathcal{AB}, \mathcal{IC} \rangle$ *and a query* $\mathcal{Q}$*, the tuple* $\langle \Delta, \theta \rangle$ *is an abductive explanation if*

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

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

Many ALP systems exist. They are capable of finding an abductive solution given an abductive framework and a query. One of them is the ASystem (Kakas et al. (2001)), on which both DAREC and TAL are based.

2.4 Top-directed Abductive Learning

The notion of similarity between ILP and ALP tasks raises question about the possibility of mapping a task of one kind to the other. Indeed, this is the approach adopted by Corapi (2011) in the Top-directed Abductive Learning (TAL) algorithm. TAL is one of the most robust ILP algorithms. It is capable of non-monotonic ILP and non-observational predicate learning. Furthermore, its soundness and completeness have been proven (Corapi (2011)).

The DTAL system, introduced in this thesis, builds upon TAL, extending it for use in distributed systems.

2.4.1 Meta-level Learning as abductive search

In TAL, the learning task's mode declaration is translated into a set of meta-rules, defining how mode predicates can be combined to create new rules. During the abductive search, selected meta-rules are used to resolve the original goals and gradually construct the newly induced rules. The process terminates with abductive assumption of a fact containing encoded description of a rule explaining one or more examples. If the abduction is successful, the rules can be easily extracted from their abduced representation to form the final hypothesis. The meta-rules, together with the background knowledge for the ILP learning task is referred to as *top theory*.

2.4.2 Rule representation

In this section, the procedure of constructing a TAL top theory (Corapi (2011)) is briefly explained. The DTAL system uses a very similar procedure, with slight modifications which enable its use in distributed systems. The mode declaration for an ILP task is expressed using a set of predicate declarations, indicating which predicates may be used as heads of the induced rules, which predicates may be included in the rule bodies and imposing restrictions on the predicate arguments. The mode declaration syntax can be summarised as follows:

- *modeh(p)* denotes that the predicate *p* may be used as the head of an induced rule,
- *modeb(p)* denotes that the predicate *p* may be used in the body of an induced rule,
- declaring a predicate's argument as *+type* states that the argument should be an input argument of type *type*,

- declaring a predicate's argument as *–type* states that the argument should be an output argument of type *type*,
- declaring a predicate's argument as *#type* states that the argument should be a constant of type *type*,
- special type *any* can be used to skip imposing a type requirement on the argument.

For example, mode declaration for a learning task enabling induction of rules with predicate *flies/1* in the head and predicates *bird/1*, *reptile/1*, *eats/2* in the body, may be written as follows:

$$modeh(files(+any))$$
$$modeb(bird(+animal))$$
$$modeb(reptile(+animal))$$
$$eats(+animal, \#food)$$

Assuming that the fact *food(grain)* is in the background knowledge, an example of a rule that could be induced using this mode declaration is:

$$files(X) \leftarrow bird(X), animal(X), eats(X, grain)$$

Abductive search through the hypothesis space requires defining a way to represent the rules. TAL approaches this by representing rules as lists of so called MBLs (mode-based literals): tuples of the form (m, c, v), where m is the mode identifier, c is the list of constants used in the predicate, and v is the list of input variables used in the predicate. The list of input variables is expressed as list of variable numbers, counting from the beginning of the rule.

The rule representation used by TAL is presented here with an example, also sourced from Corapi (2011).

Example 2.1. *(Corapi (2011)) Consider the following mode declarations:*

$$M = \begin{cases} m1 : modeh(uncle(+person, +person)) \\ m2 : modeb(father(-person, +person)) \\ m3 : modeb(gender(+person, \#mf)) \\ m4 : modeb(sibling(+person, +person)) \end{cases}$$

The rule

$$uncle(X_1, X_2) \leftarrow father(X_3, X_2), sibling(X_3, X_1), gender(X_1, m)$$

is represented as $rt_M[(m1, [], []), (m2, [], [2]), (m4, [], [3, 1]), (m3, [m], [1])]$. *Each tuple in the list only contains the mode declaration, the constants, and the indexes of the input variables. For example, 2 in the second MBL refers to the second input variable in the head,* X_2. *Note that the particular name of the variables is not relevant.*

2.4.3 Top theory generation

Definition 2.4 by Corapi (2011) formalises the process of translating the mode declaration into the top theory T_M. The full top theory is the union of the background knowledge and T_M. In Definition 2.4, $inp(s*)$, $out(s*)$ and $con(s*)$ denote, respectively, the input variables, output variables and constants used in the literal $s*$.

Definition 2.4. *(Corapi (2011)) Given a set M of mode declarations, the top theory* T_M *is constructed as follows:*

- *For each head declaration M : modeh(s), the following rule is in* T_M

$$\begin{aligned} &s* \leftarrow \\ &\quad body(inp(s*), [(m, con(s*), [])]) \end{aligned} \tag{2.1}$$

- *For each body declaration m : modeb(s), the following clause is in* T_M

$$\begin{aligned} &body(I, R) \leftarrow \\ &\quad link(inp(s*), I, Links), \\ &\quad s*, \\ &\quad append(R, (m, con(s*), Links), NR), \\ &\quad append(I, out(s*), NI), \\ &\quad body(NI, NR) \end{aligned} \tag{2.2}$$

- *The following rules are in* T_M *together with a standard definition for the append predicate.*

$$\begin{aligned} &body(I, NR) \leftarrow \\ &\quad r(NR) \end{aligned} \tag{2.3}$$

$$\begin{aligned} &link([HL1|TL1], L2, [X|TV]) \leftarrow \\ &\quad nth1(X, HL1, L2), \\ &\quad link(TL1, L2, TV). \end{aligned} \tag{2.4}$$

$$link([], L2, []).$$

Example 2.2, also sourced from Corapi (2011), shows application of Definition 2.4 to sample mode declaration.

Example 2.2. *(Corapi (2011)) Consider the following set of mode declarations M:*

$$M = \begin{cases} m1 : modeh(p(+any)) \\ m2 : modeb(q(+any, \#any)) \\ m3 : modeb(q(+any, -any)) \end{cases} \tag{2.5}$$

The corresponding top theory T_M *contains, together with rules* (2.3) *and* (2.4) *the following rules:*

$$
\begin{aligned}
&p(V1) \leftarrow \\
&\quad body(V1, [(m1, [], [])]) \\
&body(I, L) \leftarrow \\
&\quad link([V1], I, Links) \\
&\quad q(V1, C1), \\
&\quad append(L, (m2, C1, Links), M), \\
&\quad append(I, [], NI), \\
&\quad body(NI, M) \\
&body(I, L) \leftarrow \\
&\quad link([V1], I, Links) \\
&\quad q(V1, V2), \\
&\quad append(L, (m2, [], Links), M), \\
&\quad append(I, [V2], NI), \\
&\quad body(NI, M)
\end{aligned}
\tag{2.6}
\tag{2.7}
\tag{2.8}
$$

2.4.4 Learning example

Top theory generated as defined in section 2.4.3 can be used by an ALP system to solve an ILP learning task. In this section, a very simple example is presented. For more examples with detailed descriptions, see Corapi (2011).

Example 2.3. *Consider an ILP learning task* $T = \langle E, B, M \rangle$*, where*

$$
\begin{aligned}
E &= \{can\_fly(armin), can\_fly(becky), \neg can\_fly(cesar)\} \\
B &= \{bird(armin), bird(becky)\} \\
M &= \{m1 : modeh(can\_fly(+any)), m2 : modeb(bird(+any))\}
\end{aligned}
$$

Top theory T_M *together with rules* (2.3) *and* (2.4) *contains:*

$$
\begin{aligned}
&can\_fly(V1) \leftarrow \\
&\quad body([V1], [(m1, [], [])]). \\
&body(Input, Rule) \leftarrow \\
&\quad bird(V1), \\
&\quad link([V1], Input, Links), \\
&\quad append(Rule, (m2, [], Links)), \\
&\quad append(Input, [], NewInput), \\
&\quad body(NewInput, NewRule).
\end{aligned}
\tag{2.9}
\tag{2.10}
$$

The successful abductive derivation for query E uses rule 2.9 to prove the positive examples. The $body$ goal is resolved using rule 2.10. The $body$ goal coming from rule 2.10 is in turn resolved using the terminating rule (2.3)*. This yields abducible fact $r([(m1, [], []), (m2, [], [1])])$ representing rule $can\_fly(X) \leftarrow bird(X)$. If the set of examples did not contain the negative example $\neg can\_fly(cesar)$, a simpler solution $can\_fly(X)$, represented by $r([(m1, [], [])])$ would also be possible.*

2.4.5 Soundness and Completeness

TAL is a sound and complete ILP algorithm which is stated by Corapi (2011) in Theorem 2.1.

In the theorem, $H = \text{TAL}(E, B, M, \emptyset)$ denotes that the hypothesis H is computed by aplying the TAL algorithm to the set of examples E, background knowledge B and mode declaration M. The $\emptyset$ argument stands for integrity constraints, optionally resolved by the algorithm.

Theorem 2.1. *(Corapi (2011)) Let $\langle B, E, \mathcal{R}_M \rangle$ be an ILP task, T_M be the top theory generated from M and $\mathsf{T}_M \cup B$ an acyclic program. If $\mathsf{T}_M \cup B$ is abductive nonrecursive for the query E then $H = \text{TAL}(E, B, M, \emptyset)$ iff H is a subset-minimal solution for the ILP task $\langle E, B, M \rangle$.*

Soundness and completeness of TAL are used in chapter 3 as the basis for proving soundness and completeness of DTAL.

2.5 Distributed Abductive Reasoning with Constraints

Distributed Abductive Reasoning with Constraints (DAREC) is a distributed ALP system introduced by Ma (2011). The DAREC abductive procedure is based on ASystem, which also underlies TAL. DAREC features include handling of inequalities, arithmetical constraint solving and ability to compute non-ground conditional answers.

The basic idea behind DAREC is to treat the distibuted abductive process as a sequence of *state* rewriting operations. The *state* contains information about remaining goals, already assumed abducibles and relevant integrity and CLP constraints. Agent processing a state at any given moment may attempt to resolve one of the goals using its knowledge or decide to transfer the state to another agent, possibly able to continue the derivation. Each assumed abducible needs to be checked by all of the agents in the system for compliance with local integrity constraints.

DAREC assumes that the set of collaborating agents is fixed. No agents join or leave during the derivation and all of them take part in it (e.g. when an abducible is

assumed, it must be checked for consistency by all the agents). Furthermore, it assumes the set of abducibles is common and known to all the agents, as well as that any agent can exchange messages with any other agent in the system.

Definition 2.5, cited here from Ma (2011), introduces the concept of a DAREC global abductive framework. Definition 2.6, also coming from Ma (2011), sets the requirements for the solutions computed by DAREC.

Definition 2.5. *(Ma (2011)) The (DAREC) global abductive framework for a system of abductive agents, is a tuple $\langle \Sigma, \widehat{\mathcal{F}} \rangle$, where Σ is the set of all agent identifiers and $\widehat{\mathcal{F}}$ is the set of abductive agent frameworks, i.e. $\{\mathcal{F}_i \mid i \in \Sigma\}$. For any pair of agents $i, j \in \Sigma$, $\mathcal{AB}_i = \mathcal{AB}_j$.*

Definition 2.6. *Ma (2011)) Given a (DAREC) global abductive framework $\langle \Sigma, \widehat{\mathcal{F}} \rangle$ and a query $\mathcal{Q}$, let $\widehat{\Pi} = \bigcup_{i \in \Sigma} \Pi_i$, let $\widehat{\mathcal{IC}} = \bigcup_{i \in \Sigma} \mathcal{IC}_i$, and let $\widehat{\mathcal{AB}} = \bigcup_{i \in \Sigma} \mathcal{AB}_i$. A pair $\langle \Delta, \theta \rangle$ is a (DAREC) global abductive answer for $\mathcal{Q}$ if and only if:*

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

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

Note that by Definition 2.5, $\widehat{\mathcal{AB}} = \bigcup_{i \in \Sigma} \mathcal{AB}_i = \mathcal{AB}_j$ for any $j \in \Sigma$.

