Reasoning and interaction in logic-based multi-agent systems

Paolo Torroni

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Dottore di Ricerca
of the
University of Bologna.

Dipartimento di Elettronica, Informatica e Sistemistica
Università di Bologna, Italy

January 15, 2002

to Zeynep

Acknowledgments

I would like to thank all those people who have supported me throughout this PhD. Firstly, I would like to thank my brother Stefano and my parents. Thanks also to my supervisor Maurelio Boari, for his constant presence, for his words, always precious, for his support and his genuine interest in my work. Thanks to Paola Mello for the perception she showed in guiding me in my scientific education and development, and for being a prestigious reference in all kinds of situations I encountered over the years my work took to complete. Thanks to Francesca Toni and Fariba Sadri for looking after me in London, for their friendship, hospitality, and intellectual inspiration.

In addition, I would like to thank all those who have played a role in my academic life and education, and in my personal life within the University. Thanks to my co-authors, particularly to Anna Ciampolini, for the past and present collaborations, and for carefully reading this work, and to Evelina Lamma, Michela Milano and Rosy Barruffi, for all that I learned from them. Thanks to Marco Gavanelli, Fabrizio Riguzzi, Paolo Bellavista, Andrea Roli, Silvia Vecchi, Marco Prandini, Andrea Omicini, Franco Zambonelli, Dario Maio and Quinto Fattori. Thanks also to Ken Satoh, João Leite, Antonio Brogi and Jürgen Dix for an enjoyable time spent together and for several inspiring conversations.

Last, but not least, special thanks to all my friends who have always been around me over these years. I feel very lucky to have met them: for their kinship, intelligence, trust and, in some cases, love. Many thanks to Luigi Marvasi for the long discussions at the Des Arts about the Cytochrome P450, to Francesco Panciroli for waking me up last time we went to the mountains, to Ombretta Conti and her love for the middle-east countries, of which I am being a victim myself, to Alessandro Arcangeli (DJ Brad) for the truffled cheese and paté de foie d'oie, to Valentina Branchini for recognizing me in Sevilla despite my hair cut, to Mauro Bovina for relieving my feet from the pain caused by the sea urchin thorns in Cape Sounio, to Francesca Rimondi and Francesco Ficarra for the dedication in their book [42], to Annalisa Coliva for being in many respects the first one, to Rocco Musolino for his coherence, and to Cesare Strappaveccia, who never failed to explore with me seas and countries. Thanks to Elena Giovannini and Simonetta Enrico for taking care, in different times, of my friend Francesco Panciroli. Thanks to zia Maria, my cousins Cristina, Angelo, Francesco and Michela, zia Cecchini, zio Paolo, zia Bianca and Maurizio, for giving shelter and peace to my restless feelings, and to Enrica Gasparroni, Sophie Noucher, and Magdalena Bierza for shaking them up. Thanks to Yannis Xanthakos, Yvonne Morguet, Olli Kamarainen, Fabien Beugre, Angelo Maggiore and all the Kofia Society, the Dog Man, Leo Liberti, Laura Basile, Alessio Lomuscio, Caterina Chironi, Tito and Toby, Mariagiulia (DJ Janet) and Cécile Poirier for cheering up my staying in the dump County of Clayponds. Thanks to Tiziana and Enrico, who made it possible for me to see what cannot be seen any more, to Monika Kwiatosz for offering me the humidifier that I still use, to Elena Zeni for believing in my fishing skills, to Jana Scheibner, who once visited me in Monzuno, to Helen Down, who once visited me in London and who made at least 3 very valuable corrections to this part of my work, to Stefan Kreutmüller, who founded the Blubbern Club at the Left Wing Students', to Federica Bonetti and Benedetta Bonfiglioli, who survived the end of the secondary school, to the Bob Marella theater company, whose past glory still shines, and to the Associazione Piccolo Formato, the Accademia 96, and Radio Città del Capo, who gave an essential contribution to my cultural development.

Very special thanks to Zeynep, for the feelings that she has for me, for her intelligence, for her smile, her songs, and her sweetness.

Abstract

In late years, agent systems have been massively studied, both from a theoretical and a practical viewpoint. In particular, intelligent agents require both reasoning capabilities, and social capabilities, which make possible their interaction in a society. In order to model interaction among agents, two aspects have to be taken into account and properly combined. The first aspect is concerned with the language that agents use to communicate; the second one deals with the knowledge representation and logic used to model the agent's internal knowledge, i.e., both the agent local behavior and its knowledge about the outer world. Social capabilities of agents are obtained through suitable models and languages that allow to represent agent interaction. In particular, depending on the kind of reasoning adopted by each agent, different interaction and coordination strategies could be defined. Thus, it is necessary to adopt models and mechanisms that could be possibly expressed through a language for interaction among the agents.

As far as modeling agent local behavior and its knowledge about the external world, it is not possible, in general, to make assumptions either on knowledge uniqueness (i.e., more agents can hold different or equivalent views of the world) or on knowledge completeness (each agent can have, in general, a partial and incomplete knowledge about the outer world). It is thus necessary to extend classical logic and reasoning techniques in order to make the agents able to manage incomplete and multiple knowledge, and to let them interact with one another in order to acquire the information necessary for their own computation. In this context, logic languages, and logic programming in particular, thanks to its declarative and operational aspects, if suitably extended, are a good tool for modeling interaction among agents, integrating local reasoning but also embracing openness towards other agents. Besides, logic programming would also be a good formalism for possibly verifying some global properties of the resulting system.

The main objective of this thesis is to study the application of abductive reasoning to a multi-agent setting. Several possibilities are considered to coordinate the agent reasoning, with a special focus on abduction, leading to the design of an agent architecture, Abductive LogIc AgentS (ALIAS, for short), where (logic-based) agents are intelligent and have social abilities, and where local reasoning and global interaction are specified in a logic programming-based paradigm.

In particular, in this architecture a form of intelligence stems from the agents' capability to perform, with their own knowledge, abductive reasoning beside deduction. With reference to social ability, ALIAS agents can coordinate their reasoning with other agents, following several coordination schemes. To this end, I introduce the concept of "bunch of arguing agents". In particular, ALIAS agents could be grouped in statically defined groups, or they can "collaborate"

or “compete” in the solution of problems when required, forming bunches that are dynamically created, on demand, depending on the problem to solve.

The system architecture is layered, in particular reasoning and social behaviour are kept separate and correspond to different levels of abstraction: the upper level specifies, by means of LAILA expressions, possibly complex schemata of interaction among agents; it can also exploit the reasoning capabilities local to the agents (located at the lower level, and specified by abductive logic programming).

One of the main advantages in modeling agent reasoning and interaction through logics is the possibility to describe formally (in an operational and/or in a declarative fashion) the behaviour of the system and its properties. In this respect, one original contribution of this work is the possibility to express local behaviour, interaction and integration of local and global reasoning through inference rules of a proof system. These inference rules, introduced and discussed in the thesis, are a suitable description of the operational semantics of the resulting system and can be considered the basis of a formal characterization of abductive logic agents, subject to be completed with a declarative counterpart and also suitable for inferring some properties about the global behaviour.

The scope of this thesis does not want o restrict itself to abductive reasoning. The reasoning coordination language is generalized, to define a set of operators, formally specified in terms of syntax and operational semantics, that can be applied to a generic agent system where agents play the role of servers, that provide services under certain conditions. Based on this more general architecture, some correctness and completeness results are proven, which also apply to the previous architecture, but that turn out to be independent on the kind of local computation or service that agents provide.

Despite its foundational approach, this work also wants to put a stress on the applications that can be possibly tackled by the framework. In the last part of the thesis, the results of this thesis are applied to a typical multi-agent problem, that of negotiation. A dialogue-based negotiation framework is presented, along with some interesting results regarding termination, soundness, and completeness. Another domain addressed by this work is that of auctions, which can be also modeled within an abductive framework, as a particular case of negotiation.

Contents

List of Figures

Chapter 1

Introduction

Many real problems deal with distributed, complex environments and require sophisticated reasoning capabilities. In many cases, it is required the resolution of more aspects at the same time. A centralized approach, where one single program tackles all the aspects of the problem, may result too complex and too computationally expensive. A more promising approach consists, instead, of the cooperation of several components, often called agents. These agents must be autonomous and have independent computational capabilities; on the other hand, they have to interact with other agents and with the "world" they reason about and act upon.

In late years, agent systems have been massively studied, both from a theoretical and a practical viewpoint [93, 85, 103]. In particular, intelligent agents require both reasoning and social capabilities, which make possible interaction, coordination, collaboration and competition among different agents.

In order to model interaction among agents, two aspects have to be taken into account and properly combined. The first aspect is concerned with the language that agents use to communicate; the second one deals with the knowledge representation and logic used to model the agent's internal knowledge, i.e., both the agent local behavior and its knowledge about the outer world. On both sides, the literature in the Multi-Agent Systems community is enormous.

Social capabilities of agents are obtained through suitable models and languages that allow to represent agent interaction. In particular, depending on the kind of reasoning adopted by each agent, different interaction and coordination strategies could be defined. Thus, it is necessary to adopt models and mechanisms that could be possibly expressed through a language for interaction among the agents. It is worth to mention here various works in the literature concerning models and languages to express communication [44, 43] and coordination [50, 19] among agents. In particular, agent communication languages such as KQML [43] have been formally described, and are currently used in both academic research and industrial / commercial applications.

As far as modeling agent local behavior and its knowledge about the external world, it is not possible, in general, to make assumptions either on knowledge uniqueness (i.e., more agents can hold different or equivalent views of the world) or on knowledge completeness (each agent can have, in general, a partial and incomplete knowledge about the outer world). It is thus necessary to extend classical logic and reasoning techniques in order to make the agents able to manage incomplete and multiple knowledge, and to let them interact with one another in order to acquire the information necessary for their own computation. Some proposals use a *Belief, Desire and Intention* (BDI) model [84] for agent beliefs, goals, and intentions and adopt a modal logic formalism. The claim of such an approach is its effectiveness when dealing with domains where it is important to model other agents' beliefs, desires, and intentions, but very few applications use this potential in practice, because the modal logic formalism for BDI agents is still rather descriptive than operational. In this context, logic languages, and logic programming in particular, thanks to its declarative and operational aspects, if suitably extended, would be a good tool for modeling interaction among agents, integrating local reasoning but also embracing openness towards other agents. Besides, logic programming would also be a good formalism for possibly verifying some global properties of the resulting system. It is worth to mention here that there exists significant work concerning intelligent agent systems based on (possibly extended) logic programming (see, for instance, [65, 29]) in which various reasoning techniques based on logic programming have been developed and extended for dealing with incomplete knowledge.

In order to model local and global behavior, the language used for agent communication and the one used to specify the agent local knowledge should be properly combined. The problem of how to (smoothly) combine these aspects in an expressive and consistent way is still an open issue. Several possibilities have been studied, to coordinate the agent reasoning, with a special focus on abduction. This led to the design of an agent architecture, *Abductive LogIc Agent System* (ALIAS, for short), where (logic-based) agents are intelligent and have social abilities, and where local reasoning and global interaction are specified in a logic programming-based paradigm.

In particular, in this architecture a form of intelligence stems from the agents' capability to perform, with their own knowledge, abductive reasoning beside deduction. In a multi-agent setting, in fact, it is often the case that an agent requires some sort of guess about a computation which cannot be performed locally, since for it only incomplete information is known. In this respect, the Closed World Assumption [20] can be no longer assumed, and some form of open or abductive reasoning has to be considered. Abduction has been widely recognized as a powerful mechanism for hypothetical reasoning in presence of incomplete knowledge [25, 40, 59]. Abduction is generally understood as reasoning from effects to causes, and also captures other important issues such as reasoning with defaults and beliefs (see for instance [64, 81]). In a single agent context, hypotheses are made provided that they are consistent with the agent's current knowledge. In a multi-agent context, the need emerges for specific protocols for global consistency maintenance, as agents interacting each another, although reasoning with their own local knowledge base, cannot completely set aside from other agents beliefs. In particular, the hypotheses generated by an individual agent on behalf of a second one need to be re-checked for consistency with the beliefs of the second.

With reference to social ability, ALIAS agents can coordinate their reasoning with other agents, following several coordination schemes. In particular, ALIAS agents could either "collaborate" or "compete" in the solution of problems. In order to express such different coordination schemes, a language is defined, the *Language for AbductIve Logic Agents* (LAILA, for short).

The system architecture is layered, in particular reasoning and social behaviour are kept separate and correspond to different levels of abstraction: the upper level specifies, by means of LAILA expressions, possibly complex schemata of interaction among agents; it can also exploit the reasoning capabilities local to the agents (located at the lower level, and specified by abductive logic programs).

One of the main advantages in modeling agent reasoning and interaction through logics is the possibility to describe formally (in an operational and/or in a declarative fashion) the behaviour of the system and its properties. In this respect, one original contribution of this work is the possibility to express local behaviour, interaction and integration of local and global reasoning through inference rules of a proof system. These inference rules, introduced and discussed in the thesis, are a suitable description of the operational semantics of the resulting system and can be considered the basis of a formal characterization of abductive logic agents, subject to be completed with a declarative counterpart and also suitable for inferring some properties about the global behaviour.

However, the scope of this thesis does not want o restrict itself to abductive reasoning. Drawing inspiration from the achieved results, and in particular from the reasoning coordination language, a generalization of it is provided, that results in a set of operators, formally specified in terms of syntax and operational semantics, that can be applied to a generic agent system where agents play the role of servers, that provide services under certain conditions. Integrity constraints are used for example to ensure that such conditions do not conflict with each other. A concrete scenario that is presented is that of policy management in a computer system. ALIAS agents can be also interpreted as servers, in this new perspective, enclosing constraints in their local abductive logic programs, and providing, as a service, answers to queries made on their own local knowledge. Based on this more general architecture, some correctness and completeness results are proven with respect to its declarative semantics, which also apply to its abductive counterpart, ALIAS, but that turn out to be independent on the kind of local computation or service that agents provide.

Despite its foundational approach, this work also wants to put a stress on the applications that can be possibly tackled by the framework. In the last part of the thesis, the results obtained in the first chapters are applied to a typical multi-agent problem, that of negotiation. Negotiation plays a central role in many multi-agent applications, such as e-commerce, web auctions, and distributed systems management, especially with respect to resource allocation [91]. In domains such as e-commerce, where it is natural to assume that agents are self-interested and lack complete knowledge and resources, it is likely that they need to negotiate in order to obtain the information or resources they need. The dialogue protocols used for negotiation have to be properly designed in order to ensure that each part will have a positive payoff out of the negotiation process. To this purpose, a dialogue-based negotiation framework is described, and some interesting results regarding termination, soundness, and completeness are presented. Another domain addressed by this work is that of auctions, which can be also modeled within an abductive framework, as a particular case of negotiation.

The thesis is organized as follows. Chapter 2 introduces logic-based reasoning, and abductive reasoning in particular, with respect to two concrete proof-procedures for abductive reasoning. There, I give some general notions about logic-based multi-agent systems, and some definitions. Chapter 3 discusses the problem of agent interaction and coordination of reasoning in a multi-agent system, with a special focus on abductive reasoning. I present different system architectures, where agents can group in several ways. For each architecture, I present a concrete example to show how it suits to a particular setting. Chapter 4 is devoted to describing a language that can be used to coordinate such reasoning and express several coordination patterns; the idea of coordination of reasoning is generalized, to cover situations of agents that are not necessarily capable of abductive reasoning, and not even based on logic. Chapter 5 studies in detail a possible use of abduction in multi-agent system, with respect to the case of agent dialogue and negotiation. Chapter 6 draws conclusions, along with a discussion about the related work, and a brief outline of future research directions.

Chapter 2

Preliminaries

In this chapter I will firstly focus on the concept of abductive reasoning in agents, introducing some preliminary notions on this respect that are used throughout the thesis. In particular, from a logic programming perspective, I will give a declarative semantics for abduction, and present two different abductive proof procedures, both used in this thesis: the one proposed in 1990 by Kakas and Macarella [60], and the if-and-only-if (IFF) proof procedure by Fung and Kowalski [48]. Secondly, I will give a brief overview of multi-agent systems, and give to this respect some definitions.

2.1 Abductive logic programming

An assumption that is often made in multi-agent systems is that individuals only have a partial knowledge of the environment they are situated in. Therefore, in most knowledge-intensive applications of agents, it is often the case that an intelligent agent requires some sort of *guess* about a computation which cannot be performed locally, since the local knowledge is incomplete. In a logic programming perspective, we could rephrase it by saying that a guess is needed because the agent has a *goal* which cannot be solved locally. In this situation, the Closed World Assumption [20] usually adopted in logic programming can be no longer assumed, and some form of *open* or *abductive* reasoning has to be considered.

Abduction has been widely recognized as a powerful mechanism for hypothetical reasoning in the presence of incomplete knowledge [58, 25, 40, 59]. Abduction is generally understood as reasoning from effects to causes, and captures other important issues, such as reasoning with defaults and beliefs (see for instance [64, 81]). Incomplete knowledge is handled by labeling some pieces of information as abducibles, i.e., possible hypotheses which can be assumed, provided that they are consistent with the current knowledge base. In the context of intelligent agents, abduction can be regarded as a way of *dynamically* enlarging the agent's knowledge with abducible atoms. In the case of a single agent that uses a proof to reason abductively, such proof will guarantee that the abduced hypotheses that enlarge the agent's knowledge base are consistent with her knowledge. To this purpose, an agent's knowledge base is equipped with some integrity constraints, that must never be violated. As we will see in the next chapters, in a multi-agent environment, abductive reasoning might require a form of coordination among agents in order to guarantee that abduced hypotheses are assumed *consistently* in the group of agents involved in such distributed computation. Before giving a formal definition of abductive logic programs, I will briefly recall some basic concepts about logic programs.

The vocabulary of logic programs is defined in [75]. A logic program is a set of clauses

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

where A is an atom, and each L_i is a literal, i.e., an atom B or its negation *not* B. An *atom* is an n-ary predicate symbol followed by a list of n terms $p(t_1, \ldots, t_n)$, or `true`, or `false`. A *literal* is an atom or the negation of an atom. A *term* is a constant, a variable or a compound term. Also, in this thesis I will adopt the Edinburgh notation for standard Prolog, where variable names start with upper case or '_', and constants start with lower case.

Note that `not` represents default negation or negation as failure. All variables in $A, L_1, \ldots, L_n$ are assumed to be universally quantified from the outside. A is referred to as the *head* of the clause and $L_1, \ldots, L_n$ as its *body*.

Definition 1 (*abductive logic program*) An *abductive logic program* (ALP) is a triple $\langle P, \mathcal{A}, IC \rangle$ where P is a logic program, possibly with abducible atoms in clause bodies; $\mathcal{A}$ is a set of *abducible predicates*, i.e. *open* predicates which can be used to form explaining sentences; IC is a set of integrity constraints.

Given an abductive program $\langle P, \mathcal{A}, \mathcal{IC} \rangle$ and a sentence G, the purpose of abduction is to find a (possibly minimal) set of abducible atoms δ which together with P "entails" G, and such that $P \cup \delta$ "satisfies" IC. Depending on the chosen semantics for the given logic program [58], various notions of "entailment" are possible, such that, for every ground literal in the language, either the literal or its complement is entailed, and such that no literal and its complement are entailed at the same time. In the following, I will use the following notation for the concept of "abductive entailment":

$$\langle P, \mathcal{A}, IC \rangle \overset{abd}{\vdash}_{\delta} \quad G$$

and it is possible to formalize its declarative semantics in the following way:

Definition 2 (*declarative semantics of an ALP*) Let $\langle P, \mathcal{A}, IC \rangle$ be an abductive logic program. Then, given a goal G, the *declarative semantics* of

$\langle P, \mathcal{A}, IC \rangle \overset{abd}{\vdash}_{\delta} \; G$ is defined as follows:

$$\begin{cases} P \cup \delta \models G \\ P \cup \delta \text{ ``}satisfies\text{''} \;\; IC \\ \delta \subseteq \mathcal{A} \end{cases}$$

where $\models$ is the entailment symbol.

In the following, I will not distinguish between an agent x and her abductive logic program $\langle P, \mathcal{A}, IC \rangle$, thus adopting the abbreviated notation $x \overset{abd}{\vdash}_{\delta} \; G$ in place of $\langle P, \mathcal{A}, IC \rangle \overset{abd}{\vdash}_{\delta} \; G$.

There are also different notions of "integrity constraint satisfaction" in literature. Different procedures / semantics will adopt different such notions. Moreover, there is not a unique *syntax* to write integrity constraints: depending on the adopted proof procedure, they could be written for example as denials, as in the case of the Kakas-Mancarella, or as forward rules (implications), as in the IFF. All proofs require that each integrity constraint contains at least one abducible, and that abducible predicates are not defined, i.e., there is no clause in an ALP that has an abducible in the head. In fact, this would mean that the abducible could be derived by deduction, while we consider abducibles as open predicates that can be assumed true or false, depending on the goal to solve. It is worthwhile noticing that, given a program P, possibly containing definitions of abducible predicates, it is possible to syntactically transform P into another semantically equivalent program, by means of auxiliary predicates, where no abducible is defined. Such transformation is shown in [58]. The following example shows a simple abductive logic program, written according to the syntax of integrity constraints required by the Kakas-Mancarella proof procedure.

Example 1 Let us consider the following program, inspired by [79], where $ained\_last\_night$ and $sprinkler\_was\_on$ are the only abducible predicates in $\mathcal{A}$:

$$\begin{array}{ll} grass\_is\_wet & \leftarrow rained\_last\_night \\ grass\_is\_wet & \leftarrow sprinkler\_was\_on \\ shoes\_are\_wet & \leftarrow grass\_is\_wet \end{array}$$

with integrity constraint:

$\leftarrow rained\_last\_night, sprinkler\_was\_on.$

In order to satisfy such constraint, the predicates $rained\_last\_night$ and $sprinkler\_was\_on$ cannot be assumed both true at the same time. The observation $shoes\_are\_wet$ is therefore explained by two (minimal) sets of sentences, respectively:

$Abd_1 = \{rained\_last\_night, not\ sprinkler\_was\_on\}$ and

$Abd_2 = \{sprinkler\_was\_on, not\ rained\_last\_night\}$

According to [40], negation as default, possibly occurring in clause bodies, can be recovered into abduction by replacing negated literals of the form *not a* with a new positive, abducible atom *not_a* and by adding the integrity constraint $\leftarrow a, not\_a$ to IC. The natural syntactic correspondence between a standard atom and its negation by default is given by the following notion of complement:

$$\bar{l} = \begin{cases} \alpha & \text{if } l = not\_\alpha \\ not\_\alpha & \text{otherwise} \end{cases}$$

where α is an atom. All negated literals in an ALP are considered abducibles.

I will always leave implicit the integrity constraints used for handling negation as default, like the constraint $\leftarrow p, not\ p$, and suppose that they are replicated in each agent's knowledge base.

I will briefly introduce now two proof procedures for abductive derivation, that implement in different ways the declarative semantics defined above. These two are not the only abductive proof procedures defined in literature; among the others, one could cite [33, 24].

2.1.1 The Kakas-Mancarella proof procedure

The Kakas-Mancarella (KM) proof procedure [60] is an extension of the proof procedure for ordinary logic programming proposed by [40]. Both procedures extend the basic resolution mechanism adopted in SLD and SLDNF in ordinary logic programming by introducing the notion of *abductive* and *consistency* derivations.

I will only give here an intuition of the KM; the interested reader may refer to [60] for an exhaustive description. It extends the basic resolution mechanism adopted in logic programming by introducing the notion of *abductive* and *consistency* derivation. Intuitively, an *abductive* derivation is the usual logic programming derivation suitably extended in order to consider abducibles. When an abducible atom h is encountered during this derivation, it is assumed, provided this is consistent. The consistency check of a hypothesis, then, starts the second kind of derivation. The *consistency* derivation verifies that, when the hypothesis h is assumed and added to the current set of hypotheses, any integrity constraint containing h fails (i.e., the bodies of all the integrity constraints containing h are false). During this latter procedure, when an abducible L is encountered, in order to prove its failure, an abductive derivation for its complement, $\overline{L}$, is attempted. In other words, as I said before, an abductive derivation computes a set of hypotheses "entailing" (with respect to the given logic program) a given goal/query/observation, starting from a given set of initial hypotheses. In order to compute such set and guarantee that the integrity constraints are "satisfied", an abductive derivation might require subsidiary consistency derivations, which in turn might require additional abductive derivations to ensure integrity constraint "satisfaction".

Instead, I will give some detail about the syntax of such ALPs. In its original formulation, the KM requires that P is a propositional logic program, expressed in form of ground clauses. The integrity constraints are expressed in the form of denials of atoms, as in Example 1; at least one of such atoms must be abducible. Integrity constraint satisfaction in KM requires that at least one atom in the body of an integrity constraint be false. For instance, given the constraint

$\leftarrow a, b, c.$

and the set $\{a, b\}$ of abducibles, if we want to assume a to be true, we must either assume b false (by abducing *not* b) or prove that c is false. Both ways may require a deductive derivation in P, and possibly some more consistency steps. Such process is iterated until we find a consistent and "stable" set of abducibles δ, in the sense that further consistency steps do not result in an enlargement of such δ. The KM can be adopted also with first order logic programs (not only propositional): in that case, all variables in the body of an integrity constraint are assumed to be universally quantified from the outside. The KM proof procedure has been proven sound [12], for some specific notions of "entailment" and "satisfaction", with respect to the semantics defined in [38] and weakly complete for the 3-valued stable model semantics of [12], as well as the argumentation-semantics defined in [98].

2.1.2 The IFF proof procedure

Fung and Kowalski's IFF [48] is a rewriting abductive proof-procedure consisting of a number of inference rules. It does not implement the concept of consistency derivation introduced by the KM, but allows for constructive negation and abduction. In fact, integrity constraints and queries are treated identically [70]. This proof procedure is general, and has been shown to unify abductive logic programming, constraint logic programming and semantic query optimization. Two basic inference rules are *unfolding* (backward reasoning), which replaces an atom with its definition , and *propagation* (forward reasoning), which resolves an atom with an implication. Implications are obtained by repeatedly applying the inference rules of the proof procedure to either an integrity constraint in the given program (ALP), or to the result of rewriting negative literals *not* A as $false \rightarrow A$. The abductive logic programs for the IFF are syntactically different from those seen in the previous section. As with the KM, I will not describe the proof-procedure in detail here, but I will show the syntax of an ALP for the IFF. Let us first rewrite the ALP of Example 1 in a form that is suitable for the IFF. The program P turns out:

$grass\_is\_wet \leftrightarrow rained\_last\_night \vee sprinkler\_was\_on.$
$shoes\_are\_wet \leftrightarrow grass\_is\_wet.$

while the set of integrity constraints IC turns out:

$rained\_last\_night \Rightarrow not\ sprinkler\_was\_on.$

$sprinkler\_was\_on \Rightarrow not\ rained\_last\_night.$

In general, an abductive logic program as defined in [48] is composed of backward rules that constitute the program P, expressed in the form of if-and-only-if definitions, while the integrity constraints are forward rules, expressed in the form of implications[1]. As I did before in the general case, the semantics of abductive logic programs is given by specifying what constitutes an answer to a query. An answer is a set of definitions for the abducible predicates which, together with the initially given program P, implies an instance of the query and satisfies the integrity constraints. In this way, the IFF combines backward and forward reasoning: the former by unfolding ground instances of the if-and-only-if definitions, the latter by propagating on the ground instances of the integrity constraints whose body has become true. Given a query Q, i.e., a conjunct of literals, as in the KM, the abductive derivation starts from an initial formula, which is Q conjoined with the set of integrity constraints, producing an answer to Q. The IFF proof procedure is both sound and complete relative to the three valued completion semantics.

2.2 Logic-based multi-agent systems

The agent concept is widely used in distributed artificial intelligence, although no real agreement has ever been reached on its definition. In fact, the term 'agent' has been used to describe a broad range of computational entities, ranging from simple systems to very large ones [56]. A rather general idea of an intelligent agent is that of a software or hardware system that is autonomous, interactive with and *reactive* to its environment and other agents. An agent can also be *pro-active* in taking the initiative in goal-directed behavior. Thus, the agent concept is systematically used to represent entities with the ability to solve problems, reflecting the results on the environment. Moreover, intelligent agents must solve these problems while working in environments not completely under their control. Intelligent agents need deductive and pattern-matching capabilities to accomplish tasks and fulfill activity requests. Thus, the knowledge of an agent can be specified by using the logic programming paradigm [100], and particularly, a logic language.

I will define here the concept of agent system, from a logic-programming perspective.

Definition 3 (*agent system*) An *agent system* is a finite set A, where each $x \in A$ is a ground term, representing the name of an agent, equipped with a *knowledge base* $\mathcal{K}(x)$.

The definition of knowledge base $\mathcal{K}(x)$ for an agent x may change, depending on the agent architecture. In the case of abductive logic agents, i.e., agents

[1] The implications in [48] are written with a different but equivalent notation

capable of abductive reasoning, $\mathcal{K}$ will include an ALP. We will assume that, in an agent system, the agents share a common language as well as a common content language. In particular, if the context of abductive logic agents, the individuals will have separate knowledge bases but will refer to a common set of abducibles $\mathcal{A}$.

Logic-based intelligent agents are the result of the contribution of computational logics to agent research, providing conceptual and practical tools for developing agents that are both reactive and pro-active in the sense intended above. These tools have been listed in [69] and include, among others:

- **definitions in logic programming form**, to represent the knowledge and beliefs of an agent and to implement "rational" reduction of goals to sub-goals;
- **integrity constraints**, to represent an agent's persistent goals and to implement "reactive" behaviour in the manner of condition-action rules and active database rules;
- **abduction**, to generate alternative partial plans of action that an agent can follow to achieve its goals;
- **argumentation semantics and proof procedures**, to facilitate reasoning about conflicting beliefs and goals and to facilitate conflict avoidance and reconciliation.

Aim of this thesis is to study the application of abductive reasoning in a multi-agent context: therefore, we do not limit the use of abduction to planning, but rather consider it as the main form of agent reasoning. This does not exclude a priori the possibility of providing the agents with other forms of reasoning. Furthermore, I do consider integrity constraints, in that they are used to trigger other abductive computations. Thus, as far as conflicting beliefs and goals are concerned, I am proposing several techniques aimed to give agents the means to maintain consistency in an environment made of multiple individuals, and broadly, to drive agent reasoning through possible conflicts of beliefs. I anticipate an example in this respect.

Example 2 Let us consider again Example 1, using the KM notation, where all the knowledge needed to find an explanation to the observation $shoes\_are\_wet$ is enclosed in a single agent, that we will call x. Now, let us suppose that another agent, say y, has a different knowledge about the same problem. In particular, y knows that the sprinkler takes the water from a tank, which is emptied after the sprinkler has been on; thus, let $\mathcal{K}(y)$ include the following integrity constraint:

$$:-\ sprinkler\_was\_on, tank\_is\_full.$$

where $tank\_is\_full$ is not an abducible, but a predicate of y's knowledge base. Both agents could profit from collaboration on the explanation of the observation made by x, provided that it is consistent with both knowledge bases. In order for them to communicate the result of their reasoning, expressed in form of conjuncts of abducibles, we assume that they share the same set of abducibles. Let such set be $\mathcal{A} = \{sprinkler\_was\_on, rained\_last\_night\}$ plus all negated literals. The two agents therefore form a bunch together, $B = \{x, y\}$.

Now, there are two possibilities: either y can prove that $tank\_is\_full$ is true, or she cannot, in which case she will consider *not* $tank\_is\_full$ to be true, since it is a closed predicate. In the first case, it could be reasonable for x to consider $Abd_1 = \{rained\_last\_night, not\ sprinkler\_was\_on\}$ as a possible explanation to $shoes\_are\_wet$, not $Abd_2 = \{sprinkler\_was\_on, not\ rained\_last\_night\}$. In the second case Abd_2 is a more reasonable explanation than Abd_1. In both cases, it could be useful for both agents to store the newly acquired knowledge, represented by the abduced hypotheses, for future reference: if this is the case, an observation made by one agent results in an enlargement of the knowledge base of all agents in B.

In a context where an agent (x) considers another agent (y) trustworthy, and y has some knowledge that x doesn't have, that is relevant to x's problems (for example it can help motivate the observation that the shoes are wet), x and y could participate at the same statically defined bunch. Otherwise, if for example x only wants to consult y on a given problem, and interacts with different agents at different times, we need to define a more flexible framework where agents can express several grouping and interaction patterns, depending on the problem to solve.

2.3 Application scenaria

An architecture which binds abduction capabilities with distributed constraint satisfaction is suitable in several fields of application. In particular, we shall consider situations in which knowledge is incomplete and could be scattered among several weakly dependent entities. In fact, interest in having distributed knowledge could rise when most of the computation is performed locally. In the following, I want to give a flavor of some possible applications in the real world. In particular, in this thesis, I will study in detail the application of abduction in negotiation.

On-board diagnosis of automotive systems. On-board diagnostic systems play an important role in the current and future generations of cars [13]. Several electronic apparatuses devoted to controlling different subsystems of the car could benefit from coordination in terms of raised hypotheses and constraint satisfaction. That is a case in which several weakly connected knowledge bases are present.

User profiling. Abduction could be used to profile users by guessing (making hypotheses on) her preferences. This kind of application is becoming more and more interesting, especially in contexts such as the Internet, where it is often the case that the domain of customers considerably changes over time, and either it is not practical to keep a record for each user, or there is anyway not enough information about the customers can be obtained to build up a profile for each customer. An example in the domain of travel agents is shown in the course of the thesis; another one comes from the music market, where an interesting issue is that of generating customized music programs [77]. The idea is to propose to the user sequences of music titles, instead of sets of individual titles, thus customizing on-the-fly the offer of CDs with respect to some constraints. A level of abduction over the constraint satisfaction mechanisms could provide the system with descriptive capabilities: some unknown a priori customer preferences could be treated as hypotheses to abduce, in order to build a set of hypotheses to tag and characterize the generated sequence, in a similar way to that adopted for travel agents.

Agent negotiation Negotiation is one of the main research areas in multi-agent systems. In many cases agents need to negotiate because they operate in environments with limited resource availability. The area of negotiation is broad and applies to different scenarios [54]. Some kinds of negotiation are based on dialogue. In [89, 88] the authors show how an abductive proof procedure can be used to produce dialogues among agents. This issue will be explored in detail in Chapter 5.

In the following chapter I will present several architectures that exhibit different degrees of flexibility in this respect.

Chapter 3

Coordination of (abductive) reasoning in multi-agent systems

In this section I will consider the issue of coordinating multiple logic-based agents, proposing in that respect a multi-agent architecture. The focus will be on agents capable of abductive reasoning, in order to design a system that is suitable for the solution of problems in a distributed environment where knowledge might be incomplete, multiple and even conflicting.

We can think of a multi-agent system a composed of many different agents, some of which capable of (abductive) reasoning. An individual may need to coordinate her reasoning activity with one or more partners. There could be different ways to do that. In this thesis, we will consider two different approaches to it. A first possibility is that agents are statically grouped into *bunches*, where they coordinate their reasoning, making sure in particular that, whatever hypothesis or set of hypotheses is produced by an abductive derivation of a member of a bunch, there is no constraint in the other agents' programs that is violated. This could be the case of static systems, such as those in charge of the diagnosis of distributed components of a car, or of a (closed) computer system. Observations could be the symptoms of a component's failure, such as a lamp being faulty [31], but there could be different causes to the observed "symptoms": for example, the lamp could be broken, or there could be a fault in the power supply system. Rather than modeling the entire system as a monolithic system whose fault has to be diagnosed, we could delegate the diagnosis activity to several dedicated devices, with the obvious advantage brought by a modular architecture: if we want to use different components for building several systems, we do not need to design each time a new diagnostic component from scratch, provided that we have a mechanism to coordinate such separated components and ensure that they are consistent with each other. This is also the approach used in component based software engineering. I will present one such mechanism, called Distributed Abduction Algorithm (DAA).

Another possibility could be that of open system. We can imagine an extension to the previous architecture for DAA, where again we have bunches, that are persistent, but in this case some agents from the external world can join a given bunch B, and of course leave it when there is no more reason for them to be there. We will call such agents *rambling*, to reflect the idea that they can move from bunch to bunch. It is easy to imagine situations where rambling agents could be needed. If we think of a network scenario, such as for instance the Internet, agents can move from one host to another, asking for being introduced into existing bunches of agents – o maybe just form a bunch by joining one single agent – and coordinate her reasoning with that of her partners, in order to find a set of hypotheses that, starting from an initial query or goal, is consistent with all their constraints. We could equip rambling agents with an already abduced set of hypotheses that they bring with them wherever they go. This is the case of an agent that is sent by its peer to different places, with the purpose to collect some information that is consistent with all visited bunches. The desired information is obtained as a result of an abductive derivation. The mechanism that can be used for this purpose can be again the DAA, but the architecture must be suitably extended in order to support incoming agents. In fact, the abduced set of hypotheses δ that a rambling agent brings with her must be checked for integrity within a bunch, before she can join it and make any query, whose result is required consistent with δ. I will present an example with a travel agency that makes uses of such extended architecture.

Generalizing this architecture, we can imagine that agents are not constrained to operate with their reasoning within static bunches, but they can rather issue the construction of a new bunch "on demand" at every goal they wish to prove. In this case, we need a "reasoning coordination language", that provides some operators to express the dynamic grouping of agents, the proof of abductive queries in the bunch, and the request for an abductive derivation or for a consistency check that involves the dynamically obtained bunch. We will introduce a language, LAILA (Language for AbductIve Logic Agents), that provides such coordination operators, and allows the dynamic grouping of agents in different patterns. I will formally define such language in the next chapter, where I give its syntax and an operational semantics. Here, I will give an example of such approach in the medical diagnosis domain.

I will conclude this chapter by describing with some detail an implementation of ALIAS an by discussing some experimental results.

3.1 Static agent bunches

In this section I will describe the architecture of the *Abductive LogIc Agent System* (*ALIAS*) with statically defined bunches. In this version of ALIAS,[1]

[1] In this thesis, several version of the system for coordinating abductive reasoning among agents are presented. I will refer to such different versions with the same name, *ALIAS*. In case of possible ambiguity, I will distinguish among such architectures by specifying if I

logic agents are equipped with hypothetical reasoning capabilities, obtained by means of abduction.

Let us now start by defining what is a bunch of agents.

Definition 4 (*bunch of agents*) Given an agent system A, a *bunch of agents*, or more simply a *bunch*, is a subset B of A, $B \subseteq A$.