The process of finding an abductive answer for a given query interleaves abductive inference performed locally by an agent using its local abductive framework, and communication between agents, which means transfering the computational state to a different agent to allow it to process the state further. Definition 2.7 sourced from Ma (2011) formalises the concept of DAREC computational state. A short summary of procedural steps that an agent can take to process a state is presented in Table 2.1.

Definition 2.7. *(Ma (2011)) A (DAREC) computational state (or state in brief) is a tuple $\Theta = \langle (\mathcal{G}, \mathcal{G}^d), \mathcal{ST}, \tau \rangle$, where*

- *each element in $\mathcal{G}$ is a remaining goal, and can be either a literal or a denial of the form $\forall \vec{X}. \leftarrow \phi_1, \ldots, \phi_n$ ($n < 0$) where $\vec{X}$ is the set of universally quantified variables of the denial (i.e. $\overrightarrow{X}$ are variables appearing in the denial and are withing the scope of entire denial);*
- *each element in $\mathcal{G}^d$ is a delayed goal and must be a non-abducible;*

- *$\mathcal{ST}$ is a tuple of four stores $(\Delta, \mathcal{N}, \mathcal{E}, \mathcal{C})$, where*
  - *Δ is a set of abducibles assumed so far;*
  - *$\mathcal{N}$ is a set of denials $\forall \vec{X} \leftarrow \phi_1, \ldots, \phi_n$ ($n < 0$), where the ordering of $\phi_1, \ldots, \phi_n$ matters and ϕ_1 is either an abducible or nonabducible;*
  - *$\mathcal{E}$ is a set of (in-)equalities;*
  - *$\mathcal{C}$ is a set of CLP constraints;*
- *τ is a set of tags. All free variables appearing in the state Θ are existentially quantified within the scope of the whole state.*

DAREC tags are used for two purposes. A delayed goal is tagged with the identifier of the agent which delayed it, so that the goal is not left to be proven by the same agent. An abducible is tagged by the agent that abduced it, with the identifiers of all the other agents, to keep track of which agents have yet to check the abducible for consistence. When performing the check, each agent updates the list by removing its own identifier.

Rule	Description
LD1	Resolve non-abducible by replacing a goal with the body of a corresponding rule.
LA1	Resolve abducible by assuming it or reusing already assumed one.
LC1	Reduce CLP constraint (not used for ILP learning).
LE1	Reduce equality or inequality by adding it to $\mathcal{E}$.
LN1	Rewrite negation – convert a negative goal into a denial.
LD2	Resolve denial through non-abducible by replacing a goal with the bodies of corresponding rules. Denied goal has to be tagged and remembered as a global constraint because other agents may have knowledge to prove it.
LA2	Resolve denial through abducible – the denial still needs to be kept in $\mathcal{N}$, in case the abducible needs to be assumed later on.
LC2	Resolve denial through CLP constraint (not used for ILP learning).
LE2	Resolve denial through equality.
LN2	Resolve denial through negation – try to prove the negated goal or remember that the rest of the denial must fail.

Table 2.1: DAREC inference rules

2.5.1 Soundness and Completeness

Ma (2011) proves soundness and completeness of the DAREC algorithm. The soundness of DAREC is stated by Theorem 2.2. Its completeness is expressed by Theorem 2.3.

Definition 2.8. *(Ma (2011)) Given a global abductive framework $\langle \Sigma, \widehat{\mathcal{F}} \rangle$, let $\langle \Delta, \theta \rangle$ be an answer computed by the DAREC algorithm for a query $\mathcal{Q}$ such that $\Delta\theta$ is ground, then the completion of the hypotheses $\Delta\theta$ is given by formula δ, which is the conjunction of the literals $\{A \mid A \in \widehat{\mathcal{AB}} \wedge A \in \Delta\theta\} \cup \{\neg A \mid A \in \widehat{\mathcal{AB}} \wedge A \notin \Delta\theta\}$, where $\widehat{\mathcal{AB}} = \bigcup_{i \in \Sigma} \mathcal{AB}_i$*

Theorem 2.2. *(DAREC Soundness, Ma (2011)) Given a global abductive framework $\langle \Sigma, \widehat{\mathcal{F}} \rangle$ and a query $\mathcal{Q}$, if there is a successful global abductive derivation for $\mathcal{Q}$ with global abductive answer $\langle \Delta, \theta \rangle$, then:*

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

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

Theorem 2.3. *(DAREC Completeness, Ma (2011)) Let $\langle \Sigma, \widehat{\mathcal{F}} \rangle$ be a global abductive framework. If there is a finite global abductive derivation tree T for the query $\mathcal{Q}$, and $comp(\widehat{\Pi}) \cup \widehat{IC} \cup \exists\mathcal{Q}$ is satisfiable under the three-valued semantics (Fitting (1985)), then T contains a successful branch.*

Chapter 3

Distributed ILP

In this chapter, *Distributed Top-directed Abductive Learning* (DTAL) algorithm is introduced. It is a new approach to performing ILP learning in a multi-agent setting. DTAL builds upon TAL and DAREC algorithms presented in chapter 2. Before DTAL is presented, the context of distributed learning is defined. In particular, the concepts of *distributed ILP task* and its solution are defined.

3.1 Distributed learning task definition

A distributed ILP task is defined as a tuple of agents' representations of their partial knowledge $\langle A_1, A_2, \ldots, A_n \rangle$, where each agent's partial knowledge is represented in the format of a plain ILP task $\langle E_i, B_i, M_i \rangle$, where E_i is the set of examples, B_i is the background knowledge and M_i is mode declaration stating what predicates can be used in the heads and bodies of the induced rules, according to the local knowledge of the agent. This is a very general definition, allowing for distribution of all ingredients of the task: the examples, the background knowledge and the mode declaration.

By analogy to Definition 2.1, a distributed learning task can be defined as follows.

Definition 3.1. *A distributed learning task is a tuple of agent representations* $T = \langle A_1, A_2, \ldots, A_n \rangle$*, where, for each i,* A_i *is the agent's (partial) ILP learning task, defined as* $\langle E_i, B_i, M_i \rangle$*.* E_i *is the set of examples known to the agent i,* B_i *is its background knowledge and* M_i *is the mode declaration known to agent i. The language* $\mathcal{L}_H$ *of the hypothesis space is defined by the union of mode declarations* $\bigcup_i M_i$*. A hypothesis* $H \in \mathcal{L}_H$ *is a solution of task T if and only if:*

1. $\bigcup_i B_i \cup H \models \bigcup_i E_i$,
2. $\bigcup_i B_i \cup H$ *is consistent.*

For the purpose of reasoning about the similarities and differences between a (centralised) learning task and a distributed learning task, the concept of *correspondence* between them is defined (Definition 3.2).

Definition 3.2. *Let* $T_c = \langle E_c, B_c, M_c \rangle$ *be an ILP task. Any distributed ILP task* $\langle A_1, A_2, \ldots, A_n \rangle$*, such that:*

1. $\bigcup_i E_i = E_c$
2. $\bigcup_i B_i = B_c$
3. $\bigcup_i M_i = M_c$

is a distributed ILP task corresponding to task T_c*. Task* T_c *is the (centralised) ILP task corresponding to* T_d*.*

3.2 Task transformation

Similarly to TAL, DTAL transforms a distributed learning task into a distributed abductive task. The transformation of agents' local knowledge can be performed locally and does not require any communication between them.

Transformation of a distributed ILP task into DAREC distributed abductive framework and query is composed of two basic steps:

1. Agents' local mode declarations are locally translated to form agents' local top theories.
2. The learning examples are locally translated into integrity constraints. The query is set to contain only the special abducible, necessary to trigger integrity constraint resolution.

Step 1 has similar meaning to top theory generation preformed by TAL. The most important difference is that it has to ensure that mode identifiers are unique across the system. This can be achieved by prefixing each mode identifier with an agent identifier (e.g. its number). Step 2 adapts the learning task to the fact that the DAREC system requires the query to be issued to one specific agent. This means that the examples cannot be directly used to form the query (this would require assembling all the examples by one agent, disrupting the distribution of the task). Alternative solution used by DTAL is based on expressing examples as agents' local integrity constraints. Intuitively, the constraints exclude solutions not covering the positive examples or covering the negative ones. Detailed description of the solution can be found in Definition 3.4.

Definition 3.3, based on definition 2.4 used by TAL, formalises the process of local mode translation (step 1 of task transformation).

Definition 3.3. *Let m_i be a mode identifier prefixed with the identifier of agent i. Given a set M of mode declarations, stored by agent* i*, the agent's local top theory* $\mathsf{T}_{M,i}$ *is constructed as follows:*

- *For each head declaration M : modeh(s), the following rule is in* $\mathsf{T}_{M,i}$

$$\begin{aligned} s* \leftarrow \\ \quad body(inp(s*), [(m_i, con(s*), [])]) \end{aligned} \tag{3.1}$$

- *For each body declaration m : modeb(s), the following clause is in* $\mathsf{T}_{M,i}$

$$\begin{aligned} body(I, R) \leftarrow \\ \quad link(inp(s*), I, Links), \\ \quad s*, \\ \quad append(R, (m_i, con(s*), Links), NR), \\ \quad append(I, out(s*), O), \\ \quad body(O, NR) \end{aligned} \tag{3.2}$$

- *The following rules are in* $\mathsf{T}_{M,i}$ *together with a standard definition for the append predicate.*

$$
\begin{aligned}
&body(I, NR) \leftarrow \\
&\quad rule(NR)
\end{aligned}
\tag{3.3}
$$

$$
\begin{aligned}
&link([HL1|TL1], L2, [X|TV]) \leftarrow \\
&\quad nth1(X, HL1, L2), \\
&\quad link(TL1, L2, TV).
\end{aligned}
\tag{3.4}
$$

$$link([], L2, []).$$

Note that, thanks to embedding agent identifiers in all their mode identifiers, there exists a global, bidirectional mapping between mode declarations and meta-rules. A mode identifier contains all the information necessary to refer to a particular mode declaration across entire system: unique agent identifier and a declaration identifier unique within agent.

Definition 3.4 describes the set of example-based constraints of a given agent, which formalises the idea of expressing examples as constraints, mentioned above.

Definition 3.4. *Let $A_i = \langle E_i, B_i, M_i \rangle$ be partial distributed learning task representation of agent A_i. The set of examples-based integrity constraints of agent i, $\mathcal{IC}_{E,i}$ is constructed as follows:*

- *for each positive example $e^+ \in E_i$, a denial of the form $\leftarrow \neg e_+, a$ is in $\mathcal{IC}_{E,i}$,*
- *for each negative example $e^- \in E_i$, a denial of the form $\leftarrow e_-, a$ is in $\mathcal{IC}_{E,i}$,*

where a *is a special abducible, not appearing anywhere in B_i.*

Definition 3.5 uses concepts of agent's local top theory and examples-based constraints to define transformation of a DTAL learning task into a DAREC abductive task.

Definition 3.5. *Let $T = \langle A_1, A_2, \ldots A_n \rangle$ be a distributed learning task. The corresponding* DAREC *abductive task is a pair $\langle D, Q \rangle$ where*

- $D = \{\Sigma, \widehat{\mathcal{F}}\}$ *is a* DAREC *framework, constructed as follows:*
  - *for each $A_i = \langle E_i, B_i, M_i \rangle$, an abductive framework $\langle B_i \cup \mathsf{T}_{M,i}, \{rule/1, a/1\}, \mathcal{IC}_{E,i} \rangle$ is added to Σ, where $\mathsf{T}_{M,i}$ is the local top theory of agent i and a is the special abducible mentioned in Definition 3.4, common to all the agents,*
  - *for each agent $A_i \in T$, i is added to $\widehat{\mathcal{F}}$.*
- $Q = \{a\}$ *is a* DAREC *query,*

The abductive answers computed by DAREC for framework D and query Q contain the special abducible *a* and representations of the rules included in the hypothesis. The representations are *rule(L)* facts, where *L* is a list of mode-based literals (MBLs), indicating what predicates are used in the rules.

The hypothesis can be extracted from the abduced representations, using reverse mapping between the mode declaration and identifiers. In cases when particular mode is declared at an agent different than the one performing extraction, this requires querying the other agent, for the initial mode declaration of the predicate. It is known which agent included a particular MBL, thanks to the fact, that agent identifiers have been embedded in the mode identifiers.

Detailed examples of DTAL operation, including mode translation and abductive derivation are presented in section 3.3.

3.3 Examples

In this section, some examples of how rule induction is performed by DTAL are presented. The tables present agents' initial knowledge, as well as sequences of steps, including local derivation and communication between agents, that are taken on the successful path in the derivation tree.

3.3.1 Example 1

Agent	Background	Mode	Examples
1	bird(armin)	modeh(can_fly(+any)) modeb(bird(+any))	can_fly(a) can_fly(b) ¬can_fly(c)
2	bird(becky)	-	-

Table 3.1: Example 1 - agent specifications

Agent	Top theory
1	1.1: can_fly(V1) ← body([V1], [(1m1,[],[])]). 1.2: body(Input, Rule) ← bird(V1), link([V1], Input, Links), append(Rule, (1m2, [], Links)), append(Input, [], NewInput), body(NewInput, NewRule). 1.3: body(Input, Rule) ← rule(Rule). 1.4: bird(armin).
2	2.1: bird(becky)

Table 3.2: Example 1 - agents' top theories

Agent	Constraints
1	← ¬can_fly(armin), a ← ¬can_fly(becky), a ← can_fly(cesar), a
2	-

Table 3.3: Example 1 - agents' examples expressed as integrity constraints

Table 3.1 defines the learning task. Tables 3.2 and 3.3 present agents top theories and integrity constraints, respectively. Let the computation begin at agent 1. The initial DAREC state for the abductive query $Q = \{a\}$ is $\langle (G, G^d), (\Delta, N, E, C), \tau \rangle = \langle (\{a\}, \emptyset), (\emptyset, \emptyset, \emptyset, \emptyset), \emptyset \rangle$. This is the standard translation of an abductive query to the

initial DAREC state. The initial set of goals is equal to the query. The sets of delayed goals, collected abducibles, collected denials, collected equalities and tags are empty. The set of CLP constraints is empty and will remain unused, but is included here to preserve full compliance with the general DAREC algorithm, which is capable of solving CLP constraints.

St	Ag	Changes	Comment
1	1	$\Delta' = \Delta \cup \{a\} = \{a\}$, $\tau' = \tau \cup \{\langle a, \{2\}\rangle\} = \{\langle a, \{2\}\rangle\}$	The special abducible *a* is abduced.
2	1	$G' = \{$ $\quad \leftarrow \neg can\_fly(armin),$ $\quad \leftarrow \neg can\_fly(becky),$ $\quad \leftarrow can\_fly(cesar)$ $\}$	Resolving the local integrity constraints results in adding the denial goals to the goal list.
3	1	$G' = \{$ $\quad can\_fly(armin),$ $\quad \leftarrow \neg can\_fly(becky),$ $\quad \leftarrow can\_fly(cesar)$ $\}$	The denial goal with negative term turns into non-denial goal with positive term (DAREC rule LN2).
4	1	$G' = G \setminus \{can\_fly(armin)\} \cup \{body([armin], [(1m1, [], [])])\}$	The first goal is resolved using rule 1.1.
5	1	$G' = G \setminus \{body([armin], [(1m1, [], [])])\} \cup \{$ $\quad bird(armin),$ $\quad link([armin], [armin], Links),$ $\quad append(Rule, (1m2, [], Links), NewRule),$ $\quad append([armin], [], NewInput),$ $\quad body(NewInput, NewRule)$ $\}$	The existing *body* goal is resolved using rule 1.2. The body of the rule replaces the original *body* goal in the goal list.
6	1	$G' = \{$ $\quad body([armin], [(1m1, [], []), (1m2, [], [1])]),$ $\quad \leftarrow \neg can\_fly(becky),$ $\quad \leftarrow can\_fly(cesar)$ $\}$	The *bird(armin)* goal trivially succeeds. Procedural goals *link* and *append* also succeed, binding *NewInput* to $[armin]$ and *NewRule* to $[(1m1, [], []), (1m2, [], [1])]$.
7	1	$G' = \{$ $\quad rule([(1m1, [], []), (1m2, [], [1])]),$ $\quad \leftarrow \neg can\_fly(becky),$ $\quad \leftarrow can\_fly(cesar)$ $\}$	The *body* goal is resolved using rule 1.3 and replaced with the *rule* goal.

St	Ag	Changes	Comment
8	1	$\Delta' = \Delta \cup \{rule([(1m1, [], []), (1m2, [], [1])])\}$, $\tau' = \tau \cup \{\langle rule([(1m1, [], []), (1m2, [], [1])]), \{2\}\rangle\}$, $G' = \{\leftarrow \neg can\_fly(becky), \leftarrow can\_fly(cesar)\}$	The abducible *rule* is abduced and removed from the goal list (DAREC rule LA1).
9	1	$G' = \{can\_fly(becky), \leftarrow can\_fly(cesar)\}$	As in step 3, the denial of a negative goal is turned into a positive goal (DAREC LN2).
10	1	$G' = G \setminus \{can\_fly(becky)\} \cup \{body([becky], [(1m1, [], [])])\}$	The first goal is resolved using rule 1.1.
11	1	$G' = G \setminus \{body([becky], [(1m1, [], [])])\} \cup \{$ $bird(becky)$, $link([becky], [becky], Links)$, $append(Rule, (1m2, [], Links), NewRule)$, $append([becky], [], NewInput)$, $body(NewInput, NewRule)$ $\}$	Similarly to step 5, the existing *body* goal is resolved using rule 1.2. The body of the rule replaces the original *body* goal in the goal list.
12	1	$G^{d\prime} = G^d \cup \{bird(becky)\}$, $\tau' = \tau \cup \{\langle bird(becky), \{1\}\rangle\}$, $G' = \{$ $body([becky], [(1m1, [], []), (1m2, [], [1])])$, $\leftarrow can\_fly(cesar)$ $\}$	The *bird(becky)* goal is delayed because agent 1 has no way of prooving it on its own. A corresponding tag is added to mark which agent was unable to prove the goal. Procedural goals *link* and *append* also succeed, binding *NewInput* to $[armin]$ and *NewRule* to $[(1m1, [], []), (1m2, [], [1])]$.
13	1	$G' = \{$ $rule([(1m1, [], []), (1m2, [], [1])])$, $\leftarrow can\_fly(cesar)$ $\}$	The *body* goal is resolved using rule 1.3 and replaced with the *rule* goal.
14	1	$\Delta' = \Delta$, $\tau' = \tau$, $G' = \{\leftarrow can\_fly(cesar)\}$	The abducible *rule* fact, abduced previously in step 8, is reused. The *rule* goal is removed from the goal list.

St	Ag	Changes	Comment
15	1	$N' = N \cup \{\leftarrow can\_fly(cesar)\} = \{\leftarrow can\_fly(cesar)\}$, $\tau' = \tau \cup \{\langle \leftarrow can\_fly(cesar), \{2\}\rangle\}$, $G' = \{body([cesar], [(1m1, [], []), (1m2, [], [1])])\}$	The *can_fly* goal in the denial is resolved using rule 1.1. However, the original goal needs to be remembered and is added to the set of dynamic constraints. This is to ensure that the denied goal can never be proven using a different rule (e.g. one stored by another agent).
16	1	$N' = N \cup \{\leftarrow body([cesar], [(1m1, [], []), (1m2, [], [1])])\}$, $\tau' = \tau \cup \{\langle \leftarrow body([cesar], [(1m1, [], []), (1m2, [], [1])]), \{2\}\rangle\}$, $G' = \{\leftarrow ($ $bird(cesar)$, $link([cesar], [cesar], Links)$, $append(Rule, (1m2, [], Links), NewRule)$, $append([cesar], [], NewInput)$, $body(NewInput, NewRule)$ $)\}$	The denied *body* goal is resolved further using rule 1.2. Another dynamic constraint is collected, as in the previous step.
17	1	$N' = N \cup \{\leftarrow bird(cesar)\}$, $\tau' = \tau \cup \{\langle \leftarrow bird(cesar), \{2\}\rangle\}$ $G' = \emptyset$	The first goal of the denial fails. However, it still has to be added to the dynamic constraints so that it can be checked by remaining agents. The goal list is now empty but there are still delayed goals and constraints to be checked by other agents.
18	2	$\tau' = \{$ $\langle bird(becky), \{1\}\rangle$, $\langle \leftarrow can\_fly(cesar), \emptyset\rangle$, $\langle \leftarrow body([cesar], [(1m1, [], []), (1m2, [], [1])]), \emptyset\rangle$, $\langle \leftarrow bird(cesar), \emptyset\rangle$ $\}$, $G^d = \emptyset$, $G = \{$ $\leftarrow can\_fly(cesar)$, $\leftarrow body([cesar], [(1m1, [], []), (1m2, [], [1])])$, $\leftarrow bird(cesar)$, $bird(becky)$ $\}$	The computational state is now transfered to agent 2 (DAREC rule TR). The dynamic constarints are extracted to the goal list, and agent 2 label is removed form the corresponding tags. The delayed goal *bird(becky)* is also added to the goal list, so that agent 2 can try to prove it.

St	Ag	Changes	Comment
19	2	$G = \emptyset$	The denial goals succeed since agent 2 has no way of proving them. The goal *bird(becky)* also succeeds because this fact is contained in the background knowlege of agent 2 (rule 2.1).

Table 3.4: Example 1 - successful rule derivation.

The abductive answer computed for this example is $rule([(1m1, [], []), (1m2, [], [1])])$, which represents the rule $can\_fly(X) \leftarrow bird(X)$. Given joint background knowledge of the agents, this rule indeed explains all the positive examples (*armin* and *becky* are birds) and none of the negative examples (*cesar* is not known to be a bird).

3.3.2 Example 2

Agent	Background	Mode	Examples
1	bird(armin)	modeh(can_fly(+any)) modeb(bird(+any))	can_fly(ben)
2	plane(ben)	modeh(can_fly(+any)) modeb(plane(+any))	can_fly(armin)

Table 3.5: Example 2 - agent specifications

Table 3.5 defines the learning task. Tables 3.6 and 3.7 present agents top theories and integrity constraints, respectively. Let the computation begin at agent 1. As in Example 1, the initial DAREC state for the abductive query $Q = \{a\}$ is $\langle (G, G^d), (\Delta, N, E, C), \tau \rangle = \langle (\{a\}, \emptyset), (\emptyset, \emptyset, \emptyset, \emptyset), \emptyset \rangle$.

Agent	Top theory
1	1.1: can_fly(V1) ← body([V1], [(1m1,[],[])]). 1.2: body(Input, Rule) ← bird(V1), link([V1], Input, Links), append(Rule, (1m2, [], Links)), append(Input, [], NewInput), body(NewInput, NewRule). 1.3: body(Input, Rule) ← rule(Rule). 1.4: bird(armin).
2	2.1: can_fly(V1) ← body([V1], [(2m1,[],[])]). 2.2: body(Input, Rule) ← plane(V1), link([V1], Input, Links), append(Rule, (2m2, [], Links)), append(Input, [], NewInput), body(NewInput, NewRule). 2.3: body(Input, Rule) ← rule(Rule). 2.4: plane(ben).