Agents can both perform standard deduction and *abduce* new hypotheses, provided that they are consistent with the knowledge of the other agents in the bunch. To this purpose, a mechanism to coordinate agent reasoning is introduced. In particular, reasoning and coordination are integrated within the Distributed Abduction Algorithm (DAA, for short), which is executed within each logic agent. The DAA algorithm is based on the form of abductive reasoning defined by Kakas and Mancarella (see 2.1.1 for details). Each agent is characterized by statically defined local knowledge represented by an abductive logic program [58]. Moreover, a global knowledge (represented by the assumed hypotheses posted into a shared tuple space) is dynamically built. While static knowledge is peculiar to each agent and might differ from agent to agent, all agents must agree on the global set of assumed abducibles. To this purpose, a set of integrity constraints is used - together with program clauses - to confirm or discard new hypotheses. In this respect, an agent is *pro-active* when it executes (its own) goals, while it is *reactive* to its environment, and cooperate with other agents when it checks the consistency of hypotheses raised by other agents.

3.1.1 The architecture of *ALIAS* for static bunches

In this version of *ALIAS*, the computation is carried out by several agents of a multi-agent system, A, that cooperate for solving goals. Each agent $x \in A$ has its own (possibly incomplete) knowledge base $\mathcal{K}(x)$, and uses abduction as a way of hypothetical reasoning. Agents can be grouped into *bunches*: each bunch B of *arguing* agents is associated with a set Δ of *abducible* hypotheses (*i.e.*, the set of hypotheses that could be possibly raised). Moreover, agents of the same bunch refer to the common set of hypotheses assumed so far (the dynamic knowledge of the bunch). When an agent x tries to raise a new hypothesis h, agents belonging to the same bunch cooperate with x (by means of the coordination protocol) in order to check that h is consistent with the knowledge of the other agents and with the assumptions made so far: if all agents agree about the assumption of h, h is assumed; otherwise it is discarded. There could be more bunches in A, but the composition of a bunch is statically specified. For each bunch there is a blackboard, used as a shared space to facilitate the communication of hypotheses among agents. The blackboard is implemented as a tuple space, where tuples can be retrieved and posted by means of operators that are equivalent to the `rd`, `in` and `out` operators of LINDA [49].

am considering the version "with static bunches", "with rambling agents", or "with dynamic

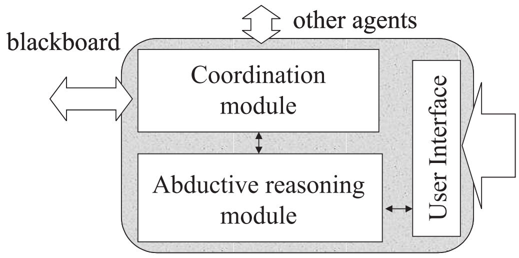

Figure 3.1: *ALIAS* agent functional structure

The inner structure of each *ALIAS* agent is shown in Figure 3.1: each agent is composed of three functional blocks: a *reasoning* module, which contains the abductive (and deductive) reasoning mechanisms, a *coordination* module which interfaces the agent with other agents in the system, and a user interface module for the interaction with external users. Each agent x can accept queries from external users by means of the *user interface* module; each query q is passed to the *reasoning* module which performs a local computation in order to calculate an answer for q. If the local knowledge of x is insufficient to solve the query, the solution of q could be possibly obtained by making additional assumptions. Each time x tries to abduce a new hypothesis h, the coordination module starts a message exchange session involving x and other agents in the bunch, in order to check the consistency of h with the knowledge of the other agents and with the assumptions made so far. At the end of this coordination activity, if all agents agree in assuming h, the hypothesis is inserted in the common set of assumptions associated with the bunch, and the computation can proceed. At the end of the computation (if successful) the set of assumptions is stored in the bunch tuple space, and the answer is returned to the external user (through the user interface). Agents are autonomous and parallel: this means that within the same bunch more than one computation could be started in parallel. For this reason, accesses to the tuple space have to be suitably synchronized, in order to preserve the consistency of the dynamic knowledge.

3.1.2 Agent reasoning and coordination

Each bunch consists of $n+1$ agents *arguing* out a common set of abducibles. Each bunch is also associated with a dynamic knowledge, represented by the set Δ, containing the hypotheses so far assumed by all agents in the group. Each agent x_i, for $i = 0, \ldots, n$, encapsulates (in its reasoning module) an abductive logic program $\langle P_i, \mathcal{A}_i, IC_i \rangle$, as defined in Chapter 2. In this framework, the KM notation for abductive logic programs is adopted, and, for the sake of simplicity, only ground programs are considered, thus assuming that P and IC have already been instantiated.

bunches".

It is worthwhile reminding that each integrity constraint in IC_x of an agent x has at least one abducible in the body, and that abducible predicates have no definition, as in [60]. As concerns integrity constraints, the user-defined ones are partitioned among the various agents, while those for handling negation as default, like the constraint $\leftarrow p, not\ p$, are left implicit and replicated in each agent's knowledge base. The set Δ contains the hypotheses so far assumed by all agents. The set of program clauses and integrity constraints might differ from agent to agent, but we assume that the set of abducible predicates (default predicates included) is the same for all the agents in a bunch.

3.1.3 The Distributed Abduction Algorithm

The DAA algorithm adopted for abductive reasoning in a multi-agent system extends the Kakas and Mancarella proof-procedure in the sense of distribution: now knowledge is distributed among various agents, and consistency derivations have to be coordinated within a pool of logic agents.

Definition 5 (*abductive derivation*) An *abductive derivation* from $(G_1\ \Delta_1)$ to $(G_n\ \Delta_n)$ for an agent x (with arguing agents $x_1, \ldots, x_m$) with knowledge base $\mathcal{K}(x)$ given by the abductive logic program $\langle P_x, \mathcal{A}, IC_x \rangle$ via a selection rule R is a sequence $(G_1\ \Delta_1), (G_2\ \Delta_2), \ldots, (G_n\ \Delta_n)$ such that each G_i has the form $\leftarrow L_1, \ldots, L_k$, $R(G_i) = L_j$ and $(G_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:

(1) If L_j is not abducible, then $G_{i+1} = C$ and $\Delta_{i+1} = \Delta_i$ where C is the resolvent of some clause in P_x with G_i on the selected literal L_j;

(2) If L_j is abducible and $L_j \in \Delta_i$ then
$G_{i+1} = \leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k$ and $\Delta_{i+1} = \Delta_i$;

(3) If L_j is abducible, $L_j \notin \Delta_i$ and $\overline{L_j} \notin \Delta_i$ and for each arguing agent A_k $(k = 1, \ldots, m)$ there exists a *consistency derivation* from $(\{L_j\}\ \Delta_i \cup \{L_j\})$ to $(\{\}\ \Delta'_k)$ and the union set $\cup_{k=1,\ldots,n} \Delta'_k$ is consistent (i.e., L and $\overline{L}$ do not belong to the union set for any literal L), then
$G_{i+1} = \leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k$ and
$\Delta_{i+1} = \cup_{k=1,\ldots,n} \Delta'_k$.

Steps (1) and (2) are local SLD-resolution steps using only the rules of x and abductive hypotheses, respectively. Step (3) requires the coordination of all agents in the bunch: the new hypothesis L_j could be possibly assumed if all arguing agents perform a successful consistency derivation for L_j, starting from $\Delta_i \cup \{L_j\}$. In general, the consistency of each hypothesis L_j is checked concurrently by all the arguing agents. Each consistency check, when successful, might require the assumption of other hypotheses. If all consistency derivation are successful, L_j, together with the additional hypotheses Δ_i returned by the arguing agents, is added to the current set of hypotheses, provided that the union set of these hypotheses is consistent; then, the computation proceeds within x by assuming L_j and the set $\Delta_{out} = \bigcup_{i=1,\ldots,n} \Delta_i$.

Definition 6 (*consistency derivation*) Given an agent x (with arguing agents $x_1, \ldots, x_m$) whose knowledge base $\mathcal{K}(x)$ is given by the abductive logic program $\langle P_x, \S, IC_x \rangle$, *a consistency derivation* for an abducible literal α from $(\alpha\ \Delta_1)$ to $(F_n\ \Delta_n)$ is a sequence $(\alpha\ \Delta_1), (F_1\ \Delta_1), (F_2\ \Delta_2), \ldots, (F_n\ \Delta_n)$ where :

(i) F_1 is the union of all goals of the form $\leftarrow L_1, \ldots, L_n$ obtained by resolving the abducible α with the denials in IC_x with no such goal been empty, $\leftarrow$;

(ii) for each $i > 1$, F_i has the form $\{\leftarrow L_1, \ldots, L_k\} \cup F'_i$ and for some $j = 1, \ldots, k$ $(F_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:

(C1) If L_j is not abducible, then $F_{i+1} = C' \cup F'_i$ where C' is the set of all resolvents of clauses in P_x with $\leftarrow L_1, \ldots, L_k$ on the literal L_j and $\leftarrow \notin C'$, and $\Delta_{i+1} = \Delta_i$;

(C2) If L_j is abducible, $L_j \in \Delta_i$ and $k > 1$, then $F_{i+1} = \{\leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k\} \cup F'_i$ and $\Delta_{i+1} = \Delta_i$;

(C3) If L_j is abducible, $\overline{L_j} \in \Delta_i$ then $F_{i+1} = F'_i$ and $\Delta_{i+1} = \Delta_i$;

(C4) If L_j is abducible, $L_j \notin \Delta_i$ and $\overline{L_j} \notin \Delta_i$, and there exists a (local) *abductive derivation* from $(\leftarrow \overline{L_j}\ \Delta_i)$ to $(\leftarrow\ \Delta')$ then $F_{i+1} = F'_i$ and $\Delta_{i+1} = \Delta'$.

It is worth to notice that the consistency derivation involves only local computation. In fact, abducible predicates have no definition (as in [40, 59]); if, during a consistency derivation, an abducible atom h is encountered, a local abductive derivation is performed. That corresponds to assume h and then, during this local abductive derivation, to ask each arguing agent for the *consistency* check of h.

3.1.4 An example

Agents coordinate themselves on the basis of the set of assumed hypotheses Δ by using a global repository for knowledge. All agents have to access the global repository for getting the current set of assumed hypotheses to be considered in their further computations. The global repository can be realized by a tuple space accessed in a reading, reading-and-consuming, and writing mode through Linda-like primitives `rd`, `in` and `out` [49].

It is worth noting that several agents may issue distinct hypotheses, in parallel and independently one of each other. This may lead to a situation where two agents may try and store into the tuple space two conflicting sets of hypotheses.

Example 3 Let us consider a system composed of four agents (A_0, A_1, A_2, A_3) whose knowledge base is structured as follows:

$A0)$	$\texttt{s} :- \texttt{p}.$	$A1)$	$:- \texttt{q}, \texttt{not b}.$
$A2)$	$:- \texttt{p}, \texttt{not c}.$	$A3)$	$\texttt{r} :- \texttt{q}.$ $:- \texttt{b}, \texttt{c}.$

where p, q, b, and c are abducible atoms. Let the current set of hypotheses Δ be empty. Suppose that agent A_0 raises the goal $\leftarrow$s and agent A_3 the goal $\leftarrow$r. Each single request is processed in parallel, producing two different and conflicting sets of hypotheses: $\{\texttt{p}, \texttt{c}, \texttt{not b}, \texttt{not q}\}$ for A_0 and $\{\texttt{q}, \texttt{b}, \texttt{not c}, \texttt{not p}\}$ for A_3.

A possible solution to the conflict is to guarantee the mutual exclusion of agents in accessing the set of hypotheses in the tuple space, so that the first agent succeeding in extracting the current Δ blocks further accesses until the consistency check terminates. Other computations are *de facto* serialized on the basis of the agent relative speeds in accessing the tuple space. However, two different computations starting with the same set of current hypotheses might result in two sets which are consistent with each other. In that case, agents may read in parallel the tuple space, while the blackboard checks the consistency of consequent updates.

In the DAA, the knowledge is distributed among various agents, and consistency derivations have to be coordinated within a pool of logic agents. In that sense, we say that the whole "(distributed) abductive logic program" embedded in a bunch architecture is not just the union of the agent abductive logic programs embedded in a single agent. In fact, since the consistency derivations run locally, the Prolog clauses of an agent's abductive logic program are not globally visible, and are not accessed in the way they would be if they were part of one single logic program. On the other hand, this separation among the agent knowledge bases does not involve the set of currently abduced hypotheses, which are globally visible, nor the integrity constraints, which operate globally. We say that *ALIAS* implements a *union* of logic programs, but just as far as the hypotheses assumed true and the integrity constraints are concerned. Formally, the declarative semantics of the DAA is defined as follows:

Definition 7 (*declarative semantics of the DAA*) Let B be a bunch composed of $n+1$ agents, $n \geq 0$. Then, for each $x \in B$, such that $\mathcal{K}(x) = \langle P, \mathcal{A}, IC \rangle$, given a goal G, let $x_B \overset{DAA}{\vdash}_{\delta} G$ be the distributed aductive derivation in B issued by x, starting from G and producing δ. The *declarative semantics* of $x_B \overset{DAA}{\vdash}_{\delta} G$ is defined as follows:

$$\begin{cases} P \cup \delta \models G \\ P \cup \delta \not\models \bigcup_{x_j \in B} IC_j \\ \delta \subseteq \mathcal{A} \end{cases}$$

Being the DAA an extension of the KM, we see in this definition how the

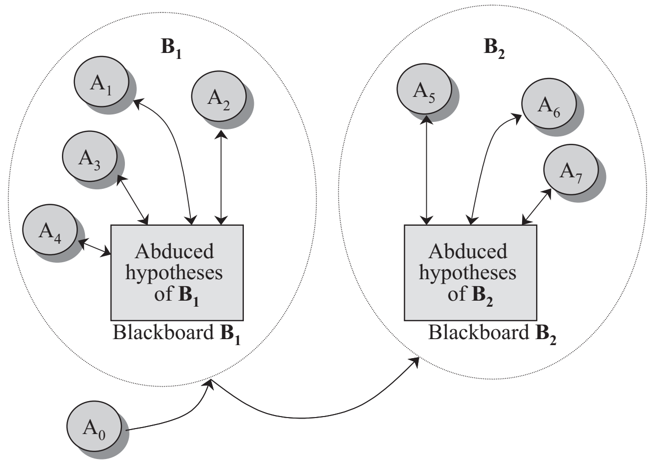

Figure 3.2: The architecture of *ALIAS*

concept of "integrity constraint satisfaction" has also a precise semantics, that given by the symbol $\not\models$.[2] The set of abducible $\mathcal{A}$ plays the role of a content language for the agent communication.

A prototype of this architecture has been implemented, based on Java and *Amzi!* Prolog, to execute on a TCP/IP network of computers. I will discuss such implementation, along with some experimental results, at the end of this chapter.

3.2 Rambling agents and dynamic bunches

We have seen so far how agents can be grouped into static bunches, and a distributed extension to the KM proof-procedure has been formalized, to coordinate their abductive reasoning. In this section, I introduce the possibility to compose the bunches dynamically. To this purpose, ALIAS agents are classified in two classes: *rambling* agents and *still* agents. Each still agent is statically anchored to its bunch, and it cannot migrate to a different bunch. Its knowledge base contributes to computations occurring inside its bunch during the entire lifetime of the system.

A scheme is presented in Figure 3.2. Each still agent refers to the dynamic knowledge of the bunch it belongs to, as the set of current hypotheses. Rambling agents, instead, can move from one bunch to another at runtime; moreover, each rambling agent handles a local dynamic knowledge that represents the set of hypotheses acquired during its life. For this reason, we say that rambling

[2] In this case, since it is an extension of the Kakas Mancarella proof procedure, the notion of "satisfaction" of an integrity constraint ic recovers into the fact that the logic program plus the set of hypotheses does not entail it, i.e., the head of such constraint cannot possibly be derived: $P \cup \delta$ "satisfies" ic if and only if $P \cup \delta \not\models ic$.

agents are *agents with experience.* When an agent dynamically wants to enter a bunch, it expresses its interest by raising a suitable event thus starting a protocol aimed at checking the consistency of its local knowledge base with the global knowledge of the bunch, Δ.

A rambling agent x may dynamically enter a bunch B for a variety of reasons. For instance: it might want to communicate its own experience to the bunch, thus enriching the dynamic knowledge of B; it might want to contribute to the reasoning activities of B, offering its knowledge base to the bunch; it might want to execute a query within that bunch; it might simply want to acquire the hypotheses assumed so far inside the bunch. Movements of rambling agents could be determined directly by the user, or they could be driven by higher level (e.g., planning) policies. In order to correctly deal with the dynamic management of bunches, each bunch is equipped with proper mechanisms that support both the entering and the exiting of rambling agents. In particular, the mechanism to enter a bunch must guarantee the consistency of the hypotheses in the dynamic knowledge of the bunch with the knowledge base of the new agent.

3.2.1 Rambling agents = agents with experience

As anticipated, we can say that rambling agents may have *experience*: this means that each rambling agent x encapsulates the set of hypotheses δ collected so far, as the result of having visited one or more bunches. Of course, by a more general point of view, such δ could be processed by a higher level layer as the agent is out of any arguing bunch, e.g. in order to follow data management policies based on persistence, relevance or other attributes of the assumed hypotheses.

Still agents, as opposed to rambling agents, do not have an own "experience", but refer to the "global experience" associated to a bunch, represented by the current set of abduced hypotheses Δ that they retrieve from the bunch's blackboard before starting a computation. A dynamic management of bunches requires a suitable protocol, that I am going to define in the following. I anticipate that the protocol such that for the set of hypotheses Δ stored on the blackboard of a bunch B the following property will hold: for all rambling agents x with experience δ that entered B, $\delta \subseteq \Delta$.

3.2.2 Entering a bunch

A major feature of *ALIAS* is to allow an agent logged in a bunch to keep its own coherent knowledge base, which could be different with those of its bunch partners. The only point in which the agents of a bunch must agree is that they are referring to a global dynamic knowledge, that is, the dynamic set of abduced hypotheses, Δ. The Distributed Abduction Algorithm guarantees that Δ is always consistent with each agent's integrity constraints.

A problem arises when an agent, x, which did not participate in formulating Δ, wants to join the bunch. In fact, x might contain some integrity constraints which are in contrast with Δ itself. The dual problem rises when an incoming agent has got its own set of abduced hypotheses, δ, and wants to join the bunch proposing it to accept δ. In that case, δ has to be the object of a distributed consistency check step.

By a general point of view, agent knowledge is constituted by an abductive logic program, plus a set of assumed hypotheses δ, representing the experience of the agent itself. I will describe the protocol to join a bunch with reference to a bunch B, and an incoming (rambling) agent x with experience δ, $\delta \in \mathcal{K}(x)$. The protocol is two-fold: the first phase takes place within x, while the second one involves all agents in the bunch, plus x.

Phase 1. Firstly, x asks B for being logged in. Then, the bunch awaits, until all the running computations terminate, prevents other distributed abductions from being started during agent entering and returns the current Δ to x. In the first phase of the protocol, x checks if Δ is consistent with δ[3]. This requires the *local* evaluation of the query $\Delta \cup \delta$, obtained by a local abduction process within x. If the query fails, x is not let in the bunch. Otherwise, a new additional set of hypotheses Δ_1 is obtained, such that $x \overset{abd}{\vdash}_{\Delta_1} \Delta \cup \delta$.

Phase 2. The second phase of the protocol aims to check the consistency of Δ_1 in the whole bunch. If Δ_1 is empty, x is simply let in. Otherwise a distributed abduction process is started by x within B in order to find a Δ_2 which together with the global knowledge entails Δ_1: $x_{B\cup\{x\}} \overset{DAA}{\vdash}_{\Delta_2} \Delta_1$. Again, this step could succeed or not: in this latter case x will have to leave the bunch. Otherwise, x is confirmed part of the bunch, which becomes $B' = B \cup \{x\}$, and the global knowledge becomes Δ_2.

In plain words, we could say that Δ_1 represents the hypotheses x must add to his δ if he wants to get in, and that Δ_2 is the result of the negotiation of the whole set of hypotheses, once x is let in the bunch.

3.2.3 Exiting a Bunch

There could be two different ways of exiting of one agent from a bunch: a *traceful* leaving and a *traceless* leaving.

A traceful leaving implies that an agent x exiting a bunch B leaves in the set Δ associated with B a set of hypotheses abduced with its contribution, i.e., the hypotheses are also supported by x knowledge base. In that case, x can exit after notifying the event to the bunch. This can be useful when we consider rambling agents in some way as *control* agents, whose movements from bunch to bunch aim at updating some knowledge distributed among several bunches.

In the case of a traceless leaving, an agent x going out of B doesn't leave any hypothesis assumed with its contribution within the bunch after its exiting. In this case, the exiting protocol would be more complicated, since the introduction

[3] Given two sets of abduced hypotheses, Δ and δ, they are considered *syntactically* consistent if their union $\Delta \cup \delta$ does not contain both a literal and its negation.

of some form of belief revision could be required. A simple solution could be to require that an incoming agent who intends to leave the bunch without leaving trace of its passage is allowed to participate to the bunch only for the time needed by the log in protocol. This could be interesting when we use rambling agents for testing the state of the arguing process of a certain bunch. In that case we implement traceless leaving if we simply don't store the login delta in the blackboard. The definition of more elaborated protocols is subject for current and future work.

As x exited B, a process of filtering could run on the *experience* matured by the agent during its staying in the bunch. In general, we could think to associate some attributes with abducibles in order to provide them with a meta-level semantics to be of support to a higher level reasoning (e.g., a *planning*) module, or an *agent behavior module* as we will see in the next section.

3.2.4 An example with an Internet travel agency

As promised at the beginning of this chapter, we show a possible use of rambling agents, by means of an example in the domain of Internet services. In particular, we considered an environment composed of several bunches, that are part of a distributed architecture which implements a *web travel agency.*

When a customer enters a traditional travel agency she is usually asked her preferences and needs, the maximum price she intends to pay for a travel and so on. Provided with such *input*, the travel agency returns in *output* a more or less wide range of possible travels. Most of the times there is no point in asking the customer a detailed description of the input, for the employee could try and imagine the customer's preferences, or could also consider several possible profiles and suitable scenarios. For example, a profile could be that of a customer who likes the mountains and wants to sleep in a comfortable hotel room, another one prefers some adventurous off-limits holidays and does not mind sleeping in a tent, while she is ready to pay a fortune to see the lions... a third one has got children and just wants to have some rest, and of course appreciates very much all kinds of baby-sitting facilities. The perfect travel agency employee has to cope with last minute offers and deals made in the background and does a just-in-time travel planning for a wide range of possible customers. Which is obviously quite a hard task, requiring good coordination skills and a lot of fantasy. That scenario is also difficult to scale up, for everything is managed by a single person. Due to its complexity, and to the fact that it is relatively easy to implement such a system, once designed, based on existing Internet databases, the theme of travel agents spurred from the early days of intelligent agents, and is still often taken as a test-bed by multi-agent systems [28].

Now, let us consider an *Abductive* Web Travel Agency, implemented using the rambling agents technology, and based on the Internet. From a Home Page, the customer is asked to fill in a form, containing preferences, which she could either select or leave blank. After a while, the Web customer is given a wide range of opportunities to choose among. They are in the form:

- *Rocky Mountains* – if you like the adventure and the mountains and flying by plane, please click here for a travel to the Rocky Mountains
- *Yucatan* – if you wish to spend less than $ 500 and you are interested in Archeological sites and you do not care moving by bus, please click here for a travel to the Yucatan
- ...

In other words, the system makes certain hypotheses about the customer, assuming things that the customer did not either think about. Let us see how this can be implemented.

In the real world, the information is scattered among different agencies, and most of the computation is made locally. If we consider the agencies being represented by bunches or by agents in bunches, and rambling agents representing the customers, moving from bunch to bunch simulates the activity that real customers do in the real world obtaining different answers to the same query (i.e., *those are the requirements of customer x: is there a suitable travel for her?*).

Let us assume that the Abductive Web Travel Agency Home Page transforms the information given by the user into a Logic Program; this could be done for example by means of a syntactical translator, but I am not going to discuss about such translation here. For the sake of this example, let us consider a family composed by two adults and two children of six and three, looking for a relaxing travel to the beach from the 7^{th} to the 14^{th} of August, and that does not want to spend too much but does not want to go camping either. For such a customer, a logic program could be the following list of facts:

$under\_4.$
$class\_economy.$
$notcamping.$
$beach.$
$high\_season.$
$notbusiness.$

Those assumptions constitute the δ of an agent that will be created for the customer. The agent could also be provided with a more complex logic program and integrity constraints, specifying some policies or needs of the Web Travel Agency. Possibly, the web site of the travel agency has also a certain number of integrity constraints and a logic program, to which respect the agent δ becomes the query to solve. For instance,

$\leftarrow under\_4, off\_limits.$
$\leftarrow over\_60, off\_limits.$
$\leftarrow class\_business, camping.$
$\leftarrow class\_business, bus.$
$\leftarrow relax, bus.$

$\leftarrow class\_economy, plane, high\_season.$
$\leftarrow class\_economy, class\_business.$
$\leftarrow under\_4, notbaby\_sitting\_service.$
$\leftarrow off\_limits, relax.$

with intuitive semantics (for example, $\leftarrow under\_4, off\_limits$ means that it is not allowed to propose a travel of kind "off-limits" to a family where there is a child under 4, and so on). As a consequence of running the query δ on that logic program, we obtain a new set of hypotheses, δ':

$\{under\_4, class\_economy, not\ camping, beach, high\_season, not\ business,$
$not\ class\_business, not\ off\_limits, not\ plane, baby\_sitting\_service\}.$

At this point, the Web Travel Agency creates several instances of the agent created for the customer, and sends them to n different bunches, each of them managing a certain kind of travel: for instance, the "Adventure Bunch", the "Business Bunch", the "Art Bunch", the "Relax Bunch", and so on. Each agent submits its δ as a query to the bunch it wants to access: depending on the failure or success of such query, the agent can be either rejected or let in. In this example, we may suppose that the agents are rejected from the "Adventure Bunch" and from the "Business Bunch". Entering a bunch causes an enlargement of the agent δ: for instance, after entering the "Relax Bunch" the hypothesis $relax$ is added. Here we can assume that the agents do not modify the bunch Δ (that is, a policy of *traceless* leaving is adopted). After one or more of such steps, which guide the agents through a multitude of bunches, the rambling agents can move to some places where different destinations are proposed, for instance we can imagine that a rambling agent who left the "Relax Bunch" is lead to another bunch which handles the following knowledge:

$suitable\_plan \leftarrow saint\_tropez.$
$suitable\_plan \leftarrow taormina.$
$suitable\_plan \leftarrow rimini.$
$suitable\_plan \leftarrow marrakech.$
$suitable\_plan \leftarrow dolomiti.$
$saint\_tropez \leftarrow beach, class\_business.$
$taormina \leftarrow beach, history\_and\_art.$
$rimini \leftarrow beach.$
$marrakech \leftarrow folklore, plane.$
$dolomiti \leftarrow mountains.$

and tries to solve the query ? $suitable\_plan$ as soon as the rambling agent is let in. The agents which are finished get back to the Web Travel Agency Home Page where they show their acquired knowledge in order for the Web Travel Agency to produce a list of readable, user-friendly choices.

Figure 3.3 shows a possible interaction of a rambling agent with two such travel agency bunches. A customer wants to relax and travel in economy class.

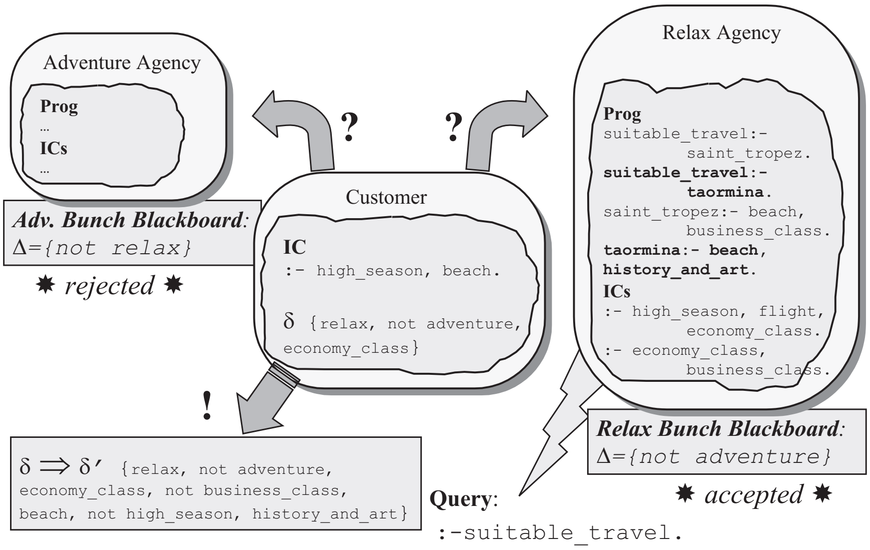

Figure 3.3: A rambling agent representing a customer of a travel agency

She doesn't want to go to the beach in high season. The travel agency generates a rambling agent which codifies her knowledge and sends her through the other sites, where she is let in or not depending on the compatibility of her knowledge base and local δ with the knowledge bases and Δ of the respective bunches. When she manages to enter a bunch, an agency tries to run a query (*suitable_travel*) which, if succeeded, generates a new δ for the agent, describing the kind of travels it could offer. The abducibles are: {*adventure, beach, business_class, economy_class, flight, high_season, history_and_art, relax*}, plus all the negated literals. The set of abducibles δ' shown in the figure will support the derivation of the predicate *taormina*, which is the option that is eventually shown to the customer by the agent representing the "relax" agency.

3.3 High level coordination of agent reasoning

I have presented so far an agent architecture where several agents, each one enclosing a local knowledge base, can perform abductive reasoning in collaboration with other agents that are stable or occasional components of a bunch. This architecture uses an extension of the KM proof-procedure for abductive reasoning.

My intent is now to generalize this architecture, in order to model different kinds of interaction, and in order to have bunches that do not exist independently of a computation, but that instead are dynamically created every time it is needed. Therefore, I will present an architecture where agents can either autonomously reason using their own local knowledge base or dynamically join other agents to cooperatively solve goals using abductive reasoning [15].

3.3.1 Collaboration and competition in ALIAS

Two basic patterns are provided to coordinate the reasoning of agents, that can be suitable combined to form more complex ones. Therefore, I will briefly introduce the concepts of *collaboration* and *competition* before I show the architecture of the agent.

In the *collaborative* case, the solution of a (possibly complex) task is distributed among several agents. Let us consider, as an example, a group of medical doctors, each one expert in a particular area (e.g. gastroenterology, ematology, etc.) who have to collaborate in order to formulate a diagnosis for a given set of symptoms. Each expert can be modeled by an abductive agent whose task is to find a hypothesis (i.e., a disease) as an explanation for a subpart of the symptoms (i.e., those relevant to her area) given as observations to the agent. In some cases, the hypotheses made by an agent can be inconsistent with other hypotheses made by a collaborative agent (for instance, a certain symptom s does not occur when the disease d is present).

In the *competitive* case, instead, several agents could be asked to solve the same problem. Let us consider again a group of medical doctors, each one expert in a particular area, who have to find a diagnosis for the symptom. Different, alternative diagnoses can be proposed by the different agents, and the *best* one can be chosen according to some policy (for instance, the one with major incidence, or the most plausible one, with respect to the clinical history of the patient, etc.).

In this last version of ALIAS (featuring dyamic bunches), agent interaction and coordination is specified by using a logic language named LAILA (Language for AbductIve Logic Agents) that allows to model agent structures, especially from the viewpoint of social behaviour. In particular, LAILA provides agents with the possibility to express different cooperation schemata by means of high level declarative operators, such as the communication operator $>$, the competition operator ;, the collaboration operator &, while a down-reflection operator $\downarrow$ allows a local abductive resolution to be triggered. I will define LAILA in detail in the next chapter.

3.3.2 ABM and ARM

The inner structure of each agent, shown in Figure 3.4, is basically composed of two modules: the *Abductive Reasoning Module*, ARM, and the *Agent Behavior Module*, ABM. Each agent is characterized by two knowledge bases: the *abductive knowledge base*, encapsulated within the ARM, which corresponds to the knowledge $\mathcal{K}$ I have referred to so far, and the *behavior knowledge base*, encapsulated within the ABM.

The abductive knowledge base is represented by an abductive logic program, which is not necessarily expressed in the form required by the KM, as opposed

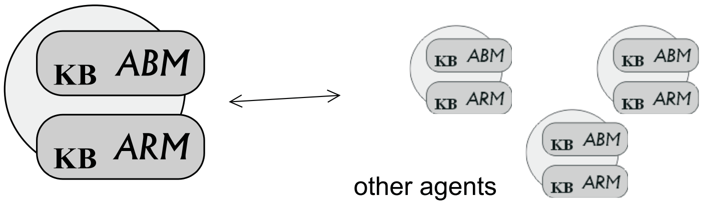

Figure 3.4: The structure of an *ALIAS* agent

to the architecture for the DAA. It describes for each agent its local knowledge and is used to perform abductive reasoning, e.g., for planning towards the achievement of the agent's goals or for explaining the agents' observations. The behavior knowledge base is a set of clauses in the *LAILA* language (see Chapter 4), used to describe the desired actions and interactions of the agent with its environment. In particular, the social behavior of each agent is expressed within the *ABM*, by means of collaborative/competitive queries submitted to other agents. Sometimes the agent's behavior may require the abductive explanation of an observation: this situation requires an interaction between local *ABM* and *ARM*, in order to locally start an abductive computation.

In this framework, each ALIAS multi-agent application is represented by a set of abductive agents, each modeled by its knowledge bases (located respectively at the *ARM*-level and at the *ABM*-level). I assume that agents can be grouped within *bunches*, which are dynamically constructed according to the requirements of interaction between agents, with the purpose, for instance, of finding the explanation for a given observation in a collaborative way within a bunch. As we will better see in the following, and especially in Chapter 5, abductive reasoning can be used in a multi-agent setting to produce negotiation dialogues, and solve problems such as that of resource allocation. In order to do that, it is required that agent coordinate their reasoning activity, and exchange information, as ALIAS agents do when they exchange consistent sets of hypotheses. I will start and introduce now the problem of agent negotiation, putting it in the context of abductive reasoning agents. The example shown below is inspired to [78].

Example 4 Let us consider three agents, say a, b, and c. Let us suppose that a is missing a resource, and in particular a *nail* that she needs in order to hang a picture to the wall. Then a might decide to request the nail to b. As we will see, the abductive mechanism can be used to implement agent dialogues, and in this case to produce a reply to the request. In fact, this request can be seen as the abductive explanation of a query, submitted by a to b, within b's own local knowledge base. b's reply will be enclosed in a set of possible hypotheses. It is up to a to decide whether such hypotheses are acceptable or not. Agent b, for instance, could try to sell a the nail for too much, asking 10 for a nail that – according to a's knowledge base – is worth at most 8.

It could also be the case that b implements the following policy in giving her resources: when asked the nail, b both tries and see if she has it and can give it, *and* asks c for it. This could be seen as a competitive process, where b tries to return a suitable set of hypotheses, possibly choosing among several alternatives (the one produced by c and the other one produced by herself).

If we implement this example within ALIAS, the ABM module will contain all the behavior rules, such as *'if I need a nail then I will ask it to b'*, or *'if I am requested a nail then I will try to obtain it either from my set of resources or from c'*, and the ARM will contain the knowledge to be used locally, such as *'I am agent b'*, *'I have a nail'*, *'the cost of a nail is not greater than 8'*, and so on.

3.3.3 Abductive reasoning in ALIAS

In this last architecture, we also have bunches, as before, but bunches are dynamically created, and the "life" of a bunch is no longer than that of the computation that required its creation. Again, the abductive explanation of a goal within a bunch of agents is a set of abduced hypotheses, agreed by all agents in the bunch.

In order to perform abduction in a multi-agent environment we need to introduce some mechanism to support agent bunches, local abduction and global consistency checks. As far as local abduction is concerned, I adopted the Kakas-Mancarella proof procedure. In ALIAS, I adopt a two-phase algorithm, which first collects the results of local[4] abduction processes running the KM, and then, depending on the chosen coordination policy, checks their consistency, in order to find a (possibly larger) set of hypotheses which is consistent with all the constraints present in the knowledge bases of the agents of the bunch. Thus, the first step involves a local abductive derivation, while the second one involves a consistency derivation within a group of agents. An operational semantics for the notion of consistency derivation in a multi-agent context is given in 4.3.

Despite the choice to use the KM for agent abductive reasoning, the ALIAS architecture is not strictly bound to it. The same high-level features of the system could exploit different abduction algorithms. In particular, the major drawback of the current approach is that it applies only to ground predicates, thus limiting the real exploitation of the system. Therefore, in future work, in order to test the system in real applications, I intend to experiment with other abductive proof procedures, such as *SLDNFA* [33] or that adopted in the ACLP framework [57, 1], a constraint-based extension of the Kakas-Mancarella which makes use of the ECL^{i}PS^e framework to allow the treatment of non-ground predicates.

3.3.4 Abduction and dialogue: an introductory example

As already mentioned, the abductive mechanism can be effectively adopted to implement agent dialogue. Let us consider a dialogue as a sequence of *dialogue moves*, i.e., requests, proposals, etc. that agents alternatively make to

[4]In this context, *local* means *performed within the agent that is demonstrating the query.*

one another (see Chapter 5 for a formal definition of dialogue and dialogue move). For the sake of simplicity, the time is left implicit. In a dialogue for obtaining a resource, let the predicates representing communication acts, such as $req(Sender, Recipient, Resource)$ and $sell(Seller, Buyer, Item, Price)$, be abducible predicates. The first one represents a request made by agent $Sender$ and addressed to $Receiver$, aimed to obtaining a $Resource$; the second one represents the offer of an $Item$ at a certain $Price$, the offer being made by agent $Seller$ to agent $Buyer$. Now, we could use the integrity constraints as a tool to express how we want an agent to reply to a certain move, under certain conditions. Let us consider again Example 4. Let agent b's ARM contain the following integrity constraint:

`% IC b:  offer a resource to a friend`
$\leftarrow req(X, b, R),\ have(R),\ friend(X),\ not\ sell(b, X, R, 0).$

Namely, if requested a resource by a friend, agent b will give it to her for free. By an operational point of view, if a requests a nail, the goal to be proven by b becomes $req(a, b, nail)$. As this is abducible, it will be assumed true and checked against the integrity constraints. These, together with the facts $have(nail)$ and $friend(a)$, will produce the abducible $sell(b, a, nail, 0)$ that is used to express that a resource is offered for free.

Let us consider now a's integrity constraint, used to decide whether a proposed price for a resource is to be accepted or not:

`% IC a:  do not over-pay a resource`
$\leftarrow sell(\_, a, R, P),\ max\_price(R, MP),\ P > MP.$

In this case, if $sell(c, a, nail, 10)$ is a hypothesis made by an agent, say c, telling that she will sell the nail to a for 10, and if in a's abductive program there is the fact $max\_price(nail, 8)$, such hypothesis will be rejected because it would produce an inconsistency within a.

In the next chapter I will formally define the LAILA language, that allows to map such problems into declarative sentences that become part of the ABM KB of agents. I would like to conclude this chapter by briefly describing an implementation of the system with static bunches, supporting rambling agents, that has been subject for experimentation.

3.4 An implementation of ALIAS with static bunches

In this section, I present a distributed implementation of ALIAS with static bunches, which has been published in [17]. The implementation has been obtained by using Java and *Amzi!* Prolog [4]. Java is used to implement the framework of the system, and in particular for supporting the communication between agents, while Prolog is used to implement both abductive and consistency derivation local mechanisms. With reference to the terminology adopted

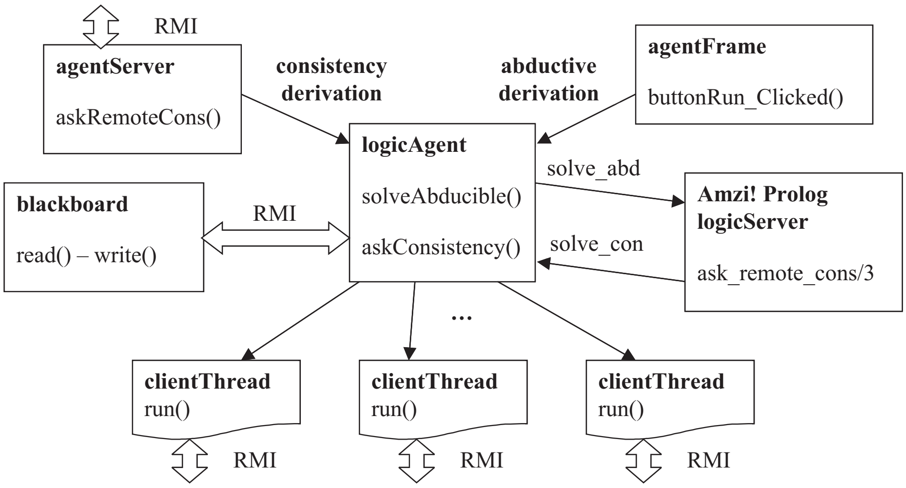

Figure 3.5: The implementation scheme of a single agent.

in Figure 3.1, Java is adopted to implement the *User Interface* and the *Coordination Module*, Prolog to implement the *Abductive Reasoning Module*.

The implementation scheme of a single agent is shown in Figure 3.5, where each block represents a software module. The user interacts with the system through the *User Interface* by invoking the methods provided by the class *agentFrame*. In particular, it is possible to start an abductive derivation process by specifying the goal to prove. Moreover, a blackboard is used to store the current list of abducibles. In this way, it is possible to separate shared *dynamic* knowledge (the list of abducibles which is available from the blackboard and can be updated at run-time by the agents) from agent-specific local knowledge base, which is statically specified.

Class *blackboard*, which implements the tuple space, handles the current list of abducibles and basically provides the access to the global knowledge. For a better comprehension of the relationships among the classes, let us consider the case of a user that asks a query to an agent through the *User interface*. Each agent of the bunch has an instance of class *agentServer* listening to an *IP address:port* waiting for remote requests. Then, the *User Interface* creates a *logicAgent* which is asked to demonstrate the goal (method *solveAbducible()*). First of all, *logicAgent* invokes the remote method *read()* of the blackboard in order to obtain the list of abducibles, Δ. As it receives an answer, it gives control to a Prolog engine, instantiating an object of class *logicServer*. During the *logicServer* initialization phase, it loads its knowledge base and its interface with Java. Then it starts the abductive derivation for the query. If no abducible is encountered during the abductive derivation phase, the *logicServer* returns control to the *logicAgent*, which displays the result. In that case, no coordination protocol among agents is required since the computation is performed locally. In some other cases, instead, the Prolog engine could attempt to abduce a hypothesis h. The consistency of the new abducible has to be checked by all the agents of the bunch. Therefore, Prolog returns the control to Java (*ask_remote_cons/3*), which creates one thread for each other agent in the bunch (*clientThread*) for communication purposes. Each thread sends the consistency request (by means of method *askRemoteCons()* of *agentServer*) to a specific remote agent. The remote *agentServer* receiving the request starts a *consistency derivation* for h, for which a new Prolog engine is dynamically created. It is worth to notice that the consistency derivation could start nested abductive and consistency derivations. After a remote consistency check returns the result (which will be positive iif all of the *clientThreads* returned *true*) the initial Prolog engine proceeds in its abductive derivation. This process of asking remote consistency might be invoked several times before a goal is demonstrated. As *logicServer* returns control to the *logicAgent*, it invokes method *write()* of object *blackboard* in order to store in the blackboard the current list of abducibles.

The source code, whose latest version is available at [2], has been written in Java 1.2 and imports class *logicSever* of *Amzi!* Prolog [4] in order to interface with the Prolog engine. *Amzi!* has been chosen among the other Prolog interpreters since it allows multiple instances of class *logicServer* to be created at the same time inside a single Java process. In fact, for the sake of deadlock avoidance, the *agentServer* which answers to remote requests must create a new instance of class *logicAgent* any time the consistency of a new abducible has to be proved. As the tests demonstrate, the sequence of operations devoted to creating an object *logicAgent* is responsible for most of the overhead.

3.4.1 Implementing rambling agents

The implementation of ALIAS for static bunches can be easily extended to the case of rambling agents. In fact, the prototype that is available on the ALIAS Home Page [2] also supports rambling agents. I would like to add here a comment on the protocol for entering a bunch proposed in Section 3.2. Since the protocol is made of two distinct phases, the acceptance of an incoming agent x in the bunch depends on the first solution proposed by x in the first phase (that called Δ_1 in Section 3.2). In general, x could produce more than one abductive explanation of Δ. Therefore, a backtracking mechanism is needed to repeat the two phases with distinct values of Δ_1, in case the first one cannot be accepted by the bunch.

Another brief comment: in the implemented system, when an agent wants to log in, no other distributed abduction algorithms may run concurrently, in order to ensure the coherence of the whole knowledge. By an implementative point of view, as an agent asks the bunch (i.e., the blackboard) for being accepted, the blackboard awaits all the running computations to terminate until it can lock itself, and prevents other distributed abductive computations from starting, until the login protocol is finished. At the end of the login protocol, the login delta is stored in the blackboard.

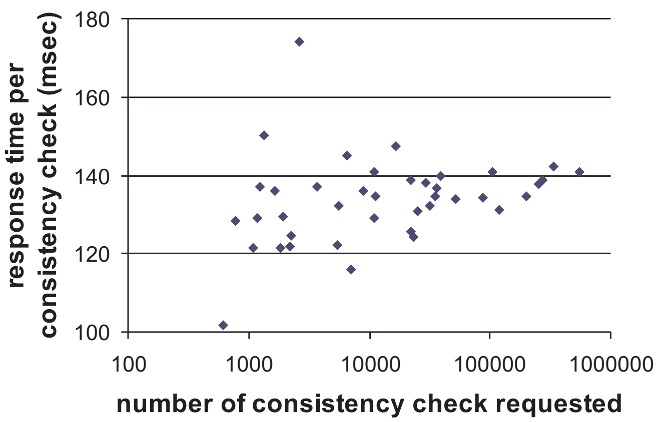

Figure 3.6: Average consistency check cost

3.5 Experimental results

The prototype has been tested in order to trace the behavior of the protocol when specific parameters change. The Prolog knowledge bases consist in general of a certain number of goals to prove, a set of rules organized in a hierarchy to form an AND/OR tree, a set of integrity constraints and a list of abducibles. The experiments included bunches from two to four agents and with different depths of the program (depth is intended to be the maximum number of nodes between a goal and the leaves, constituted by rules having abducibles in the body). Other parameters are the number of atoms, of abducibles, of integrity constraints, and the average length of the body of a clause.

From the experimental results emerged that there is evidence of a dependency between the number of consistency checks (c) requested by a computation, and the response time (t).

Figure 3.6 reports t/c, calculated for different values of c. From these results we can see that t/c remains almost constant. We can say that in this architecture t depends linearly on c. A closer analysis of the single consistency request cost allows us to focus on two different sources of overhead. The *communication overhead* is the cost of pure message exchange. The *Prolog engine load time* is the time spent by the *agentServer* creating a thread and loading a Prolog engine. In order measure these sources of overhead, a bunch composed of two agents has been considered, the former running a standard program, the latter playing the role of an *agentServer* having no computation to do but a list of abducibles. Prolog engine load time resulted responsible for 82% of the whole response time, while communication overhead is of 17%.

The performance results show that the implementation can be improved and optimized. In particular, one could consider to provide the *agentServer* with a pool of threads ready to answer to remote queries, in order to reduce the overhead produced by both thread creation and Prolog engine management. Different Prolog engines, for instance some lighter interpreter with a shorter load-time, could improve the system's perfomance. Another way is to reduce the number of consistency check requests itself, for instance by indexing the broadcast and implementing a consistency derivation driven by the characteristics of each agent constraints (multicast in place of broadcast).

The next chapter is devoted to describing LAILA and its features, and to presenting some results. There, I will also discuss an implementation of ALIAS that incorporates LAILA and supports dynamic agent bunches.

Chapter 4

A language to coordinate the reasoning of agents

In this chapter I will describe a language that can used to express the coordination of the reasoning activity of multiple agents. It is based on ALIAS, and its semantics is defined for the case of agents capable of abductive reasoning. In particular, in the first part of the chapter, I will formally define the syntax of LAILA, and its semantics. In the last part, I abstract away from the agent reasoning technique, and I will not only consider abductive reasoning agents, but rather refer to a generic agent system where agents play the role of service providers, that provide services under certain conditions. Integrity constraints are used, for example, to ensure that such conditions do not conflict with each other. A concrete scenario that I present is that of policy management in a computer system.

The relationship between this architecture and that based on ALIAS emerges from the operational semantics defined. In particular, in the consistency check phase it is clear how the link with abductive reasoning that characterizes the approach of LAILA is lost, and there is no abductive logic programs where definitions interact with integrity constraints, but constraints are kept separate from the agent code, that can be written in any programming language. If we want to frame ALIAS in this new perspective, we can consider the service provided by agents as answers to queries made on their own local ALP. Based on this more general architecture, I prove some correctness and completeness results, which also apply to the previous architecture, but that turn out to be independent on the kind of local computation or service that agents provide.

4.1 Reasoning and coordination

In a multi-agent setting, we can equip each abductive agent with a distinct abductive logic program. Each agent may thus reason on its own knowledge base and find abductive explanations to goals in the way explained before. However, multi-agent systems offer more than single reasoning entities. We saw in the previous chapter an algorithm that maps the abductive explanation of a goal within a bunch of agents into a distributed process that returns a set of abduced hypotheses, that in some cases could require a collective agreement by all agents in the bunch.

In this chapter I will introduce a language that can be used to issue and coordinate multiple abductive reasoning procedures in a society of logic agents [18]. This language allows to model agent actions and interactions in a logic programming style. In particular I will focus on agent social behavior, and especially on how each agent can request proofs of goals to other agents in the system. For this purpose, I introduce two composition operators: the collaborative operator (&) and the competitive operator (;) that can be used by each agent to formulate and coordinate abductive queries to other (set of) agents. The language provides also a communication operator (>) that is used to submit queries to other agents, and a *down-reflection* (↓) operator to trigger a local abductive derivation.

Before moving on to define the syntax and the operational semantics of the language, I would like to spend a few lines about the terminology used when I talk about "coordination of agent reasoning". Coordination is one of the main research areas in multi-agent systems. Coordination-based methods aim at providing a clean separation between individual agents and their interaction within the overall system organization, and also at making large applications more tractable in general. Most modern coordination infra-structures are based on the idea of a shared environment where software components may post the knowledge they want to exchange with one or more other components. This idea has led to the development of Linda [49] and to its extension to more expressive coordination languages and models. Integrated programming environments such as those used by Java and Prolog developers are often provided with libraries that implement Linda-like primitives [94]. In this way it is possible to engineer agent coordination spaces in a data-driven manner rather than in an often more complex, and less open, control-driven style.

The understanding of coordination, in this work, is different from that above of most multi-agent researchers. When I use the expressions "coordination patterns", "coordination of agent reasoning", and so on, I do not refer to tuple spaces, but to the coordination that is needed at the higher level of agent reasoning. Of course, we can refer to a coordination model such as a blackboard implementing a tuple space, as it has been done in the actual implementation of ALIAS for static bunches depicted in Section 3.4, but this approach is independent of the underlying coordination model adopted at that level.

In particular, the approach adopted to realize the agent coordination mechanisms is data-driven, in assuming that agents communicate by means of one or more tuple spaces accessed via Linda-like primitives. This form of coordination supports the implementation of a higher level language for the coordination of agent reasoning, LAILA. It is a goal-driven high level declarative language that allows a clean separation between the agent internal knowledge and reasoning, and its behavior with respect to other agents. Thus, LAILA allows the programmer to concentrate on the declarative aspect of distributed reasoning.

4.2 The syntax of LAILA

Let us consider a system composed by multiple agents. Each agent encapsulates a LAILA program describing its behavior. Let $\mathcal{V}$ be the vocabulary of the language, based upon logic programming. The set $\{\ \&\ ,\ ;\ ,\ >\ ,\ \downarrow\ \}$ of operators is added to the standard logic programming vocabulary [75], where:

- & is the *collaborative* coordination operator;
- ; is the *competitive* coordination operator;
- $>$ is the *communication* operator;
- $\downarrow$ is the *down-reflection* operator;

A *LAILA program* is a set of clauses. A *LAILA clause* (*L-Clause* for short) is defined as follows by a BNF grammar:

L-Clause	::=	*Atom* $:-$ *L-Body.*
L-Body	::=	*L-Expr* , *L-Body* $\mid$ *L-Expr*
L-Expr	::=	*L-Fact* & *L-Expr* $\mid$ *L-Fact* ; *L-Expr* $\mid$ *L-Fact*
L-Fact	::=	$\downarrow$ *LiteralList* $\mid$ *Agent* $>$ *LiteralList* $\mid$
		Literal $\mid$ (*L-Body*)
LiteralList	::=	*Literal* $\mid$ (*Literals*)
Literals	::=	*Literal* , *Literals* $\mid$ *Literal*
Agent	::=	*Atom*

I already defined in Section 2.1 the syntax of atoms, literals, and terms. *L-Body*, *L-Expr* and *L-Fact* respectively stand for *LAILA Body*, *LAILA Expression* and *LAILA Factor.* A computation can be started by a *Query*, defined as *L-Query* ::= ? *L-Body.* To clarify the sense of *L-Clauses*, let us consider, for instance, the following LAILA competitive query, formulated by agent A_0:

$$?\ \downarrow \texttt{G1}\ ;\ \texttt{A1} > \texttt{G2}$$

It means that A_0 must either perform a local abductive derivation for the goal G_1, *or* ask agent A_1 to demonstrate goal G_2. In case the local derivation fails the system will have to wait for A_1's answer, and vice versa. In case both fail, the whole query fails.

Let us consider, now, this collaborative query issued by agent A_0:

$$?\ \texttt{A1} > \texttt{G3}\ \&\ \texttt{A2} > \texttt{G4}$$

It means that agent A_0 asks agent A_1 to prove G_3 and A_2 to prove goal G_4; the result is obtained by merging in a unique consistent set of abducibles the answers respectively produced by A_1 and A_2. In case any fails, the whole query fails.

4.3 The semantics of LAILA

For the sake of simplicity, the following inference rules refer to the case of propositional LAILA programs. In the following, $\mathcal{U}$ is the universe of agents, and $A_0, \ldots, A_n$ denote agents in the system, i.e., $\{A_0, \ldots, A_n\} \subset \mathcal{U}$. The entailment symbol adopted is $\vdash_\delta^B$, whose superscript denotes the bunch B of agents involved in the computation, and whose subscript the set of assumed hypotheses δ, both of which are output parameters of the derivation process. Given $A \vdash_\delta^B F$, F is the *formula* to prove (a *formula* is in general a *L-Body*), and A is the program used for the derivation. Such program is an agent's ABM: for the sake of simplicity, and with abuse of notation, I will write the name of the agent instead of its ABM. Therefore, $A \vdash_\delta^B F$ means that the formula F is proven within A's ABM, producing as an output a set of hypotheses δ and a bunch B. Finally, I will adopt the following notation for the abductive and consistency derivations:

- $A \overset{abd}{\vdash}_\delta G$, where G is a list of literals, and $A \in \{A_0, \ldots, A_n\}$ is an agent enclosing in its ARM an abductive logic program $\langle P, \mathcal{A}, IC \rangle$ denotes the local abductive proof ("entailment") for the conjunction of all literals in G, performed by A within its ARM.

- $B \overset{cons}{\vdash}_\delta \Delta$, where Δ is a set of abducibles, denotes the "consistency check" of the conjunction of all atoms in Δ, with respect to the integrity constraints of all agents in bunch B; in particular, let $S \subseteq \{1, \ldots, n\}$. Given a bunch B of agents $\{A_i | i \in S\}$, each enclosing an abductive logic program $\langle P_i, \mathcal{A}, IC_i \rangle$, the declarative semantics of $B \overset{cons}{\vdash}_\delta \Delta$ is defined as follows:[1]

$$\begin{cases} \forall A_i \in B \quad P_i \cup \delta \quad \textit{“satisfies”} \ \bigcup_{j \in S} IC_j \\ \Delta \subseteq \delta \subseteq \mathcal{A} \end{cases}$$

  We achieve in that a separation among the agents knowledge bases, ensuring the absence of conflicts between δ and each of their integrity constraints. Moreover, I would like to notice that, while in describing the declarative semantics, I refer – for the sake of simplicity – to the union of the agent integrity constraints, the actual implementation of the system does not require at all that the agents share any knowledge in that respect (see the consistency check operational semantics further on for details). Knowledge bases and therefore integrity constraints are kept separate, as we would expect in a multi-agent setting.

Let us describe now the operational behavior of the system, by giving the inference rules of the LAILA operators.

[1] As in Section 2.1, I am abstracting away from the notion of integrity constraint "satisfaction". Different procedures/semantics will adopt different such notions.

Definition 8 (*down reflection formula*)

$$\frac{A \overset{abd}{\vdash}_{\delta} G}{A \vdash^{A}_{\delta} \ \downarrow G}$$

where A is an agent, $A \in \{A_0, ..., A_n\}$, and G a goal or *query* (i.e., a list of literals). The goal $\downarrow G$ triggers a local abductive derivation for G.

Definition 9 (*communication formula*)

$$\frac{Y \vdash^{B}_{\delta_1} G \quad \wedge \quad B \cup X \overset{cons}{\vdash}_{\delta} \ \delta_1}{X \vdash^{B \cup X}_{\delta} \ Y > G}$$

where X and Y are two distinct agents, $X, Y \in \{A_0, ..., A_n\}, X \neq Y$.

Therefore, the communication formula $Y > G$ used by X to ask Y to solve G requires a consistency check within the bunch composed by X and by those agents that participated in the derivation of G. Clearly, this bunch will at least include X and Y. This makes the communication operator more than just a simple call to another agent's knowledge base and reasoning module: the answer possibly returned back from Y to X is put in the form of a set of abducibles, δ_1, and needs to be agreed upon and possibly modified (enlarged) by a whole bunch of agents, through a consistency step.

Definition 10 (*competitive formula*)

$$\frac{A \vdash^{B}_{\delta} f}{A \vdash^{B}_{\delta} f; F}$$

$$\frac{A \vdash^{B}_{\delta} F}{A \vdash^{B}_{\delta} f; F}$$

where A is an agent, $A \in \{A_0, ..., A_n\}$, f is a *LAILA factor*, F is a *LAILA expression* and B is a bunch of agents, $B \subseteq \{A_0, ..., A_n\}$. The competitive formula results in a non-deterministic choice of one inference rule between the two listed above.[2]

[2] As it is, the competitive operator is equivalent to a disjunction. The reason why it is explicitly included in LAILA is that it allows for further extensions aimed to letting the choice be deterministic, e.g., by the introduction of a (partial) ordering function mapping from the cartesian product $\mathcal{A}^2 \times \mathcal{U}^2$ to a real range of numbers $R \subseteq \mathbb{R}$. Such function, used to choose among the two possible inference rules, would necessarily be domain dependent and does not seem to be of much interest for the general case. Therefore, I omitted it here in order to let the competitive operator have a simpler notation.

Definition 11 (*collaborative formula*)

$$\frac{A \vdash^{B'}_{\delta'} f \quad \wedge \quad A \vdash^{B''}_{\delta''} F \quad \wedge \quad B' \cup B'' \overset{cons}{\vdash}_{\delta} \quad \delta' \cup \delta''}{A \vdash^{B' \cup B''}_{\delta} f\&F}$$

where A is an agent, $A \in \{A_0, ..., A_n\}$, f is a *LAILA factor*, F is a *LAILA expression* and B, B' and B'' are three possibly distinct bunches of agents, $B, B', B'' \subseteq \{A_0, ..., A_n\}$.

The semantics of goal formulas is described by the following inference rules[3,4]:

Definition 12 (*goal formula*)

$$\frac{A \vdash^{B'}_{\delta'} h \quad \wedge \quad A \vdash^{B''}_{\delta''} G}{A \vdash^{B' \cup B''}_{\delta' \cup \delta''} \quad h, G}$$

$$\frac{\exists\ h : -G \ \in abm(A) \quad \wedge \quad A \vdash^{B}_{\delta} G}{A \vdash^{B}_{\delta} \quad h}$$

$$\frac{}{A \vdash^{A}_{\emptyset} \quad \texttt{true}}$$

where A is an agent, $A \in \{A_0, ..., A_n\}$, h is a literal, G is a *LAILA body* and B, B' and B'' are bunches of agents, $B, B', B'' \subseteq \{A_0, ..., A_n\}$. With $abm(A)$ I mean A's ABM.

Finally, while I refer to [60] for an operational semantics for the abductive derivation, I describe the semantics of the consistency check through the following inference rules:

Definition 13 (*consistency check*)

$$\frac{\forall A_i \in B \quad A_i \overset{abd}{\vdash}_{\delta_i} \delta \quad \wedge \quad \delta \subset \bigcup_{A_j \in B} \delta_j \quad \wedge \quad B \overset{cons}{\vdash}_{\delta'} \quad \bigcup_{A_i \in B} \delta_i}{B \overset{cons}{\vdash}_{\delta'} \quad \delta}$$

$$\frac{\forall A_i \in B \quad A_i \overset{abd}{\vdash}_{\delta} \delta}{B \overset{cons}{\vdash}_{\delta} \quad \delta}$$

[3]Note that here, with an abuse of notation, h, G is considered to be equivalent to (h, G).

[4]Here, the sequential operator (the comma) does not require any consistency check between the two sets of abducibles δ_1 and δ_2. This is due to the fact that at the level of LAILA programs, there is no notion of integrity constraints, that are only part of the ARM abductive programs.

where A_i, A_j are agents, $A_i, A_j \in \{A_0, ..., A_n\}$, h is a literal, G is a *LAILA body* and B is a bunch of agents, $B \subseteq \{A_0, ..., A_n\}$.

Therefore, the consistency check of δ is first performed individually by every single agent via an abductive derivation, which could result in an enlargement of δ (in particular, this happens if exists a δ_i such that $\delta \subset \delta_i$). In case no agent raises new hypotheses, i.e., for all $A_i \in B$, $\delta \equiv \delta_i$, then $\delta \subset \bigcup_{A_j \in B} \delta_j$ is not true any more and the consistency check terminates, otherwise the union of the δ_i has to be checked again within the bunch.

Finally, it is possible to introduce the definition of a successful top-down derivation.

Definition 14 (*successful top-down derivation*)
Let A be an agent and F a *LAILA body.* A *successful top-down derivation* for F in A's ABM, with δ a set of abduced hypotheses and B the bunch of agents dynamically involved in it, can be traced in terms of a (possibly infinite) tree such that:

- The root node is labeled by $A \vdash_\delta^B F$, where A is an agent;
- The internal nodes are derived by using, backwards, the inference rules defined above;
- All the leaves are labeled by the empty formula, or represent a successful abductive/consistency derivation.

For instance, we have seen how a collaborative formula $A \vdash_\delta^B \quad f \ \& \ F$ develops into three branches (sub-trees), one for demonstrating *f*, another one for F and a last one for the consistency check. Similarly, the communication formula produces two sub-trees, while the down reflection and the competitive formulas produce only one branch, and so on.

As a final consideration, I would like to stress that the semantics given to the consistency check is different from the one that we would have if we joined all the programs together.

Let us consider the example shown in Figure 4.1, which describes the knowledge bases of two agents, A_0 and A_1. r, g, q, and all the negated literals are the abducibles.

If A_0 starts form the query $\mathtt{A_1} > \mathtt{k}$, triggering A_1's proof of $(\downarrow \mathtt{p}, \downarrow \mathtt{s})$, the consistency check, run on separate programs, according to the given operational semantics of LAILA, will fail.

A different result would be obtained, instead, if we put the two programs together, since the query $(\downarrow \mathtt{p}, \downarrow \mathtt{s})$ would return successfully. In particular, it would return $\delta = \{$ `r, not q, g` $\}$.

The inference steps made are the following: on running the `query`, A_0 asks A_1 to prove, within its ABM, the goal `k`. In the given semantics of the *communication* operator, after A_1 proves `k` on her own, and possibly produces a δ,

A_0)	$\mathcal{ABM}$ $\texttt{query} :- \texttt{A}_1 > \texttt{k}.$ ——— $\mathcal{ARM}$ $\texttt{f} \leftarrow \texttt{not r}.$ $\leftarrow \texttt{g, not f}$	A_1)	$\mathcal{ABM}$ $\texttt{k} :- \downarrow \texttt{p}, \downarrow \texttt{s}.$ ——— $\mathcal{ARM}$ $\texttt{p} \leftarrow \texttt{r}.$ $\texttt{s} \leftarrow \texttt{g}.$ $\texttt{f} \leftarrow \texttt{not q}.$ $\leftarrow \texttt{r, q}.$

Figure 4.1: Failure of the consistency check made on separate programs

such δ's consistency needs to be checked within the bunch B consisting of both agents: $B = \{A_0, A_1\}$. In particular, the proof of `k` requires the proof of both `p` and `s` within A_1's ARM. The first one produces $\delta_p = \{\texttt{r}, \texttt{not q}\}$, the second one produces $\delta_s = \{\texttt{g}\}$. Putting them together, A_1 obtains $\delta = \{\texttt{r}, \texttt{not q}, \texttt{g}\}$. Eventually, A_0 is called to check if δ is consistent with its ARM. According to its integrity constraint, `g` and `not f` can not possibly be true at the same time, therefore, in order to accept δ, `f` must be proven true. Which is not possible within A_0's knowledge base, because $\texttt{r} \in \delta$, i.e., it is already assumed to be true. If we adopted a different semantics of consistency, and the two knowledge bases were put together, such consistency check would not fail to prove `f` to be true, based on the assumptions made so far ($\texttt{not q} \in \delta$), because an alternative definition of `f` would be available ($\texttt{f} \leftarrow \texttt{not q}.$).

The motivation why I do not want to put together all the agent knowledge bases lies upon the concept of multi-agents itself, which is in a way a distributed computing paradigm and fosters independence and autonomy. On the other hand, when an agent asks for collaboration and obtains an answer to a query, in the form of a set of abduced hypotheses, it would be desirable that such set was at least consistent with her own knowledge base, otherwise she would most likely fail to use it. An alternative to the proposed consistency check, for all that an agent universe has potentially infinitely many members, could be to jointly check on consistency all the agents' knowledge bases together before they interact, but this would lead to serious computational problems.

4.3.1 Back to the Abductive Web Travel Agency

In order to show how this framework differs from that described in the last chapter, based on static bunches plus rambling agents, let us consider again the example of the travel agency introduced in Section 3.2.4, and see how it can be modeled now with LAILA. I will also use this example to explain the implementation of the language in the next section.

As an example of use of the LAILA language I will discuss the case of a group

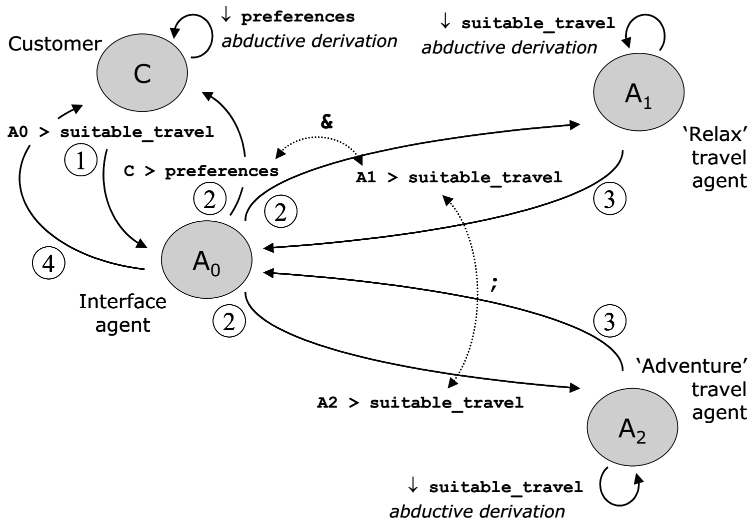

Figure 4.2: Tour operators and customers modeled like agents

of agents whose task is to act as travel agencies. I suppose that agents enclose the information regarding trips in a declarative database in the form of a logic program, and that they make use of customer profiles and web forms in order to present suitable offers. Therefore, on one side, the customer approaching a travel agency fills in a form on the web, specifying her preferences and needs, while on the other side the agent playing the role of the travel arrangements agency tries to find some travel which may suit the customer's needs and which are not in contrast with her constraints. When the agent needs to know something about the customer which is not contained in the customer profile nor in the data provided by the customer herself, it may abduce it.

Moreover, I assume that each agency is in turn composed of several agents, reflecting the structure of an organization that keeps information distributed. Also, when needed, agents from different agencies may interact, for example in order to retrieve some information that they are missing. In this setting, the LAILA language is used as the means to coordinate the agent reasoning activity.

In the following examples, I will consider the case of an agency composed of several agents, the first one providing the user interface and asking more specialized agents for specific services. I will consider the customer itself an agent. I will give one example of collaborative and one of competitive behavior. I will describe the agents by means of their abductive logic programs that I will refer to as *knowledge*, and their coordination, *behavioral knowledge*, the latter expressed in terms of LAILA phrases.

Let the set of abducible hypotheses be the following: {`category/1`, `class/1`, `location/1`, `destination/1`, `accommodation/1`, `travel/1`, `season/1`}, where terms are used to refer to the customer's requirements. For instance, `class(economy)` indicates that it is assumed that the customer would refuse travel that is not in economy class, while `not location(mountains)` indicates the hypothesis that the customer has not requested travel to mountains. The customer stores their assumptions in an initial δ_0 which will be used to formulate the LAILA queries.

Let us suppose the customer agent's knowledge, composed as follows:

```
← destination(rimini), season(high).
```

meaning that she does not want to go to Rimini in high season. Then, she issues the query ← `suitable_travel` to the interface agent, asking it to produce travel suiting the requirements of travelling in economy class and going to the beach, expressed in her knowledge base in the form of the clause
`preferences ← class(economy), location(beach)`.

Let us suppose the agents' behavioral knowledge expressed as follows:

Customer agent (C):
Query: `? A1 > suitable_travel &`
`preferences ← ↓ preferences.`
Interface agent (A0):
```
suitable_travel ← C > preferences & A1 > suitable_travel ;
                  C > preferences & A2 > suitable_travel
```
'Relax' travel agent (A1):
`suitable_travel ← ↓ suitable_travel.`
'Adventure' travel agent (A2):
`suitable_travel ← ↓ suitable_travel.`

Let the agents' abductive knowledge bases be the following:

Customer agent (C):
```
preferences ← location(beach), travel(flight).
← destination(rimini), season(high).
```
'Relax' travel agent (A1):
```
suitable_travel ← category(relax), destination(rimini),
    location(beach), travel(flight), season(high), class(economy),
    accommodation(bed_and_breakfast).
suitable_travel ← category(relax), destination(casablanca),
    location(beach), travel(flight), season(high), class(economy),
    accommodation(hotel).
suitable_travel ← category(relax), destination(sardinia),
    location(sea), travel(yacht), season(middle),
    class(business), accommodation(cabin).
```
'Adventure' travel agent (A2):
```
suitable_travel ← category(adventure), destination(amazon),
    location(river), travel(flight), class(economy),
    accommodation(camping).
suitable_travel ← category(adventure), destination(tibet),
    location(mountains), travel(flight), season(middle),
```

```
class(economy), accommodation(camping).
```

As shown in Figure 4.2, as the interface agent receives the query `?` *suitable_travel* from the customer (step 1), it asks the customer her preferences and redirects the query to two competing agents (step 2), each specialized in a certain kind of travel (relaxing ore more adventurous), each trying to produce suitable travel arrangements for the customer, in competition (step 3). The interface agent selects one answer, checks its consistency and forwards it to the customer (step 4), who in turn, with the interface agent, checks the consistency of the answer with her constraints – such as that of not wanting to go to Rimini in high season mentioned above – and decides whether to accept it or not. In case she decides not to accept it, a backtracking mechanism tries to provide her with more offers.

4.4 Towards a framework for agent negotiation

The next chapter will focus on agent negotiation as a concrete application domain of abductive reasoning in multi-agent systems. Here, I would like to anticipate some concepts in order to both introduce the issue of negotiation and to show how LAILA can be applied to it.

4.4.1 Nails and hammers

Let us consider once more the example introduced in Section 3.3.4, and see how LAILA can be used in the negotiation domain. we can now explicit the content of the three agents' $\mathcal{ABM}$ and $\mathcal{ARM}$[5], and see how agent a can obtain a nail and have the picture hung (see Figure 4.3).

The meaning of the predicates and their arguments is the following: `obtain(List)` is usually the initial query, that will return successful if the agent manages to obtain all the missing resources in the `List`. Such list of missing resources could be the output of a planning module, or could be provided by a planning agent, but at his stage I do not commit to a particular design choice to this respect, and for the sake of this example I will simply assume that such output is given.

`max_price(Item, Price)` represents the maximum price that an agent is willing to pay for a given `Item`; `cost(Item, Price)` is the price that a seller agent is going to ask for a given `Item`; `get(Buyer, Seller, Item)` represents a request of `Buyer` to obtain from `Seller` a certain `Item`; `req/3` and `sell/4` are dialogue moves that may become part of the abduced set of hypotheses, as we have seen in Section 2.1, and in particular `req/3` represents a goal for the abductive reasoning module. It is worth to notice that these hypotheses are the result of the agents' reasoning, and that they do not represent communicative acts. The way agents communicate is coded in the LAILA programs, contained in the agents' behaviour modules.

[5] Of course, I am giving here only those bits of information that are strictly needed in order to follow the example.

a) $\mathcal{ABM}$

```
obtain([R|Tail]) :-
    b > get(a, b, R) &
    obtain(Tail).
obtain([]) ← true.
```

$\mathcal{ARM}$

```
i_am(a).
have(picture).
have(hammer).
max_price(nail, 8).
← max_price(R, M),
  sell(_, a, R, M1),
  M1 > M.
```

b) $\mathcal{ABM}$

```
get(X, Y, R) :-
    ↓ req(X, Y, R) ;
    c > ( get(X, Y, R),
          get(Y, c, R) ).
```

$\mathcal{ARM}$

```
i_am(b).
friend(a).
have(nail).
← req(X, Y, R), i_am(Y)
  have(R), friend(X),
  friend(X),
  not sell(b, X, R, 0).
← sell(Z, Y, R, P), i_am(Y),
  req(X, Y, R), X ≠ Z,
  P1 is P * 1.2,
  not sell(Y, X, R, P1).
```

c) $\mathcal{ABM}$

```
get(X, Y, R) :-
    ↓ req(X, Y, R).
```

$\mathcal{ARM}$

```
i_am(c).
have(nail).
cost(nail, 5).
← req(X, Y, R), i_am(Y),
  have(R), cost(R, P),
  not sell(Y, X, R, P).
```

$\mathcal{A} = \{\texttt{req/3}, \texttt{sell/4}\} \cup$ *negated literals*

Figure 4.3: The negotiating agents' knowledge bases

Now, let us suppose as I were saying that agent a runs the $\mathcal{ABM}$ query ? $\texttt{obtain([nail])}$. As a consequence, a will ask b to prove the goal $\texttt{get(a, b, nail)}$, and b will competitively issue a local abductive derivation for $\texttt{req(a, b, nail)}$ and ask c to solve the goals $\texttt{get(a, b, nail)}, \texttt{get(b, c, nail)}$. The set of hypotheses returned by her local $\mathcal{ARM}$, say δ_b, is $\delta_b = \{\texttt{req(a, b, nail)}, \texttt{sell(b, a, nail, 0)}\}$, that is also consistent with a's knowledge base. The set of hypotheses returned by c is $\{\texttt{req(a, b, nail)}, \texttt{req(b, c, nail)}, \texttt{sell(c, b, nail, 5)}\}$, that in order to be consistent with b's integrity constraints will have to be enlarged and becomes $\delta'_c = \{\texttt{req(a, b, nail)}, \texttt{req(b, c, nail)}, \texttt{sell(c, b, nail, 5)}, \texttt{sell(b, a, nail, 6)}\}$. Both δ_b and δ'_c are acceptable by a. As a conclusion, a will either accept b's nail for free or pay 6 for c's nail. As I already said before, LAILA does not specify which one will be returned by the competitive operator, leaving this choice open. This is the case where the ordering function mentioned in Definition 3 (Competitive formula) could be useful, setting a preference in the order in which b will return the δs to a.

4.4.2 A two-agent dialogue framework

In [89], building on [3], the authors introduce a framework where negotiation is carried out by means of a distributed abduction process. Here, the agents try to achieve their goals by means of plans that may require some resources that they miss. In that case, the negotiation process aims at obtaining such resources. The authors represent the beliefs of agents as logic programs and integrity constraints, as in abductive logic programming, with the dialogue performatives treated as abducibles. The integrity constraints play the role of dialogue / negotiation protocols. In this section I first will give a brief description of the basic elements that are needed to build a dialogue, then I will show how ALIAS can effectively support the collaborative / competitive negotiation framework.

The dialogue framework introduced in [89] is composed of two agents sharing a *language* and a *protocol*, and having each a *knowledge representation*, a *reasoning mechanism*, and a *planner* capable of producing plans (i.e., sets of actions) for a given goal. The approach is totally independent of the planner.

- The *language* is a set of performatives, e.g., `request`, used to request a resource, such as an information, or a physical tool, `accept` and `refuse`, used to reply affirmatively or negatively to a request or proposal, and finally `challenge`, `justify`, and `promise`, used for proper negotiation, allowing the agents to share more information than just that attached to a request and a reply. An example of dialogue, adapted from [78], is the following, where agent a wants to hang a picture on the wall and plans to do it by using a nail and a hammer. Unfortunately, a does not have a nail and needs to negotiate with b to try to obtain it.