Table 3.6: Example 2 - agents' top theories

Agent	Constraints
1	← ¬can_fly(ben), a
2	← ¬can_fly(armin), a

Table 3.7: Example 2 - agents' examples expressed as integrity constraints

St	Ag	Changes	Comment
1	1	$\Delta' = \Delta \cup \{a\} = \{a\}$, $\tau' = \tau \cup \{\langle a, \{2\}\rangle\} = \{\langle a, \{2\}\rangle\}$	The special abducible *a* is abduced.
2	1	$G' = \{\leftarrow \neg can\_fly(ben)\}$	Resolving the local integrity constraints results in adding the denial goal to the goal list.
3	1	$G' = \{can\_fly(ben)\}$	The denial goal with negative term turns into non-denial goal with positive term (DAREC rule LN2).
4	1	$G' = \{body([ben], [(1m1, [], [])])\}$	The goal is resolved using rule 1.1.
5	1	$G^{d'} = \{body([ben], [(1m1, [], [])])\}$ $\tau' = \tau \cup \{\langle body([ben], [(1m1, [], [])]), \{1\}\rangle\}$ $G' = \emptyset$	The *body* goal is delayed for resolution by another agent. The goal list is now empty.
6	2	$G^{d'} = \emptyset$ $\tau' = \{\langle a, \emptyset\rangle, \langle body([ben], [(1m1, [], [])]), \{1\}\rangle\}$ $G' = \{body([ben], [(1m1, [], [])]), \leftarrow \neg can\_fly(armin)\}$	The computational state is transfered to agent 2 (DAREC rule TR). The delayed *body* goal is added to the goal list, so that agent 2 can try to prove it. Integrity constraints for abducible *a* are resolved and agent 2 label is removed form the corresponding tag.
7	2	$G' = \{$ $plane(ben)$, $link([ben], [ben], Links)$, $append(Rule, (2m2, [], Links))$, $append(ben, [], NewInput)$, $body(NewInput, NewRule)$ $\}$	The *body* goal is resolved using rule 2.2. The $plane(ben)$ goal succeeds, as well as procedural goals *link* and *append*, binding *NewInput* to $[ben]$ and *NewRule* to $[(1m1, [], []), (2m2, [], [1])]$.
8	2	$\Delta' = \Delta \cup \{rule([(1m1, [], []), (2m2, [], [1])])\}$, $\tau' = \tau \cup \{\langle rule([(1m1, [], []), (1m2, [], [1])]), \{1\}\rangle\}$, $G' = \{\leftarrow \neg can\_fly(armin)\}$	The abducible *rule* is assumed and removed from the goal list (DAREC rule LA1).
9	2	$G' = \{can\_fly(armin)\}$	As in step 3, the denial of a negative goal is turned into a positive goal (DAREC LN2).

St	Ag	Changes	Comment
10	2	$G' = \{body([armin], [(2m1, [], [])])\}$	The first goal is resolved using rule 2.1.
11	2	$G^{d'} = \{body([armin], [(2m1, [], [])])\}$ $\tau' = \tau \cup \{\langle body([armin], [(2m1, [], [])]), \{2\}\rangle\}$ $G' = \emptyset$	Similarly to step 5, the existing *body* goal is delayed.
12	1	$G^{d'} = \emptyset$ $\tau' = \{$ $\langle a, \emptyset\rangle,$ $\langle rule([(1m1, [], []), (1m2, [], [1])]), \emptyset\rangle,$ $\langle body([armin], [(2m1, [], [])]), \{2\}\rangle$ $\},$ $G' = \{body([armin], [(2m1, [], [])])\}$	The computational state is transfered to agent 2 (DAREC rule TR). The delayed *body* goal is added to the goal list, so that agent 2 can try to prove it. The tag for agent 2 is removed from the *rule* abducible.
13	1	$G' = \{$ $bird(armin),$ $link([armin], [armin], Links),$ $append(Rule, (1m2, [], Links)),$ $append(ben, [], NewInput),$ $body(NewInput, NewRule)$ $\}$	The *body* goal is resolved using rule 2.2. The *bird(armin)* goal succeeds, as well as procedural goals *link* and *append*, binding *NewInput* to $[armin]$ and *NewRule* to $[(2m1, [], []), (1m2, [], [1])]$.
14	1	$\Delta' = \Delta \cup \{rule([(2m1, [], []), (1m2, [], [1])])\},$ $\tau' = \tau \cup \{\langle rule([(2m1, [], []), (1m2, [], [1])]), \{2\}\rangle\},$ $G' = \emptyset$	The *rule* abducible is assumed. Corresponding tag is added to mark the abducible for consistency check by the other agent.
15	2	$\tau' = \{$ $\langle a, \emptyset\rangle,$ $\langle rule([(1m1, [], []), (1m2, [], [1])]), \emptyset\rangle$ $\langle rule([(2m1, [], []), (1m2, [], [1])]), \emptyset\rangle$ $\},$ $G' = G = \emptyset$	The state is transfered to agent 2 for consistency check of the most recent abducible. Since there are no integrity constraints containing the *rule* abducible, the check trivially succeeds.

Table 3.8: Example 2 - successful rule derivation.

The abductive answer computed for this example is

$$\{rule([(1m1, [], []), (2m2, [], [1])]), rule([(2m1, [], []), (1m2, [], [1])])$$

which represents the hypothesis:

$$can\_fly(X) \leftarrow plane(X)$$
$$can\_fly(X) \leftarrow bird(X)$$

This is the most complex hypothesis that can be computed for this example. Other possibilities include the simplest hypothesis containing only rule $can\_fly(X)$ with empty body and hypotheses containing the simple rule as well as one of the complex ones:

$$can\_fly(X) \qquad\qquad can\_fly(X)$$
$$can\_fly(X) \leftarrow bird(X) \qquad\qquad can\_fly(X) \leftarrow plane(X)$$

In both of these hypotheses, the second rule is clearly redundant. However, they can be computed because of the fact that the algorithm attempts to explain all the examples separately. In the future, the system can easily be extended to avoid returning such solutions. This could be achieved by additional post-processing of the hypotheses to detect redundant rules or by iterative removing of positive examples which are already covered (the *cover loop approach*, Corapi (2011)).

3.4 Soundness and Completeness

In this section, theorems about fundamental properties of DTAL learning algorithm, namely its soundeness and completeness, are stated and proven. Corresponding theorems about soundness and completeness of TAL and DAREC are used as the basis.

When a *distributed* learning task is considered, a fundamental question to ask is how the fact of distribution of knowledge affects the overall reasoning capability of the system. In Definition 3.6, the concept of a *centralised top theory* is introduced. It is then used in reasoning about the relations between a set of local top theories generated by the agents and a global (*centralised*) top theory, which could be generated for a hypothetical global ILP learning task, based on the knowledge of all agents aggregated in just one reasoning entity.

Definition 3.6. *Let $T_d = \langle A_1, A_2, \ldots, A_n \rangle$ be a distributed learning task. Let T_c be a centralised task corresponding to T_d. The top theory Th_c generated by TAL for task T_c is the centralised top theory of task T_d.*

Theorem 3.1 expresses both soundness and completeness of DTAL. Intuitively, it states that all the solutions computed by DAREC are indeed solutions of the global ILP task corresponding to the distributed ILP task for which the algorithm is run. Furthermore, every existing solution to the task can be computed by DAREC.

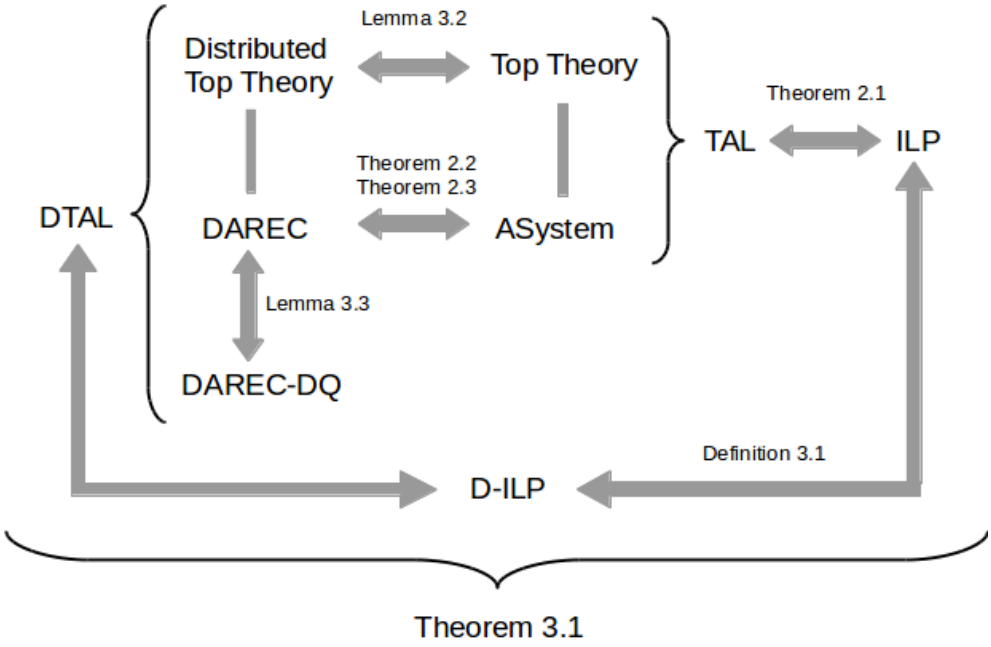

Figure 3.1: Theorem 3.1 illustration

Theorem 3.1. *Let* $T_d = \langle A_1, A_2, \ldots, A_n \rangle$ *be a distributed learning task and* Th_c *its centralised top theory and an acyclic program. Let* $T_c = \langle \bigcup_i E_i, \bigcup_i B_i, \bigcup_i M_i \rangle$ *be an ILP task corresponding to* T_d*. If* Th_c *is abductive nonrecursive for query* $\bigcup_i E_i$ *then:*

- *if the hypothesis H is computed by applying the DTAL algorithm to task* T_d*,* $H \in \text{DTAL}(T_d)$*, then H is a solution of task* T_c*;*
- *if H is a subset-minimal solution of task* T_c*, then H is computed by applying the DTAL algorithm to task* T_d*,* $H \in \text{DTAL}(T_d)$*.*

To state a proof for Theorem 3.1, some lemmas about the properties of DAREC algorithm need to be introduced. At the end of the section, they are brought together with existing theorems provided by Corapi (2011) and Ma (2011) to from the complete proof. The Figure 3.1 is used to give an intuition of how the elements of the proof are used together. Theorems 2.1 (Corapi (2011)) state soundness and completeness of TAL in its ability to compute answers of an ILP learning task. Theorems 2.2 and 2.3 (Ma (2011)) state soundness and completeness of DAREC as an abductive proof procedure. Lemma 3.2 expresses the equivalence between a centralised top theory used by TAL and the distributed top theory employed by DTAL. Lemma 3.3 provides the formal correctness of expressing the learning examples as integrity constraints.

Lemma 3.1 formalises the similarity between a centralised top theory of a DTAL task and a simple set-theoretical union of DTAL agents' locally generated partial top

theories.

Lemma 3.1. *Let $T = \langle A_1, A_2, \ldots, A_n \rangle$ be a distributed learning task. Let Th_i be a local top theory of agent i, generated from its local mode declaration and background knowledge. Let $Th_{all} = \bigcup_i Th_i$. Let Th_c be the centralised top theory for task T. For every clause in Th_{all} there is exactly one clause in Th_c which differs exactly by the mode identifier constant. For every clause in Th_c there is at least one clause in Th_{all} which differs exactly by the mode identifier constant.*

Proof. The process of top theory generation, either for the centralised TAL or for DTAL, is composed of three basic steps:

1. Each clause from the background knowledge is added to the top theory unchanged.
2. Each mode declaration is translated into exactly one mode clause.
3. A special clause terminating the rule generation and *link* predicate are added.

The three sets of clauses added to the top theory in the above steps are disjoint and can be considered separately.

1. If a background clause is contained in the centralised top theory Th_c, then it must also be contained in the centralised background knowledge B_c, mentioned in Definition 3.6. If that is the case, the clause must come from the local background knowledge of at least one agent A_i. This implies that it will also be contained in the locally generated partial top theory Th_i of agent *i*, and, by that fact, also in Th_{all} which is the union of all agents' locally generated theories.
   If a background clause is contained in the theory Th_{all}, then it must be contained in at least one locally generated partial top theory Th_i of agent *i*. If that is the case, the clause must come from the agent's local background knowledge B_i. This implies that it is contained in the union of all agents' background knowledge and, by definition of the centralised top theory, in Th_c.
2. If a mode clause $C_1$is contained in the centralised top theory Th_c, then a corresponding mode declaration must be contained in the centralised mode M_c, mentioned in Definition 3.6. If that is the case, the declaration must come from the local mode declaration of at least one agent A_i. This implies that a clause C_2, different from C_1 only by the agent-specific mode identifier, will also be contained in the locally generated partial top theory Th_i of agent *i*, and by that fact also in Th_{all} which is the union of all agents' locally generated theories. In case when agent's mode declarations overlap, Th_{all} may actually contain more clauses representing the same mode declaration, differing by the agent-specific

identifiers.
If a mode clause is contained in the theory Th_{all}, then it must be contained in at least one locally generated partial top theory Th_i of agent i. If that is the case, a corresponding mode declaration must exist in the agent's local mode M_i, and also in the union of all agents' mode declaration and, by definition of the centralised top theory, a corresponding clause must be included in Th_c.

3. The terminating clause and *link* predicate are present in each top theory, independently from the mode declaration. It is included Th_c and each of the agents' locally generated top theories, by which fact it is also included in Th_{all}
□

The connection between top theories stated in Lemma 3.1 can be further used to show that, whether an ALP system is applied to a centralised top theory or to a set union of local top theories, it finds precisely the same ILP task solutions (hypotheses).

Lemma 3.2. *Let $T = \langle A_1, A_2, \ldots, A_n \rangle$ be a distributed learning task. Let E_i be the set of examples known to agent i. Let Th_i be a local top theory of agent i, generated from its local mode declaration and background knowledge. Let $Th_{all} = \bigcup_i Th_i$. Let Th_c be the centralised top theory for task T. Let A_{all} and A_c be abductive logic frameworks, defined as follows:*

$$A_{all} = \langle Th_{all}, \{\text{rule}/1\}, \emptyset \rangle$$

$$A_c = \langle Th_c, \{\text{rule}/1\}, \emptyset \rangle$$

For a given sound and complete ALP system S, a hypothesis H can be found by post-processing of an answer to query $\bigcup_i E_i$, returned by S for framework A_{all} if and only if H can be found by post-processing an answer returned by S for framework A_c.

Proof. By Lemma 3.1, every clause in top theory Th_c has an equivalent clause in top theory Th_{all} and every clause in Th_{all} has an equivalent clause in Th_c. The differing mode identifiers do not influence the resolution procedure until a *rule* fact is abduced, which contains identifiers of clauses used up to this point. The identifiers are only used to keep record of the rules used in the derivation and they are removed by the abductive answer post-processing. Since every clause in Th_{all} has an equivalent clause in Th_c, every solution found for Th_{all} can also be found for Th_c. Since every clause in Th_c has at least one equivalent clause in Th_{all}, every solution found for Th_c can also be found for Th_{all}. The potential presence of duplicate clauses in Th_{all}, caused by overlap between agents' mode declarations, can only introduce additional ways of inducing the same solutions and does not change the set of all solutions found for the query □

The DTAL system uses DAREC integrity constraints to express the examples, as noted in 3.3. The main reason for that is to enable actual distribution of examples - if the

examples were used to form the query, they would have to be known to a single agent before starting the learning process. Lemma 3.3 states that this way of expressing examples does not violate the soundness and completeness of the system. For the purpose of the lemma, a *distributed-query-equivalent* of given DAREC framework and query is defined.

Definition 3.7. *Let D be a DAREC framework and Q a query for the framework D. The pair $\langle D, Q\rangle$. The query $Q = \{q_1, q_2, \ldots, q_i\}$ can be transformed into an equivalent query for a helper goal, $Q_{DQ} = \{a\}$ (where a does not appear in Q nor in D) by transforming each of the original goals $q_1 \ldots q_n$ into an integrity constraint of the form $\leftarrow$ not q_i for positive goals and $\rightarrow q_i$ for negative goals and distributing them arbitrarily among the agents of the framework D and adding them to agents' existing integrity constraints. Let D_{DQ} denote the resulting framework. The pair $\langle D_{DQ}, Q_{DQ}\rangle$ is called the distributed-query equivalent of the pair $\langle D, Q\rangle$.*

Lemma 3.3. *Let D be a DAREC framework and Q a query for the framework D. Let $\langle D_{dist}, Q_{dist}\rangle$ be the distributed-query equivalent of the pair $\langle D, Q\rangle$. Let S be the set of answers computed by DAREC for query Q and framework D. Let S_{dist} be the set of answers computed by DAREC for query Q_{dist} and framework D_{dist}.*

$$A \in S \iff A \cup \{a\} \in S_{dist}$$

Proof. The Definition 2.6 of a (DAREC) global abductive answer $\langle \Delta, \theta\rangle$ requires satisfaction of both of the following conditions:

1. $\Delta\theta \subseteq \widehat{AB}$
2. $\widehat{\Pi} \cup \Delta\theta \models \mathcal{Q}\theta$
3. $\widehat{\Pi} \cup \Delta\theta \models \widehat{\mathcal{IC}}$

We will show that both of these conditions are met for a pair of a DAREC framework and a query $\langle D, Q\rangle$ if and only if corresponding conditions are met for its distributed-query equivalent:

1. $\Delta\theta_{dist} \subseteq \widehat{AB}_{dist}$
2. $\widehat{\Pi} \cup \Delta\theta_{dist} \models \mathcal{Q}\theta_{dist}$
3. $\widehat{\Pi} \cup \Delta\theta_{dist} \models \widehat{\mathcal{IC}}_{dist}$

The conditions 2 and 3 can be rewritten as follows:

$$\widehat{\Pi} \cup \Delta\theta \models \mathcal{Q}\theta \cup \widehat{\mathcal{IC}} \tag{3.5}$$

Under the generalised 3-valued semantics, fact 3.5 holds if and only if:

$$\widehat{\Pi} \cup \Delta\theta \models \{\leftarrow \neg q : q \in Q_+\theta\} \cup \{\leftarrow q : q \in Q_-\theta\} \cup \widehat{\mathcal{IC}} \tag{3.6}$$

Let $\Delta\theta_{dist}$ be the $\Delta\theta$ extended by adding the special abducible a, not appearing anywhere in $\widehat{\Pi}$, nor in $\widehat{\mathcal{IC}}$. Let $\widehat{AB}_{dist} = \widehat{AB} \cup \{\mathsf{a}\}$.

$$\Delta\theta_{dist} = \Delta\theta \cup \{\mathsf{a}\} \tag{3.7}$$

Since $a \in \Delta\theta_{dist}$, fact 3.6 holds if and only if:

$$\widehat{\Pi} \cup \Delta\theta_{dist} \models \{\leftarrow \neg q, \mathsf{a} : q \in Q_+\theta\} \cup \{\leftarrow q, \mathsf{a} : q \in Q_-\theta\} \cup \widehat{\mathcal{IC}} \tag{3.8}$$

By Definition 3.7:

$$\widehat{\mathcal{IC}}_{dist} = \{\leftarrow \neg q, \mathsf{a} : q \in Q_+\theta\} \cup \{\leftarrow q, \mathsf{a} : q \in Q_-\theta\} \cup \widehat{\mathcal{IC}} \tag{3.9}$$

From (3.8) and (3.9) we get that:

$$\widehat{\Pi} \cup \Delta\theta_{dist} \models \widehat{\mathcal{IC}}_{dist} \tag{3.10}$$

By Definition 3.7, $\mathcal{Q}_{dist} = \mathcal{Q}\theta_{dist} = \{\mathsf{a}\}$ and:

$$\widehat{\Pi} \cup \Delta\theta_{dist} \models \mathcal{Q}_{dist}\theta \tag{3.11}$$

Together with the fact that $\{\mathsf{a}\} \subseteq \mathcal{AB}$, facts 3.10 and 3.11 mean that $\Delta\theta_{dist}$ satisfies the definition of an abductive answer for framework D_{dist} and query $\mathcal{Q}_{dist}$ □

With lemmas introduced in this section, as well as results by Corapi (2011) and Ma (2011) quoted in Chapter 2, Theorem 3.1 can be proven. Soundness and completeness of DTAL are proven here by considering the stages of mode translation and abductive search separately.

Proof of Theorem 3.1 By Theorem 2.1 TAL is known to be a sound and complete ILP algorithm. Lemma 3.2 states, that the top theory generated by TAL for a centralised learning task can be replaced by union of agents' locally generated partial DTAL top theories $Th_{all} = \bigcup_i Th_i$. By Theorems 2.2 and 2.3, DAREC can be used on the Th_{all} instead of a sound and complete centralised ALP algorithm, to find exactly the same solutions to the task. In DAREC, the background knowledge for the abductive task (in case of a learning task, this is the top theory Th_{all}) can be distributed arbitrarily among agents. In particular, the locally generated top theories, considered as their set union Th_{all} before, can be stored locally by agents that generated them. Finally, distribution of the learning examples and expressing them as agents' local integrity constraints does not violate soundness and completeness of the system, as stated by Lemma 3.3. This implies that, for any distributed ILP task T_d and its corresponding (centralised) ILP task T_c, the following holds:

$$H \in \text{DTAL}(T_d) \iff H \in \text{TAL}(T_c) \tag{3.12}$$

Given soundness and completeness of TAL and the fact that by Definition 3.1, T_d and T_c have the same solutions, it can be concluded that any $H \in \text{DTAL}(T_d)$ is indeed a solution of the ILP learning task $T_c = \langle \bigcup_i E_i, \bigcup_i B_i, \bigcup_i M_i \rangle$. Similarly, by fact 3.12, and completeness of TAL, any subset-minimal solution of T_c is computed by DTAL □

Chapter 4

Implementation and Evaluation

The DTAL system has been implemented in SWI-Prolog as a task-translating layer that can be put on top of an existing DAREC implementation. The implementation was tested with the DAREC implementation created by Jiefei Ma as a part of his PhD thesis (Ma (2011)). Brief specification of the implementation is provided in Section 4.1. The implementation code can be found in Appendix A.

The DAREC implementation had to be modified so that it can be used with SWI-Prolog. Unfortunately, the YAP-Prolog, for which the implementation was originally intended, no longer supports some constructions used in DAREC. The availability of older versions of YAP is very limited. Moreover, even the newest version is difficult to obtain, especially in a variant with multi-threading support. Section 4.2 lists the changes that were introduced in the DAREC implementation.