  Dialogue 1
  $\mathtt{tell}(\mathtt{a}, \mathtt{b}, \mathtt{request}(\mathtt{give}(\mathtt{nail})), 1)$
  $\mathtt{tell}(\mathtt{b}, \mathtt{a}, \mathtt{refuse}(\mathtt{request}(\mathtt{give}(\mathtt{nail}))), 2)$
  $\mathtt{tell}(\mathtt{a}, \mathtt{b}, \mathtt{challenge}(\mathtt{refuse}(\mathtt{request}(\mathtt{give}(\mathtt{nail})))), 3)$
  $\mathtt{tell}(\mathtt{b}, \mathtt{a}, \mathtt{justify}(\mathtt{refuse}(\mathtt{request}(\mathtt{give}(\mathtt{nail}))), \{\mathtt{not\ have}(\mathtt{nail})\}), 4)$

  In this example, a uses the `challenge` performative to ask the reason why b refuses to give the nail: agent b says that she does not have it. The dialogue could go on with other moves, depending on the agent abductive logic programs that produce them, as we will see next.

- The *protocol* used for negotiation is a set of constraints that must not be violated, and that describe the dialogue, the possible moves, and its termination.

- The *agent knowledge* is represented by the tuple $< \mathcal{B}, \mathcal{R}, \mathcal{I}, \mathcal{D}, \mathcal{G} >$, containing respectively $\mathcal{B}$*eliefs*, $\mathcal{R}$*esources*, $\mathcal{I}$*ntentions*, past $\mathcal{D}$*ialogue* moves, and of course $\mathcal{G}$*oal(s)*. An intention is a plan intended to be used to achieve a goal.

- The *reasoning* mechanism is that of *abduction*, and is used to generate a dialogue move starting from a knowledge base, including the past dialogue. The abductive logic program used is contained in the agent knowledge base, namely in $\mathcal{B}$. The proof-procedure adopted is not necessarily the same for both agents, but for the sake of clarity I will write the ICs as condition-action rules, as in the case of the if-and-only-if (IFF) abductive proof-procedure. This will put a stress on the dialogue move to be uttered (the 'action') in case certain 'conditions' (including a previous move from the counterpart) hold.

To start up, let us suppose that all agents are provided with the same set of integrity constraints. Let us write only one of them as an example:

```
% accept a request
```

$i\_am(X) \ \wedge \ tell(Agent, X, request(give(R))) \ \wedge \ have(R) \ \wedge \ not \ need(R) \Rightarrow tell(X, Agent, accept(request(give(R))))$

This rule models the behavior of a collaborative agent, that gives a resource that she has and does not need, in case she is request to give it. As already mentioned in the introduction, it is possible to rewrite such constraint and express it as a denial, as following:

```
:- i_am(X), tell(Agent,X,request(give(R))), have(R), not need(R),
   not tell(X,Agent,accept(request(give(R))))
```

Of course, more integrity constraints are needed to produce the example of dialogue 1. Also, In Chapter 5 I will define the exact semantics of predicates such as `need` and `have`, whose rough meaning I will consider by now intuitive.

Before moving on to the next part, I would like to stress a couple of points. First of all, the negotiation process, as in [88], is driven by the need for a specific resource for a specific plan. Second, the dialogue is intended to be between (only) two agents. Bearing such considerations in mind, let us see how this framework could be used in a multi-agent environment, where agents could ask for multiple resources and try different plans, if provided with an appropriate support for collaboration and competition.

4.4.3 Negotiating ALIAS agents

The problems addressed by multi-agent systems extend in many respects the situation where two agents negotiate to make a plan feasible, involving in general more than two agents, possibly provided with more than one plan for each goal, while the resources needed for each plan are not necessarily all owned by the same agent. Moreover, it is reasonable not to assume that the sets of integrity constraints modeling the dialogue are the same for all the agents[6], this last possibility leading to the risk of non-terminating dialogues, and other unpleasant effects.

[6]The assumption that all the agent constraints are the same, would also lead to computationally hard problems in terms of verification, and seems quite unlikely to be easily feasible.

In this setting, for a given goal to achieve, an agent will have to manage multiple dialogues, either in 'competition', if referring to alternative plans, or in 'collaboration', if aimed at obtaining all the resources needed to make a certain plan feasible. If we want to frame the dialogue in an argumentative abduction based system, it is clear that we need a mechanism to express and manage collaboration and competition between autonomous abductive reasoning systems. Here, consistency enforcement can be needed in order to guarantee that certain dialogue rules are not violated, and that some desired properties still hold, despite the possible difference in the agent dialogue rules and constraints. In such scenario, ALIAS comes in hand since it provides collaboration and competition patterns within a solid formal theory, that can be used to prove if such properties will hold or not.

Example 4, that I have been considering throughout the thesis, can be considered a first rough sketch of such application. Let us consider again a's knowledge base, as reported in Section 4.4.1. The predicate `obtain` was in charge of trying to get all the missing resources, needed for a certain plan. To that purpose, several communication formulas of the kind $\mathtt{X} > \mathtt{get}(\mathtt{a},\mathtt{X},\mathtt{R})$ were issued in collaboration, since all the resources (and not only part of them) must be obtained in order to make a plan executable.

The role of consistency checking has already been exemplified in the case of the communication formula, when a asks for a nail but refuses to spend more than 8 to buy it (Figure 4.3). But there is another, more general, use of the constraints that becomes important in an open, Multi-Agent, environment: that is, the implementation of the dialogue protocol and the verification that it is not being violated.

In 4.4.2 we assumed that two agents that are having a dialogue share the same set of integrity constraints. This is because in that way we are able to prove, for instance, that a certain kind of dialogue will terminate [89, 88]. When an agent is put in an open context, this assumption cannot be made any more. Each agents must check within herself that, no matter if the other agents have or not the same set of constraints, at least they produce compatible hypotheses with respect to their own programs, without violating the dialogue protocol. For instance, one of the properties of a dialogue as described in [89] is that of termination in a finite number of steps. This is possible because there are some constraints that rule the dialogue (e.g., it is not possible to tell the same thing twice at different times). This kind of constraints can be easily checked within a bunch of agents, that will accept the other agents' moves, i.e., abducibles, as soon as they do not violate the protocol.

4.5 An implementation of LAILA

LAILA has been implemented on the top of ALIAS. In order to add LAILA, ALIAS has been re-designed and its code re-written using Jinni [97]. This has allowed us to write the meta-interpreter LAILA in a subset of the Prolog language, extended with multi-threading and Linda-like coordination primitives,

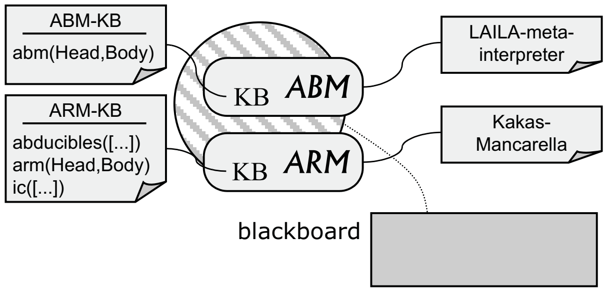

Figure 4.4: Prolog implementation of *ALIAS* agents' modules

and to reuse the implemented code for the KM proof-procedure. Agents enclose two different knowledge bases, one used for performing (local) abductive reasoning, and the other for expressing their social behavior. In this section, I will briefly describe the architecture of ALIAS and its implementation, focusing on the LAILA meta-interpreter.

The inner structure of each ALIAS agent, composed of two modules, has been already discussed in the previous chapter, and is shown in Figure 3.4. The abductive knowledge base is an abductive logic program used to perform abductive reasoning; the behavior knowledge base is a set of LAILA clauses that describe actions and interactions of the agent within the environment.

In this framework each ALIAS multi-agent application is represented by a set of abductive agents, each modeled by its knowledge bases (located respectively at ARM and ABM levels), and, in particular, interacting in accordance with rules enclosed at the upper level of their structure.

As I just mentioned, the current implementation of *ALIAS*, which supports LAILA, relies on Jinni [97], a lightweight, multi-threaded, logic programming language extended with Linda-like primitives, used as a development platform for building multi-agent applications based on logic agents. Although I am aware that a more direct and customized implementation should be more efficient than the current one, I chose Jinni mainly for its high-level and declarative features that allow us to achieve a rapid prototyping of the *ALIAS* system. The adoption of Jinni allows an effective qualitative leap in terms of code readability, homogeneity, easy maintenance, and expressiveness. In fact, unlike the previously described implementation for static bunches, both the reasoning and the coordination issues introduced by *ALIAS* are solved by means of logic clauses, leading to the development of a fully declarative design. Moreover, *ALIAS* is platform-independent, since Jinni is written in Java.

Jinni threads are controlled with tiny interpreters following a scripting language based on a subset of Prolog, while the synchronization among the threads is obtained through a blackboard via a set of Linda-like primitives.

As shown in Figure 4.4, an agent in *ALIAS* is implemented by two meta-interpreters, the first one supporting the execution of *LAILA* programs, the second supporting abductive reasoning, through the KM proof-procedure.

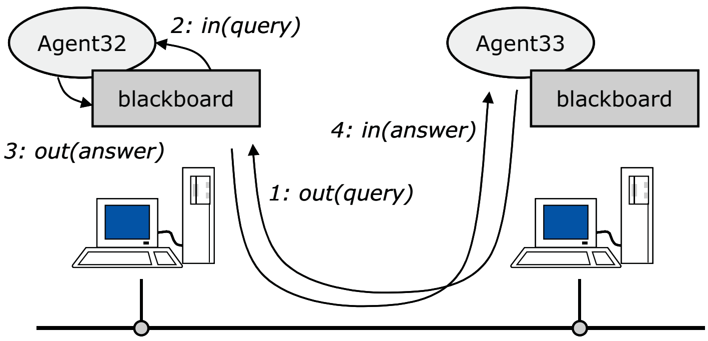

Figure 4.5: Networked *ALIAS* agents

ALIAS agents can run on different networked machines. Agents communicate through Linda-like primitives, by posting and consuming tuples in a shared blackboard space. To this purpose, each agent provides a blackboard as a communication space (see Figure 4.5), where it fetches the queries coming from the other agents, such as, for instance, the request for the demonstration of a given *LAILA* query. In particular, the ABM runs a background thread waiting for tuples of the form: `laila_query(MyName,LailaQuery, DeltaIn)`. As it fetches a tuple from the blackboard, it processes `LailaQuery` within its $\mathcal{ABM}$ knowledge base, and tries to demonstrate the goal `laila( LailaQuery, DeltaIn, DeltaOut )`, possibly obtaining a `DeltaOut`. Once it produces an answer, it puts it on the blackboard.

I decided to use the blackboard as a means to update the status of a goal demonstration, besides using it to trigger the query demonstration itself as we have just seen. In particular, as the demonstration of a query is started, a coordination tuple is put on the blackboard by the issuing agent. When an agent has some update about the query to publish, either because it produced a new answer or because it wants to change the status of the computation (e.g. if it wants the computation to terminate), it consumes the tuple (through an `in` operation), updates it and then posts the updated tuple on the blackboard again (through an `out` operation). This mechanism ensures conflict avoidance among the threads.

4.5.1 The LAILA meta-interpreter

I would like to discuss some code fragments to show how the *LAILA* meta-interpreter is structured[7]. I will take back the example presented in Section 4.3.1: therefore, in the following, I will refer to agent `C` as the customer agent, to agent `A0` as to the interface agent, and so on. Also, I will refer to the operational

[7] The code listed in this section is a simplified version of the actual LAILA meta-interpreter, showing a condensed version of the clause bodies; still it gives a flavor of the implementation of LAILA and should be sufficiently explanatory for the purposes of this work.

semantics introduced in 4.3, and observe how it maps into the actual Prolog implementation[8].

Let us start with the down-reflection operator. It is mapped in the following meta-interpreter clause:

```
% down reflection
laila(down(Goal),Delta) :-
  solve_ARM_goal(Goal,Delta).
```

Agent `C`'s ABM includes the LAILA clause `preferences` $\leftarrow$ $\downarrow$ `preferences`, which means that when agent `A0` will ask her for preferences about the travel, her ABM will map that query into the query for the $\mathcal{ARM}$ meta-interpreter `solve_ARM_goal(preferences,Delta)`, returning in `Delta` the abductive explanation of predicate `preferences`. Let agent `C`'s ARM again be that listed in Section 4.3.1 (the agent knowledge base contains facts about her identity and location in the internet as well): one solution to `solve_ARM_goal(preferences,Delta)` is therefore $\delta = \{$`location(beach), travel(flight)`$\}$.

The communication operator is mapped into the following clause:

```
% communication
laila(ask(Agent,Goal),Delta) :-
  label(Q),
  out(answer(Q,comm,[],0,[],0,N,[])),
  bg(solve_Comm(Q,Agent,Goal)),
  select_answer(Q,Delta1),
  consistency([<me>,Agent],Delta1,Delta)).
```

The `solve_Comm` predicate, called in background inside the `bg`, has the function of communicating to agent `Agent` the query `Goal`, posting a `laila_query` tuple on its blackboard. The `select_answer` retrieves one delta, whose consistency is checked by the `consistency` predicate.

In the example of 4.3.1, agent `A0` ABM includes the LAILA clause
`suitable_travel` $\leftarrow$ `C > preferences & A1 > suitable_travel ;`
`C > preferences & A2 > suitable_travel`, which contains several communication operators.

In particular, agent `A0` is not able to solve the query `suitable_travel` on its own, therefore it requires the intervention of other partners, including the customer agent herself. In order to process the communication operator `laila(ask(C,preferences),Delta)`, agent `A0`'s ABM meta-interpreter issues a background thread which will manage the communication with the partner

[8]The operational semantics introduced in Section 4.3 is used to define whether a δ is an acceptable explanation of a goal. In the implementation, though, we shall consider the δ as an output parameter. That results in a gap between semantics and implementation. For an alternative operational semantics that is closer to the implementation of the system, the reader could refer to [16].

agent (`C`) and will block on the blackboard, waiting for an answer (clause `select_answer`). After obtaining one, it performs a consistency step, as the operational semantics of the communication operator requires.

Now, let us consider the collaboration and competition cases. As far as the collaborative and competitive operators are concerned, both of them issue the background execution of both left and right branches (`solve_Formula` and `solve_SFormula`). The choice of the abductive set δ among those possibly produced is left to the `select_answer` clause, that can be customized inside the agent.

```
% competition
laila(or(G1,G2),Delta) :-
  label(Q),
  out(answer(Q,comp,[],0,[],0,N,[])),
  bg(solve_Formula(Q,left,G1)),
  bg(solve_Formula(Q,right,G2)),
  select_answer(Q,Delta).
```

According to the operational semantics we do not need a consistency check in the competitive case. The `select_answer` selects one delta among those produced by the two solving branches (operands) of the competitive operator, and may be customized within the agent to implement some selection policy. This operator is not implemented by means of two different clauses, in order not to leave control (and therefore the choice of the `Delta`) to Prolog. Therefore, if we want to have two parallel solving threads, we issue them both in one clause.

```
% collaboration
laila(and(G1,G2),Delta) :-
  label(Q),
  out(answer(Q,coll,[],0,[],0,N,[])),
  bg(solve_SFormula(Q,left,G1)),
  bg(solve_SFormula(Q,right,G2)),
  select_answer(Q,Delta1),
  bunch(and(G1,G2),Bunch),
  consistency(Bunch,Delta1,Delta).
```

While in the competitive case, the `select_answer` puts in `AbdOut1` either a δ_1 from the left branch or a δ_2 from the right branch of the query, in the competitive case a pair is considered (δ', δ''), and the bunch is created in order to check its consistency.

Once again, the clause `suitable_travel` $\leftarrow$ `C > preferences & A1 > suitable_travel ; C > preferences & A2 > suitable_travel` of agent `A0` contains both competition and collaboration operators. Some possible answers to `A0`'s query could be the set of abducibles {`category(relax)`, `destination(rimini)`, `location(beach)`, `travel(flight)`, `season(high)`, `class(economy)`, `accommodation(bed_and_breakfast)`} produced by agent

A1 and {category(adventure), destination(amazon), location(river), travel(flight), class(economy), accommodation(camping)} produced by agent A2. Agent A0 could decide to propose the first or the second answer, depending on some agency policy (most probably, the most expensive one). Should it propose the first one, though, the consistency check with agent C's integrity constraint $\leftarrow$ destination(rimini), season(high) would provoke failure, causing it to backtrack to another solution. Again, the backtracking mechanism is implemented inside the select_answer clause.

This brief discussion illustrates how LAILA allows a clear separation between the agent (abductive) knowledge bases and the higher behavior level, and how this feature can simplify the task of agent programming.

4.6 Beyond abduction: coordinating agents with constraints

In the past sections, a language has been described that can be used to coordinate the reasoning activity of multiple agents, capable of abductive reasoning. Now, I abstract away from the internal reasoning of agents. I will refer to an agent system where agents play the role of generic service providers, and where the service is not necessarily implemented by an abductive derivation.

The concept of multi-agent system introduced in Chapter 2 envisages societies of entities that are *sociable* and *autonomous*, and exhibit both a *reactive* and a *pro-active* behaviour, in that they should act in anticipation of future goals, while coping with the (generally unpredictable) changes of a dynamic environment. The metaphor of software components as agent societies is appealing for many reasons, that made it very popular, but there are some obstacles to its applications to problems such as software engineering. According to [53], 'there are two major drawbacks associated with the very essence of an agent-based approach: (1) the patterns and the outcomes of the interaction are inherently unpredictable; and (2) predicting the behaviour of the overall system based on its constituent components is extremely difficult (sometimes impossible) because of the strong possibility of emerging behaviour.'

This could be a drawback not only for software engineering. In fact, the need for making agents "predictable", and – for most applications – as deterministic as possible, is indeed in contrast with the concept itself of autonomy. Nonetheless, it is reasonable to believe that societies (of agents) can only exist as long as the individuals' autonomy does not represent a threat for the other individuals, and for the society in general. Therefore, the important counterpart of autonomy is represented by private constraints and public laws, that can make the agents, if not predictable, at least not colliding with each other needs and constraints. I will call 'safe' the execution of tasks in an agent system that does not violate any of the constraints in the system itself.

Much work has been done towards safety ensuring in agent systems, especially for inherently open systems such as mobile agent systems, where mobile agents could move to remote machines [47]. For instance, a typical problem in this area is, from the hosting system viewpoint, the problem of enforcing some authorization policy, and make sure that hosted external agents do not consume more resources than are allowed. From the mobile agent viewpoint, the problem is instead to ensure that she can have the right resources that she needs in order to carry out a certain task. In this thesis, I tackle the problem of ensuring that the execution of tasks in a constrained multi-agents setting is consistent with respect to its constraints. In order to make the discussion more concrete, I take inspiration from the area of distributed system policy management [26], using it as an example throughout the last part of the present chapter. Policies can be positive or negative, and are of different kinds: authorization policies which specify access control, obligation policies which tell the subject what he must do, refrain policies which define the actions that subjects must not perform on target objects, delegation policies which specify which actions subjects are allowed to delegate to others, and so on. Constraints can be specified to limit the applicability of all types of policy. In the example that I show, the tasks represent services, and are associated with abstract specifications that express conditions on such services. The constraints, embedded in the agents' code, may include – but are not limited to – policies on provided services, and limitations about the use and allocation of bounded resources.

Drawing inspiration from the work made with LAILA, I introduce a formalism that is used to express the way agents can coordinate the requests of services, and to verify that they do not collide with each other's conditions, in dynamic, possibly adaptive systems [14]. To this purpose, the communication, collaboration, and competition operators are re-defined in a way that is independent on the abductive reasoning. The idea is to map such definitions, in a concrete implementation, into a library of the language or languages used to encode the agents in the system. The operators provided by such library could be used to specify the need to obtain different services form diverse sources, in a *coordinated* fashion. For instance, an agent A_0 needs both a service s_1 from agent A_1 and a service s_2 from agent A_2: if either of the two cannot be provided, the other becomes useless. From a policy viewpoint, the two services could require conflicting authorizations (e.g., there is a policy stating that an agent cannot be provided with both service s_1 and s_2), and therefore it could be impossible to provide them both. If we want the system to work in accordance with such constraints, the agents should check the requests against any possible threat to its integrity that could arise from the execution of their tasks. The things are complicated by the fact that the agent system is inherently distributed, which introduces a semantic and computational problem, and that the number of *conditions* related to those services can grow bigger while constraints are checked. In fact, it may turn out that the set of conditions of service generated by the two original requests collides with other policies of the system. We need that an integrity ensuring mechanism be dynamic, since all the hypotheses on the external world are not necessarily known a priori by the system programmer, or programmers, and we need a protocol that is able to guarantee that a request or set of requests is allowed only if at execution time there is no constraint in

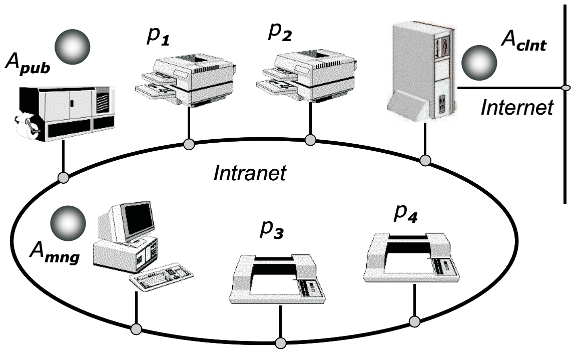

Figure 4.6: A local area network composed of several printers and servers, hosting three agents.

the system that is violated.

Once a formalism is defined, I give such formalism an operational semantics, in the form of inference rules which can be used as an abstract specification of an algorithm, that guarantees the correct execution of multiple, possibly conflicting, tasks. We see how such semantics is very close to that referring to abductive reasoning agents defined for LAILA. With respect to it, general results can be proven: in particular, I prove that, if the abstract machine underlying the agent code implements such operational semantics, all requests involving integrity constraints are allowed only if no constraint in the system is violated. In doing that, each agent performs consistency checks only within its own (private) set of constraints. That is, the consistency of the constraints is ensured in the whole system, without each agent needing to disclose its constraints to other agents.

4.7 Requesting services: a formalism

In this section, I introduce the formalism for expressing the way agents can coordinate the requests of services. The idea is to have a tool that is expressive enough to allow verification that they do not collide with each other's conditions. In doing this, I will draw inspiration from LAILA. As I said, I now abstract away from the specific reasoning capabilities and knowledge representation of the agents, and generalize the language to cope with service requirements and constraints in general. The only assumption that I make is that it is possible to describe the problem in question, in terms of constraints and conditions on the services, possibly expressed in the form of logic predicates.

An example in the printer management domain will clarify the idea to the reader that is not confident with logics. Let us consider a LAN, where there are several agents providing printing services, plus other agents acting on the behalf of clients that need such services. A possible situation is represented in Figure 4.6. The agents in the system are characterized for providing specific

Table 4.1: Printer authorizations

group	p_1	p_2	p_3	p_4
staff	YES	YES	NO	YES
students	NO	NO	YES	YES

services and for having certain integrity constraints. An example could be the following: A_{clnt} is a client agent that needs to print a document, *doc*. A_{prn} provides printing services, checking the client authorizations and scheduling the authorized jobs. A_{prn} manages several printers: the laser printers p_1 and p_2, and the ink-jet printers p_3 and p_4. We can suppose that it contains an authorization table, such as the following, where we imagine that agents are part of groups, and there are two groups, *students* and *staff*:

The content of Table 4.1 can be put in the form of constraints of the kind:

$$\leftarrow condition_1, condition_2, \ldots$$

meaning that *condition*$_1$, *condition*$_2$, ... can not be assumed all true at the same time. For instance, the first line could be translated into the constraint: $\leftarrow$ *staff*, p_3, where by *staff* I mean '*group* = *staff*', and by p_3 I mean '*printer* = p_3'. Therefore, A_{clnt} cannot possibly be both part of group *staff* and be authorized to print on p_3. Constraints can also include negated conditions, meaning obligations: $\leftarrow$ *students*, *not inkjet*, that is, students must print on ink-jet printers.

As far as the conditions initially returned by the functions representing services, I do not commit to any programming language or virtual machine implementing them. I will assume that such particular functions, possibly associated with interfaces that are public and known by other agents, return a set of conditions that are transmitted to the requester following a common ontology. For example, if A_{clnt} requests to print a document *doc*, calling A_{prn}'s service *auth_print*(*doc*), we can imagine that A_{prn}, based on her private system tables, recognizes A_{clnt} to be part of the staff, and therefore creates the list { *staff*, *laser* }. This means that *doc* can be printed on a *laser* printer, giving A_{clnt} the *staff* rights.

In addition to request of services, I want to model the possible combination of service requests in an agent system. Similarly to what I did with LAILA, I refer to two basic combination schemes, and define two composition operators: the collaborative operator (`&`) and the competitive operator (`;`). As before, a communication operator (`>`) let agents formulate and coordinate requests to other agents. Finally, since I do not refer to any particular reasoning mechanism, it is a call to a local computation, which is in general the authorization for a service, and not a local abductive derivation, that produces a list of conditions. Such call is preceded by the down reflection operator ($\downarrow$). In the following subsections I will describe the syntax and operational semantics of such operators.

4.7.1 Syntax

Before I proceed with the syntax, I define what I intend by agent and by agent system, in this perspective.

Definition 15 (*agent*) An agent is a triple $< P, S, IC >$, representing a program P written in a language that allows the definition of *functions*, a set S of such functions called *services*, and a set IC of *integrity constraints*.

The services $s \in S$ are annotated with pairs $< s, \delta >$, that represent the agent's denotation, as defined below. I do not make assumptions on the syntax of the agent programs, services, and ICs, although I assume that the agents are provided with a mechanism, that we call 'local consistency check', that is able to determine if the constraints are violated. In the examples of this thesis I use a syntax for the ICs and a semantics for the (local) consistency check that is derived from the abductive logic programming literature, inspired in particular to the KM abductive proof-procedure.

Definition 16 (*agent denotation*) Let A be an agent, s a service and δ a set of conditions. We call *denotation of agent* A ($Den(A)$) the following set:
$Den(A) \stackrel{def}{=} \{< s, \delta >$: s is locally provided by A under the conditions δ and $\delta \cup IC_{\{A\}} \not\models \perp\}$

I assume that $Den(A)$ is sound and complete with respect to the notion of local computation.

Definition 17 (*agent system*) An *agent system* is a finite set, whose elements are atoms representing the name of an agent.

The syntax of the operators is given in BNF. Let $\mathcal{V}$ be the vocabulary of the language used to write the agents' program P. To its vocabulary, it is added the set $\{\ \&\ ,\ \ ;\ ,\ \ >\ ,\ \downarrow\}$ of operators, where, differently from the LAILA down-reflection operator, $\downarrow$ is the *local execution* operator.

Such operators are part of *expressions*, possibly enclosed in the agent programs, where the *local execution* operator has a maximum priority, followed by the *communication* operator; the *competitive* coordination operator has the minimum priority. As far as associativity, they are all left-associative. An expression is defined as follows:

$$
\begin{array}{lcl}
Expression & ::= & Formula\ ;\ Expression \mid Formula \\
Formula & ::= & SingleFormula\ \&\ Formula \mid \\
 & & SingleFormula \\
SingleFormula & ::= & Agent > SingleFormula \mid \\
 & & \downarrow Service \\
Agent & ::= & \mathbf{A_0} \mid \mathbf{A_1} \mid \ldots \mid \mathbf{A_n}
\end{array}
$$

The agents are labeled $\mathbf{A_0}\ \ldots\ \mathbf{A_n}$. *Service*, as introduced in Definition 15, is a function call in the language that embeds the communication, competitive and collaborative coordination operators; such services are often s_0, s_1, etc. Services are associated with *conditions*. Conditions are grouped into sets that in the operational semantics are usually labelled δ, δ', δ_1, etc. The communication operator can also be used to issue requests to 'local' services, that could involve some constraints in their execution.

Let us consider the same examples of Section 4.2. Let the following expression be enclosed in an agent program, say A_0:

$$\texttt{A0} > \downarrow \texttt{s1} \; ; \; \texttt{A1} > \downarrow \texttt{s2}$$

It means that A_0 must either perform a local service, s_1, *or* ask agent A_1 for service s_2. Should both service be available, possibly under different conditions, the system will select non-deterministically only one of them.

Let us consider, now, the following expression, also embodied in agent A_0's program, representing a collaborative request composed of two different sub-requests, whose conditions must be coherent with one another:

$$\texttt{A1} > \downarrow \texttt{s3} \; \& \; \texttt{A2} > \downarrow \texttt{s4}$$

Agent A_0 asks agent A_1 for the service s_3 and A_2 to for the service s_4; after both A_1 and A_2 reply to A_0 by giving each a set of conditions for the requested service, the result is obtained by merging such sets of conditions in a unique consistent set, with respect to the bunch of agents (A_0, A_1, and A_2) dynamically considered along the computation. Such set could be bigger than the union of the parts, due to additional constraints that are fired in the cross checking phase.

4.8 Operational semantics

I give now an operational semantics for the operators introduced in the previous section, in the form of inference rules which can be used as an abstract specification of an algorithm, that guarantees the correct execution of multiple and possibly conflicting tasks. The inference rules could be implemented to extend the operating system or virtual machine that supports the execution of agents.

For the sake of simplicity, the rules I define refer to the case of propositional expressions, not including variables, but they can easily be generalized. The same notation as in Section 4.3 is used here for agents, for the universe of agents, as well as for the entailment symbol and the consistency derivation symbol. The agent's code embodies both the integrity constraints IC and the program P of Definition 15: for the sake of simplicity, and with abuse of notation, I will write the name of the agent instead of its code. As before, $A \vdash_{\delta}^{B} F$ means that the formula F is proven within A's code, producing as an output a set of hypotheses δ and a bunch B. As far as the consistency derivation, $B \overset{cons}{\vdash}_{\delta} \Delta$, where Δ is a set of conditions on a (set of) service(s), denotes the "consistency check" of the conjunction of all atoms in Δ, with respect to the integrity constraints of all agents in bunch B. In particular, let $S \subseteq \{1, \ldots, n\}$. Then, given a bunch

B of agents $\{A_i|i \in S\}$, let IC_i denote the set of integrity constraints of agent A_i. The declarative semantics of $B \overset{cons}{\vdash}_\delta \Delta$ is defined as follows:

$$\begin{cases} \forall A_i \in B \quad \delta \cup IC_B \not\models \perp \\ \Delta \subseteq \delta \end{cases}$$

where $IC_B = \bigcup_{j \in S} IC_j$. That is, δ satisfies[9] all the integrity constraints of agents in B. In Section 4.10 it is proven that the operational notion of consistency in a bunch and his declarative counterpart are equivalent (Lemma 1). The same considerations made in Section 4.3 hold with respect to the separation among agent programs and constraints.

I also define a concept of "local consistency", and adopt the following notation: $A \overset{l-cons}{\leadsto}_\delta \Delta$, where Δ is a set of conditions. By 'local consistency' I mean that Δ is consistent with agent A's integrity constraints, IC, returning a (possibly enlarged) set of conditions δ:

$$\begin{cases} \delta \cup IC \not\models \perp \\ \Delta \subseteq \delta \end{cases}$$

The consistency derivation of the KM also implements this more general concept of consistency, being compliant with its declarative semantics.

Beside understanding the role that integrity constraints play to ensure the correct behaviour of a system, as briefly exemplified above, I would like to show that the consequences that it brings to extend the concept of consistency to the multi-agent case are quite similar to those studied in Section 4.3 with respect to the particular case of consistency derivation for abductive programs. In particular, again we have that, while the union of two given sets of hypotheses (conditions) could be enough to ensure the consistency of two separate sets of integrity constraints, it may become insufficient if we want to ensure the consistency of the union of the integrity constraints. The following example will enlighten the reader on this respect.

Example 5 Let us consider two different sets of constraints, $IC_1 = \{\leftarrow a, c\}$ and $IC_2 = \{\leftarrow b, not\ c\}$, a, b and c being possible conditions on a service. If we check the consistency of the set of hypotheses $\{a, b\}$ against either of IC_1 and IC_2 separately, we obtain no inconsistency. But, if we join the constraints in a unique set $IC_3 = IC_1 \cup IC_2$, it is understood that $\{a, b\}$ cannot be the case, since we would obtain both $\{a, notc\}$ from IC_1 and $\{b, c\}$ from IC_2, and therefore both c and its negation in the set of hypotheses that includes a and b.

For this reason, in a distributed setting such as that of multi-agents, the individuals must communicate to each other the possible sources of inconsistency,

[9]There are several notions of constraint satisfaction. The one adopted here refers to a formalization of integrity constraints used in abductive logic programming of the kind: $false \leftarrow body$, where $body$ is a set of conditions. $\delta \cup IC \not\models \perp$ means that $\forall ic \in IC, body(ic) \not\subseteq \delta$, i.e., δ does not make $body(ic)$ true.

like it was c in Example 5. The sets of conditions may therefore grow bigger while the consistency check is made, and the enlarged set of conditions checked for consistency again by all the involved agents, until it gets stable.

Let us describe now the operational behavior of the system, by giving the inference rules of the operators.

The *communication* operator is used to request a service to an agent, e.g. $Y > F$ is used to ask Y the service F. In general, F is a *single formula* that contains properly combined service requests. Its semantics is the same as in Definition 9, where in place of a goal G, we have F. In the communication formula $Y > F$ contained in the code of an agent X, if F is simply a local execution such as $\downarrow s$, X asks Y to locally perform the service s. In general, however, F may include several nested communication formulas, involving several agents possibly enclosing some private integrity constraints. The same considerations made in Section 4.3 with respect to the consistency check still hold here, that make of the communication operator more than just a simple call to another agent's knowledge base and reasoning module.

There could be the case that an agent requires a service that can be performed in several ways, but it does not really matter which one is selected for execution. This is why I introduce the *competitive* operator, that introduces a degree of non-determinism, since in the expression $f\ ;\ F$ there is no precedence relationship between the two operands, f and F. The semantics is the same as in Definition 10.

As far as the *collaborative* coordination operator, again its semantics is the same as in Definition 11, that requires a consistency step involving all agents that contribute to provide the service. It is used to indicate that two services are both required, and must be consistent with each other.

Finally, we need to re-define the semantics of the consistency check, making reference to a concept of local consistency that does not directly relate to the abductive mechanism. To this purpose, we write the following inference rules:

Definition 18 (*consistency check*)

$$\frac{\forall A_i \in B\ A_i \overset{l-cons}{\rightsquigarrow}_{\delta_i} \delta \ \wedge\ \delta \subset \bigcup_{A_j \in B} \delta_j \ \wedge\ B \overset{cons}{\vdash}_{\delta'} \bigcup_{A_i \in B} \delta_i}{B \overset{cons}{\vdash}_{\delta'} \delta}$$

$$\frac{\forall A_i \in B \quad A_i \overset{l-cons}{\rightsquigarrow}_{\delta} \delta}{B \overset{cons}{\vdash}_{\delta} \delta}$$

where the symbols have the same meaning as in Definition 13. This semantics differs from that of Definition 13 in that it does not refer to the concept of abductive reasoning in any way. A new operator, instead, is the local computation operator, that in fact substitutes the concept of abductive derivation. Its semantics is at follows:

Definition 19 (*local computation formula*)

$$\frac{local\_computation(A, s, \delta') \quad \wedge \quad A \overset{l-cons}{\rightsquigarrow}_{\delta} \quad \delta'}{A \vdash_{\delta}^{\{A\}} \quad \downarrow s}$$

where A is an agent, $A \in \{A_0, ..., A_n\}$, s is a service, and $\{A\}$ is the bunch composed by the only agent A. *local_computation* is a function evaluated by agent A, that returns the conditions δ' associated with a certain service s, that A is asked to execute. δ' must be checked for consistency with the other constraints of A, which could result in an enlargement of δ' leading to a $\delta \supseteq \delta'$. The local computation formula, characterized by the $\downarrow$ operator, is intended not as an actual execution of a service, but rather as a preliminary step that anticipates the execution of a service.

Finally, I re-define the definition of a successful top-down derivation.

Definition 20 (*successful top-down derivation*) Let A be an agent and F an expression that possibly describes the request for a service. A *successful top-down derivation* for F in A, which returns a set of conditions δ and the bunch of agents B dynamically involved in the service, can be traced in terms of a (finite) tree such that:

- The root node is labeled by $A \vdash_{\delta}^{B} F$;
- The internal nodes are derived by using, backwards, the inference rules defined above;
- All the leaves are labeled by the empty formula, or represent a successful local computation.

4.9 Examples

In this section, I show two examples in the printer management domain introduced at the beginning of Section 4.7. The first example is about the consistency check that follows a call to a service made through a communication operator, the second shows the collaborative conjunction of two service requests.

Example 6 Let the system be composed of two agents: A_{clnt} and A_{prn}. Let A_{prn}'s integrity constraints be $IC_1 = \leftarrow$ *students, not inkjet* and $IC_2 = \leftarrow$ *students,* p_1, representing authorization policies, meaning that if the client is of group *student* then the printer must be an *ink-jet*, but not p_1. Let A_{clnt}'s integrity constraint be: $IC_3 = \leftarrow$ *inkjet, not* p_1, *not* p_2, representing a constraint on the quality of the printer service, meaning in particular that if an ink-jet printer is proposed, then it must be either p_1 or p_2 (one can imagine that there are several printers, and that p_1 and p_2 are the only ink-jet printers known by A_{clnt} that meet her requirements).

Now, if A_{clnt} asks A_{prn} for printing a document, for instance with the expression $A_{prn} \ > \ \downarrow$ *auth_print(doc)*, one can assume that, as soon as A_{prn} receives the requests (Definition 9), according to the rule in Definition 19 she runs a local procedure (*auth_print*), checking A_{clnt}'s authorizations, recognizing her as being a student, and producing an initial set of conditions, $\delta_1 = \{$ *students* $\}$. Checking this against her integrity constraints (as required by Definition 19), she finds she must include the condition *inkjet* in order to satisfy IC_1, and transmits to A_{clnt} the set of conditions, $\delta_1 = \{$ *students, inkjet* $\}$. From now on, the negotiation process continues automatically, following the rules of Definition 18, i.e., enlarging the set of conditions until it gets stable. In particular, A_{clnt}'s IC_3 fires; let us assume that A_{clnt} produces $\delta_2 = \{$ *students, inkjet,* $p_1\}$ (the choice between p_1 and p_2 is non deterministic). This means, 'it is ok to print with students rights on an inkjet printer, and please use p_1'. A_{prn} checks it and finds that it violates IC_2, therefore rejects it. As A_{clnt} has another possibility to satisfy its constraint, she proposes another alternative delta, $\delta_3 = \{$ *students, inkjet,* $p_2\}$, that A_{prn} checks and accepts with no additional conditions. The consistency check is therefore terminated. When A_{prn} executes the service, it will not violate any of the involved agents' integrity constraints.

The communication formula used in Example 6 joins one agent (A_{clnt}) to a bunch ($\{A_{prn}\}$) for the consistency check. The following example shows how the collaborative coordination operator can be used to join one agent (A_{clnt}) and two bunches ($\{A_{prn}\}$ and $\{A_{pub}\}$) together in order to print a document according to the publisher's requirements.