4.1 Specification of the DTAL implementation

The top-level procedures of the implementation, forming the user interface of the DTAL layer, as well as their intended use, are presented in this section. The figure 4.1 gives a high-level overview of the implementation. The parts in black are intended to be run by all the agents in the system. The parts in grey should be run by just one, arbitrarily chosen agent.

```
preprocess(Background, Mode, Examples, Theory)
```

The `preprocess` procedure processes background knowledge, mode declaration and examples into a complete local top theory of an agent. The `Background`, `Mode` and `Examples` arguments are the paths to files containing the corresponding components of an agent's partial learning task definition. The `Theory` argument is the path to the output file containing preprocessed task. The background knowledge is

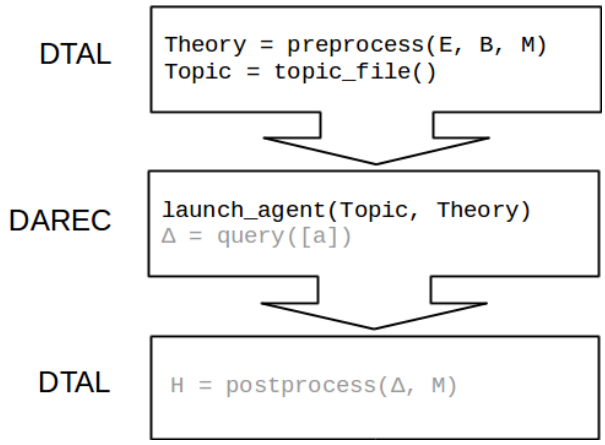

Figure 4.1: DTAL implementation overview

copied to the theory file without changes. The mode declaration is translated into corresponding meta-rules, containing agent-specific mode identifiers. The examples are translated into integrity constraints. The output theory file is intended for use as the theory file for a DAREC agent. Typically, the `preprocess` predicate is called by every DTAL agent once, before the standard DAREC derivation begins. The source code of the procedure is presented in Listing 4.1. The code of `toptheory` and `examples_as_constraints` procedures can be found in Appendix A.

Listing 4.1: The `preprocess` procedure

```
preprocess(Background, Mode, Examples, Theory) :-

    open(Background, read, BStream),
    clauses_from_file(BStream, BClauses),
    close(BStream),

    open(Mode, read, MStream),
    clauses_from_file(MStream, MClauses),
    close(MStream),

    open(Examples, read, EStream),
    clauses_from_file(EStream, EClauses),
    close(EStream),

    toptheory(MClauses, MTheory),
    examples_as_constraints(EClauses, Constraints),
    flatten([BClauses, MTheory, Constraints], TopTheory),

    open(Theory, write, TStream),
    clauses_to_file(TopTheory, TStream),
    close(TStream).
```

`topic_file(Topic)`

The `topic_file` procedure generates the topic file for the DAREC system. The file indicates abducibles and built-in predicates to enable resolving them properly by DAREC (built-in predicates are resolved using a standard Prolog call, rather than DAREC derivation based on the background knowledge). The content of this file is the same for all agents and independent of a particular learning task. Typically, the procedure may be run by a DTAL agent once, before the standard DAREC derivation begins. Alternatively, the topic file generated by the procedure may be shipped with agents' code - its content does not change. If many agents are run in an environment sharing the same file system, a single topic file may be used by all of them. The source code of the procedure is presented in Listing 4.2.

Listing 4.2: The `topic_file` procedure

```
topic_file(Topic) :-
    open(Topic, write, Stream),
    clauses_to_file([
        (abducible(a)),
        (abducible(rule(_))),
        (builtin(link(_,_,_))),
        (builtin(append(_,_,_))),
        (builtin(nonmember(_,_)))
    ], Stream),
    close(Stream).
```

`postprocess(Mode, HypoRep, Hypothesis)`

The `postprocess` procedure performes extraction of the hypothesis form its abduced representation, given as a list of *rule* facts. The `Mode` is the path to the file containing the agent's local mode declaration. `HypoRep` is the list of abduced rule representations. `Hypothesis` is the answer to the original learning task, expressed as a list of Prolog rules. During the execution of `postprocess` procedure, agents possessing particular mode declarations are identified using the agent-specific mode identifiers and asked for the necessary mode declarations. If the mode declaration in question comes form the current agent, the query is resolved using local knowledge, bypassing the TCP communication. The source code of the procedure is presented in Listing 4.3. The code of `extract_hypothesis` procedure can be found in Appendix A.

Listing 4.3: The `postprocess` procedure

```
postprocess(Mode, HypoRepA, Hypothesis) :-
    open(Mode, read, Stream),
    clauses_from_file(Stream, MClauses),
    close(Stream),
    delete(HypoRepA, a, HypoRep),
    extract_hypothesis(MClauses, HypoRep, Hypothesis).
```

`run_mode_server(Mode, Port)`

The `run_mode_server` procedure enables testing of the DTAL code without a DAREC implementation. The procedure starts a simple TCP server, capable of responding to mode queries with the corresponding mode declaration. This is necessary during the post-processing phase, as noted in the `postprocess` predicate description. The `Mode` argument is the path to a file containing agent's local mode declaration. `Port` is the number of TCP port on which the server will by accepting mode queries.

Listing 4.4: The `run_mode_server` procedure

```
run_mode_server(Mode, Port) :-
    open(Mode, read, Stream),
    clauses_from_file(Stream, MClauses),
    close(Stream),
    sample_server(Port, MClauses).
```

4.2 Changes in the DAREC implementation

In this section, the changes that were introduced in the DAREC implementation are briefly described. The changes are independent of the characteristic of the learning environment and were necassary to enable using DAREC with SWI-Prolog. This in turn was needed because of very limited availability of YAP-Prolog, for which the implementation was initially developed (Ma (2011)).

The main changes are as follows:

- The code using global variables has been altered to accommodate to the fact that in SWI global variables are not shared between threads. A procedure was implemented, that wraps around thread creation by injecting additional goals to a newly created thread. The added goals are redefining necessary global variables inside the new thread.
- The code directly performing TCP communication has been replaced with code using SWI-specific TCP communication predicates.
- The predicates `new_variables_in_term` and `variables_within_term`, not available in SWI, have been reimplemented.
- A number of YAP-specific predicates was replaced with semantically equivalent SWI predicates,
- The '@' operator used by DAREC to denote the fact that a particular predicate is defined by a particular agent ($< predicate\_head >@< agent\_name >$) has been replaced by the asterisk operator ('*'). This is to accommodate the syntactic differences between SWI-Prolog and (past versions of) YAP-Prolog.

- The YAP-specific inequality operator '=/=' used for dealing with inequalities has been replaced with '=\='.

4.3 Testing and Limitations

Various components of the implementation were used for testing others during the development work. For example, top theories generated by the DTAL layer were used in the process of modifying the DAREC implementation. In this section, current limitations of the implementation, as well as achieved results are presented.

The DAREC implementation developed by Jiefei Ma (Ma (2011)) is in fact an implementation of the DAREC^2 system - an extended version of DAREC, introducing the concept of confidentiality (hence the second 'C'). The background knowledge predicates of DAREC^2 are divided into two classes. One of them is publicly known, that is, agent's knowledge of the predicate is advertised so that other agents know, that the advertising agent may be helpful in resolving goals expressed using that predicate. The other class is confidential and private to the agent - the agent may use the knowledge when performing local resolution, but other agents are unaware of the predicate existence. From their point of view, if no agent advertises the predicate, it is undefined and goals expressed using it fail. It is easy to see how DAREC^2 is a more general system than plain DAREC - if all predicates in the background knowledge of DAREC^2 agents are marked as public (*askable*), the system behaves in exactly the same way as DAREC.

Unfortunately, the case when all predicates are marked as *askable* is an extreme case of DAREC^2 application, rather than its typical use. Extensive use of *askable* predicates with the DAREC system very often leads to its malfunction. The limitations of the current version include in particular:

- floundering when resolving negated *askable* goals,
- inability to resolve integrity constraints containing *askable* predicates (essential for expressing DTAL learning examples as constraints).

Despite the limitations mentioned above, performing DTAL learning in a limited scope was possible.

The Example 1, presented in the section 3.3 was successfully solved after manually introducing some changes in the top theory. The basic idea used to overcome current limitations, was to resign from marking all the background knowledge and example predicates as *askable* by default. Instead, they were included in the background knowledge as standard (*defined*) predicates. The top theory was then manually adjusted, to turn only some selected predicates back into *askables*. It has to be admitted, that the choice of new *askables* was made using author's prediction on which

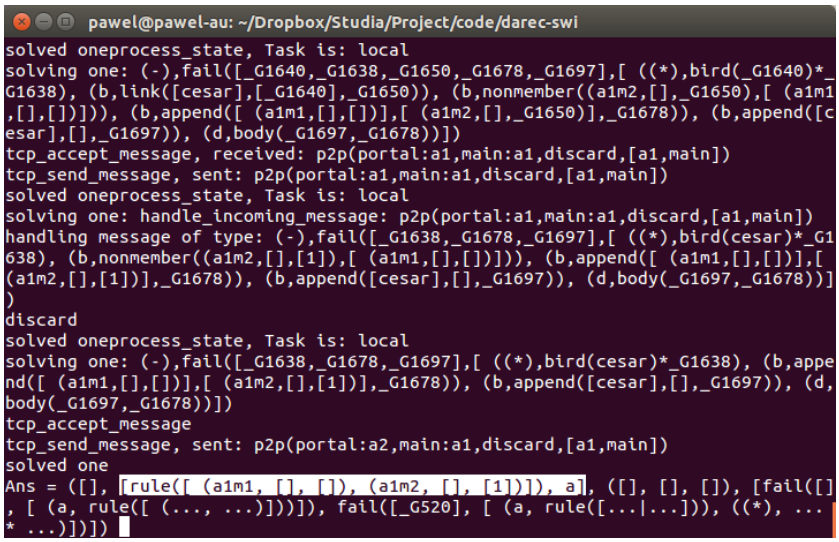

Figure 4.2: Example 1 – computed abductive answer

of the predicates may need to be consulted with another agent. With this modifications, the DAREC^2 system was able to abduce fact $rule([1m1, [], []], [1m2, [], [1]])$ corresponding to the rule $can\_fly(X) \leftarrow bird(X)$, the same one achieved manually in the derivation presented for Example 1 in section 3.3. The abductive answer computed for the example is presented in Figure 4.2. Note that for the purpose of the example, the agent names used were *a1* and *a2*. Due to the pending necessity to modify the DAREC implementation to handle *askable* predicates better, the post-processing phase has not been embedded in the system yet, and needs to be done separately.

Adjustments to the askable predicates allowed the system to solve the task presented in Example 2 from section 3.3 as well. All the hypotheses mentioned in section 3.3 were found. The abductive answer corresponding to the most sophisticated hypothesis is presented in Figure 4.3.

Further work on the implementation will focus on resolving the issues with *askable* predicates described above. The precise cause of the problem is not fully known yet. It is conjectured that the backtracking information stored for *askable* predicates may not be sufficient.

Another approach could be an attempt to 'downgrade' DAREC^2 to DAREC so that there are no separate classes of predicates and all of them are advertised – this would enable simplifying the system, in particular the internal DAREC constructions used to handle the *askable* predicates.

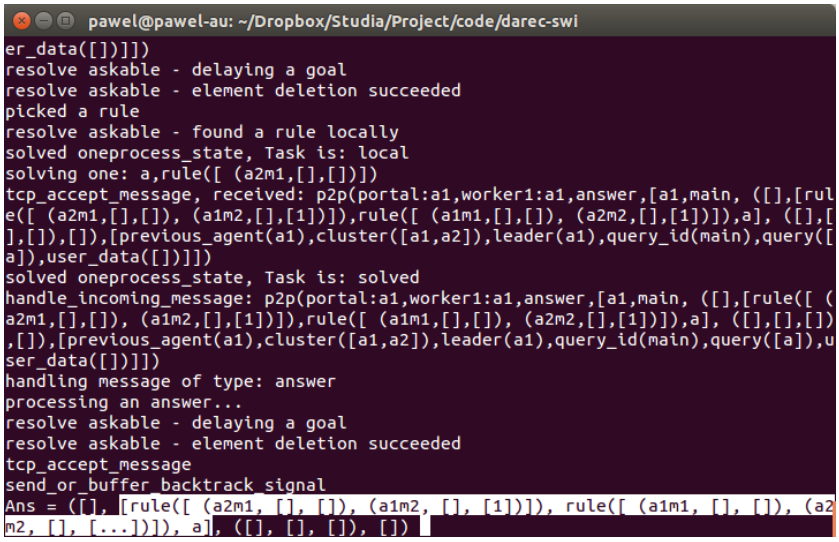

Figure 4.3: Example 2 – computed abductive answer

Chapter 5

Related Work

DTAL is not the first attempt to bring logic-based machine learning to a multi-agent environment. Two other algorithms enabling distribution of a learning process are presented in this section and briefly compared to DTAL.