Example 7 Let us consider a system composed of three agents: a client A_{clnt}, a printer manager A_{prn}, and a publisher A_{pub}. The scenario is similar to the previous one. This time, A_{clnt} is a member of the staff, that produced a report for the annual department book and needs to print it out for the publisher. She therefore submits it to the printer manager, and at the same time asks the publisher agent for the printout requirements. The publisher is represented by agent A_{pub}. The agents' constraints are as follows. A_{prn} has the constraint: $\leftarrow$ *staff, not inkjet, not laser*, representing authorization policies, meaning that if the client is of group *staff*, then the printer must be either an *ink-jet*, or a *laser* printer. A_{clnt} contains an integrity constraint of the kind: $\leftarrow$ *inkjet, laser*, whose meaning is, trivially, that a printer cannot be both a laser and an ink-jet. A_{pub} has no constraints, but contains a function *requirements* that returns a set of conditions that must be met. In particular, *requirements(doc, book)* returns the set δ_1 containing *laser* (the printout must be of laser quality).

Now, let us assume that A_{clnt} needs to evaluate the following expression: $A_{prn} \ > \ \downarrow$ *auth_print(doc)* & $A_{pub} \ > \ \downarrow$ *requirements(doc, book)*. A_{prn} recognizes A_{clnt} as a member of the staff, generates $\delta' = \{$ *staff* $\}$, checks it against her integrity constraint producing $\delta_2 = \{$ *staff, inkjet* $\}$ and returns it to A_{clnt} (Definition 9). From δ_2, the bunch $\{A_{clnt}, A_{prn}\}$ produces $\delta_2' = \{$ *staff, inkjet, not laser* $\}$. On the other hand, A_{pub} returns $\delta_1 = \{laser\}$. From

δ_1, the bunch $\{A_{clnt}, A_{pub}\}$ produces $\delta_1' = \{$ *laser, not inkjet* $\}$ (Definition 9 + 19). Then, according to the inference rule in Definition 11 that describes the & operator, a consistency check has to be made within all agents in the two bunches, put together to form the superset $B = \{A_{clnt}, A_{prn}, A_{pub}\}$. The check is on the set of conditions that collects the conditions generated from the two branches of the collaboration operator, and in particular on $\delta_3 = \delta_1' \cup \delta_2'$. It fails (according to Definition 18) because δ_3 contains an inconsistency. In fact, in δ_3 there are both a condition, *laser*, and its negation, *not laser*. At this point, A_{prn} has another set of conditions left that satisfies the constraint $\leftarrow$ *staff, not inkjet, not laser*, i.e., $\overline{\delta_1} = \{$ *staff, laser* $\}$. Then, a $\overline{\delta_1'} = \{$ *staff, laser, not inkjet* $\}$ is created by $\{A_{clnt}, A_{prn}\}$, and putting eventually all things together, a final set of conditions is generated, $\delta = \overline{\delta_1'} \cup \delta_2' = \{$ *staff, laser, not inkjet* $\}$, that satisfies all agents' integrity constraints.

4.10 Soundness and completeness

In this section, the soundness and completeness properties of the system are proven, which guarantee a safe execution of tasks. To this purpose, I start by introducing two basic properties, concerning the soundness and completeness of local consistency derivations. I assume that they hold for all the local consistency computations.[10]

Property 1 (*soundness of the local consistency derivation*) Given an agent A_i and a set of conditions δ,

$$A_i \overset{l-cons}{\rightsquigarrow}_{\delta_i} \delta \Rightarrow \delta_i \cup IC_i \not\models \perp$$

where IC_i represents the integrity A_i's constraints.

This means that, if there exists a local consistency derivation for δ in A_i that returns a set of conditions δ_i, then δ_i is consistent with the integrity constraints of A_i itself.

Property 2 (*completeness of the local consistency derivation*) Given an agent A_i and a set of conditions δ,

$$\forall \delta : \delta \cup IC_i \not\models \perp \Rightarrow \exists \delta_i : A_i \overset{l-cons}{\rightsquigarrow}_{\delta_i} \delta \wedge \delta_i \supseteq \delta$$

where IC_i represents the integrity A_i's constraints.

[10] Although this system does not rely upon a logic system, there exist in literature some proof procedures that implement a local consistency derivation, such as [48, 59], some of which are proved correct and complete.

This means that, if a set of conditions δ is consistent with the integrity constraints of an agent A_i, then there exists a local consistency computation for δ in A_i itself, which possibly adds some new conditions to δ, returning $\delta' \supseteq \delta$.

In the following, I prove two lemmas about the soundness and completeness of the consistency check in a bunch of agents of Definition 18, that I will use to prove the theorems below.

Lemma 1 (*soundness of the consistency check*) Given a bunch of agents B and a set of conditions δ,

$$B \overset{cons}{\vdash}_{\delta'} \delta \Rightarrow \delta' \cup IC_B \nvDash \bot \wedge \delta' \supseteq \delta$$

where IC_B represents the union of the integrity constraints of all the agents in B.

Proof. The proof is given by separately proving the two conjuncts in the right hand side of the implication. In particular, I prove inductively that $\delta' \supseteq \delta$.

- If $B \overset{cons}{\vdash}_{\delta'} \delta$ is 1-length computation, then by the second inference rule in Definition 18, $\delta' = \delta$.
- If $B \overset{cons}{\vdash}_{\delta'} \delta$ is n-length, I assume the lemma holds for n-1. Then, by the first inference rule in Definition 18, $\delta' \supseteq \bigcup_{A_i \in B} \delta_i$. Since $\delta \subset \bigcup_{A_i \in B} \delta_i$, then follows: $\delta' \supset \delta$.

I prove inductively that $\delta' \cup IC_B \nvDash \bot$:

- for one-length computation, $B \overset{cons}{\vdash}_{\delta} \delta$ is equivalent to $\forall A_i \in B \; A_i \overset{l-cons}{\rightsquigarrow}_{\delta} \delta$ by the second inference rule in Definition 18. By the soundness of the local consistency derivation (Proposition 1) follows that $\forall A_i \in B, \delta \cup IC_i \nvDash \bot$;
- for an n-length computation, $B \overset{cons}{\vdash}_{\delta'} \delta$ by the first inference rule of Definition 18 is equivalent to $\forall A_i \in B, A_i \overset{l-cons}{\rightsquigarrow}_{\delta_i} \delta \;\; \wedge \;\; \delta \subset \;\; \bigcup_{A_j \in B} \delta_j$ $\wedge \; B \overset{cons}{\vdash}_{\delta'} \;\; \bigcup_{A_i \in B} \delta_i$. By the inductive hypothesis, $B \overset{cons}{\vdash}_{\delta'} \;\; \bigcup_{A_i \in B} \delta_i$ implies $\delta' \cup IC_B \nvDash \bot$.

■

Lemma 2 (*completeness of the consistency check*) Given a bunch of agents B,

$$\forall \delta' : \delta' \cup IC_B \nvDash \bot \Rightarrow \exists \delta \supseteq \delta' : B \overset{cons}{\vdash}_{\delta} \delta'$$

Proof. $\delta' \cup IC_B \nvDash \bot$ implies $\forall ic \in IC_B : \delta' \cup ic \nvDash \bot$. This implies, by Proposition 2, that $\forall A_i \in B \; \forall ic_i \in IC_i \exists \delta_i : A_i \overset{l-cons}{\rightsquigarrow}_{\delta_i} \delta' \wedge \delta_i \supseteq \delta'$. By induction on the length of the computation,

- for a 1-length computation, by the second inference rule of Definition 18, the statement is trivially proven;
- for an n-length computation, by the first inference rule of Definition 18, and by the inductive hypothesis the statement is proven as well.

■

In the following, I state the soundness theorem for this system. The theorem states that, for any successful top-down derivation, under the assumption of soundness of the local consistency derivations (Proposition 1), the computed conditions satisfy all the integrity constraints of the bunch of agents dynamically involved in the service in question. This implements in a sense the introduced notion of *safe execution*.

Theorem 1 (*Soundness*) Let A be an agent and F an expression that possibly describes the request for a service. If there exists a *successful top-down derivation* for F in A, which returns a set of conditions δ and the bunch of agents B, then the computed set δ is consistent with the integrity constraints of the bunch B. Formally:

$$A \vdash_{\delta}^{B} F \Rightarrow \delta \cup IC_B \not\models \bot$$

.

Proof. The proof is given by induction on the length of the derivation:

- for a 1-length derivation, F has the form $\downarrow s$, where s is a service. By Definition 19, $A \vdash_{\delta}^{\{A\}} \downarrow s$, which implies that there exists in A a local computation for s returning δ' and $A \overset{l-cons}{\rightsquigarrow}_{\delta} \delta'$. By Proposition 1, it follows that $\delta \cup IC_{\{A\}} \not\models \bot$;
- For any n-length computation, let us consider several cases:
  - *(i)* F is in the form $Y > f$. Then, $A \vdash_{\delta}^{B \cup \{A\}} f$ by Definition 9 implies $Y \vdash_{\delta'}^{B} f$ and $B \cup \{A\} \overset{cons}{\vdash}_{\delta} \delta'$. By the inductive hypothesis, $\delta' \cup IC_B \not\models \bot$, and by Lemma 1, $\delta \cup IC_{B \cup \{A\}} \not\models \bot$ and $\delta \supseteq \delta'$. Then, it follows that $\delta' \cup IC_{B \cup \{A\}} \not\models \bot$;
  - *(ii)* When F is in the form $f; F'$, the statement is trivially proven.
  - *(iii)* F is in the form $f \& F'$. Then, $A \vdash_{\delta}^{B' \cup B''}$ by Definition 11 implies $A \vdash_{\delta'}^{B'} f \quad \wedge \quad A \vdash_{\delta''}^{B''} F' \quad \wedge \quad B' \cup B'' \overset{cons}{\vdash}_{\delta} \quad \delta' \cup \delta''$. The first two conjuncts imply, by the inductive hypothesis, $\delta' \cup IC_{B'} \not\models \bot \wedge \delta'' \cup IC_{B''} \not\models \bot$. The third conjunct implies, by Lemma 1, $\delta \cup IC_{B' \cup B''} \not\models \bot \wedge \delta \supseteq \delta' \cup \delta''$

■

In this latter part of the section, I also state a completeness theorem for a subset of the possible expressions occurring in a top-level request. The completeness theorem ensures that, if a δ can be found, satisfying the integrity constraints of a bunch of agents, and it belongs to the meaning of some agent, as defined below, then there exists an expression F such that there exists a successful top-down derivation for F in A leading to the bunch B of agents, and to conditions δ.

Theorem 2 (*Completeness*) Let B be a bunch of n agents, B, s a service and δ a set of conditions.
$\forall B = \{A_1, \ldots, A_n\}, \forall < s, \delta > \in Den(A_i), A_i \in B : \delta \cup IC_B \not\models \bot \Rightarrow \exists F, A \in B :$ $A \vdash^B_{\hat{\delta}} F \wedge \hat{\delta} \supseteq \delta$ where F is an expression occurring in a top-level request.

Proof. The proof is given in the case of a communication formula F expressed in the form: $F = A_{1i} > A_{2i} > \ldots > A_{ki} > A_i >\downarrow s$, where $k \geq n-2$, and $\{A_{ji} : j = i..k\} \cup \{A\} \cup \{A_i\} = B$. By Definition 16, being $< s, \delta > \in Den(A_i)$, it follows that

$$(i) \qquad local\_computation(A_i, s, \delta) \wedge \delta \cup IC_{A_i} \not\models \bot$$

By Proposition 2, it follows that

$$(ii) \qquad \exists \delta \supseteq \delta' \; s.t. \; A \overset{l-cons}{\rightsquigarrow}_{\delta'} \delta$$

Therefore, by Definition 19, (i), and (ii), it follows that

$$(iii) \qquad A_i \vdash^{\{A_i\}}_{\delta'} \downarrow s, \delta' \supseteq \delta$$

Now, being $\delta \cup IC_B \not\models \bot$, by Lemma 2 it follows that $B \overset{cons}{\vdash}_{\delta''} \delta, \delta'' \supseteq \delta$. Now, by Lemma 1, $\delta'' \cup IC_B \not\models \bot$, and, again by Lemma 2, $B \overset{cons}{\vdash}_{\tilde{\delta}} \delta''$. From Definition 18 it is possible to prove that $\delta' \subseteq \delta''$. Therefore, $\{A_i\} \cup \{A_{ki}\} \overset{cons}{\vdash}_{\delta'''} \delta'$. From this, Definition 9 and *(iii)*, it follows that

$$\{A_i, A_{ki}\} \overset{cons}{\vdash}_{\delta'''} A_i >\downarrow s$$

and $\delta''' \supseteq \delta'$. Now, by iteratively applying Definition 9, we can prove that $\exists \hat{\delta} : A \vdash^B_{\hat{\delta}} A_{1i} > A_{2i} > \ldots > A_{ki} > A_i >\downarrow s \wedge \hat{\delta} \supseteq \delta$
■

In a similar way it is possible to give the proofs referring to other kinds of formula. (such as, for instance, $A \vdash^B_{\delta} A_1 >\downarrow s_1 \; \& \; A_2 >\downarrow s_2 \; \& \; \ldots \; \& \; A_n >\downarrow s_n$, or $A \vdash^B_{\delta} A_{11} > \ldots > A_{1m_1} \downarrow s_1 \; \& \; \ldots \; \& \; A_{k1} > \ldots > A_{km_k} >\downarrow s_k$)

4.11 Concluding remarks

The first part of this chapter was devoted to defining LAILA, a language for abductive logic agents coordination. Then, drawing inspiration from it, a formalism has been defined to express the combination of service requests, and an operational semantics that can be used to extend an operating system or virtual machine supporting the agents' execution, to ensure a safe task execution. I presented two results, with respect to the semantics. The first one is a soundness result, which implements the notion of safe execution, ensuring that no constraint is violated by a service request which is possibly composed of several services, asked to several agents. In other words, if the operational semantics allows the execution of a certain service, then it is safe to execute it. The second is a completeness result, which states that if such service could be safely executed, then its execution is allowed by the operational semantics. This result has been proven for a particular service composition pattern, but it is possible to extend the proof for other more general patterns. The correctness and completeness results hold for any system composed of agents, that provide a local consistency procedure that implements the declarative semantics defined in Section 4.8. ALIAS agents based on the KM proof-procedure for abductive reasoning are an instance of such class of agents.

I have therefore concluded he study of the coordination of agent reasoning, that led us to proposing different alternative models, with different degrees of openness and generality, ranging from static bunches of agents performing distributed abduction, to completely dynamic bunches of agents providing computational services of any kind. In all cases, the need for a kind of coordination of the agent reasoning arises from the presence of constraints in the agent knowledge base, and from the need to keep the consistent, whatever computation is performed within the agent, possibly in relationship with other partners. I presented several "toy examples" that could nevertheless be exported to real-world situations.

In the next chapter, I will study in detail an application of the abductive reasoning in multi-agent systems, that is the dialogue between agents. I will mostly refer to a well-known problem, that of resource negotiation, but I will also study the extension of the proposed framework from dialogue to auction-like interactions.

Chapter 5

Dialogue and negotiation

I have so far discussed different possibilities for coordinating the reasoning of logic-based and non logic-based agents, introducing to this purpose several architectures and a language. I presented examples taken from different domains, such as medical diagnosis, web based services, home improvement, computer network management. In this chapter, I start by showing some applications of abductive reasoning in the negotiation domain, that has already been introduced in the past chapters. I build on the work by Sadri et al. [88], including both dialogue-based (one-to-one) negotiation for resource allocation, and auction-like (one-to-many) negotiation patterns.

5.1 Dialogue based negotiation

Negotiation is one of the main research areas in multi-agent systems. In many cases agents need to negotiate because they operate in environments with limited resource availability. The area of negotiation is broad and applies to different scenarios [54]. For example, *one-to-many* negotiation is used for auctions, where auctioneer and bidders reach an agreement on the cost of the items on sale. Most research in the area of one-to-many negotiation adopts a game-theoretic approach, with interesting results but under the often unrealistic assumptions that the computational resources of agents are unbounded.

Another example of negotiation is *one-to-one* negotiation, used, for instance, for task reallocation [90] and for resource reallocation [78], where the limited resources may be time, the computational resources of agents, or physical resources needed to carry out some tasks. Many approaches in the area of one-to-one negotiation are heuristic-based and, in spite of their experimentally proven effectiveness, they do not easily lend themselves to expressing theoretically provable properties. Other approaches present a good descriptive model, but fail to provide an execution model that can help to forecast the behaviour of any corresponding implemented system.

Persuasion also plays a role when the agent goals are conflicting and the resources in the system are not enough to carry out all agent plans, therefore agents try to modify each other goals or intentions. Many argumentative frameworks are proposed to tackle the problem [3, 72], that often try to embrace very hard problems and result in very good descriptive models for them. Still, an execution model for most argumentation based systems is not straightforward to obtain, and it is also difficult to give properties and forecast the behaviour of an implemented system, if any.

Dialogue is one of the most flexible interaction patterns in Multi-Agent Systems, being something between completely fixed protocols and totally free conversations [34]. Intuitively, a dialogue is not the (purely reactive) question-answer sequence of a client server architecture framed in a rigid protocol, but is rather a kind of interaction grounded on an expressive enough knowledge representation. Dialogues start, in general, from the need to achieve an explicit goal. The goal of initiating a dialogue could be for example to persuade another party, to find an information, to verify an assumption, and so on [101]. In the case of negotiation dialogues, for instance, agents need to negotiate because they operate in environments with limited resource availability, and the goal of a dialogue is to obtain a resource. This general idea applies to different scenarios with different meanings.

The point is to use an abductive framework to tackle one-to-one negotiation, between agents that share the same language for negotiation but may adopt different negotiation policies. The logic-based approach to the negotiation problem allows for proving properties such as termination, correctness and completeness. I also propose an execution model to be used in order to implement the system. Although the focus is on the specific problem of resource reallocation, the approach presented here is more general, as far as the representation of policies and the reasoning that generates dialogues and their sequences, and could be extended to be applicable to other negotiation domains.

The proposed framework is based on agent dialogues, as a way to request resources, propose resource exchanges, suggest alternative resources, etc. I assume that the agents are equipped with a planner that, given a goal, is able to produce a sequence of actions to achieve it and an associated set of missing resources that are needed to carry out the plan. I study some properties of dialogues, sequences of dialogues, and sets of sequences of dialogues in an agent system. I define the properties of termination, correctness and completeness, and show whether they hold for some concrete agent varieties I propose. Finally, I study how a heterogeneous system, i.e., a system composed by a variety of agents, can be used to implement resource reallocation policies. Note that no concrete assumption on the internal structure of agents is made, except for requiring that they hold beliefs, goals, intentions and, possibly, resources. In the last part of this chapter, I extend the proposed framework towards one-to-many negotiation schemes, those used in auction contexts.

5.2 Language for negotiation

There are some assumptions that I made about the agents. I presume a logic language for the knowledge base of agents, but make only three assumptions on the syntax of this language. In particular, I assume that it contains notions of *literal*, *complement* of sentences, *conjunction* of sentences, *true* and *false*, and that such language is equipped with a notion of *entailment*, such that, for every ground literal in the language, either the literal or its complement is entailed from the set of sentences that represent the knowledge of agents, and such that no literal and its complement are entailed at the same time. I also make concrete assumptions on the syntax of a component of the knowledge base, consisting of "dialogue constraints" (described in Section 5.4). The purpose of negotiation is for the agent to obtain the missing resources, while retaining the available ones that are necessary for the plan in its current intention.

In Section 4.4.2 I anticipated, informally, that the "ingredients" that I use to build up a negotiation framework based on abduction are:

- a *communication language*, as well as a *content language*,
- one or more *protocols* to specify allowed interactions,
- a way to represent the *agent knowledge*,
- a *reasoning mechanism*, and
- an *execution model*.

Let us start with the language for negotiation. A language is a set of performatives, i.e., possible utterances that agents my direct to each other in the course of a conversation.

Definition 21 (*performative or dialogue move*) A *performative* or *dialogue move* is an instance of a schema $tell(X, Y, \mathbf{Subject}, T)$, where X is the *utterer* and Y is the *receiver* of the performative, and T is the *time* when the performative is uttered. **Subject** is the *content* of the performative, expressed in some given *content language*.[1]

A concrete example of a performative is

$$tell(a, b, \mathbf{request}(\mathbf{give}(nail)), 1)$$

where **Subject** is $\mathbf{request}(\mathbf{give}(nail))$. Intuitively, this performative represents a's request to b for a nail, at time 1.

I will often refer to a performative $tell(X, Y, \mathbf{Subject}, T)$ simply as $p(T)$, if I want to emphasize the time of the performative, or p, where p stands for "performative".

[1] As usual, Capital and lower-case letters stand for variables and ground terms, respectively.

Definition 22 (*language for negotiation*) A *language for negotiation* $\mathcal{L}$ is a (possibly infinite) set of (possibly non ground) performatives. For a given $\mathcal{L}$, I define two (possibly infinite) subsets of performatives: $\mathcal{I}(\mathcal{L})$ and $\mathcal{F}(\mathcal{L}) \subseteq \mathcal{L}$, called respectively *initial moves* and *final moves*. Each final move is either *successful* or *unsuccessful*.

Let us define a possible concrete language for negotiation, that is a subset of that proposed in [3]:

- $tell(X, Y, \mathbf{request}(\mathbf{give}(Resource)), T)$ is a dialogue move, whose content is $\mathbf{request}(\mathbf{give}(Resource))$, used to request a resource R;
- $tell(X, Y, \mathbf{promise}(\mathbf{give}(Resource), \mathbf{give}(Resource')), T)$ } is a dialogue move, of content, whose content is $\mathbf{promise}(\mathbf{give}(Resource), \mathbf{give}(Resource'))$, used to propose and to commit to exchange deals, of $Resource'$ in exchange for $Resource$.
- if $Content$ is the content of a dialogue move, so are

  $\mathbf{accept}(Content)$, $\mathbf{refuse}(Content)$ (used to accept/refuse a previous dialogue $Move$)

  $\mathbf{challenge}(Content)$ (used to ask a justification for a previous move of content $Content$)

  $\mathbf{justify}(Content, Support)$ (used to justify a past move of content $Content$, by means of a $Support$)
- there are no other dialogue moves, except the ones given above.

I will use this language in the following sections, therefore I give it a name, $\mathcal{L}_1$:

$$
\begin{aligned}
\mathcal{L}_1 = \{ \; & tell(X, Y, \mathbf{request}(\mathbf{give}(Resource)), T) \\
& tell(X, Y, \mathbf{accept}(Move), T) \\
& tell(X, Y, \mathbf{refuse}(Move), T) \\
& tell(X, Y, \mathbf{challenge}(Move), T) \\
& tell(X, Y, \mathbf{justify}(Move, Support), T) \\
& tell(X, Y, \mathbf{promise}(\mathbf{give}(Resource), \mathbf{give}(Resource')), T) \; \}
\end{aligned}
$$

The initial and final moves are:

$$
\mathcal{I}(\mathcal{L}_1) = \{tell(X, Y, \mathbf{request}(\mathbf{give}(Resource)), T)\}
$$

$$
\begin{aligned}
\mathcal{F}(\mathcal{L}_1) = \{ \quad & [\textit{successful moves:}] \\
& tell(X, Y, \mathbf{accept}(\mathbf{request}(\mathbf{give}(Resource))), T), \\
& tell(X, Y, \mathbf{accept}(\mathbf{promise}(\mathbf{give}(Resource), \mathbf{give}(Resource'))), T), \\
& [\textit{unsuccessful moves:}] \\
& tell(X, Y, \mathbf{refuse}(\mathbf{request}(\mathbf{give}(Resource))), T), \\
& tell(X, Y, \mathbf{refuse}(\mathbf{promise}(\mathbf{give}(Resource), \mathbf{give}(Resource'))), T) \\
\}. &
\end{aligned}
$$

Once uttered, dialogue performatives are recorded on a blackboard (the *dialogue store*) shared amongst the agents, in the order in which they have been uttered. For example, the dialogue store could contain the history of Dialogue 1, introduced in Section 4.4.2. Note that times are *transaction timestamps*. Note further that the dialogue store grows monotonically over a dialogue, as new performatives are uttered. I will assume the dialogue store is reset at the end of each dialogue.

As I am now interested in negotiation for the exchange of resources, I will assume that there always exists a **request** move in the (initial moves of any) language for negotiation.

5.3 Knowledge

As far as the content of the knowledge base, I will assume that it describes, in addition to the dialogue constraints,

- domain-specific as well as domain-independent beliefs that the agent can use to generate *plans* for its *goal*, such as the knowledge describing preconditions and effects of actions. I will not make any assumptions on how such plans are generated; in particular, they could be obtained from a planner, either within the agent or external to it, or be extracted from a given library of plans, which is part of the knowledge of the agent;
- information about the resources available to the agent;
- information about the dialogues in which the agent has taken part;
- information on the selected *intention*, consisting of the given *goal*, a *plan* for the given goal, as well as the set of resources already *available* to the agent and the set of *missing* resources, both required for the plan to be executable.

More formally, each agent's knowledge $\mathcal{K}$ is defined as a tuple $< \mathcal{B}, \mathcal{R}, \mathcal{I}, \mathcal{D}, \mathcal{G} >$, where:

- $\mathcal{B}$ contains domain-dependent beliefs about the world, such as the information about itself and the other agents, as well as domain-independent beliefs, used to regulate the negotiation dialogues;
- $\mathcal{R}$ contains the resources owned *initially* (i.e., before the dialogue starts) by the agent (the current state of owned resources can be inferred through $\mathcal{R} \cup \mathcal{D}$, see further on);
- $\mathcal{I}$ is the agent's current intention, i.e., the plan that the agent intends to carry out in order to achieve its goal, together with the available resources and the missing resources, that need to be obtained for the plan to be executable. In $\mathcal{I}$ the agent goal is also included;

- $\mathcal{D}$ contains the current dialogue store, represented by the past (*time-stamped*) dialogue performatives. The dialogue store is shared amongst agents, so all agents involved in the same dialogue are assumed to have the same component $\mathcal{D}$.

- $\mathcal{G}$ is the agent goal;

In the sequel, I will refer to the knowledge base, intention, goal, ...of an agent x as $\mathcal{K}(x)$, $\mathcal{I}(x)$, $\mathcal{G}(x)$, ..., respectively, or simply as $\mathcal{K}$, $\mathcal{I}$ and $\mathcal{G}$, ..., respectively, if clear from the context. An example of an agent x's knowledge is the following $\mathcal{K}(x)$ (0 stands for the initial time):

$\mathcal{B}(x)$ domain-specific beliefs $\{\ is\_agent(y), i\_am(x)\ \}$, as well as domain-independent beliefs, e.g., those described in Section 5.5;
$\mathcal{R}(x)$: $\{\ have(picture, 0),\ have(hammer, 0),\ have(screwdriver, 0)\ \}$;
$\mathcal{I}(x)$: $\{\ available(\{hammer, picture\}, 0),$
$plan(\{obtain(nail), hit(nail),\ hang(picture)\}, 0),$
$goal(\{hung(picture)\}),$
$missing(\{nail\}, 0)\ \}$;
$\mathcal{D}(x)$: $\emptyset$.
$\mathcal{G}(x)$: $hung(picture)$;

Note that I assume that each agent is equipped with a *single (possibly conjunctive) goal*. For the sake of simplicity, I will also assume that this goal does not change during the dialogues, it is therefore not timestamped. The assumption that the agents' goals are static is functional to the (successful) termination results that I present in the last sections. In fact, it would be difficult to define what I mean by successfully solve a problem, whose conditions change while it is being solved. In particular, the success associated with the solution of a resource reallocation problem, as discussed in Section 5.11, depends on the satisfiability of an initial set of goals enclosed in the agent intentions. Such assumption could be relaxed, and this framework extended, for example by letting the agents' modify their goals after they have been achieved, or proven impossible to achieve, in accordance with the Rao and Georgeff's BDI theory [83] that requires goals to be believed feasible.

I assume that the agents are provided with a *cost* function that maps from the domain of the possible intentions (plans) to $\mathbb{R}^+$. Such function will have to be *acceptable*[2] in order for some results to hold, as it will be discussed in the

[2] A *cost* function $|\,.\,|\ :\ \mathcal{I} \rightarrow \mathbb{R}^+$ is ***acceptable*** if it satisfies the following properties:

- $missing(\ \emptyset\ ) \in I \ \Rightarrow\ |\ I\ |\ =\ 0$
- $missing(\ Rs\ ) \in I, Rs \neq \emptyset \ \Rightarrow\ |\ I\ |\ > 0$
- $missing(\ Rs\ ) \in I, missing(\ \bar{Rs}\ ) \in \bar{I}$, $Rs = \bar{Rs} \ \Rightarrow\ |\ I\ |\ =\ |\ \bar{I}\ |$
- $missing(\ Rs\ ) \in I, missing(\ \bar{Rs}\ ) \in \bar{I}$, $Rs \supset \bar{Rs} \ \Rightarrow\ |\ I\ |\ >\ |\ \bar{I}\ |$

sequel. In this context, I will adopt as a cost function for a given intention the number of its missing resources.

I assume that plans and missing resources in intentions are generated by a planner, but I do not rely upon any specific planner: for instance one could adopt a STRIPS-, situation calculus- or event calculus-based planner, or even a plan library. Moreover, each agent might rely upon a different planner. However, I assume that, when given a goal (in $\mathcal{G}$) and the available resources (in $\mathcal{R}$), the planner of an agent provides a suitable plan in the form of an ordered sequence of actions, as well as a (possibly empty) list of missing resources (namely, it returns an intention in $\mathcal{I}$). The planner of an agent might be used to generate plans for other agents, if a resource exchange is required in order to obtain some missing resources in the agent's current plan, and to re-plan (see later in this section). I also assume that the missing resources in the intention and those available to the agent, as given by the knowledge base, have no intersection, and that the available resources in the intention are contained in the resources available to the agent, as given by the knowledge base.

The planner has access to domain-dependent beliefs in $\mathcal{B}$, describing pre-conditions and effects of actions or plan libraries. I assume that agents cannot exchange rules such as '*you can have a picture hung if you have a screw and a screwdriver*'. Therefore, I assume that negotiating agents share knowledge about actions. Such knowledge is also stored in $\mathcal{B}$. This assumption allows agents to re-plan for other agents, if an exchange of resources is required.

In the sequel, given any intention $\mathcal{I}$,

- $goal(G) \in \mathcal{I}$ will stand for 'G is the goal in $\mathcal{I}$'
- $plan(P,T) \in \mathcal{I}$ will stand for 'P is the plan in $\mathcal{I}$, time-stamped with time T'
- $missing(Rs,T) \in \mathcal{I}$ will stand for 'Rs are the resources needed to make the plan in $\mathcal{I}$ executable but currently missing, time-stamped with time T'
- $available(Rs,T) \in \mathcal{I}$ will stand for 'Rs are the resources needed to make the plan in $\mathcal{I}$ executable and currently available, time-stamped with time T'.

I will omit the time T if clear from the context.

5.4 Dialogues

For a given agent $x \in A$, where A is equipped with $\mathcal{L}$, we define the sets

- $\mathcal{L}^{in}(x)$, of all performative schemata of which x is the receiver, but not the utterer;
- of all performative schemata of which x is the utterer, but not the receiver.

Note that it is not allowed for agents to utter performatives to themselves. In the sequel, I will often omit x, if clear from the context, and simply write $\mathcal{L}^{in}$ and $\mathcal{L}^{out}$.

Negotiation policies and protocols can be specified by sets of dialogue constraints:

Definition 23 (*dialogue constraint*) Given an agent system A, equipped with a language for negotiation $\mathcal{L}$, and an agent $x \in A$, a *dialogue constraint* for x is a (possibly non-ground) if-then rule of the form: $p(T) \wedge C \Rightarrow \hat{p}(T+1)$, where

- $p(T) \in \mathcal{L}^{in}(x)$ and $\hat{p}(T+1) \in \mathcal{L}^{out}(x)$,
- the utterer of $p(T)$ is the receiver of $\hat{p}(T+1)$, and the receiver of $p(T)$ is the utterer of $\hat{p}(T+1)$,
- C is a conjunction of literals in the language of the knowledge base of x. [3]

Any variables in a dialogue constraint are implicitly universally quantified from the outside. The performative $p(T)$ is referred to as the *trigger*, $\hat{p}(T+1)$ as the *next move* and C as the *condition* of the dialogue constraint.

Intuitively, the dialogue constraints of an agent x express policies between x and any other agent. The intuitive meaning of a dialogue constraint $p(T) \wedge C \Rightarrow \hat{p}(T+1)$ of agent x is as follows: if at a certain time T in a dialogue some other agent y utters a performative $p(T)$, then the corresponding instance of the dialogue constraint is triggered and, if the condition C is entailed by the knowledge base of x, then x will utter $\hat{p}(T+1)$, with y as receiver, at the next time $T+1$. This behaviour of dialogue constraints can be achieved by employing an automatic abductive proof procedure such as the IFF within an *observe-think-act* agent cycle [67]. The procedure implements a concrete concept of entailment with respect to knowledge bases expressed in abductive logic programming terms [58]. The execution of the proof procedure within the agent cycle allows agents to produce dialogue moves immediately after a dialogue constraint is fired.

The use of an abductive proof procedure, equipped with an operational semantics and theoretical results, has several advantages, beside the availability of an execution model. In particular, the proofs of the theorems stated later in the chapter are implicitly grounded on the property of correctness, which holds for the IFF proof-procedure.

An example of a dialogue constraint allowing an agent x to accept a request is:

$$
\begin{aligned}
&\mathit{tell}(Y, x, \mathit{request}(\mathit{give}(R)), T) \ \wedge\ [\mathit{have}(R, T) \wedge \mathit{not\ need}(R, T)] \\
&\ \Rightarrow \mathit{tell}(x, Y, \mathit{accept}(\mathit{request}(\mathit{give}(R))), T+1)
\end{aligned}
$$

[3] Note that C in general might depend on several time points, possibly but not necessarily including T; therefore I will not indicate explicitly any time variable for it.

where the trigger is $tell(Y, x, request(give(R)), T)$, the condition is within square brackets and the next move is $tell(x, Y, accept(request(give(R))), T + 1)$.

This is a simple example, but in general C might contain other dialogue performatives, uttered in the past by other agents, within the same dialogue or other dialogues. For instance, one could decide to accept a request only if another agent made a certain move in the past.

We will refer to the set of dialogue constraints associated with an agent $x \in A$ as $\mathcal{S}(x)$, and we will call it the *agent program* of x. We will often omit x if it is clear from the context or unimportant. As already mentioned in Section 5.3, the agent program of an agent is part of its knowledge base $\mathcal{K}$.

Note that different agents might be equipped with different agent programs.

Definition 24 (*dialogue*) A *dialogue* between two agents x and y is a set of ground performatives, $\{p_0, p_1, p_2, \ldots\}$, such that, for some given $t \geq 0$:

1. $\forall\ i \geq 0$, p_i is uttered at time $t + i$;

2. $\forall\ i \geq 0$, if p_i is uttered by agent x (resp. y), then p_{i+1} (if any) is uttered by agent y (resp. x);

3. $\forall\ i > 0$, p_i can be uttered by agent $\alpha \in \{x, y\}$ only if there exists a (grounded) dialogue constraint $p_{i-1} \wedge C \Rightarrow p_i \in \mathcal{S}(\alpha)$ such that $\mathcal{K}(\alpha) \wedge p_{i-1}$ entails C.

A dialogue $\{p_0, p_1, \ldots p_m\}$, $m \geq 0$, is *terminated* if p_m is a ground final move, namely p_m is a ground instance of a performative in $\mathcal{F}(\mathcal{L})$.

By condition 1, a dialogue is in fact a *sequence* of performatives. By condition 2, agents alternate utterances in a dialogue. By condition 3, dialogues are generated by the dialogue constraints, together with the given knowledge base to determine whether the condition of triggered dialogue constraints is entailed.

Intuitively, a dialogue should begin with an initial move, according to the given language for negotiation. The kind of dialogue that is relevant to the purposes of this dissertation is that started with a request for a resource R. Such a dialogue will be initiated by an agent x whose intention $\mathcal{I}$ contains R in its set of *missing* resources.

Definition 25 (*request dialogue*) A *request dialogue* with respect to a resource R and an intention $\mathcal{I}$ of agent x is a dialogue $\{p_0, p_1, p_2, \ldots\}$ between x and some agent $y \in A$ such that

- $p_0 = tell(x, y, request(give(R)), t)$,

- $missing(Rs, t) \in \mathcal{I}$ and

- $R \in Rs$,

for some $t \geq 0$.

In general, after a terminated request dialogue d with respect to a resource R and an intention $\mathcal{I}$ of some agent x, the agent might have been successful or not in obtaining the resource R. If it has been successful, then its intention changes so that R is not missing anymore, and the agent will engage in new dialogues to obtain the remaining missing resources, if any, in the intention. If d has been unsuccessful, then the agent's intention does not change, and the agent might decide to engage in a dialogue to obtain R from a different agent.

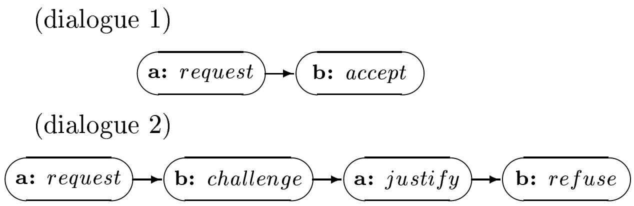

Figure 5.1: Two dialogues, a successful one (dialogue 1) and an unsuccessful one (dialogue 2)

In Figure 5.1 two possible dialogues are sketched: the former terminates successfully while the latter does not. In both cases, the plan in the intention does not change. We also allow a third possibility, of a successful termination with a change of intention possibly (and typically) involving a change of plan. This could happen, for example, if during the dialogue the other agent persuaded the requesting agent to modify its plan, with the promise of a different resource deal.

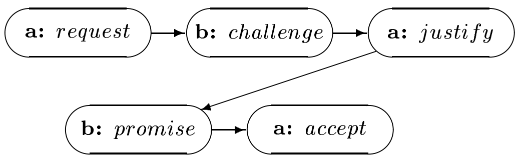

Figure 5.2: c-successful dialogue: on receiving the justification from a, b promises a new resource that makes the original request obsolete

In Figure 5.2 there is an example of c-successful terminated dialogue. In the sequel, I will assume that *a terminated request dialogue, for a given resource R and intention $\mathcal{I}$, returns an intention $\mathcal{I}'$.*

Definition 26 (*types of terminated request dialogues*) Let $\mathcal{I}$ be the intention of some agent a, and R be a missing resource in $\mathcal{I}$. Let d be a terminated resource dialogue with respect to R and $\mathcal{I}$, and $\mathcal{I}'$ be the intention resulting from d. Then, if $missing(Rs), plan(P) \in \mathcal{I}$ and $missing(Rs'), plan(P') \in \mathcal{I}'$:

i) d is *successful* if $P' = P$, $Rs' = Rs \setminus \{R\}$;

ii) d is *conditionally* or *c-successful* if $Rs' \neq Rs$ and $Rs' \neq Rs \setminus \{R\}$;

iii) d is *unsuccessful* if $\mathcal{I}' = \mathcal{I}$.

Note that, in the case of c-successful dialogues, typically, but not always, the agent's plan will change ($P' \neq P$).

As it is done in most approaches to task reallocation and negotiation in general, in order to ensure convergence of the negotiation process we could impose that terminated request dialogues return intentions whose missing resources are no more than those in the original intention, in order to ensure the termination of the negotiation process of an agent, achieved by dialogue sequences.

More in general, we could refer to a property of 'convergence', defind as follows:

Definition 27 (*convergence*) An agent $x \in A$ is *convergent* iff, for every terminated request dialogue of x, with respect to some resource R and some intention $\mathcal{I}$, the *cost* of the returned intention $\mathcal{I}'$ is not higher than the *cost* of $\mathcal{I}$.

The *cost* of an intention can be defined as the number of missing resources in the intention.

5.5 Dialogue protocols

Dialogue protocols can be expressed by means of dialogue constraints, whose conditions can be satisfied by retrieving information from the knowledge of the agent, e.g. the missing resources and the currently owned resources as well as the dialogue store. I will assume that this information is compiled into if-rules defining the predicates *have* (the agent has a resource R), *miss* (the agent has not yet obtained a resource R needed for a plan P to achieve a goal G), *need* (the agent has a resource R needed for a plan P to achieve a goal G), and so on. Before I introduce the dialogue protocols, let us present a simplified version of the if-rules for some such predicates:[4]

`% rule 1: the agent has resource` R `at time` T `if she had it initially`
`% and has not given it away`

$have(R, T) \leftarrow$
$\quad have(R, 0) \;\wedge 0 < T \wedge not\; [gave\_away(R, T1), 0 < T1 \leq T]$

`% rule 2: the agent has resource` R `at time` T `if she has obtained it and`
`% did not give it away`

[4] The *not* here has the meaning of negation as failure

$have(R, T) \leftarrow$
$\quad obtained(R, T1) \ \wedge T1 < T \wedge not\ [gave\_away(R, T2), T1 < T2 \leq T]$

```
% rule 3:  the agent obtained R at time T from Agent after she
% requested it
```

$obtained(R, T) \leftarrow$
$\quad i\_am(X) \wedge tell(Agent, X, accept(request(R)), T)$

```
% rule 4:  the agent gave away R at time T in reply to the Agent's
% request
```

$gave\_away(R, T) \leftarrow$
$\quad i\_am(X) \wedge tell(X, Agent, accept(request(R)), T)$

```
% rule 5:  agent a misses R at time T if it is in the missing list in I_a
% for a plan P and a goal G
```

$miss(R, P, G, Rs, T) \leftarrow$
$\quad intend(I) \ \wedge plan(P, T) \in I \ \wedge goal(G) \in I \wedge missing(Rs, T) \in I \ \wedge R \in Rs$

```
% rule 6:  agent a needs R at time T if it is in the available list in
% I_a for a plan P for a goal G
```

$need(R, P, G, T) \leftarrow$
$\quad intend(I) \ \wedge plan(P, T) \in I \ \wedge goal(G) \in I \wedge available(Rs, T) \in I \ \wedge R \in Rs$

Note that resources are considered not to be owned if they have been promised to another agent, even if the actual delivery has not yet been carried out. Indeed, here we are not concerned with the execution of a plan, and we assume that agents will actually obtain the resources they have been promised by the time of the execution of the plan.

A dialogue protocol can be seen as a property that must be satisfied at all time, by enforcing (uttering) the conclusion of such protocol whenever it is triggered and the conditions in its premise are satisfied by the if-rules. In this sense, they behave like active rules in databases and integrity constraints in abductive logic programming [58]. Indeed, the knowledge of an agent can be seen as an abductive logic program, with the logic program consisting of the if-rules above plus the owned resources, the integrity constraints consisting of the dialogue protocols given below, and the abducibles (possible hypotheses) consisting of the performatives.

I will now describe one protocol in particular, that allows to prove some properties. It is represented by the following set of dialogue constraints. Note that, from now on, I will drop the time parameter, for simplicity.

`% IC 1: accept a request`

$$i\_am(X) \ \wedge \ tell(Agent, X, request(give(R))) \ \wedge \ have(R) \wedge \ not \ need(R) \\ \Rightarrow tell(X, Agent, accept(request(give(R))))$$

This set of constraints describes the behaviour of a collaborative but selfish agent. In particular, the agent will *challenge* the *request* and start a negotiation thread (`IC2`) only if there exists a resource that she is actually missing, i.e., only if she can obtain a positive payoff from further dialogue.

`% IC 2(a): challenge a request if` R `is not held`

$$i\_am(X) \ \wedge \ tell(Agent, X, request(give(R))) \wedge \ miss(\_,\_,\_,\_) \ \wedge \ not \ have(R) \\ \Rightarrow tell(X, Agent, challenge(request(give(R))))$$

`% IC 2(b): challenge a request if` R `is needed`

$$i\_am(X) \ \wedge \ tell(Agent, X, request(give(R))) \ \wedge \ have(R) \wedge \ miss(\_,\_,\_,\_) \ \wedge \ need(R) \\ \Rightarrow tell(X, Agent, challenge(request(give(R))))$$

`% IC 3(a): refuse a request if not convenient to propose an exchange`

$$i\_am(X) \ \wedge \ tell(Agent, X, request(give(R))) \wedge \ not \ miss(\_,\_,\_,\_) \ \wedge \ not \ have(R) \\ \Rightarrow tell(X, Agent, refuse(request(give(R))))$$

`% IC 3(b): refuse a request if not convenient to propose an exchange`

$$i\_am(X) \ \wedge \ tell(Agent, X, request(give(R))) \wedge \ not \ miss(\_,\_,\_,\_) \ \wedge \ need(R) \\ \Rightarrow tell(X, Agent, refuse(request(give(R))))$$

`% IC 4: justify a request`

$$i\_am(X) \ \wedge \ tell(Agent, X, challenge(request(give(R)))) \\ \wedge \ miss(R, Plan, Goal, Missing) \\ \wedge \ Support = \{missing(Missing), plan(Plan), goal(Goal)\} \\ \Rightarrow tell(X, Agent, justify(request(give(R)), Support))$$

In `IC5` and `IC6`, the policy adopted is that when X cannot give R to $Agent$, X tries to be helpful to $Agent$ and looks for an alternative plan for $Agent$'s $Goal$, in that $Agent$ changes intention (R is no longer required, but the new plan requires a new resource R'' that X has and can give) and X gets a resource R' that she is missing. In doing that, X looks for an exchange that is favorable to both, which could be thought of as a *selfish* behaviour. If no such alternative can be found, or if X does not miss anything at all, as in `IC3`, X refuses the request.

`% IC 5: refuse a request after a justification:` $Agent$ `cannot find an`

`%` alternative plan for X's goal that could be proposed for a deal

$i\_am(X) \;\wedge\; tell(Agent, X, justify(request(give(R)), Support)) \;\wedge\; goal(Goal) \in Support \;\wedge\; miss(R', \_, \_, \_) \wedge \; not\ exists\_alternative\_plan(Goal, without(\{R', R\}))$
$\quad \Rightarrow tell(X, Agent, refuse(give(R)))$

`%` IC 6: propose R'' in exchange for R': X finds a different plan for
`%` $Agent$'s goal that makes use of the resource R'' that she has and does
`%` not need, and that requires neither of $\{R, R'\}$, while she needs R',
`%` therefore can propose a deal: R'' in exchange of R'

$i\_am(X) \;\wedge\; tell(Agent, X, justify(request(give(R)), Support)) \wedge \; goal(Goal) \in Support \;\wedge\; miss(R', \_, \_, \_) \wedge \; have(R'') \;\wedge\; not\ need(R'')$
$\wedge \; choose\_alternative\_plan(Goal, NewPlan, without(\{R, R'\}), with(\{R''\}))$
$\quad \Rightarrow tell(X, Agent, promise(give(R''), give(R')))$

In order to evaluate a proposal, the agent that receives the *promise* must check if it allows for a lower cost plan. To that purpose, the predicate *choose_better_plan* will have to compare the old plan and the new one with respect to the cost function. It is clear now that such function plays a central role in the whole negotiation process, ensuring monotonicity and termination. In particular, such cost function must be acceptable, with respect to the criteria given above. I will discuss this in more detail in the next section.

`%` IC 7: accept a deal: X finds the deal proposed by $Agent$ convenient
`%` (the predicate $choose\_better\_plan$ is successful only if $NewPlan$
`%` costs less than $OldPlan$)

$i\_am(X) \;\wedge\; tell(X, Agent, request(give(R)))$
$\wedge \; tell(Agent, X, promise(give(R''), give(R')))$
$\wedge \; need(R, OldPlan, Goal) \;\wedge\; have(R') \;\wedge\; not\ need(R')$
$\wedge \; choose\_better\_plan(Goal, NewPlan, OldPlan, without(\{R, R'\}), with(\{R''\}))$
$\quad \Rightarrow tell(X, Agent, accept(promise(give(R''), give(R'))))$

`%` IC 8: refuse a deal

$i\_am(X) \;\wedge\; tell(X, Agent, request(give(R)))$
$\wedge \; tell(Agent, X, promise(give(R''), give(R'))) \;\wedge\; need(R, OldPlan, Goal)$
$\wedge \; not\ choose\_better\_plan(Goal, NewPlan, OldPlan, without(\{R, R'\}, with(\{R''\}))$
$\quad \Rightarrow tell(X, Agent, refuse(promise(give(R''), give(R'))))$

In the sequel, I will denote with $\mathcal{S}$ the set of integrity constraints in $\mathcal{B}$, and with $\mathcal{S}_0$ this particular set, $\{IC_1, \ldots, IC_8\}$, describing this particular agent behaviour. In the next section I will introduce some definitions, with respect to a generic set of integrity constraints, and some properties that hold for $\mathcal{S}_0$ in particular.

5.6 Properties of S_0

It is now possible to prove a first result of the integrity constraint set $\mathcal{S}_0$, about termination or request dialogues: if two agents involved in a dialogue have as dialogue constraints in their knowledge bases a subset of $\mathcal{S}_0$, and if their program is terminating with respect to the abductive procedure used to generate dialogue moves, then the dialogue terminates in finitely many moves.

Property 3 (*termination of the request dialogue*) A request dialogue between two agents, a and b, terminates in a finite number of steps, provided the rules in $\mathcal{B}$ (including the planner) constitute a terminating program, and the ICs in $\mathcal{K}$ are a subset of $\mathcal{S}_0$.

Proof. (*sketch*) Since the dialogue is logically inferred within an abductive framework (and in particular, this framework is based on the IFF proof-procedure), where the abductive derivation is performed interleaving resolution steps made by the two agents, a dialogue will terminate if and only if such resolution terminates. In particular, the if-definitions in $\mathcal{B}$ (including those of the planner) constitute a terminating program by hypothesis, and the integrity constraints in $\mathcal{S}_0$ are such that the same constraint cannot be selected for propagation a second time in the same dialogue, with a different grounding of the body. Finally, the form of the integrity constraints is such that there cannot be a cycle involving an if-definition and an integrity constraint. Therefore, since $\mathcal{S}_0$ is a finite set, for any grounded query (i.e. for any resource request) it is possible to draw a finite IFF-tree. ■

It is also possible to prove a result of convergence, with respect to $\mathcal{S}_0$:

Property 4 (*convergence of the request dialogue*) Let us consider a request dialogue between agents a and b, started by agent a, with respect to an intention I and a resource R. If the agents' integrity constraints are a subset of $\mathcal{S}_0$, after the dialogue terminates, a's intention will be $\bar{I}$, and for a given acceptable cost function, $\mid \boldsymbol{.} \mid \; : \; \mathcal{I} \rightarrow \mathbb{R}^+$, it is always the case that $\mid \bar{I} \mid \leq \mid I \mid$.

Proof. (*sketch*) By definition, a dialogue can be *successful*, *c-successful*, or *unsuccessful*. In the first case, given that $missing(Rs) \in I, R \in Rs$, $\bar{I}$ is such that $missing(\bar{Rs}) \in \bar{I}$, therefore $Rs \supset \bar{Rs} \Rightarrow |I| > |\bar{I}|$. If the dialogue is c-successful, we have that this property holds by definition: again, we have $|I| > |\bar{I}|$. Finally, if the dialogue is unsuccessful, after it has terminated the agent intention has not changed, therefore we have $I = \bar{I}$. ■

Intuitively, there is no constraint in $\mathcal{S}_0$ that will force a change of intention if it does not produce a positive payoff in terms of cost. Therefore, a request dialogue will never result in an increased cost of the plan. In particular, if the dialogue is successful, it is possible to substitute the '$\leq$' relation with '$<$' (and in the case of the particular cost function that we have been considering, we have $| \bar{I} |=| I | -1$).

When adopting other sets of integrity constraints, different from $\mathcal{S}_0$, such as those that we will see below, we can ensure their effectiveness for negotiation purposes, by ensuring that Property 4 holds for them.

5.7 The execution model

In Section 5.4 I have already given some directions about a possible execution model of this abductive-based dialogue framework. The formulation of the knowledge of agents as given in Section 5.3 paves the way to the use of the agent architecture in [67] to model the behaviour of each agent engaged in a dialogue, compliant with the dialogue protocols expressed by the integrity constraints in $\mathcal{S}$.

The elements introduced so far are: a knowledge base $\mathcal{K}$ including a set $\mathcal{S}$ of integrity constraints, a planner and a shared blackboard for communication. The agent architecture that I propose is therefore the combination of a communication layer, a planning module, and a reasoning module. The *communication layer* implements the shared blackboard that contains the dialogue. I assume that once an agent writes a performative on the dialogue store, the recipient can read it. The writing onto the dialogue store is assumed to be incremental (at least within a dialogue, until it terminates). The *planning module* is initially used to build an intention for an agent. The *reasoning module* is in charge of activating a sequence of dialogues until either there are no more missing resources in the intention, or the intention turns out not to be viable because it is impossible to obtain all its missing resources.

While the initial plan can be obtained statically, the dialogue must be performed dynamically. Therefore, in order to select a resource, trigger the suitable *request* move, and carry on with the dialogue itself, the reasoning must be part of an *agent cycle* that combines rationality with reactivity. To this purpose an observe-think-act agent cycle à la Kowalski-Sadri [67] is adopted, where observations are blackboard events, thinking is the reasoning part that makes use of $\mathcal{S}$ and acting is writing a dialogue move.

These agents adopt the abductive proof-procedure of [48] as their underlying engine, with the if-then-rules in $\mathcal{S}$ used for forward reasoning, and the given rules (if-definitions) used backwards.

5.8 Properties of agent programs

In Section 5.6 I proved properties of a system where the agents have a program that is a subset of a given one ($\mathcal{S}_0$). I would like now to tackle one-to-one negotiation between agents that share the same language for negotiation but may adopt different negotiation policies. A single dialogue is used to obtain a resource. I describe an agent dialogue cycle that can be used to trigger a new dialogue after one is terminated, in order to allow the agent to collect all the resources needed to achieve its goal. In this way, the agents of a system produce sequences of dialogues, and in general sets of sequences of dialogues (a sequence for each agent that needs to obtain any missing resources). A single dialogue is used to obtain a resource.

In this section we consider an individual agent $x \in A$, equipped with a language for negotiation $\mathcal{L}$ and an agent program $\mathcal{S}$ that, for the purposes of this section, we assume to be grounded.

An agent program should ideally guarantee that the agent's dialogues terminate, and that the agent is able to produce one and only one move at a time, in finite time, in response to any non-final move of other agents. In this section, I show how some such desirable properties can be ensured. I discussed the 'termination' property in Section 5.6 for $\mathcal{S}_0$, and I am going to discuss it more in general further on in this chapter.

Given a *grounded* agent program $\mathcal{S}$ and a *ground* performative p, we define the set $\mathcal{S}(p)$ of *conditions* associated with p:

$\mathcal{S}(p) \stackrel{def}{=} \{C \mid \text{ there exists } (p \wedge C \Rightarrow \hat{p}) \in \mathcal{S}\}$.

Definition 28 (*determinism*) An agent $x \in A$ is *deterministic* iff, for each performative $p(t)$ which is a ground instance of a schema in $\mathcal{L}^{in}$, there exists at most one performative $\hat{p}(t+1)$ which is a ground instance of a schema in $\mathcal{L}^{out}$ such that $p(t) \wedge C \Rightarrow \hat{p}(t+1) \in \mathcal{S}$ and $\mathcal{K} \wedge p(t)$ entails C.

Namely, an agent is deterministic if for every given performative, at most one of the dialogue constraints triggered by it actually fires, in that it produces a next move.

Determinism can be guaranteed for "non-overlapping" agent programs:

Definition 29 (*non-overlapping agent program*) $\mathcal{S}$ is *non-overlapping* iff for each performative p which is a ground instance of a schema in $\mathcal{L}^{in}$, for each $C, C' \in \mathcal{S}(p)$ such that $C \neq C'$, then $C \wedge C' \equiv false$.

Namely, an agent program is non-overlapping if, for any given incoming dialogue move, there do not exist any two triggered dialogue constraints firing at the same time.

Theorem 3 If the (grounded) agent program of x is non-overlapping, then x is deterministic.

Proof. (*sketch*): for each incoming performative there exists at most one triggered dialogue constraint whose conditions are true, thus $\mathcal{K}$ only entails at most the condition of one constraint. ∎

In order for a dialogue not to end before a final move is reached, it is important that an agent be "exhaustive":

Definition 30 (*exhaustiveness*) An agent $x \in A$ is *exhaustive* iff, for each performative $p(t)$ which is a ground instance of a schema in $\mathcal{L}^{in} \setminus \mathcal{F}(\mathcal{L})$, there exists at least one performative $\hat{p}(t+1)$ which is a ground instance of a schema in $\mathcal{L}^{out}$ such that $p(t) \wedge C \Rightarrow \hat{p}(t+1) \in \mathcal{S}$ and $\mathcal{K} \wedge p(t)$ entails C.

Namely, an agent is exhaustive if, for every given performative, at least one of the dialogue constraints triggered by it actually fires, in that it produces a next move.

Exhaustiveness can be guaranteed for "covering" agent programs:

Definition 31 (*covering agent program*) Let $\mathcal{L}(\mathcal{S})$ be the set of all (not necessarily ground) performatives $p(T)$ that are triggers in dialogue constraints:

$\mathcal{L}(\mathcal{S}) \stackrel{def}{=} \{p(T) \mid \text{ there exists } p(T) \wedge C \Rightarrow \hat{p}(T+1) \in \mathcal{S}\}$. (Obviously, $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}^{in}$)

Then, $\mathcal{S} \neq \emptyset$ is *covering* iff for every performative p which is a ground instance of a schema in $\mathcal{L}^{in}$, $\bigvee_{C \in \mathcal{S}(p)} C \equiv \mathit{true}$ and $\mathcal{L}(\mathcal{S}) = \mathcal{L}^{in} \setminus \mathcal{F}(\mathcal{L})$.

Namely, an agent program is covering if, for any given incoming dialogue move that is not a final move, there exists at least one triggered dialogue constraint which actually fires.

Theorem 4 If the (grounded) agent program of x is covering, then x is exhaustive.

Proof. (*sketch*): for each incoming performative there exists at least one dialogue constraint which is triggered and whose condition is true, thus $\mathcal{K}$ entails the condition of at least one constraint. ∎

If an agent is exhaustive and deterministic, then at each step of the dialogue the agent will produce *exactly one* reply to a (non final) move made by the other agent. If both agents involved in a dialogue are exhaustive and deterministic, then at each step exactly one agent is guaranteed to produce only one dialogue move, unless a final move is made.[5]

The IFF proof procedure and the agent cycle have features that allow us to produce dialogues as defined earlier, given agent programs that are exhaustive and deterministic.

[5] Note that I use *transaction*, rather than *real*, time, i.e., I assume that there is no time-point between two *consecutive* dialogue moves in a dialogue.

5.9 Agent varieties: concrete examples of agent programs

In this section I show some concrete agent programs and I discuss which properties hold for them and which ones do not. All the examples refer to a given agent a, equipped with a language for negotiation $\mathcal{L}_2 = \mathcal{I}(\mathcal{L}_2) \cup \mathcal{F}(\mathcal{L}_2)$, where:

- $\mathcal{I}(\mathcal{L}_2) = \{\ \mathit{tell}(X, Y, \mathbf{request}(\mathbf{give}(\mathit{Resource})), T)\ \}$;
- $\mathcal{F}(\mathcal{L}_2) = \{\ \mathit{tell}(X, Y, \mathbf{accept}(\mathit{Move}), T),\ \mathit{tell}(X, Y, \mathbf{refuse}(\mathit{Move}), T)\ \}$.

Agent program `P0`

$$
\begin{array}{l}
\mathit{tell}(X, a, \mathbf{request}(\mathbf{give}(R)), T)\ \wedge\ \mathit{have}(R, T) \\
\ \Rightarrow \mathit{tell}(a, X, \mathbf{accept}(\mathbf{request}(\mathbf{give}(R))), T+1) \\
\mathit{tell}(X, a, \mathbf{request}(\mathbf{give}(R)), T)\ \wedge\ \mathit{need}(R, T) \\
\ \Rightarrow \mathit{tell}(a, X, \mathbf{refuse}(\mathbf{request}(\mathbf{give}(R))), T+1)
\end{array}
$$

This program is neither deterministic nor exhaustive. In fact, if a has and needs the resource that is requested at time T, the program allows to return two answers at time $T+1$, namely both *accept* and $refuse$, while if a does not have R, then the program will not be able to return any answer at time $T+1$.

Agent program `P1`

$$
\begin{array}{l}
\mathit{tell}(X, a, \mathbf{request}(\mathbf{give}(R)), T)\ \wedge\ \mathit{have}(R, T) \\
\ \Rightarrow \mathit{tell}(a, X, \mathbf{accept}(\mathbf{request}(\mathbf{give}(R))), T+1) \\
\mathit{tell}(X, a, \mathbf{request}(\mathbf{give}(R)), T)\ \wedge\ \mathit{not\ have}(R, T) \\
\ \Rightarrow \mathit{tell}(a, X, \mathbf{refuse}(\mathbf{request}(\mathbf{give}(R))), T+1)
\end{array}
$$

An agent programmed with `P1` simply gives away the requested resource if it has it, no matter whether it needs it or not. This program is exhaustive and deterministic.

Agent program `P2`

$$
\begin{array}{l}
\mathit{tell}(X, a, \mathbf{request}(\mathbf{give}(R)), T)\ \wedge\ \mathit{have}(R, T)\ \wedge\ \mathit{not\ need}(R, T) \\
\ \Rightarrow \mathit{tell}(a, X, \mathbf{accept}(\mathbf{request}(\mathbf{give}(R))), T+1) \\
\mathit{tell}(X, a, \mathbf{request}(\mathbf{give}(R)), T)\ \wedge\ \mathit{not\ have}(R, T) \\
\ \Rightarrow \mathit{tell}(a, X, \mathbf{refuse}(\mathbf{request}(\mathbf{give}(R))), T+1) \\
\mathit{tell}(X, a, \mathbf{request}(\mathbf{give}(R)), T)\ \wedge\ \mathit{need}(R, T) \\
\ \Rightarrow \mathit{tell}(a, X, \mathbf{refuse}(\mathbf{request}(\mathbf{give}(R))), T+1)
\end{array}
$$

An agent programmed with `P2` gives away the requested resource only if it has it and it does not need it. This program is exhaustive and deterministic.

While the properties of exhaustiveness and determinism can guarantee that an agent program is able to produce a dialogue in accordance with the definition introduced in Section 5.4, the property of self-interested individual rationality can be used to forecast the status of the system after a set of negotiation dialogue sequences, as I will show in the sequel.

5.10 Dialogue sequences

In this section I model the negotiation process of an agent by means of "sequences of (request) dialogues" with respect to an intention, I, of a given agent x. The request dialogues aim at obtaining *all* the missing resources in I. I assume that all dialogues between x and any other agents are *atomic*, i.e., it is not possible for x to be engaged in two different dialogues at the same time.

In order to start a request dialogue with respect to an intention I, the set of missing resources of I must be non-empty, i.e., $missing(Rs) \in I, Rs \neq \emptyset$.

Intuitively, given a goal G such that $goal(G) \in I, missing(Rs) \in I$ and $plan(p) \in I$, a sequence of dialogues with respect to I aims at decreasing the cost of I, either by obtaining one by one all the missing resources and making p executable, or by finding an alternative plan (and thus an alternative intention) which can be made executable. Indeed, after each dialogue, the intention might change. I associate with a dialogue sequence $d_1, d_2, \ldots, d_n$ the corresponding sequence of intentions $I_1 = I, I_2, \ldots, I_{n+1}$. In general, after a request dialogue $d_i, i < n$, with respect to an intention I_i and a resource R, is terminated, the agent will have an intention I_{i+1}. All such intentions have the same goal, but possibly different plans and missing/available resources.

Definition 32 (*sequence of dialogues*) A *sequence of dialogues* $s(I)$ with respect to an intention I of an agent x with $goal(G) \in I$ is an ordered set $\{d_1, d_2, \ldots, d_n, \ldots\}$, associated with a sequence of intentions I_1, I_2, …, I_{n+1}, … such that

- for all $1 \leq i$, d_i is a request dialogue between x and some other agent (not necessarily the same for all i), with respect to a resource R_i such that $R_i \in Rs_i$, $missing(Rs_i) \in I_i$, I_i is the intention of agent x, $goal(G) \in I_i$, and I_{i+1} is the intention after dialogue d_i;
- $I_1 = I$;

We are interested in dialogue sequences that terminate:

Definition 33 (*termination of a dialogue sequence*) A sequence of dialogues $\{d_1, d_2, \ldots, d_n\}$ with respect to an initial intention I of an agent x and associated with the sequence of intentions $I_1, I_2, \ldots, I_{n+1}$ is *terminated* iff there exists no possible request dialogue with respect to I_{n+1} that x can start.

Termination could occur for two reasons: either there are no missing resources in I_{n+1} or x could not obtain all the resources in I_{n+1} but "decided" to give up the intention without forming a new one for the time being. Accordingly, we can assess the success of dialogue sequences:

Definition 34 (*success of a dialogue sequence*) A terminated sequence of dialogues $\{d_1, d_2, \ldots, d_n\}$ with respect to an initial intention I of an agent x and associated with a sequence of intentions $I_1, I_2, \ldots, I_{n+1}$ is *successful* if I_{n+1} has an empty set of missing resources; it is *unsuccessful* otherwise.

In the remainder of this section I will discuss how dialogue sequences can be generated by agents, by an "agent dialogue cycle" which repeatedly generates individual dialogues until a terminated sequence is produced. I will call SD and SI, respectively, the sequence of dialogues and intentions, produced within the dialogue cycle:

Definition 35 (*agent dialogue cycle*) Given an initial intention I of agent x, containing a set of missing resources Rs, the *agent dialogue cycle* is the following:

1. set $SD = \emptyset$ and SI to $\{I\}$;
2. until $Rs = \emptyset$ repeat
   (a) select a resource $R \in Rs$ and update Rs to $Rs - \{R\}$;
   (b) until a successful or c-successful dialogue d is found, between x and some other agent, repeat
      i. select an agent $y \in A - \{x\}$ such that there does not exist in SD a request dialogue between x and y with respect to R;
      ii. if such y does not exist, then **failure** and restore Rs to $Rs \cup \{R\}$;
      iii. construct a request dialogue d between x and y with respect to R and I, returning an intention I';
   (c) add d to SD and I' to SI;
   (d) update I to I' and Rs to Rs' such that $missing(Rs') \in I'$;
3. **success**.

Note that the above agent dialogue cycle does not allow for the plan in an intention to change unless the change is achieved by an individual dialogue between x and the chosen y.

Note also that the above agent dialogue cycle has the following properties:

- no agent is asked twice for the same resource within the dialogue sequence;
- if a resource is not obtained from one agent, then it is asked from some other agent, if any;
- if a resource is not obtained after asking all agents, then the agent dialogue cycle terminates with **failure**.

The idea behind the third property is that the agent will not carry on asking for the other resources, since, as at least one resource in the current intention cannot be obtained, the intention will not possibly be executable.

Theorem 5 Given an agent $x \in A$, if x's agent dialogue cycle returns **success** then there exists a successful dialogue sequence with respect to the initial intention I of x.

Proof. (*sketch*): The dialogue sequence SD, associated with the sequence of intentions SI, as generated by the agent dialogue cycle at step 2(c) during the last execution of the loop at step 2, is a successful dialogue sequence.
■

If the agent $x \in A$ is convergent, then all dialogues of a sequence with respect to an intention I of x will result in a non-increased cost of x's intention. Thus, it is possible to set an upper bound to the number of dialogues in a sequence initiated by x:

Theorem 6 Given an agent x with intention I, and a successful dialogue sequence $s(I)$ generated by x's dialogue cycle, if x is convergent, then the number of dialogues in $s(I)$ is bounded by $m \cdot |Rs|$, where $missing(Rs) \in I$ and $|A \setminus \{x\}| = m$, A being the set of agents in the system.

Proof. (*sketch*): $|Rs|$ can never increase, since x is convergent. In the worst case, x will have to request each resource $R \in Rs$ to all the other m agents of the system, thus generating m dialogues with respect to each resource. The cardinality of the set of missing resources decreases by only one unit after each successful dialogue.
■

5.11 Using dialogue sequences for resource reallocation

In this section, I study the use of multiple sequences of dialogues for the solution of the well-known and widely studied *resource reallocation problem* [8, 91]. The problem can be rephrased in this system as follows.
Definition 36 (*resource reallocation problem – r.r.p.*) Given an agent system A, with each agent $x \in A$ equipped with a knowledge base $\mathcal{K}(x)$ and an intention $\mathcal{I}(x)$,

- the *r.r.p. for an agent* $x \in A$ is the problem of finding a knowledge base $\mathcal{K}'(x)$, and an intention $\mathcal{I}'(x)$ (for the same goal as $\mathcal{I}(x)$) such that $missing(\emptyset) \in \mathcal{I}'(x)$.

- the *r.r.p. for the agent system* A is the problem of solving the r.r.p. for every agent in A.

A *r.r.p.* is *solved* if the required (sets of) knowledge base(s) and intention(s) is (are) found.

The new knowledge bases will differ from the old ones in the resources available to the agents and in the intentions. Typically, agents need to acquire new resources from other agents so that their intentions have no missing resources. In this framework, agents can obtain the new resources by means of dialogue sequences.

Note that, the knowledge and intention of an agent x might change not only as a result of dialogues which have been initiated by x, but also as a result of dialogues initiated by other agents, but engaging x. Indeed, in such more 'passive' dialogues, x might agree to give away a resource that it needs. This 'double change' of knowledge and intentions (of both agents involved in a dialogue) plays a role when agents need to engage in multiple concurrent dialogue sequences in order to achieve their goal. [6]

In the remainder of this section, I will assume *generalised notions of (terminated) dialogue*, modifying the intentions of both agents engaged in it, and *dialogue sequence*, modifying the intentions of all agents involved in all dialogues in it. Then, the notion of *convergence* can be generalized as well, so that the cost of both agents' intentions returned by a dialogue is not increased, with respect to the cost of the agents' initial intentions. Note that convergent agents, in this generalized sense, correspond to *self-interested individual rational* agents as proposed in the literature (see Sandholm [90]).

I consider two properties of agents:

- *correctness*: if the dialogue cycle of some agent x returns success, then the r.r.p of x is solved, and, if the dialogue cycle of every agent in A returns success, then the r.r.p. of the agent system is solved;

- *completeness*: if there exists a solution to the r.r.p of an agent system, then the dialogue cycle of every agent in A returns success, producing such a solution.

The proof of correctness for a single agent is straightforward, by definition of successful dialogue sequence. In the case of an agent system, instead, correctness as given above can be proven only for convergent agents.

Theorem 7 (*Correctness of the agent dialogue cycle with respect to the r.r.p.*) Let A be the agent system, with the agent programs of all the agents in A being convergent. If all agent dialogue cycles of all agents in A return **success** then

[6] Note that this is not in contrast with the assumption that agents engage in at most one dialogue at a time. As I assume that dialogues are atomic, there will always be at most one dialogue taking place at any time.

the r.r.p. for the agent system is solved.

Proof. (*sketch*): Since all agent programs are convergent, there will never occur a dialogue after which, and as a consequence of which, either of the two agents involved has more missing resources in its intention than it had before. Therefore, having m successful dialogue sequences, one for each agent in A, is in the worst case equivalent to having m *independent* dialogue sequences, after each of which the relevant agent will have solved its r.r.p., thus solving the r.r.p. of A.

■

In order to address completeness I need to deal with the issue of how intentions change, which is beyond the scope of this thesis. But to illustrate the problem, consider the following example. Let $A = \{a, b, c\}$ be an agent system, with knowledge bases containing:

$a : \mathcal{R}(a) = \emptyset$, $\mathcal{I}(a) = \{$ $\texttt{plan}(\{\texttt{p(a)}\})$, $\texttt{missing}(\{\texttt{r}_2\})$, $\texttt{available}(\emptyset)$, $\texttt{goal}(\{\texttt{g(a)}\})\}$
$b : \mathcal{R}(b) = \{r_2\}, \mathcal{I}(b) = \{$ $\texttt{plan}(\{\texttt{p(b)}\}), \texttt{missing}(\{\texttt{r}_3\}), \texttt{available}(\{\texttt{r}_2\}), \texttt{goal}(\{\texttt{g(b)}\})\}$
$c : \mathcal{R}(c) = \{r_3, r_4\}, \mathcal{I}(c) = \{$ $\texttt{plan}(\{\texttt{p(c)}\}), \texttt{missing}(\emptyset), \texttt{available}(\{\texttt{r}_3\}), \texttt{goal}(\{\texttt{g(c)}\})\}$

where $\mathcal{R}(x)$ stands for the resources available to agent x, as described within x's knowledge base. Let us assume that there exists an alternative plan $\texttt{p}'\texttt{(b)}$ for b's goal $\texttt{g(b)}$, and therefore an alternative intention, $\mathcal{I}'(b)$:
$\mathcal{I}'(b) = \{$ $\texttt{plan}(\{\texttt{p}'\texttt{(b)}\}), \texttt{missing}(\{\texttt{r}_4\}), \texttt{available}(\emptyset), \texttt{goal}(\{\texttt{g(b)}\})\}$

It is easy to see that there exists a possible resource reallocation that makes all goals achievable (r_2 to a, r_4 to b and r_3 to c), but in order to be found by negotiation, b must be able to change its intention to $\mathcal{I}'(b)$.

I consider a weak notion of completeness, whereby agents have, as their own initial intentions, the intentions whose plan does not need to be changed by negotiation.

Definition 37 (*weak completeness*) Let A be an agent system consisting of n agents. Let $\mathcal{R}(A)$ be the union of all resources held by all agents in A, and $\mathcal{R}(\mathcal{I}(A))$ be the union of all resources needed to make all agents' initial intentions $\mathcal{I}(A)$ executable.

A is *weakly complete* if, given that $\mathcal{R}(\mathcal{I}(A)) \subseteq \mathcal{R}(A)$, then **there exist** n successful dialogue sequences, one for each agent in A, such that the intentions $\mathcal{I}'(A)$ returned by the sequences have the same plans as $\mathcal{I}(A)$ and all have an empty set of missing resources.

Theorem 8 (*weak completeness with respect to* `P1` + `P2`) Let A be an agent system consisting of n agents. Then, if the agent program of every agent in A is either `P1` or `P2`, then A is weakly complete.

Proof. (*sketch*): The sub-system composed by the agents programmed with `P2` is weakly complete, therefore the agents programmed with `P2` will obtain all

missing resources, either from the agents programmed with P1 or from the ones programmed with P2. The agents programmed with P1 will obtain the missing resources either from the agents programmed with P2 that have such resources and do not need them, or by the agents programmed with P1, which represent also a weakly complete sub-system.
■

Thus, I have shown that the r.r.p. can be solved within this approach, by diverse agents, equipped with different agent programs, negotiating with one another. Note that such diversity can be used to implement a concept of priority between agents. Indeed, agents programmed with P2 always retain all resources they need, whereas agents programmed with P1 give them away, if asked. Thus, P2-agents have a higher priority than P1-agents, with respect to the allocation of resources.

5.12 On the termination of agent dialogues

In this section and in the following I tackle the problem of termination of agent dialogues [99]. I start by discussing the meaning of termination in agent dialogue, and show a trade-off between ensuring termination of dialogues, and therefore robustness in the agent system, and achieving completeness in problem solving. Then, I show how such ideas apply in practice to the concrete case of the negotiation framework proposed above. I show how the agent programs can be transformed in order to ensure termination in the negotiation process, and I define three different degrees of such transformation. I show how they make the agents more robust against possible situations of non-terminating dialogues, while reducing the class of solutions that can be found in a specific application domain, that of resource reallocation.

5.12.1 Agent dialogue termination: robustness vs. completeness

In [27] agent societies are categorized in terms of openness, flexibility, stability and trustfulness, and it is claimed that whereas open societies support openness and flexibility, closed societies support stability and trustfulness. The author suggests two classes of societies (semi-open and semi-closed), that balance the trade-off between these aspects, because in many situations there is a need for societies that support all of them.

In the case of dialogue, the problem of determining such trade-off still holds, because the dialogue can be used as a means to let heterogeneous agents communicate, despite the differences among them, and without necessarily sticking to a given protocol. On the other hand, if we let agents openly join societies, with no control on the individuals that access them, problems could arise from their diversity. For instance, dialogues can last forever.

The concept of *termination* of a negotiation dialogue can be recovered into the idea that at a certain point an agent makes a final move [104]. Of course, the other agent is supposed to recognize that such move is intended to terminate the dialogue. If no agent makes any final move, both agents could keep exchanging messages, without getting to an end.

Example 8
$tell(a, b, hello, 1)$
$tell(b, a, hello, 2)$
$tell(a, b, hello, 3)$
...

Example 8 shows a dialogue between two (particularly overpolite) agents that keep exchanging greetings. The situation of Example 8 could be due to the fact that both agents' programs force them to reply to an incoming greeting with an equal greeting. We can imagine that loop conditions of the like could unpredictably arise each time we put together, in an open society, agents that were independently programmed. An obvious solution to this problem could be to force agents not to tell the same thing twice. But this measure does not really solve the problem, as the following example shows: agents could keep exchanging slightly different messages, and still get stuck in a loop condition.

Example 9
$tell(a, b, hello(1), 1)$
$tell(b, a, hello(2), 2)$
$tell(a, b, hello(3), 3)$
...