5.1 Multi-agent Inductive Learning System

Huang and Pearce Huang and Pearce (2006) introduced the Multi-Agent Inductive Learning System (MAILS). The system works by interlacing deduction and induction. An agent tries to explain observed example using its background knowledge. At some point it may reach a goal expressed with a predicate for which it lacks definition. At this point, it may either induce the necessary predicate itself or ask other agents for help, supplying them with corresponding examples for the new predicate. In this approach, distribution of the learning process is rather coarse-grained: a single predicate needs to be defined by a single agent, without help from other agent's except for the initially provided examples. Further work by Huang and Pearce Huang and Pearce (2007) builds upon MAILS, providing an interesting example of applying it in the task of finding a path in a graph.

The limitations of MAILS system can be seen in the Example 1, introduced in section 3.3 and presented again in Table 5.1.

The hypothesis solving Example 1 is `can_fly(X) :- bird(X).` However, it would not be possible to compute it using the MAILS system. The reason is that induction of a single predicate has to be executed by a single agent. The agent starting the learning process could explain just one of the positive examples using this rule. Since communication between agents can only involve a request to induce a new predicate and examples for the same predicate, there is no way of using a fact from the other agent's background knowledge. This is potentially a serious limitation,

Agent	Background	Examples
1	bird(armin)	can_fly(armin) can_fly(becky) ¬can_fly(cesar)
2	bird(becky)	-

Table 5.1: Example - background knowledge split across agents

because in many applications of multi-agent systems it is natural for the agents to have knowledge similar in nature (expressed by the same predicate) but referring to a different entity. For example, an agent placed in the hall could know that `closed(front_door)` holds, while only the agent in the living room would be aware that `closed(window)` is also true.

Tasks solved by Huang and Pearce (2006) can also be solved by DTAL – the delegation of the task of inducing a particular predicate can be expressed using the concept of delayed goals, introduced by DAREC and used by DTAL.

5.2 Sound Multi-agent Incremental Learning

Bourgne et al. define an algorithm called SMILE (*Sound Multi-agent Incremental LEarning*) Bourgne et al. (2008). This is a method of performing concept learning in a multi-agent environment. The idea behind SMILE is to perform a local revision of a global hypothesis whenever a new example is considered. After that, consistency of the new hypothesis with examples known by other agents is checked. If some agent knows an example contradicting the proposed version of the hypothesis, it objects the changes and sends relevant example to the proposer, which updates the hypothesis. This process is repeated iteratively, until all agents agree on and share the same hypothesis.

This method however is limited to direct learning of concepts defined by a tuple of boolean variables. Compared to more general ILP algorithms, this means that it is only capable of observational predicate learning (only rules for the predicate in which examples are expressed can be generated). Furthermore, the learning process cannot use any background knowledge. In general, tuple-based (*attribute-value*) learning is less expressive than ILP, as noted by Raedt (2010).

This limitation can also be seen using example presented in the Table 5.1. Reasoning about the three entities in question (*armin*, *becky* and *cesar*) would require describing each of them as a tuple of boolean values and using this knowledge as the examples, which must themselves contain entire available knowledge. While this could seem feasible in this case (just one boolean value describing whether some-

thing is a bird), if more kinds of animals were defined, the examples would need to provide a value for each type of animal and each entity. This knowledge could not only be difficult to assemble, it may also be incomplete, especially from the point of view of a single agent.

In further work on SMILE Bourgne et al. (2009) Bourgne et al. investigate a slightly modified approach, in which the hypothesis is not necessarily shared between the agents, which are now allowed to keep their own versions, possibly inconsistent with union of all knowledge in the system. Many hypotheses created in this way can than be used to classify new examples using some aggregation method, e.g. majority voting.

Chapter 6

Conclusions and Further Work

In this thesis, the concept of *Distributed ILP* learning has been introduced. The DTAL algorithm capable of solving distributed ILP task has been devised. Furthermore, its fundamental properties, such as soundness and completeness have been stated and proven. The algorithm has also been implemented as a top theory and meta-rule processing layer for an existing implementation of DAREC.

The DTAL algorithm is considerably more general than existing distributed logic-based learning algorithms, described in short in Chapter 5, by allowing use of distributed background knowledge and non-observational predicate learning.

The robustness of DTAL may largely be seen as coming at cost of its scalability. Similarly to DAREC, which DTAL is based on, the state transfers between agents may require transmitting substantial amount of information, representing the current computational state. Resolving negative goals may be particularly expensive, because it requires remembering all the intermediate steps (all the possible ways of proving the negated goal) for future reference. This problem may be seen even in the Example 1 (Section 3.3), for which the set of dynamically collected constraints becomes seemingly very large, given the size of the learning task. Future work on DTAL can focus on investigating the scalability of the algorithm and finding ways to improve it.

One possible way of decreasing the communication cost would be modifying the abductive search process to introduce communicating incremental state information. Instead of passing complete computational state, only information about its change could be transmitted. This would certainly increase the complexity of the system itself, because agents would have to be aware of the version of state known to other agents. However, sending incremental state information does not have to be mandatory. If not certain about the current state version at one of its peers, an agent may resort to sending the full state. In many cases the state is transfered from agent A to agent B and then send back immediately. In such cases, it should be easy to express the state change incrementally.

As many ILP algorithms, DTAL in its current form does not allow any noise in the data, and attempts to find a precise logical definitions of the learned concepts, based on all of the examples. Especially in large distributed environments, it may be desired to search for a hypothesis *best* describing observed examples, rather then finding an exact solution or failing altogether. In the future, the possibility of extending DTAL to enable dealing with noisy data may be investigated. Since assuming noise in the data implies resigning from formal soundness with respect to entire set of examples, the work in this area could be combined with further research on reducing the cost of communication between agents - it could now be acceptable to deliberately drop some information from the state.

Bibliography

Bourgne, G., Bouthinon, D., Seghrouchni, A. E. F., and Soldano, H. (2009). Collaborative concept learning: Non individualistic vs individualistic agents. In *21st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2009, November 2, 2009 - November 5*, pages 653–657. IEEE Computer Society. pages 41

Bourgne, G., El, F. S., and Soldano, H. (2008). Smile: Sound multi-agent incremental learning. In *6th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS'07, May 14, 2008 - May 18*, pages 164–171. Association for Computing Machinery. pages 40

Corapi, D. (2011). *Nonmonotonic Inductive Logic Programming as Abductive Search*. PhD thesis, Imperial College London. pages 1, 5, 6, 7, 8, 9, 26, 27, 31

Corapi, D., Russo, A., and Lupu, E. (2010). Inductive logic programming as abductive search. In *26th International Conference on Logic Programming, ICLP 2010, July 16, 2010 - July 19*, volume 7, pages 54–63. Schloss Dagstuhl - Leibniz-Zentrum fur Informatik GmbH. pages 1

Fitting, M. (1985). A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295 – 312. pages 12

Huang, J. and Pearce, A. R. (2006). Distributed interactive learning in multi-agent systems. In *21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference, AAAI-06/IAAI-06, July 16, 2006 - July 20*, volume 1, pages 666–671. American Association for Artificial Intelligence. pages 39, 40

Huang, J. and Pearce, A. R. (2007). Toward inductive logic programming for collaborative problem solving. In *2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT'06, December 18, 2006 - December 22*, pages 284–290. Inst. of Elec. and Elec. Eng. Computer Society. pages 39

Kakas, A. C., Kowalski, R. A., and Toni, F. (1992). Abductive logic programming. *Journal of logic and computation*, 2(6):719–770. pages 4

Kakas, A. C., Van Nuffelen, B., and Denecker, M. (2001). A-system: Problem solving through abduction. pages 591 – 596. pages 4

Kazakov, D. and Kudenko, D. (2001). Machine learning and inductive logic programming for multi-agent systems. In *9th ECCAI Advanced Course, ACAI 2001 and Agent Links 3rd European Agent Systems Summer School, EASSS 2001*, Multi-Agent Systems and Applications, pages 246–70. Springer-Verlag. pages 3

Lavrač, N. and Džeorski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York. pages 3

Ma, J. (2011). *Distributed Abductive Reasoning: Theory, Implementation and Application*. PhD thesis, Imperial College London. pages 1, 4, 9, 10, 12, 27, 31, 32, 35, 36

Panait, L. and Luke, S. (2005). Cooperative multi-agent learning: the state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434. pages 1

Raedt, L. D. (2010). Inductive logic programming. In *Encyclopedia of Machine Learning*, pages 529–537. Springer US. pages 40

Weiß, G. and Dillenbourg, P. (1999). What is multiin multi-agent learning. *Collaborative learning. Cognitive and computational approaches*, pages 64–80. pages 2