Then, we could think to introduce a more restrictive measure, e.g., based on message patterns, that prevents agents from telling a message whose "pattern" is the same as a previous one in the same dialogue.[7] But this could result in preventing agents from finding solutions (or *agreements*, in the case of negotiation), that could be found otherwise. The example below shows a possibly successful dialogue that would solve a resource reallocation problem (agent a obtains a *screw* from b and therefore can execute a plan to achieve her goal of hanging a picture). Such dialogue would not be permitted if agent b was prevented from making the move 'propose an exchange' (such is the meaning of the *promise* performative, inherited from [3]) twice.

Example 10
$tell(a, b, request(give(nail)), 1)$
$tell(b, a, challenge(request(give(nail))), 2)$
$tell(a, b, justify(request(give(nail)), \ldots), 3)$

[7] the pattern could be, in this case: *tell(a, b, hello(_), _)*, where the *underline* indicates whatever ground expression.

$tell(b, a, promise(give(bluetac), \ldots), 4)$
$tell(a, b, refuse(promise(give(bluetac), \ldots)), 5)$
$tell(b, a, promise(give(screw), \ldots), 6)$
$tell(a, b, accept(promise(give(screw), \ldots)), 7)$

Still, unless we consider very generic (and therefore very restrictive) patterns, the threat of non-termination remains. The following example could evoke a familiar situation to those who have spent some moments of their life dealing with small children... In the example, the *challenge* performative, also derived from [3], has the meaning of asking for a justification of what the dialogue partner just said.

Example 11
$tell(a, b, request(give(nail)), 1)$
$tell(b, a, challenge(request(give(nail))), 2)$
$tell(a, b, justify(request(give(nail)), \ldots), 3)$
$tell(b, a, challenge(justify(request(give(nail)), \ldots)), 4)$
$\ldots$

In the end, we realize intuitively what follows: the more we reduce the set of dialogue moves that the agents can exchange in the course of a dialogue, the more we reduce the universe of reachable solutions of a negotiation problem. I think that the choice about to which extent the dialogue should be constrained to certain patterns must be left to the system designer(s).

Based on the dialogue framework introduced before, I introduce several syntactic transformations of such programs, that make them robust in various degrees against non-termination, and reflect the above discussed trade-off. I also prove a theorem that determines a bound in the maximum length of a dialogue, measured in terms of number of exchanged messages, and I extend such result to dialogue sequences.

5.12.2 IFF-terminating abductive logic programs

The IFF proof procedure has already been introduced in Chapter 2. The dialogue can be considered as a particular IFF derivation, obtained by interleaving resolution steps made by the two agents. Therefore, I will tackle the issue of proof termination, and give a characterization of the class of *IFF-terminating* programs, i.e., of those abductive logic programs for which all IFF-trees for grounded queries are finite. Since it is beyond the scope of this research thesis to give a formal proof of the termination results for the IFF in the general case, we restrict to the case of terminating dialogue programs, and give some intuition of the general case.

Intuitively, the reason why a program is not IFF-terminating can be recovered into the presence in the program of rules / constraints whose combination leads to infinite propagation or unfolding. It is possible to identify three cases, that can be generalized.[8] In the following, `p` and `q` represent literals.

1) *unfolding + unfolding*
   `p` $\leftrightarrow$ `q`
   `q` $\leftrightarrow$ `p`

2) *unfolding + propagation*
   `p` $\leftrightarrow$ `q`
   `q` $\Rightarrow$ `p`

3) *propagation + propagation*
   `p` $\Rightarrow$ `q`
   `q` $\Rightarrow$ `p`

In all cases, `p` unfolds / propagates to `q` and vice versa, ad infinitum. In order to characterize a class of programs that terminate, I define the property of *acyclicity*, tailored to the case of ALP in relationship to the IFF proof procedure (IFF-acyclicity). In fact, if an ALP is IFF-acyclic, then it is IFF-terminating. An ALP is IFF-acyclic if we can found a *level mapping* $||$ such that:

1. for every ground instance of every clause (if-definition) in P, say $A \leftarrow L_1, \ldots, L_i, \ldots, L_n$, we have $|A| > |L_i|$ (i.e., P is acyclic)

2. for every ground instance of every integrity constraint in IC, say $L_1, \ldots, L_j, \ldots, L_m \Rightarrow A$, we have:

   - $|L_j| > |A|$
   - if K is a negative literal, say *not* M in the body of the integrity constraint, then $|L_j| > |M|$

The presence of such level mapping ensures that the situations above do not occur in an agent program. We will call IFF-acyclic programs *acceptable*: in fact, we cannot guarantee termination if an agent gets stuck in an infinite branch of the derivation tree, producing therefore no dialogue move at all. In the following, we will always require that the agent programs are acceptable. This work builds on these results and identifies a class of agent programs that ensure the termination of negotiation dialogues and sequences of dialogues.

5.12.3 Terminating dialogue programs

It is possible now to extend the termination results of 5.12.2 to dialogue programs.

The possible reasons for an infinite derivation tree are still, *mutatis mutandis*, those of Section 5.12.2. Let us consider again the three cases, and see if they can occur when the knowledge is distributed between two agents, a and b. We put

[8]Here, for the sake of simplicity, I consider only ground programs, thus assuming that P and IC have already been instantiated. The results could be generalized to non-ground programs.

on the left side of each rule / constraint the name of the agent whose program contains it. I assume that the agents' programs are *acceptable.*

1) *unfolding + unfolding*
   $[\mathtt{a}]\ \mathtt{p} \leftrightarrow \mathtt{q}$
   $[\mathtt{a}]\ \mathtt{q} \leftrightarrow \mathtt{p}$

2a) *unfolding + propagation*
   $[\mathtt{a}]\ \mathtt{p} \leftrightarrow \mathtt{q}$
   $[\mathtt{a}]\ \mathtt{q} \Rightarrow \mathtt{p}$

2b) *unfolding + propagation*
   $[\mathtt{a}]\ \mathtt{p} \leftrightarrow \mathtt{q}$
   $[\mathtt{b}]\ \mathtt{q} \Rightarrow \mathtt{p}$

3) *propagation + propagation*
   $[\mathtt{a}]\ \mathtt{p(T)} \ \wedge\ \mathtt{C_p[T]} \Rightarrow \mathtt{q(T+1)}$
   $[\mathtt{b}]\ \mathtt{q(T)} \ \wedge\ \mathtt{C_q[T]} \Rightarrow \mathtt{p(T+1)}$

(1) and (2a) are both forbidden by the program *acceptability* requirement. (2b) does not represent a possible cause of non-termination:[9] since q is not abducible, therefore it is not possible to communicate it to b. As for the third case, let us consider, as p(T), *tell(b, a, Subject, T)*, and as q(T), *tell(a, b, Subject, T).* This is apparently a threat for termination. For instance, this is the case in Example 8, where *Subject* is *hello.* We could therefore introduce some restrictions to the dialogue protocols, in order to prevent such situation, at the cost of a reduction of the space of reachable solutions, as explained before by examples.

5.12.4 Three degrees of restrictions

In order to try and prevent *propagation* from causing infinite dialogue move generation (case 3), we should make sure that the same integrity constraint of an agent program is not triggered infinitely many times. To this purpose, I define a transformation $\mathcal{T}$ that maps an element $\mathcal{S} \in \mathcal{P}$ of the domain of (acceptable, exhaustive and deterministic) agent programs into another element, $\mathcal{T}(\mathcal{S})$ in the same domain. $\mathcal{T}$ is defined as follows:

Definition 38 (*agent program transformation*) Given a language $\mathcal{L}$, an agent x and an agent program $\mathcal{S}(x) \in \mathcal{P}$, the transformation $\mathcal{T}$ with respect to a given set of literals (*restriction*) $p_{check}(T)$, $\mathcal{T}(p_{check}(T)) : \mathcal{P} \rightarrow \mathcal{P}$ is defined in the

[9] The only way to pass the computation thread from an agent a to her dialogue partner b is through an abducible representing a dialogue move in the head of one of a's dialogue constraints.

following way:

$$
\begin{array}{l}
\forall ic \in \mathcal{S}(x), ic = p(T) \ \wedge \ C[T] \ \ \Rightarrow \ \ \hat{p}(T+1), \mathcal{T}(ic) \stackrel{def}{=} \\
\quad \left\{ \begin{array}{llll} p(T) \ \wedge \ C[T] \ \wedge \ not \ p_{check}(T) & \Rightarrow & \hat{p}(T+1) \\ p(T) \ \wedge \ C[T] \ \wedge & p_{check}(T) & \Rightarrow & f(T+1) \end{array} \right. \\
where \ f(T+1) \in \mathcal{F}(\mathcal{L}), \qquad\qquad if \ \hat{p}(T+1) \notin \mathcal{F}(\mathcal{L}); \\
\quad ic, \qquad\qquad otherwise.
\end{array}
$$

Therefore, there is a 1-1 correspondence between a *restriction* $p_{check}(T)$ and a transformation function $\mathcal{T}$. I will call *restricted according to* $\mathcal{T}$ the programs that are elements of the co-domain of $\mathcal{T}$. If an acceptable program is restricted according to $\mathcal{T}$, when the partner agent produces a move triggering a *dialogue constraint* whose condition and restriction are both verified, the agent will jump to a final state, thus interrupting the dialogue. It is easy to prove that, given a transformation function $\mathcal{T}$ associated with a restriction $p_{check}(T)$, if the $p_{check}(T)$ is ground for all possible instantiations of $p(T) \ \wedge \ C[T]$, $\mathcal{T}$ transforms exhaustive and deterministic programs into exhaustive and deterministic programs, i.e., $\mathcal{T}$ maps from $\mathcal{P}$ to $\mathcal{P}$.

The choice of the restriction can be made in several ways; I will define here three different kinds of restrictions, that formally reflect the considerations of Section 5.12.1.

(i) The check is made on ground instances of predicates,

(ii) The check is made on predicate patterns,

(iii) The check is made against an ordering.

Let us consider them one by one. Case (i), *check made on ground instances of predicates*, restricts the applicability of a dialogue constraint by preventing it from being triggered twice *by the same instance of dialogue move* at different times. This is in line with the characterization of dialogue given in [89], and prevents situations of infinite 'pure' loop, such as the one in Example 8, generated by the constraint $tell(X, a, hello, T) \ \Rightarrow tell(a, X, hello, T+1)$ of agent a. The restriction in this case could be:

$$p^{(i)}_{check}(T) = tell(X, a, hello, T1) \ \wedge \ T1 < T$$

This does not guarantee termination in a finite number of steps, though, as shown by Example 9. The *check on predicate patterns* could be implemented in that case by the restriction

$$p^{(ii)}_{check}(T) = tell(X, Y, hello(\_), T1) \ \wedge \ T1 < T$$

The case of check made on predicate patterns is a more restrictive policy, that has the already mentioned drawbacks and limitations. In particular, the situation of Example 11 could be caused by an agent that contains in her program

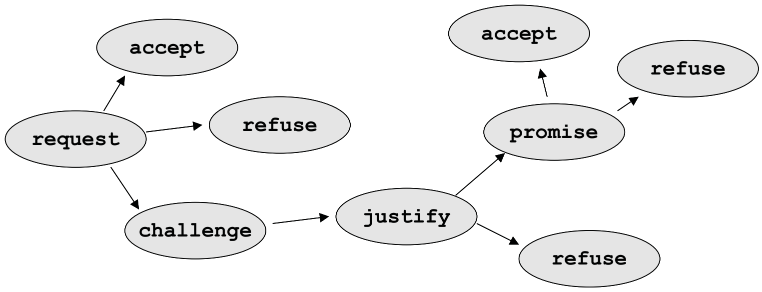

Figure 5.3: Dialogue tree.

the following dialogue constraint: $tell(b, a, Anything, T) \Rightarrow tell(a, b, challenge(Anything), T + 1)$.

A solution to this could be to establish an *ordering* among the dialogue constraints of an agent. Before going on, let us point out that the problem of non-termination is of the agent that starts the negotiation dialogue, although these results are independent on this. In fact, if it is true that the dialogue can be terminated by either agent, on the other hand, broadly speaking, the one that started it is the one that we expect to be waiting for a reply, and not *vice versa*.

Now, the intuition is that the agent that started the negotiation process, let us say a, can ideally draw a tree of possible dialogues, that has as a root his initial move, let us say, $tell(a, b, request(\ldots), 0)$. A correct tree could be drawn if a knew b's program exactly, which is not an assumption we want to make. However, in first approximation, a can assume that b has the same constraints as she has. Then, a can generate a tree that has as nodes the possible dialogue moves, and as branches the integrity constraints that lead from one move to another one. An example of such tree, for the language $\mathcal{L}_1$ and the negotiation program defined in [89], is that of Figure 5.3.

The purpose of drawing a tree, that goes from an initial move to some possible final moves, is to have an ordering function, that allows to order the dialogue moves, and consider an ongoing dialogue *valid* as far as the tree or graph is explored in one direction (the one that leads to final moves). It is important to notice that not all agent programs in $\mathcal{P}$ allow us to draw a tree: in order to do that, such ordering function must exist. We call the existence of such function *acyclicity*, as we did in Section 5.12.2 in the case of IFF-terminating programs. If such function does not exist, it is not possible to adopt policy *(iii)*, and it is easy to imagine that a dialogue between two agents having both a non-acyclic program will not likely terminate. More formally, an instance of ordering function, that we call (*rank* function) is defined as follows:

Definition 39 (*rank function*) A *rank* function, mapping from a language $\mathcal{L}$ to the set of natural numbers $\mathbb{N}$, $rank : \mathcal{L} \rightarrow \mathbb{N}$, is *procedurally* defined for a given agent program $\mathcal{S} \in \mathcal{P}$ as follows in two steps. First we *label* all the performatives of $\mathcal{L}$, by applying one of the following rules, until no rule produces any change in the labeling:

- for all $p \in \mathcal{L}$ that have not been labeled yet, if $p \in initial(\mathcal{L})$, then $label(p) = 0$;
- for all $p \in \mathcal{L}$ that have not been labeled yet, if $p \notin initial(\mathcal{L})$ and $\exists ic \in \mathcal{S} : ic = p \wedge C \Rightarrow (\bar{p})$, and $label(\bar{p}) = r$, then $label(p) = r + 1$
- for all $p \in \mathcal{L}$ that have already been labeled, let $label(p)$ be r. Then, if $\exists ic \in \mathcal{S} : ic = (\bar{p}) \wedge C \Rightarrow p$ such that $label(\bar{p}) \leq r$, then $label(\bar{p}) = r + 1$

If it is not possible to apply such labeling to the language (i.e., if it is not possible to complete this procedure in finitely many steps), it means that the program is not acyclic. Otherwise, after labeling the language, let R be $\max_{p\in\mathcal{L}} label(p)$ (R is finite). Then, $rank$ is defined as $rank(p) = R - label(p)$.

Once a rank function $rank(p, n)$, $n \in \mathbb{N}$ is defined for all $p \in \mathcal{L}$, the restriction turns out:

$$p^{(iii)}_{check}(T) = rank(p(T), R) \ \wedge \ q(T1) \ \wedge \ T1 < T \ \wedge \\ rank(q(T1), R1) \ \wedge \ R1 \leq R$$

That is, a move of higher rank has been made after a move of lower rank. It is worth to notice that the introduction of the restrictions does not modify the existing language ranking.

We will call this policy *check against an ordering*. It is more restrictive than the previous ones, since it does not allow jumping backwards from one branch to another one, and, again, some more possibly successful dialogues could be rejected. On the other hand, if applied to acceptable, exhaustive, and deterministic programs, it is enough to ensure termination, as stated by the following theorem:

Theorem 9 (*Finite termination of a dialogue with check against an ordering*) Let a and b be two agents provided with *acceptable*, *exhaustive*, and *deterministic* programs. Let a's program be *restricted* according to a *check against an ordering* policy. Therefore, a dialogue d between a and b *will terminate* before finitely many moves. In particular, if R is the maximum rank of a dialogue move, d will have *at most* $R + 2$ moves.

Proof. The proof for such theorem is given inductively. Given a dialogue $d = \{p_1, p_2, \ldots, p_j, \ldots\}$, between agents a and b, and started by a, let R be the maximum rank of a dialogue move in a's program. By definition of $rank$, all nodes ranked 0 must be leaves, and therefore final moves, while all the leaves ranked R are initial moves. Then we have, $\forall p_j$:

- if p_j is final, the dialogue is terminated;

- if p_j is not final, with $rank(p_j) = r_j > 0$, and it is uttered by b (i.e., j is even), then p_{j+1} must be computed by a, and will be either a final move, or will be ranked $rank(p_{j+1}) = r_j - 1$;
- if p_j is not final, with $rank(p_j) = r_j > 0$, and it is uttered by a (i.e., j is odd), then p_{j+1} must be computed by b, and it will be either a final move, or a move p_{j+1} ranked $rank(p_{j+1})$. If $rank(p_{j+1}) \geq rank(p_j)$, then the next move p_{j+2} will be final and the dialogue will terminate; otherwise p_{j+2} will be ranked $rank(p_{j+2}) = rank(p_{j+1}) - 1 \leq rank(p_j) - 2$;
- $rank(p_1) \leq R$ by definition.

Therefore, each move p_j is ranked r, with $r \leq R + 1 - j$. Now, if in the dialogue there are two moves with the same rank, the next move will be the last one. Since there cannot be more than two moves with the same rank, and the maximum rank is R, the maximum number of moves computed in a dialogue is $R + 2$. For all non-final move p_j uttered by either agent, there exists a (unique) next move p_{j+1}, since both agent programs are exhaustive and deterministic. Moreover, the reasoning required by either party to compute a dialogue move terminates in a finite number of steps, since both programs are acceptable. Therefore, the dialogue terminates in at most $R + 2$ moves. ■

5.12.5 Termination of dialogue sequences

This termination result can be easily extended to the case of dialogue sequences, formally defined in Section 5.10.

Definition 40 (*termination of a sequence of dialogues*) A sequence of dialogues $s(\mathcal{I})$ with respect to an initial intention $\mathcal{I}_a$ of an agent a is *terminated* when, given that $\mathcal{I}_n$ is the intention after d_n, there exists no possible request dialogue with respect to $\mathcal{I}_n$ that a can start.

This could be due to two reasons: either after s there are no more missing resources in the intention, i.e. $cost(\mathcal{I}_n) = 0$, or $cost(\mathcal{I}_n) > 0$ and in a's program there is no constraint that can start a request.

In order to ensure termination, we could program the agent such that after every single dialogue the set of missing resources rather shrinks than grows in size: $\forall i > 0, cost(\mathcal{I}_i) > (\mathcal{I}_{i-1})$. If an agent program is such that a dialogue can only decrease the cost of an intention, we call such agent *self-interested rational.* In that case, given a system of $n + 1$ agents and an intention $\mathcal{I}$, the length $length(s(\mathcal{I}))$ of a sequence $s(\mathcal{I})$ of dialogue with respect to an intention $\mathcal{I}$, i.e., the number of dialogues in $s(\mathcal{I})$, is bounded by the product $n \cdot cost(\mathcal{I})$. It is possible to prove that termination is a property that holds for *self-interested rational* agents whose programs are *restricted* according to a *check against an ordering* policy:

Theorem 10 (*Termination of a sequence of dialogues for a restricted agent program*) Let A be a system of $n+1$ agents, and $s(\mathcal{I})$ be a sequence of dialogues with respect to an initial intention $\mathcal{I}$ of a *self-interested rational* agent $a \in A$. Let the agent programs be acceptable, exhaustive, deterministic, and in particular let a's program be *restricted* according to a *check against an ordering* policy. Then $s(\mathcal{I})$ *will terminate* before finitely many dialogue moves. In particular, if R is the maximum rank of a dialogue move, according to a's ranking function, $s(\mathcal{I})$ will terminate after at most $n \cdot cost(\mathcal{I}) \cdot (R+2)$ dialogue moves.

Proof. (*sketch*) As the number of dialogues in $s(\mathcal{I})$ is bounded by the product $n \cdot cost(\mathcal{I})$, and each dialogue, according to Theorem 6, is terminated in at most $R+2$ moves, the whole sequence of dialogues will terminate, and will terminate after at most $n \cdot cost(\mathcal{I}) \cdot (R+2)$ dialogue moves.[10]
■

In the end, I would like to make a parallel between the concept of restriction introduced to ensure the termination of a (negotiation) dialogue and the self-interested rationality assumption of Theorem 10. Indeed, self-interested rationality could be considered a limitation, in that it reduces the space of agent programs. It reflects into a reduction of the space of the achievable solutions of a resource reallocation problem (it is easy to imagine situations where two negotiating agents get stuck in a 'local maximum' because none of them wants to give away a resource). There are some results in this respect in [90]. Due to such reduction, a *weak* notion of completeness has been introduced in [88].

5.13 Extensions

Auctions could be interpreted as an alternative way to negotiate and retrieve resources. They provide an alternative to one-to-one negotiation, e.g. the negotiation carried out by means of dialogues as introduced in the previous section. Indeed, in a framework that includes auctions as a negotiation mechanism, if an agent x needs a resource, it can either ask it to another agent in the system, or find out an auction where the resource is sold and try to obtain it.

Differently from the general dialogue framework, in auctions the *price* of items plays a major role. In particular, the main resource limit is on the initial budget of the single bidders (otherwise agents would buy *at any price*). The participants of an auction are called *auctioneer*, if they sell resources, and *bidders*, if they compete against one another to buy resources. Reasonably, we will assume that there are at least two bidders (otherwise we simply have one-to-one negotiation). In the following, we will call a the auctioneer, b_1, b_2, …, b_n the bidders.

[10] It could easily be proven that the bound is less than that, but it does not seem to make a significant difference, therefore I will consider as a reference this overestimating, still easy to understand and to use, upper bound.

Figure 5.4: Basic steps of an auction

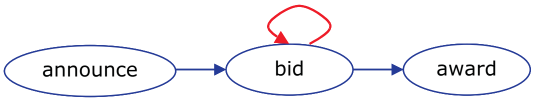

Figure 5.5: English auction: price goes up, 1 bid at a time

In general, an auction consists of at least three steps, as shown in Figure 5.4. The *announcement* step is an utterance made by the auctioneer a, where it declares that an item is on sale at a certain price, for a certain timeout. The *bid* phase is an offer made by a bidder, who is willing to buy the item on sale for a certain price. The *award* phase is a statement made by a to notify everybody that the item has gone[11].

There are various kinds of auctions in literature. I will cite here only four of them, that I consider representative, and in particular: the *English* auction, the *Dutch* auction, the *First Price Sealed Bid (FPSB)* auction, and the *Vickrey* auction.

English auction

In the case of the English auction, the price of the item that is currently on sale is increased until there is only one bid. The highest bid wins. This behaviour can be represented as in Figure 5.5, where there can be any number of bids before moving on to the award.

Dutch auction

In the case of the Dutch auction, the price of the item that is currently on sale goes down until there is one bid. The first bid wins. This behaviour can be represented as in Figure 5.6, where the announcement can be repeated several times before a bid is made.

[11] In general, items could also be declared unsold. For the sake of simplicity, we will consider this as a sub-case of this last award phase.

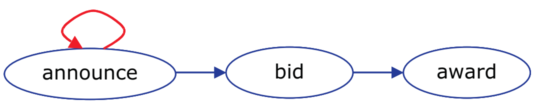

Figure 5.6: Dutch auction: price goes down, 1 bid at a time

Figure 5.7: FPSB auction: k bids are collected, highest wins

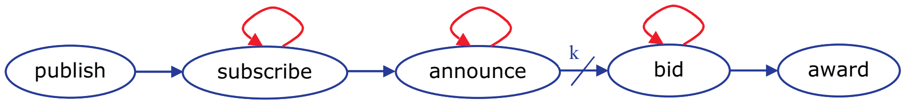

Figure 5.8: General steps of an auction

FPSB auction

In the case of the First Price Sealed Bid auction, k bids are simultaneously made by k bidders, $k \leq n$. Bids are sealed, i.e., bidders do not know each other's bids. The highest bid wins. This behaviour can be represented as in Figure 5.7, where the announcement is made once and k simultaneous bid are made. This protocol is in general faster that the two previous ones, since it is guaranteed to terminate in three steps.

Vickrey auction

In the case of the Vickrey auction, again, k bids are simultaneously made by k bidders, $k \leq n$. Bids are sealed, i.e., bidders do not know each other's bids. The difference with the FPSB case is that the second highest bid wins. This behaviour is not different from that represented in Figure 5.7.

It is worth to notice that, by a protocol point of view, Vikrey and FPSB auctions are the same. In fact, the only difference is the final price that the winner pays, which can play a role in terms of strategies, but not in terms of protocols.

Before the auction takes place, the auctioneer must declare *which objects* are about to be sold, *when* the auction is starting, *which kind* of auction it is going to be (English, Dutch, etc.), how long the *timeout* is going to be, and so on. We could call the period of such declarations the *publishing* phase. Then, auctions could be assigned unique *identifiers* (IDs), that will be used by all those who need to refer to the auction.

In general, the auctioneer must know who is participating at the auction. We will therefore assume that bidders *subscribe* to the auction before it starts[12] and form a group. The group is named after the auction ID. After all the objects have been either awarded or declared unsold, the auction is *closed*.

In general, we could represent the four protocols as a particular case of the one represented in Figure 5.8:[13]

[12] More generally, we could assume that bidders can subscribe up until the auction is closed, i.e., up until all the objects have been either awarded or declared unsold.

[13] I omit here the closure phase. Note that such phase does not play a role, anyway, for the purposes of this work.

It is clear that it is necessary to have a notion of *timeout*, as the whole auction mechanism is based on it. If we implement auctions by means of dialogue primitives with an associated *transaction* time, we still have to relate it to the *actual* time. To this respect, the blackboard comes in hand, since it can be used to assign primitives both an actual timestamp and a transaction time, and keep them bound somehow. In the sequel, I will assume that agents are equipped with a predicate $actual\_time(\ Transaction\_time,\ Actual\_time)$ that keeps track of such relationship.

5.13.1 Extended negotiation language and framework

If we want to extend this framework to tackle auctions, we need a language, and in addition to it we need a communication layer that provides a support for multi-casting. For this we will use a blackboard. Although I will not commit to any particular blackboard or coordination framework, such as for instance a tuple space can be, I will make on the blackboard some assumptions. In particular, we assume that the blackboard is able to assign a transaction time to incoming messages, to store transactions, to manage conflicts in case of multiple concurrently incoming messages (for instance by allowing only one of them), and we will also assume that the blackboard can be programmed in order to allow only those messages that stick to the protocol. For instance, if an English auction is on, where price can only go up, and it must be at least a certain percent higher than that of the previous bid, the blackboard can be programmed to reject those primitives that represent bids that are below the minimum threshold.

We also introduce in the framework a *locking* mechanism, implemented again within the blackboard, that allows bidding only after obtaining the blackboard lock. This is expressed in the agent programs by a logical predicate, $obtain\_lock(\ at(\ Actual\_time\ ),\ Auction\_ID,\ Transaction\_time\ )$. The semantics of such predicate is: true if the agent obtains the right to write on the blackboard (i.e., to multi-cast) at time $Actual\_time$, the $Transaction\_time$-th primitive of $Auction\_ID$. An example of use of such predicate is in Section 5.13.2.

Multi-cast primitives

As opposed to dialogue primitives, the communication acts of auctions must be multi-casted, which requires a suitable support, as I pointed out previously. Therefore, I will assume that agents do not write directly in a blackboard, not before obtaining from the blackboard the permission to write. In fact, the blackboard will coordinate the agent requests, by allowing one message (primitive) at the time, and by giving it a unique transaction time.

The format of multi-cast primitives must be different from that of one-to-one primitives, since the recipient is in general not unique. As I already said, the second parameter will not be a single agent, but a group of agents. Multi-cast primitives will have in general this format: $tell(Sender,\ Group,\ Subject,\ Time)$, where $Sender$ is an agent, $Group$ is a group of agent, intended to be the recipients of the message, $Subject$ expresses the content of the message, and $Time$ is again a transaction time.

We are now ready to introduce a *language* for auctions, i.e., a set of allowed dialogue primitives that agents use to implement an auction:

- $tell(Auctioneer, all^{14}, publish(auction(Auction\_ID), items(\{ i_1, i_2, \ldots, i_m \}), protocol(english \mid dutch \mid fpsb \mid vickrey), beginning\_at(initial\_time), timeout(2')), Time)$
- $tell(Bidder, Auction\_ID, subscribe(Auction\_ID), Time)$
- $tell(Auctioneer, Auction\_ID, announce(Item, Price, TOut), Time)$
- $tell(Bidder, Auction\_ID, bid(Item, Price), Time)$
- $tell(Auctioneer, Auction\_ID, award(Item, Price, Bidder), Time)$
- $tell(Auctioneer, Auction\_ID, close, Time)$

Given this language, an example of an English auction is the following.

$\leadsto tell(a, all, publish(auction(auction_1), items(\{ nail \}), protocol(english), beginning\_at(\mathtt{Jan\ 3^{th}, 2002, 14:30\ GMT}), timeout(2')), 1)$
$\leadsto tell(b_4, a, subscribe(auction_1), 2)$
$\leadsto \ldots$
$\leadsto tell(a, auction_1, announce(nail, 100, 14), 13)$
$\leadsto tell(b_4, auction_1, bid(nail, 100), 14)$
$\leadsto tell(a, auction_1, announce(nail, 110, 16), 15)$
$\leadsto tell(b_9, auction_1, bid(nail, 110), 16)$
$\leadsto tell(a, auction_1, announce(nail, 120, 18), 17)$
$\leadsto tell(a, auction_1, award(nail, 110, b_9), 18)$ [15]

In this example, the auctioneer starts with publishing the auction $auction_1$ for a certain point in the future ($Jan\ 3^{th}, 2002, 14:30\ GMT$), and collects subscriptions. Then it starts the auction by announcing the first (and only) item, a $nail$. Bidders keep replying to further announcements, until a timeout is reached after the last one. Similar examples for the other auctions could be given.

In the case of English and Dutch auctions, as in this example, the auctioneer announces the new (minimum) price and waits for a bid. If the timeout comes the auctioneer posts in the blackboard the award message. We will assume that

[14] Here and in the following, *all* could mean agents all the agents known or reachable by the auctioneer. I assume that the auctioneer has some visibility of the environment; I have not dealt explicitly so far with such issue, though.

[15] As I said before, we will not deal with the closure of auctions, which could be expressed by a primitive such as $tell(a, auction_1, close, 19)$, uttered by the auctioneer.

some aspects, such as filtering unsuitable bids, are managed by the blackboard (bids for items not being sold in that auction, bids from unsubscribed agents, bids for a price that is too low, ... will be rejected by the blackboard). I admit that this is a strong assumption on the environment, and needs to be further investigated. Moreover, such behaviour hardwired in the blackboard, could require a dynamic (re-)configuration. It is not obvious at all who and how should to it. Another important issue, with this respect, is how to merge such non-logical element within the logical framework.

We can assume of course that bidders can bid more than announced by the auctioneer (highest bid wins anyway). This is up to the agent. In the case of FPSB and Vickrey auctions, that we did not represent here, the timeout would be expressed in terms of transaction time, as the current time $+n$, n being the number of expected bidders. This is one more reason why it is required to subscribe an auction: n is actually the number of subscribers. This could raise some problems, in that it would require some assumptions on the presence of all subscribers for the whole duration of the auction, which is maybe too strong in a multi-agent setting. Anyway, it is possible to relax this condition, and define a protocol in which not all subscribers must bid. We could imagine that if there are less bids than bidders ($k < n$), the auctioneer will be in charge to make sure that before too late (*actual* timeout) n messages are anyway posted in the blackboard, in order to reach the (*transaction*) timeout. She will therefore put $n - k$ messages atomically together with the award message, just before it is published.

5.13.2 Implementation of the english auction protocol

In this section, I will give here an example of a possible implementation of the English auction protocol. Due to the space constraints, I will only write the basic rules and constraints of the two kinds of agents: the bidder and the auctioneer. Let us start with the auctioneer program.

auctioneer (a) program:

$$\begin{array}{l}
\mathbf{on_sale(Item, Auction_ID)} \leftarrow \\
\quad tell(a, all, publish(auction(Auction_ID), items(Items), \\
\quad protocol(Protocol), beginning_at(Initial_time), timeout(Timeout)), Time) \\
\quad \wedge\ Item \in Items \\
\quad \wedge\ actual_time(Actual_time) \\
\quad \wedge\ Initial_time < Actual_time \\
\quad \wedge\ not\ timeout_expired(Item, Auction_ID, Timeout) \\
\quad \wedge\ not\ (tell(a, Auction_ID, award(Item, Price, Bidder), Time1) \\
\qquad \wedge\ Time1 < Time)
\end{array}$$

An item is *on_sale* if it is included in the item list of an ongoing auction, and the timeout after the last bid has not expired.

$$\begin{aligned}
&\mathbf{timeout_expired(Item, Auction_ID, Timeout)} \leftarrow \\
&\quad tell(a, Auction_ID, announce(Item, Price1, Timeout1), Time1) \\
&\quad \wedge\ not\ (tell(a, Auction_ID, announce(Item, Price2, Timeout2), Time2) \\
&\qquad \wedge\ Time2 > Time1) \\
&\quad \wedge\ actual_time(Time1, Actual_time1) \\
&\quad \wedge\ actual_time(now, Actual_time) \\
&\quad \wedge\ Actual_time > Actual_time1 + Timeout
\end{aligned}$$

The timeout that refers to a certain item on sale at a certain auction is *expired* if *now* is a later time than that of the last announcement, plus the declared *Timeout*.

$$\mathbf{new\_price(Price, New\_price)} \leftarrow New\_price\ is\ Price + 0.1 * Price$$

The auctioneer announces the new price of an item adding a ten percent to the last price.

$$\begin{aligned}
&\mathbf{tell(Bidder, Auction_ID, bid(Item, Price), Time)} \\
&\wedge\ on_sale(Item, Auction_ID) \\
&\wedge\ new_price(Price, New_price) \\
&\wedge\ obtain_lock(at(now), Auction_ID, Time + 1) \\
&\quad \Rightarrow \mathbf{tell(a, Auction_ID, announce(Item, New_price, Time + 2), Time + 1)}
\end{aligned}$$

This constraints says that if a participant bids at time $Time$, and the item in question is still on sale, the auctioneer next (i.e., at $Time + 1$) announces such item for a new price. Such announcement will hold up to $Time + 2$.

$$\begin{aligned}
&\mathbf{timeout_expired(Item, Auction_ID, Timeout)} \\
&\wedge\ tell(A, all, publish(auction(Auction_ID), \ldots, timeout(Timeout)), \ldots) \\
&\wedge\ tell(Bidder, Auction_ID, bid(Item, Bidder_price), Time - 1) \\
&\wedge\ tell(A, Auction_ID, announce(Item, New_price, Time + 1), Time) \\
&\wedge\ Bidder_price \geq New_price \\
&\quad \Rightarrow \mathbf{tell(a, Auction_ID, award(Item, Bidder_price, Bidder), Time + 1)}
\end{aligned}$$

This constraints says that after the timeout has expired the highest bid wins, and therefore the bidder is awarded the item.

$$\begin{aligned}
&\mathbf{timeout_expired(Item, Auction_ID, Timeout)} \\
&\wedge\ tell(A, all, publish(auction(Auction_ID), \ldots, timeout(Timeout)), _) \\
&\wedge\ tell(A, Auction_ID, announce(Item, _, _), Time) \\
&\wedge\ not\ tell(_, Auction_ID, bid(Item, _), _) \\
&\quad \Rightarrow \mathbf{tell(a, Auction_ID, award(Item, \ldots, noone), Time + 1)}
\end{aligned}$$

This constraints says that if there are no bids, the item is declared unsold.

bidder (b) program:

The bidder program is simpler, but it introduces two predicates whose implementation is not specified here, and will depend on the individual bidder programs (we assume that each bidder may adopt a different policy from the others). Such predicates are *calculate_new_price* and *calculate_bid_time*.

$$\begin{array}{l}
\mathbf{tell}(\mathbf{a}, \mathbf{Auction_ID}, \mathbf{announce}(\mathbf{Item}, \mathbf{Price}, \mathbf{Timeout}), \mathbf{Time}) \\
\wedge\ have_subscribed(Auction_ID) \\
\wedge\ not\ (tell(Participant, Auction_ID, Anything, Time1) \wedge\ Time1 > Time) \\
\wedge\ calculate_new_price(Item, Price, New_price) \\
\wedge\ calculate_bid_time(Auction_ID, Bid_time) \\
\wedge\ obtain_lock(at(Bid_time), Auction_ID, Time + 1) \\
\quad \Rightarrow \mathbf{tell}(\mathbf{b}, \mathbf{Auction_ID}, \mathbf{bid}(\mathbf{Item}, \mathbf{New_price}), \mathbf{Time+1})
\end{array}$$

This means that if the auctioneer has announced an $Item$ at a $Price$, at the (transaction) time $Time$, and after that no other participant made any move, if the bidder b can calculate a suitable price and can obtain the blackboard to bid at a certain time (Bid_time), then b will bid at (transaction time) $Time + 1$. The two above mentioned predicates and the blackboard will decide whether the agent will actually bid or not.

$$\begin{array}{l}
\mathbf{have_subscribed}(\mathbf{Auction_ID}) \leftarrow \\
\quad tell(b, a, subscribe(Auction_ID), Time)
\end{array}$$

5.14 Concluding remarks

In this chapter, I studied the applicability of the abductive mechanism in a multi-agent system to model a negotiation problem. A framework has been defined, based on the IFF procedure and on the Kowalski-Sadri agent cycle, that allows to support negotiation dialogues and that can be extended toward auction-like negotiation mechanisms. I proved some important properties of the dialogue framework, among which, a correctness result and a completeness result, based on a notion of weak completeness and with respect to resource reallocation problem. I also discussed the issue of termination of dialogues and sequences of dialogues, showing that the intrinsic flexibility of the dialogue itself as an agent interaction pattern (compare to protocols) makes it difficult to determine the conditions for ensuring termination, without loosing in terms of completeness in problem solving.

In conclusion, I think that abductive reasoning in multi-agent systems has the potential to increase its importance in logic based multi-agents, being these results interesting not only from a theoretical viewpoint, but also from a more practical perspective. In fact, the proposed framework has both the advantage of providing a formalism that allows for proving properties such as those above mentioned, and an execution model available, that makes it possible to generate prototypes of agent system, by only giving their formal specification.

Chapter 6

Discussion

This thesis mainly focused on abductive reasoning. Both theoretical results and a practical application in the agent negotiation domain have been presented. In this final chapter I will first discuss the related work, then move on to the conclusions and present some issues for future work.

6.1 Related Work

This work embraces several fields of research, ranging from logic-based agents and abductive reasoning to multi-agent negotiation. In this section, I will give some references to the related work, roughly following the division in chapters adopted so far.

6.1.1 Logic agents

In the last years, the interest on agents that make use of logic has increased, although the notion of "logic agent" is not associated with a commonly agreed upon formal definition. An exhaustive overview covering recent work on computational logic and multi-agent systems has been written by Sadri and Toni [87]. More recent contributions can be found in [35, 36]. I would like to cite here some of the most relevant contributions in the field.

IMPACT [5, 37] is a joint effort among the Army Research Laboratory, the University of Mariland (Subrahmanian et al.) and several other institutions. It delivered an agent architecture that is able to cope with heterogeneous individuals and, and that allows for retrieving information from arbitrary data structures, semantically wrapped within a well-defined interface. It has been applied to several applications in the domains of logistics, flight control, and nontrivial U.S. Army applications. MAGIC and MENTAL are two projects, mainly carried out within the Artificial Intelligence group of the Universidade Nova of Lisbon (Pereira et al.). They propose both an intra- and inter- architecture for rational agents, providing support for multiple forms of reasoning. The characteristic of such architectures is the use of dynamic logic programming to allow for agent program updates and belief revision. InterRaP [76], developed by K. Fischer et al. at DFKI is a layered hybrid agent architecture, featuring reactivity, planning, and supporting multiple agents interactions. It has been applied to the domains of transportation, task scheduling, and loading dock.

Common Rules is a project that involved a number of academic institutions and industrial partners, aimed to integrate the industrial experience with the academic research. It delivered an architecture and a prototype, whose main characteristics are the identification of computational logic as a core declarative knowledge representation language, and the incorporation of rule execution engines, conflict handling and inter-agent communication facilities. The prototype has been applied to B2B integration, manufacturing supply chains, e-commerce, negotiation, and security authorization policies.

Relevant work concerning the architecture of logic agents has been made by Kowalski et al. [65, 66, 68], aimed to merging in a single architecture both the rational and reactive aspect of autonomous agents. Rational agents have both beliefs and goals. A reactive agent, instead, has neither an explicit knowledge base, nor an explicit representation of goals. In [66], Kowalski and Sadri propose a uniform agent architecture that aims to capture both rationality and reactivity. This one in particular is the architecture that has been proposed in the execution model for logic-based negotiating agents discussed in Chapter 5. In a rational agent, goals are explicitly represented and the knowledge can be represented as goal reduction rules of the type *conclusion if condition*. Furthermore, condition-action rules, peculiar to reactive agents, can be interpreted as integrity constraints. The two kinds of behaviour are then unified in a single logic-based architecture which is grounded on the IFF proof procedure. According to [65, 66], the logic agents of ALIAS in its various versions are to be considered rational agents, since their main capability is to answer queries (i.e. goals) from the environment (i.e. other arguing agents) or from high-level goals.

In [67] Kowalki and Sadri define the *observe-think-act* agent cycle, that we also include in the execution model proposed in Chapter 5 for agents that negotiate. It consists of three phases, and in particular of a *think* phase, where the abductive derivation process takes place. The weight of such phase within the cycle, in terms of computational resources used, can be increased or decreased, according to the need for agents to exhibit a more rational or, respectively, a more reactive behavior.

The Vivid Agents project (Wagner and Schröder) proposes an architecture for reactive agents when the agent knowledge is expressed in the form of event-condition-action rules. The agent activity is governed by a *perception-action* cycle, similar to that of Kowalski and Sadri, but especially suited for agents that need to exhibit a reactive behavior. Beliefs are expressed in the form of extended logic programs. There exist several frameworks for negotiation that make use of logic [88, 72, 78]. They will be discussed in Section 6.1.4. Logic has also been used for agent system verification, making use of model checking techniques an temporal logic. Relevant work is that by Benerecetti et al. [9], and CaseLP by Bozzano et al. [10], that has been applied to the problems of train scheduling, planning of goods transportation, and retrieval of medical information. Other worth mentioning projects have been, recently, the FX-Agents project (Genesereth and Petrie) and the "area" of Logic Agents within the European Community funded project Compulog-net [69].

Probably, the most recent research project that focuses on the combination of logic programming and multi-agent systems, is the 3 years European Community founded Project *Societies of ComputeeS* (SOCS) [95]. The project started in January 2002, and aims to investigate computational and logical models of individual and collective behaviour of computational entities – referred to as *computees* – interacting in the context of global and open computing environments.

6.1.2 Abductive reasoning and architectures for the coordination of reasoning in agents

The abductive coordination protocol introduced in Section 3.1.3 (DAA) is based on a query-based proof procedure originally defined by Eshgi and Kowalski [40] that recovers negation as default through abduction, and is extended by Kakas and Mancarella in [60] in order to manipulate arbitrary abducibles and integrity constraints. In [12] this proof procedure has been further extended to consider a second kind of negation, and proved sound and weakly complete with respect to a three-valued abductive semantics. Other relevant literature exists on proof procedure for extended and abductive logic programs. Satoh and Iwayama [92] proposed a query-based proof procedure which improves the KM, and addresses completeness with generalized stable model semantics [59], but at the price of a more complex integrity checking. Another proof procedure has been proposed by Denecker and De Schreye in [33, 32], by extending the SLDNF resolution to the case of abduction. A crucial property of this abductive procedure is the treatment of *non-ground* abductive goals, that is neither considered in [60] nor in the abductive protocol used in ALIAS. In [33], the authors do not consider general integrity constraints, but only constraints of the kind $\leftarrow a, not\ a$. To overcome this limitation, in a later work [32], they consider the treatment of general integrity constraints that are checked only at the end of the proof for a query, i.e., only when the overall set of abductive hypotheses supporting the query has been computed. However, my belief is that it is better to check consistency as soon as the hypotheses are generated.

The problem of incremental consistency checking for the generated hypotheses (possibly containing variables) is considered in *Theorist*, by Poole [82]. In *Theorist*, consistency is immediately checked for those hypotheses with no variables and delayed until the end for hypotheses containing variables at the time they are generated.

A notion of abduction for the family of Concurrent Constraint (`cc`) languages was proposed by Codognet and Saraswat in [21]. The key idea of the `cc` framework is to use constraints (and a global set of constraints called the constraint *store*) to synchronize concurrent logic computations. Multiple agents run concurrently and synchronize by adding a new constraint to the *store* (*tell* operation) or by checking whether a given constraint is entailed by the *store* (*ask* operation). An asking agent may block and suspend if information in the *store* is insufficient to determine whether the constraint is entailed or not. In (cc) computations this corresponds to a situation where only partial information in the *store* is available. Thus, in [21], abduction is used to generate some hypothetical constraint that would lead some *ask* operations to be satisfied and therefore resume some agent computations. With respect to this thesis, they consider as hypotheses constraints instead of atoms, and the consistency check is delayed during the computation, until some other agents will add to the *store* a possibly inconsistent constraint.

Relevant work has been done by Kakas et al. within the ACLP project [57, 1] in order to extend the Kakas-Mancarella proof procedure towards the treatment of non-ground abductive goals. The system, ACLP, combines abductive reasoning and constraint solving by integrating the framework of Abductive Logic Programming (ALP) with that of Constraint Logic Programming (CLP). The ACLP framework has been implemented on top of the ECL^{i}PS^e language [39] as a meta-interpreter using explicitly its low-level constraint solver that handles constraints over finite domains.

A parallel implementation of the proof procedure described in [60] has been proposed by Kakas and Papadopoulos in [61]. Various forms of parallelism (OR and dependent AND parallelism) are exploited to parallelize deductive logic programming in the framework of abductive logic programming. This parallel model for abduction is then mapped into the Andorra model. Nonetheless, the focus of [61] is on a parallel implementation of a single abductive logic programming computation rather than on the multi-agent case.

6.1.3 Languages for the grouping of agents and coordination of agent reasoning

The issue of combining agents into groups has been discussed by Barbosa and Cunha in [7], where the authors present the GroupLog coordination language for collective agent based systems. GroupLog defines extensions to the Extended Horn Clause language (EHC), that are supported at two levels: L_1 defines agents as program units and L_2 defines groups of agents. Agents are structured in classes. While in L_1 it is possible to define the agent interface to other agents, in L_2 it is possible to coordinate the behavior of a group of agents of the same class. To that extent, a group of agents is considered itself a meta-agent, allowing the modular composition of the group notion. In ALIAS, coordination within a group of agents is not achieved through the notion of group but is built in the semantics of the consistency operators: group coordination in ALIAS is needed to ensure the consistency of a given set of abduced hypotheses and is transparent to the agent, while the behavior of the individual agents can be expressed in the ABM.

Other work on the representation of complex actions is that of the language CONGOLOG [73], a programming language whose formal semantics is based on the situation calculus. CONGOLOG supports the specification of agent behaviors at a very high level of abstraction. While CONGOLOG has been thought to overcome the lack of formal semantics of the KQML definition [22], it does not aim at separating communication issues from agent reasoning. ALIAS, instead, with a separation introduced between a reasoning module, the ARM, and a communication/behavior module, the ABM, allows for a higher level programming style. It is less general, in that it considers a specific class of agents (abductive logic agents). However, there is current work aiming to extend and generalize the system towards other forms of reasoning.

Among the other agent languages, I would cite Agent-0 [93] and Concurrent MetateM [46], where the focus is on the specification of the agent's internal behavior: while the former has single communication acts built-in, the latter does not model interactions between agents at all. As far as agent communication languages are concerned, the most relevant work towards standardization has been done with KQML [43], a widely used specification of a language for inter-agent communication, and FIPA ACL [45], especially oriented to physical agents. KQML and FIPA ACL are very general purpose languages, and while not referring specifically to logic agents, are intended to be used as specifications rather than languages. An approach for expressing communication for abductive logic agents is proposed in [30], where the primitives *tell* and *ask* are treated as abducibles, similarly to what is done in ALIAS, and framed in an "observe-think-act" cycle [67]. The main difference between this approach and that followed in this thesis is that the latter provides operators to ensure the consistency of a set of hypotheses with respect to the integrity constraints of a set of agents, while in [30] the abductive reasoning process is individually performed. Also, in this thesis I opted for a strong separation between a communication / coordination meta-language, and the abductive knowledge base contained in the ARM.

Relevant work on the combination of multiple knowledge bases is that of Baral et al. [6], where the authors define the concept of combining the knowledge present in a set of knowledge bases, and present algorithms to maximally combine them, so that their combination is consistent with the integrity constraints associated with the knowledge base sets. In [63] Konieczny investigates the logical properties of knowledge base combination operators proposed in the literature, and discusses the difference between merging knowledge bases and combining them. BReLS [74], by Liberatore and Shaerf, is another system for the knowledge bases integration; the authors distinguish among three different approaches to the problem: belief revision, merging and update, and present a framework that integrates them.

In general, work on knowledge merging or combining is based on the definition of ordering and preference operators, aimed at defining a maximal amount of knowledge that can be put together without violating some integrity constraints. The point of this work is the other way round, as it aims at keeping separate the agent knowledge bases. In a way, this thesis approaches the issue of cooperation rather than integration, in that it adopts the agent model in order to allow different and possibly contradicting knowledge bases to coexist and to participate to some common reasoning activity, without preferring one over the one. In ALIAS, in particular, agent coordination is query-dependent, and it is left up to the querying agent to decide the way to combine the knowledge coming from its partners, whether in a competitive or collaborative way.

Before implementing LAILA, some work concerning various Prolog parallel implementation issues has been considered. In particular, I would cite [62], [52] and [80], where the problem of dependent AND-parallelism is broadly discussed. LAILA does not allow a full AND-parallelism, though, limiting the interaction among parallel computing branches to a consistency check among ground abducible sets.

Some work presenting an approach to concurrent logic programming that inherits the blackboard model of problem solving, is that of Shared Prolog (SP) [11]. SP is an elegant and concise specification tool for intelligent agent-based applications such as expert systems based on blackboard architectures. The programmer controls the granularity of parallelism, coordinating communication and synchronization of the agents via a centralized data structure. With respect to SP, LAILA is a higher level language aimed at masking the low level inter-agent coordination issues. The purpose of LAILA is to provide the developer with a few high level, highly expressive primitives, abstracting away from the details of the control of parallelism.

6.1.4 Agent dialogue and negotiation

Introduction and surveys of automated negotiation can be found in [86] and [71, 55], respectively. An approach to negotiation via dialogue, that makes use of an argumentation system, is that of Sycara [96], and more recently Kraus et al. [72], where the authors adopt a modal logic approach and focus on the mental states of agents that move towards an agreement and encourage cooperation. The authors' formalization of dialogues makes use of rules that, similarly to the dialogue constraints, are triggered when a performative is received, and may lead to another dialogue move under certain conditions, within an agent cycle. The execution is intertwined with the dialogue, allowing the agents to express threats, promises of rewards, etc. and to maintain a mental model of the other partners. Such an approach, focused on persuasion, leads to different results from the ones presented here, being somewhat more descriptive, and not aiming to determine specific properties of the negotiation policy. However, it could interesting to extend the framework presented in this thesis towards persuasion / argumentation based negotiation, since argumentation can also be modeled through abduction [58]. Other (more general and less operational) argumentation-based approaches to agent negotiation, are those of Amgoud et al. [3] and Parsons et al. [78]. I have already described how my work relates to [3] in Section 5.1. In [78], from which the nail-and-hammer example has been inherited, the authors adopt a formal system of argumentation in order to get the support necessary to build agents which negotiate to reach an agreement. Their approach is also modal logic based, and is more descriptive than procedural. In order to develop an agent architecture which has a clear link between its specification and its implementation, they make use of multi-context systems [51]. ALIAS can also be considered a particular instance of multi-context systems, where ABM and ARM are two *units*, each having its own *logics* and *theories*, and where the LAILA *down-reflection* operator is a *bridge rule* between one agent's ABM and ARM, and the *communication* operator is the *bridge rule* between the ABMs of two different agent.

Zlotkin and Rosenschein [105] adopt a game theory approach to the study of automated negotiation. However, their work – as most others that are game theory-based – considers agents that are not cooperative, and focuses on conflict resolution rather than collaboration. One of the interesting issues they address is the disclosure of private information. This work is somewhat orthogonal, aiming at obtaining the resources the agents need, without taking into account the problems that could arise from disclosing private information. I agree that this is very important, especially in contexts such as e-commerce and virtual enterprises.

The use of negotiation to tackle the resource reallocation problem is not a new idea; among other contributions I cite those by Faratin [41], Sandholm [90], and Sathi and Fox [91]. Most of them study the problem adopting a game-theoretic approach and prove results that I consider orthogonal to those in this thesis.

The logic-based framework for negotiation introduced in this thesis allows us to prove properties that we can use to forecast the behaviour of a system of agents with no need for simulation and experimental results. This thesis suggests an execution model that relies on the *observe-think-act* agent cycle of Kowalski and Sadri [67], discussed in Section 6.1.2, and on the adoption of the IFF abductive proof procedure of Fung and Kowalski [48]. In this way, the implementation of the system, at least as far as a prototype, is straightforward, and does not require much more than writing high-level dialogue rules of the kind presented in the thesis.

From a language point of view, much work has been done in the area of speech act theory by Cohen and Levesque [22], and Cohen and Perrault [23], to which the negotiation language adopted in this thesis is indirectly inspired and on the semantic issues that arise, especially in terms of verifiability [102] and guaranteed success [104]. This work is again orthogonal to these, and is assumes that both agents involved in the dialogue share a common semantics of the speech acts (the dialogue performatives), and that they can interpret them within the IFF proof-procedure. Also, the approach of this thesis is totally independent of the planner, while [23], aiming to integrate dialogue and plans, make some important assumptions, that they call *heuristic principles of purposeful behaviour*, describing some requirements that a planner must fulfill. On the other hand, the work of Wooldridge [102] becomes even more relevant as soon as we want to study the properties of the protocol when the set of integrity constraints changes, and thus the behaviour of the agents changes.

6.2 Conclusion

This thesis contributes in many respects to the extension of logic programming to distributed systems and agent systems in particular, and on the other hand to the adoption of logic programming techniques to the design of multi-agent systems. In the first part of the thesis, I tackled the issue of grouping agents for coordinating their reasoning, and presented several agent architectures based on abductive logic agents. To this end I provided an infrastructure where agents are grouped in bunches and can interact following different patterns. I defined a language, LAILA, to write abductive multi-agent applications in a declarative way, and independently of the particular abductive mechanism that each agent may adopt. The underlying use of abductive reasoning makes the system particularly suited for solving problems in domains where knowledge is incomplete, as it is the case of negotiation between agents. In this setting, reasonable models and protocols should assume that agents lack knowledge about other agents. Also, the presence of a consistency checking mechanism in LAILA allows for independently programmed systems to be put together and still exhibit a consistent behaviour, which is in line with the idea of multi-agents.

The contribution of this part of the work is in the integration of two reasoning mechanisms, namely deduction and abduction, and in the formalization of the agent interactions, from an operational point of view, and by means of inference rules. This allows to model both local and global behavior, properly combining the language used to model communication among agents with the one used to model the agent local knowledge, and is intended to be a starting point for the definition of formal properties and for their verification.

In the last part of the thesis, I focused on one-to-one agent negotiation for achieving goals in a system with limited resources. The solution proposed is based on agent dialogues, as a way of requesting resources, proposing resource exchanges, and suggesting alternative resources. The negotiation framework presented is a general one for representing policies and generating dialogues. The resource reallocation problem is one possible application of this framework. This framework could be deployed to tackle other problems, e.g. to deal with dialogues starting with an offer or an advertisement rather than a request.

I assumed that the agents are provided with a planner that, given a goal, is able to produce a sequence of actions to achieve it and an associated set of missing resources that are needed to carry out the actions. The framework proposed is independent of the planner, the specific application domain, the agent programs that regulate the dialogues, and even the language that agents use to communicate.

In the domain of resource reallocation, a single dialogue is used to obtain a resource. This thesis presents a logic based approach, and, in particular, the agents involved in a dialogue can use an abductive proof procedure to reason about their current beliefs and intentions, and produce a dialogue move to transmit to their partners. An agent dialogue cycle is used produce sequences of dialogues, and in general sets of sequences of dialogues in order for the agent to collect all the resources it needs to achieve its goal.

The logic-based approach allows us to define and prove some interesting properties of dialogues, sequences of dialogues, and sets of sequences of dialogues in an agent system. In particular, the adoption of an abductive proof procedure for which termination, correctness and completeness results hold, allows us to extend such general results to the specific case of the resource reallocation problem. Various agent programs have been shown, that can be used to tackle such a problem, proving for them specific results, i.e., to what extent the properties of termination, correctness and completeness, as defined in this thesis, hold for them.

ALIAS, as an architecture to coordinated abductive agent reasoning, allows to add interesting new features to this framework for agent negotiation, providing a suitable support for multiple negotiation processes. In particular, given an agent, a goal, and n alternative plans to achieve it, the agent may adopt a *collaborative* interaction pattern in order to collect all the missing resources for the execution of a chosen plan and make it feasible, and a *competitive* pattern to try to select one feasible plan, out of the n alternative plans, by exploring them concurrently.

Both the architecture, ALIAS, and the language, LAILA, have been implemented on top of Jinni [97], a logic language extended with Linda-like primitives, thus following a fully declarative approach. Although in this version of ALIAS each agents implements the Kakas-Mancarella abductive proof procedure, the system architecture is independent of any concrete algorithm. For instance, a simple two-agent negotiation dialogue has been tested also with the IFF proof-procedure.

6.3 Future research

This thesis leaves a number of issues for future research. Computational logic in multi-agent systems has the potential to play a fundamental role, as long as it manages to bind its foundational nature with some more practical aspects, to make it understood and accepted by a broader community. This idea is supported by matters of fact such as an international award [88] and the funding of an European project [95], that both go in this direction, aiming to provide a workable prototype of a computational agent model, and bridge the gap between specification and implementation.

With respect to ALIAS, as we already mentioned several times, its implementation should be extended to first-order non ground abductive programs; in doing that a possibility could be to merge in ALIAS the ACLP framework. It is in my intentions to contribute with the results of this thesis to the SOCS project, and to extend the architecture for abductive reasoning with several functionalities, such as belief revision, theory revision, reactive behavior. In order to do that, an agent architecture, possibly based on a cycle, must be defined to include meta-level reasoning. LAILA could be a starting point, to define several *down reflection* operators that can express not only a request for an abductive derivation, but also requests of different kinds, such as, e.g., for reasoning upon a set of examples in order to obtain a theory.

As far as the application domain of negotiation, I intend to explore the interaction of heterogeneous agents containing various negotiation protocols, each described by a different abductive logic program. In this respect, I plan to analyze the properties of the system, e.g., termination, convergence, etc., when putting together different agent varieties, i.e., abductive logic programs that implement different negotiation policies. Moreover, I intend to give a better formal definition of the framework extended with broadcast/multicast performatives introduced in Section 5.13, in the application domain of automated auctions. In a negotiation setting, the integration of abductive reasoning with learning capabilities could lead to a dynamic improvement of the agents' logic programs and of their interaction patterns, based on past negotiation experiences.

So far I assumed that agents do not change their goals. A natural extension would be to take into account the possibility of modifying goals as a consequence of an unsuccessful dialogue sequence, in accordance with the BDI theory that requires goals to be believed feasible. This could require an integration with dynamic programming or updating techniques, such as those cited in 6.1.1.

Another related issue is about the notion of determinism. Assuming that an agent program is deterministic could result in a loss in terms of flexibility. This problem might be solved if we let agents change their goals, or if we introduce a degree of non-determinism in the procedure, but then we would need to base on different correctness and completeness results for the modified semantics. Negotiation of sets of resources, rather that single items, is another interesting problem, as is the use of broadcast or multi-cast primitives as in electronic auctions. This would not only require to extend the language and protocols defined in Section 5.5, but would also bring along some complexity implications. In fact, the risk is that, without the integration of suitable search strategies, the risk is to exponentially increase the size of the problem, since the number of possible subsets of a set of n elements is 2^n. The integration of the negotiation framework with search techniques based on a fitness landscape is also subject for future work. Finally, it would also be interesting to study different cost functions for intentions, and to see what their impact is on the dialogues and on the dialogue properties. Currently, in moving from one intention to another, as a consequence of a deal (the acceptance of a *promise*), an agent can choose a plan that turns out not to be feasible, because of the lack of resources in the system. A more sophisticated cost function than the one proposed in the thesis could result in the agent ensuring that all the resources for a new plan exist somewhere in the system, before agreeing to a deal. This could also imply the adoption of suitable search strategies.

Bibliography

[1] ACLP: Abductive Constraint Logic Programming. Electronically available. ACLP Home Page URL: `http://www.cs.ucy.ac.cy/aclp/`.

[2] ALIAS: Abductive LogIc Agent System. Electronically available. ALIAS Home Page URL: `http://lia.deis.unibo.it/Research/ALIAS/`.

[3] L. Amgoud, S. Parsons, and N. Maudet. Arguments, dialogue and negotiation. In W. Horn, editor, *Proceedings 14th European Conference on Artificial Intelligence, Berlin, Germany*. IOS Press, August 2000.

[4] *AMZI!* prolog + logic server 4.1. Amzi! URL: `http://www.amzi.com`.

[5] K. A. Arisha, F. Ozcan, R. Ross, V. S. Subrahmanian, T. Eiter, and S. Kraus. IMPACT: a Platform for Collaborating Agents. *IEEE Intelligent Systems*, (2):64–72, March/April 1999.

[6] C. Baral, S. Kraus, and J.Minker. Combining Multiple Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 3(2):208–220, June 1991.

[7] F. Barbosa and J. C. Cunha. A coordination language for collective agent based systems: GroupLog. In *Proceedings ACM SAC'00, Villa Olmo, Como, Italy*. MIT Press, 2000.

[8] M. Barbuceanu and W. Lo. A multi-attribute utility theoretic negotiation architecture for electronic commerce. *Proceedings of the Fourth International Conference on Autonomous Agents, Barcelona, Spain*, pages 239–247, 2000.

[9] M. Benerecetti and A. Cimatti. Symbolic model checking for multi-agent systems. In Dix and Satoh [36]. Electronic proceedings available. URL: `http://research.nii.ac.jp/ ksatoh/clima01.html`.

[10] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Logic programming and multi-agent system: A synergic combination for applications and semantics, 1999.

[11] A. Brogi and P. Ciancarini. The concurrent language, shared prolog. *ACM Trans. on Programming Languages and Systems*, 13(1):99–123, 1991.

[12] A. Brogi, E. Lamma, P. Mancarella, and P. Mello. A unifying view for logic programming with non-monotonic reasoning. *Theoretical Computer Science*, 184:1–49, 1997.

[13] F. Cascio, L. Console, M. Guagliumi, M. Osella, A. Panati, S. Sottano, and D. Theseider Dupré. On-board diagnosis of automotive systems: from dynamic qualitative diagnosis to decision trees. In *Proc. IJCAI Workshop on Qualitative Reasoning dor Complex Systems and their Control*, 1999.

[14] A. Ciampolini, E. Lamma, P. Mello, and P.Torroni. Coordinating the safe execution of tasks in a constrained multi-agent system. Technical Report DEIS-LIA-01-009, University of Bologna (Italy), 2001. LIA Series no. 53.

[15] A. Ciampolini, E. Lamma, P. Mello, F. Toni, and P. Torroni. Co-operation and competition in *ALIAS*: a logic framework for agents that negotiate. In Dix et al. [35].

[16] A. Ciampolini, E. Lamma, P. Mello, and P. Torroni. Expressing collaborative and competitive coordination among abductive logic agents. In *Proceedings CLIMA'00, First International Workshop on Computational Logic in Multi-Agent Systems*, London, UK, July 2000.

[17] A. Ciampolini, E. Lamma, P. Mello, and P. Torroni. An implementation for abductive logic agents. In E. Lamma and P. Mello, editors, *Proceedings AI*IA 99, 6th Congress of the Italian Association for Artificial Intelligence*, number 1792 in LNAI. Springer Verlag, 2000.

[18] A. Ciampolini, E. Lamma, P. Mello, and P. Torroni. LAILA: A language for coordinating abductive reasoning among logic agents. *Computer Languages*, 27(4):137–161, February 2002.

[19] P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2):300–302, 1996.

[20] K. L. Clark. *Negation as Failure*, chapter Logic and Databases. Plenum Press, New York, 1978.

[21] P. Codognet and V. Saraswat. Abduction in concurrent constraint programming. Technical report, INRIA-Roquencourt, March 1992.

[22] P. R. Cohen and H. J. Levesque. Communicative actions for artificial agents. In V. Lesser and L. Gasser, editors, *Proc. 1st ICMAS, San Francisco, CA*. AAAI Press/MIT Press, 1995.

[23] P. R. Cohen and C. R. Perrault. Elements of a plan-based theory of speech acts. *Cognitive Science*, 3(3), 1979.

[24] L. Console, D. Theseider Dupré, and P. Torasso. On the relationship between abduction and deduction. *The Journal of Logic and Computation*, 2(5):661–690, 1991.

[25] P. T. Cox and T. Pietrzykowski. Causes for events: Their computation and applications. In *Proceedings CADE-86*, page 608, 1986.

[26] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proc. the Policy Workshop, (Policy'01), Bristol, UK*, number 1995 in Lecture Notes in Computer Science. Springer-Verlag, January 2001.

[27] Paul Davidsson. Categories of artificial societies. In Andrea Omicini, Paolo Petta, and Robert Tolksdorf, editors, *Engineering Societies in the Agents World II*, volume 2203 of *LNAI*, pages 1–9. Springer-Verlag, December 2001. 2nd International Workshop (ESAW'01), Prague, Czech Republic, 7 July 2001, Revised Papers.

[28] J. Delgado. *Agent-Based Recommender Systems and Information Filtering on the Web*. PhD thesis, Nagoya Institute of Technology, 2000.

[29] P. Dell'Acqua and L. M. Pereira. Updating agents. In *Proc. ICLP Workshop on Multi-Agent Sytems in Logic Programming (MAS'99), Las Cruces, New Mexico*, 1999.

[30] P. Dell'Acqua, F. Sadri, and F. Toni. Communicating agents. In *Proceedings ICLP Workshop on Multi-Agent Systems, Las Cruces, NM*, 1999.

[31] M. Denecker. *Knowledge representation and reasoning in incomplete logic programming*. PhD thesis, Department of Computer Science, K. U. Leuven, Belgium, 1993.

[32] M. Denecker and D. De Schreye. Representing Incomplete Knowledge in Abductive Logic Programming. In *Proc. International Logic Programming Symposium ILPS93*, pages 147–163. The MIT Press, 1993.

[33] M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for abductive logic programs. *Journal of Logic Programming*, 34(2):111–167, 1998.

[34] F. Dignum, B. Dunin-Keplicz, and R. Verbrugge. Dialogue in team formation. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, number 1916 in LNCS, pages 264–280. Springer-Verlag, 2000.

[35] J. Dix, F. Sadri, and K. Satoh, editors. *Computational Logic and Multi Agency*, volume to appear of *Special Issue of the Annals in Mathematics and Artificial Intelligence*. Baltzer Science Publishers, 2002.

[36] J. Dix and K. Satoh, editors. *Proceedings CLIMA'01, 2nd International Workshop on Computational Logic and Multi-Agent Systems*. Paphos, Cyprus, December 2001. Electronic proceedings available. URL: `http://research.nii.ac.jp/ ksatoh/clima01.html`.

[37] J. Dix, V. S. Subrahmanian, and G. Pick. Meta-agent programs. *Journal of Logic and Computation*, 2001.

[38] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77:321–357, 1995.

[39] ECL^iPS^e *User Manual Release 3.3*. ECRC, 1992.

[40] K. Eshgi and R. A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Proceedings 6th International Conference on Logic Programming*, page 234. MIT Press, 1989.

[41] P. Faratin. *Automated Service Negotiation Between Autonomous Computational Agents*. PhD thesis, Department of Electronic Engineering, Queen Mary College, University of London, UK, 2000.

[42] F. Ficarra and F. Rimondi. *La domenica non si lavora*. VOX noir. Derive e Approdi, 2000.

[43] T. Finin, Y. Labrou, and J. Mayfield. *Software Agents*, chapter KQML as an agent communication language. MIT Press, Cambridge, 1997.

[44] T. Finin and G. Wiederhold. An overview of KQML: A knowledge query manipulation language. Technical Report Research Report, Department of Computer Science, Stanford University, Stanford, CA, 1991.

[45] FIPA: Foundation for Intelligent Physical Agents. FIPA Home Page URL: `http://www.fipa.org/`.

[46] M. Fisher. A survey of concurrent MetateM – the language and its applications. In *Proc. 1st Int. Conf. Temporal Logic*, LNCS, pages 480–505. Springer-Verlag, 1994.

[47] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361, May 1998.

[48] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 1997.

[49] D. Gelernter. Generative communication in linda. *ACM Toplas*, 7(1):80–112, 1985.

[50] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[51] F. Giunchiglia and L. Serafini. Multilanguage hierarchical logics (or: How we can do without modal logics). *Artificial Intelligence*, 65:29–70, 1994.

[52] G. Gupta and V. Santos Costa. Optimal implementation of and-or parallel prolog. *Journal of Future Generation Computer Systems*, 10(1):71–92, April 1994.

[53] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.

[54] N. R. Jennings. Automated haggling: Building artificial negotiators (invited talk). In *AISB'01 Convention, York, UK*, March 2001.

[55] N. R. Jennings, P. Faratin, A.R. Lomuscio, S. Parsons, C. Sierra, and M. J. Wooldridge. Automated Negotiation: Prospects, Methods and Challenges. *International Journal of Group Decision and Negotiation*, 10(2), 2001.

[56] N. R. Jennings and M. J. Wooldridge. Applying agent technology. *Applied Artificial Intelligence*, 9:357–369, 1995.

[57] A. C. Kakas. ACLP: integrating abduction and constraint solving. In *Proceedings NMR'00, Breckenridge, CO*, 2000.

[58] A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in AI and Logic Programming*, 5:235–324, 1998.

[59] A. C. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proceedings 9th European Conference on Artificial Intelligence*. Pitman Pub., 1990.

[60] A. C. Kakas and P. Mancarella. On the relation between Truth Maintenance and Abduction. In T. Fukumura, editor, *Proceedings of the first Pacific Rim International Conference on Artificial Intelligence, PRICAI-90, Nagoya, Japan*, pages 438–443, 1990.

[61] A. C. Kakas and G. A. Papadopoulos. Parallel Abduction in Logic Programming. In *Proceedings International Symposium on Paralle Symbolic Computation, PASCO94*, pages 214–224. World Scientific Pub., 1994.

[62] J. C. De Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, September 1994.

[63] S. Konieczny. On the difference between merging knowledge bases and combining them. In *Proc. KR'00, Breckenridge, CO*, 2000.

[64] R. A. Kowalski. Problems and promises of computational logic. In *Proceedings Symposium on Computational Logic*, pages 1–36. Springer-Verlag, November 1990.

[65] R. A. Kowalski. Using metalogic to reconcile reactive with rational agents. In K. Apt and F. Turini, editors, *Meta-Logics and Logic Programming*. MIT Press, 1995.

[66] R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In D. Pedreschi and C. Zaniolo, editors, *Proc. International Workshop on Logic in Databases, San Miniato, Italy*, number 1154 in LNAI. Springer Verlag, 1996.

[67] R. A. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3/4):391–419, 1999.

[68] R. A. Kowalski, F. Sadri, and F. Toni. An agent architecture that combines backward and forward reasoning. In B. Gramlich and F. Pfenning, editors, *Proceedings CADE-15 Workshop on Strategies in Automated Deduction*, pages 49–56, 1998.

[69] R. A. Kowalski, F. Sadri, and F. Toni. Logic-based multi-agent systems. *Compulog Newsletters*, December 1998. Electronic version, URL: `http://www.cs.ucy.ac.cy/compulog/`.

[70] R. A. Kowalski, F. Sadri, F. Toni, and G. Wetzel. Towards a declarative and efficient glass-box clp language. In N. Fuchs and G. Gottlob, editors, *Proceedings Logic Programming Workshop WLP94*, 1994.

[71] S. Kraus. *Strategic Negotiation in Multi-Agent Environments*. MIT Press, Cambridge, MA, 2000.

[72] S. Kraus, K. Sycara, and A. Evenchik. Reaching agreements through argumentation; a logical model and implementation. *Artificial Intelligence*, 104:1–69, 1998.

[73] Y. Lesprance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J. P. Mueller, and M. Tambe, editors, *Intelligent Agents Volume II – Proc. ATAL'95*, LNCS. Springer-Verlag, 1996.

[74] P. Liberatore and M. Shaerf. Brels: A system for the integration od knowledge bases. In *Proc. KR'00, Breckenridge, CO*, 2000.

[75] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[76] J.P. Müller, K. Fischer, and M. Pischel. A Pragmatic BDI Architecture. In M. N. Huhns and M. P. Singh, editors, *Readings in Agents*, pages 217–225. Morgan Kaufmann, 1998.

[77] F. Pachet. Constraints and multimedia. *Logic Programming Newsletter*, pages 13–17, August 1999.

[78] S. Parsons, C. Sierra, and N. R. Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and Computation*, 8(3):261–292, 1998.

[79] J. Pearl. Embracing causality in formal reasoning. In *Proceedings National Conference on Artificial Intelligence, Seattle, WA*, pages 369–373, 1987.

[80] E. Pontelli and G. Gupta. Analysis of dependent and-parallelism. In *Proceedings ICLP'97*, pages 123–137, 1997.

[81] D. L. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1988.

[82] D. L. Poole. Compiling a default reasoning system into prolog. *New Generation Computing*, 9(1):3–38, 1991.

[83] A. Rao and M. Georgeff. An abstract architecture for rational agents. In *Proceedings of the International Workshop on Knowledge Representation (KR'92)*, 1992.

[84] A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.

[85] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484. Morgan Kaufmann Publishers: San Mateo, CA, April 1991.

[86] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers.* MIT Press, Cambridge, Massachusetts, 1994.

[87] F. Sadri and F. Toni. Computational logic and multi-agent systems: a roadmap. *Compulog Newsletters*, 7, December 1999. Electronic version, URL: `http://www.cs.ucy.ac.cy/compulog/`.

[88] F. Sadri, F. Toni, and P. Torroni. Dialogues for negotiation: agent varieties and dialogue sequences. In *Proceedings ATAL'01, Seattle, WA*, volume to appear of *LNAI, Intelligent Agents VIII (best paper award)*. Springer Verlag, 2001.

[89] F. Sadri, F. Toni, and P. Torroni. Logic agents, dialogues and negotiation: an abductive approach. In *Proceedings AISB'01 Convention, York, UK*, March 2001.

[90] T. Sandholm. *Negotiation among Self-Interested Computationally Limited Agents.* Computer science, University of Massachusetts at Amherst, September 1996.

[91] A. Sathi and M. Fox. Constraint-directed negotiation of resource reallocation. In L. Gasser and M. Huhns, editors, *Distributed Artificial Intelligence*, volume II, pages 163–195. Morgan Kaufmann, San Mateo, CA, 1989.

[92] K. Satoh and N. Iwayama. A Query Evaluation Method for Abductive Logic Programming. In K. Apt, editor, *Proc. Int. Joint Conf. and Symp. on Logic Programming*, pages 671–685. The MIT Press, 1992.

[93] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[94] SICStus prolog user manual, release 3.8.4, May 2000.
URL: `http://www.sics.se/isl/sicstus/`.

[95] SOCS: Societies Of ComputeeS. Electronically available documentation. URL: `http://lia.deis.unibo.it/Research/Projects/SOCS`.

[96] K. P. Sycara. Argumentation: Planning other agents' plans. In *Proceedings 11th International Joint-Conference on Artificial Intelligence*, pages 517–523. Morgan Kaufman, 1989.

[97] P. Tarau. Jinni: Intelligent mobile agent programming at the intersection of java and prolog. In *Proceedings PAAM'99, London, UK*, 1999.

[98] F. Toni. A semantics for the kakas-mancarella procedure for abductive logic programming. In M. Alpuente and M. I. Sessa, editors, *Proc. GULP'95*, pages 231–242, 1995.

[99] P. Torroni. A study on the termination of negotiation dialogues. In *Proceedings AAMAS, Bologna, Italy*, 2002. Accepted for publication. Preliminary version available as Techical Report DEIS-LIA-01-008, University of Bologna (Italy), LIA Series no. 52.

[100] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[101] D. N. Walton and E. C. W. Krabbe. *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning.* State University of New York Press, Albany, NY, 1995.

[102] M. J. Wooldridge. Semantic issues in the verification of agent communication languages. *Journal of Autonomous Agents and Multi-Agent Systems*, 2000.

[103] M. J. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[104] M. J. Wooldridge and S. Parsons. Languages for negotiation. In W. Horn, editor, *Proceedings 14th European Conference on Artificial Intelligence, Berlin, Germany.* IOS Press, August 2000.

[105] G. Zlotkin and J. S. Rosenschein. Mechanisms for automated negotiation in state oriented domains. *Journal of Artificial Intelligence Research*, 5:163–238, 1996.