Appendix A

DTAL source code

```
:-use_module(library(lists)).


%%%%% INTERFACE %%%%%

preprocess(Background, Mode, Examples, Theory) :-

    open(Background, read, BStream),
    clauses_from_file(BStream, BClauses),
    close(BStream),

    open(Mode, read, MStream),
    clauses_from_file(MStream, MClauses),
    close(MStream),

    open(Examples, read, EStream),
    clauses_from_file(EStream, EClauses),
    close(EStream),

    toptheory(MClauses, MTheory),
    examples_as_constraints(EClauses, Constraints),
    flatten([BClauses, MTheory, Constraints], TopTheory),

    open(Theory, write, TStream),
    clauses_to_file(TopTheory, TStream),
    close(TStream).


topic_file(Topic) :-
    open(Topic, write, Stream),
    clauses_to_file([
        (abducible(a)),
        (abducible(rule(_))),
```

```
        (builtin(link(_,_,_))),
        (builtin(append(_,_,_))),
        (builtin(nonmember(_,_)))
    ], Stream),
    close(Stream).


postprocess(Mode, HypoRepA, Hypothesis) :-
    open(Mode, read, Stream),
    clauses_from_file(Stream, MClauses),
    close(Stream),
    write(MClauses), nl, nl,
    delete(HypoRepA, a, HypoRep),
    extract_hypothesis(MClauses, HypoRep, Hypothesis).


run_mode_server(Mode, Port) :-
    open(Mode, read, Stream),
    clauses_from_file(Stream, MClauses),
    close(Stream),
    sample_server(Port, MClauses).



%%%%% TOP-THEORY GENERATION %%%%%

toptheory(M,TT) :-
    toptheory(M, 1, T),
    toptheory_terminator(X),
    append(T, [X], TT).


toptheory([], _, []).

toptheory([modeh(Pred)|DT], Counter, [R|RT]) :-

    %tal_mode_id(Counter, ModeId),
    dtal_mode_id(Counter, ModeId),

    Pred =.. [Name|ArgTypes],
    length(ArgTypes, Len),

    functor(Head, Name, Len),
    Head =.. [Name|Args],
    typecheck(ArgTypes, Args, TypeChecks),
    constants_and_variables(ArgTypes, Args, Constants, Variables, []),
    BodyPred =.. [body,Variables,[(ModeId, Constants, [])]],
```

```
    append(TypeChecks, [BodyPred], Body),
    list_to_tuple(Body, BodyTuple),
    R = (Head :- BodyTuple),

    Counter1 is Counter + 1,
    toptheory(DT, Counter1, RT).

toptheory([modeb(NPred)|DT], Counter, [R|RT]) :-

    % handle NBF in the same clause
    (NPred = -Pred; (NPred \= -_, NPred = Pred)),

    %tal_mode_id(Counter, ModeId),
    dtal_mode_id(Counter, ModeId),

    functor(RuleHead, body, 2),
    RuleHead =.. [body|[Input,Rule]],
    Pred =.. [Name|ArgTypes],
    length(ArgTypes, Len),
    functor(ActualPred, Name, Len),
    ActualPred =.. [Name|Args],

    typecheck(ArgTypes, Args, TypeChecks),
    LinkGoal = link(Input, Args, Links),
    constants_and_variables(ArgTypes, Args, Constants, _InputVar,
        OutputVar),
    RuleExtension = (ModeId,Constants,Links),
    RepetitionCheck = nonmember(RuleExtension, Rule),
    AppendRuleGoal = append(Rule, [RuleExtension], NewRule),
    AppendInputGoal = append(Input, OutputVar, NewInput),
    RecursiveGoal = body(NewInput, NewRule),
    include_negation(NPred, ActualPred, NActualPred),

    append(TypeChecks, [NActualPred, LinkGoal, RepetitionCheck,
        AppendRuleGoal, AppendInputGoal, RecursiveGoal], BodyList),
    list_to_tuple(BodyList, BodyTuple),
    R = (body(Input, Rule) :- BodyTuple),

    Counter1 is Counter + 1,
    toptheory(DT, Counter1, RT).


toptheory_terminator(X) :-
    X = (body(_Input, Rule) :- rule(Rule)).


tal_mode_id(Counter, Id) :-
    number_chars(Counter, CounterChars),
```

```
    atom_chars(Id, [m|CounterChars]).


dtal_mode_id(Counter, Id) :-
    alias(Alias),
    number_chars(Counter, CounterChars),
    atom_chars(Rest, [m|CounterChars]),
    atom_concat(Alias, Rest, Id).


typecheck([],[],[]).

typecheck([+any|TT], [_|VT], Checks) :-
    typecheck(TT, VT, Checks).

typecheck([-any|TT], [_|VT], Checks) :-
    typecheck(TT, VT, Checks).

typecheck([+TH|TT], [VH|VT], [CH|CT]) :-
    TH \= any,
    CH =.. [TH,VH],
    typecheck(TT, VT, CT).

typecheck([-TH|TT], [VH|VT], [CH|CT]) :-
    TH \= any,
    CH =.. [TH,VH],
    typecheck(TT, VT, CT).

typecheck([@TH|TT], [VH|VT], [CH|CT]) :-
    TH \= any,
    CH =.. [TH,VH],
    typecheck(TT, VT, CT).


include_negation(-_, Goal, (\+Goal)) :- !.

include_negation(_Pred, Goal, Goal).


constants_and_variables([], [], [], [], []).

constants_and_variables([TH|TT], [AH|AT], C, [AH|IT], O) :-
    TH = +_,
    constants_and_variables(TT, AT, C, IT, O).

constants_and_variables([TH|TT], [AH|AT], C, I, [AH|OT]) :-
    TH = -_,
    constants_and_variables(TT, AT, C, I, OT).
```

```
constants_and_variables([TH|TT], [AH|AT], [AH|CT], I, O) :-
    TH = @_,
    constants_and_variables(TT, AT, CT, I, O).



%%%%% HYPOTHESIS EXTRACTION %%%%%

extract_hypothesis(_Mode, [], []).

extract_hypothesis(Mode, [RepH|RepT], [RuleH|RuleT]) :-
    extract_rule(Mode, RepH, RuleH),
    extract_hypothesis(Mode, RepT, RuleT).


extract_rule(Mode, rule(Rep), Rule) :-
    extract_rule(Mode, Rep, [], NestedRuleList),
    flatten(NestedRuleList, RuleList),
    RuleList = [Head|BodyList],
    (list_to_tuple(BodyList, BodyTuple)
        -> Rule = (Head :- BodyTuple)
        ;  Rule = Head
    ).


extract_rule(_Mode, [], _Vars, []).

extract_rule(Mode, [RepH|RepT], Vars, [Goals|RuleT]) :-
    %RepH = (ModeId, Consts, Links),
    %atom_chars(ModeId, [m|NumberChars]), %% change for DTAL
    %%%% DTAL VERSION
    %decompose_mode_id(ModeId, Agent, Number),

    %%%%
    %number_chars(ModeNumber, NumberChars),
    %nth1(ModeNumber, Mode, ModeElem),

    RepH = (ModeId, Consts, Links),
    %tal_get_mode(Mode, ModeId, ModeElem),
    dtal_get_mode(Mode, ModeId, ModeElem),
    (ModeElem = modeh(Pred) ; ModeElem = modeb(Pred)),
    get_goal(Pred, Consts, Links, Vars, VarsOut, Goals),
    append(Vars, VarsOut, NewVars),
    extract_rule(Mode, RepT, NewVars, RuleT).


get_goal(NPred, Consts, Links, VarsIn, VarsOut, Goals) :-
```

```
    % handle NBF in the same clause %%%%%%%%%%%%%%
    (NPred = -Pred; (NPred \= -_, NPred = Pred)),

    Pred =.. [PredName|PredArgs],
    rule_args(PredArgs, Consts, Links, VarsIn, VarsOut, Guards, RuleArgs),
    Goal =.. [PredName|RuleArgs],
    include_negation(NPred, Goal, NGoal),
    Goals = [NGoal|Guards].


tal_get_mode(Mode, ModeId, ModeElem) :-
    atom_chars(ModeId, [m|NumberChars]),
    number_chars(ModeNumber, NumberChars),
    nth1(ModeNumber, Mode, ModeElem).


dtal_get_mode(Mode, ModeId, ModeElem) :-
    decompose_mode_id(ModeId, Agent, Number),
    alias(Agent),
    !,
    nth1(Number, Mode, ModeElem).

dtal_get_mode(_Mode, ModeId, ModeElem) :-
    decompose_mode_id(ModeId, Agent, Number),
    ask_for_mode(Agent, Number, ModeElem).


decompose_mode_id(ModeId, Agent, Number) :-
    atom_chars(ModeId, Chars),
    decompose_mode_id(Chars, [], AgentCharsR, NumberChars),
    reverse(AgentCharsR, AgentChars),
    atom_chars(Agent, AgentChars),
    number_chars(Number, NumberChars).


decompose_mode_id(['m'|T], BeforeR, BeforeR, T) :- !.

decompose_mode_id([H|T], Acc, Agent, Number) :-
    decompose_mode_id(T, [H|Acc], Agent, Number).


rule_args([], _, _, _, [], [], []).

rule_args([@_|PredArgT], [ConstH|ConstT], Links, VarsIn, VarsOut,
  Guards, [ConstH|RuleArgT]) :-
    write('rule_args:␣const\n'),
    rule_args(PredArgT, ConstT, Links, VarsIn, VarsOut,
```

```
        Guards, RuleArgT).

rule_args([-_|PredArgT], Const, Links, VarsIn, [X|VarsOut],
  Guards, [X|RuleArgT]) :-
    write('rule_args:␣out\n'),
    rule_args(PredArgT, Const, Links, VarsIn, VarsOut, Guards, RuleArgT).

rule_args([+any|PredArgT], Const, [Link|LinkT], VarsIn, VarsOut,
  Guards, [X|RuleArgT]) :-
    !, %% don't use the next rule, which would treat 'any' as a type
    nth1(Link, VarsIn, X),
    rule_args(PredArgT, Const, LinkT, VarsIn, VarsOut,
        Guards, RuleArgT).

rule_args([PredArgH|PredArgT], Const, [Link|LinkT], VarsIn, VarsOut,
  [GuardH|Guards], [X|RuleArgT]) :-
    PredArgH = +Type,
    write('rule_args:␣in\n'),
    write(nth1(Link, VarsIn, X)), nl,
    nth1(Link, VarsIn, X),
    write('after␣nth1\n'),
    GuardH =.. [Type,X],
    rule_args(PredArgT, Const, LinkT, VarsIn, VarsOut, Guards,
        RuleArgT).

%% For the input variable in the head.
%% Disjointness with the above clauses by empty link list,
%% may cause some invalid representations to pass.

rule_args([+any|PredArgT], Const, [], VarsIn, [X|VarsOut],
  Guards, [X|RuleArgT]) :-
    !, %% don't use the next rule, which would treat 'any' as a type
    write('rule_args:␣in2any\n'),
    rule_args(PredArgT, Const, [], VarsIn, VarsOut, Guards, RuleArgT).

rule_args([+Type|PredArgT], Const, [], VarsIn, [X|VarsOut],
  [GuardH|Guards], [X|RuleArgT]) :-
    GuardH =.. [Type,X],
    rule_args(PredArgT, Const, [], VarsIn, VarsOut, Guards, RuleArgT).



%%%%% TCP MODE INFORMATION INTERCHANGE %%%%%

ask_for_mode(Agent, Id, ModeDecl) :-
    agent(Agent, AdrPort),
    tcp_socket(Socket),
    tcp_connect(Socket, AdrPort, Streams),
```

```
    write(Streams, mode_q(Id)),
    write(Streams, '.\n'),
    flush_output(Streams),
    read(Streams, Msg),
    Msg = mode_a(ModeDecl),
    close(Streams).


mode_response(ModeList, Msg, Res) :-
    Msg = mode_q(Id),
    nth1(Id, ModeList, Mode),
    Res = mode_a(Mode).


sample_server(Port, ModeList) :-
    tcp_socket(S),
    tcp_bind(S, localhost:Port),
    tcp_listen(S, 5),
    tcp_open_socket(S, AcceptFd),
    sample_server_connection(AcceptFd, ModeList).


sample_server_connection(L, ModeList) :-
    tcp_accept(L, S, _),
    tcp_open_socket(S, Streams),
    read(Streams, Msg),
    mode_response(ModeList, Msg, Res),
    write(Streams, Res),
    write(Streams, '\n.'),
    flush_output(Streams),
    close(Streams),
    sample_server_connection(L, ModeList).


% A1: sample_server(4500, [modeh(p(+any)), modeb(q(+any))]).
% A2: assert(agent(s, localhost:4500)), ask_for_mode(s, 2, M).



%%%%% EXAMPLES AS CONSTRAINTS %%%%%

examples_as_constraints([], []).

examples_as_constraints([EH|ET], [CH|CT]) :-
    EH = (\+E),
    !,
    CH = (ic :- a, E),
    examples_as_constraints(ET, CT).
```

```
examples_as_constraints([EH|ET], [CH|CT]) :-
    CH = (ic :- a, \+EH),
    examples_as_constraints(ET, CT).



%%%%% FILE I/O %%%%%

theory_to_file(Theory, File) :-
    open(File, write, Stream),
    clauses_to_file(Theory, Stream),
    close(Stream).


clauses_to_file([], _Stream).

clauses_to_file([(H)|T], Stream) :-
    write(Stream, H),
    write(Stream, '.\n'),
    clauses_to_file(T, Stream).


clauses_from_file(Stream, Clauses) :-
    read(Stream, Term),
    (Term = end_of_file
        -> Clauses = []
        ;  Clauses = [Term|T], clauses_from_file(Stream, T)).


%%%%% THE LINK PREDICATE %%%%%

link(_Vars, [], []).

link(Vars, [H|T], [LinkH|LinkT]) :-
    link_one(Vars, H, LinkH),
    link(Vars, T, LinkT).


link_one(Vars, One, Link) :-
    link_one(Vars, One, 1, Link).


link_one([One|_VarT], One, Counter, Counter).

link_one([_|VarT], One, Counter, Link) :-
    Counter1 is Counter + 1,
    link_one(VarT, One, Counter1, Link).
```

```
%%%%% HELPER PREDICATES %%%%%

list_to_tuple([X], X).

list_to_tuple([H|T], (H,Rest)) :-
    list_to_tuple(T,Rest).


list_difference(A, B, Result) :-
    list_difference(A, B, [], ResultR),
    reverse(ResultR, Result).

list_difference([], _, Result, Result).

list_difference([H|T], ToRemove, Acc, Result) :-
    member(H, ToRemove),
    !,
    list_difference(T, ToRemove, Acc, Result).

list_difference([H|T], ToRemove, Acc, Result) :-
    list_difference(T, ToRemove, [H|Acc], Result).


nonmember(Item, List) :-
    \+ member(Item, List).


alias(Alias) :-
    nb_getval(myalias, Alias).
```