A Computational Logic-based System for Specification and Verification of Agent Interaction

Marco Alberti

Acknowledgements

I wish to thank all the people who have supported me during the three years of my Ph.D.

The first person that I would like to thank is Evelina Lamma, who was my Artificial Intelligence teacher, my *tesi di laurea* advisor, and is now my Ph.D. supervisor. I thank her for helping me continuously in all aspects of my work, and for encouraging and guiding me in this experience.

Then, I would like to thank all my co-authors: besides Evelina, Paola Mello, Anna Ciampolini, Michela Milano, Marco Gavanelli, Paolo Torroni, Federico Chesani, Alessio Guerri, and Davide Daolio, for all that I learned from them and for sharing enjoyable working time with me.

Most of my work during this Ph.D. was done in the context of the SOCS (IST-2001-32530) European project. This was an occasion for me to meet and work with many prominent people and talented researchers in the field of Computational Logic, and to share with them a complex and rewarding experience: thanks to all the SOCS people from the Universities of Ferrara, Bologna, Pisa, and Cyprus, and from the Imperial College and the City University in London.

And I wish to thank all the many people who have shared some of their time with me, in enjoyable and inspiring conversations.

This thesis would not have been possible without the collaboration of the people involved in SOCS in Ferrara and Bologna. During the past three years I worked in so close collaboration with them that it has been impossible for me to isolate my contribution to the work while keeping the thesis reasonably self-contained. As a consequence, part of the thesis describes joint work with other people: with Marco Gavanelli, Evelina Lamma, Paola Mello and Paolo Torroni for the formal $\mathcal{S}$CIFF framework, with Marco Gavanelli for the implementation

of the $\mathcal{S}$CIFF unification, with Marco Gavanelli, Alessio Guerri and Michela Milano for the combinatorial auction protocols, with Anna Ciampolini for the social ACL semantics, with Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello and Paolo Torroni for the automatic proof of properties. Moreover, I gave no significant contribution to the theoretical definition of the $\mathcal{S}$CIFF and g-$\mathcal{S}$CIFF proof procedures (defined by Marco Gavanelli, Evelina Lamma, Paola Mello and Paolo Torroni), and to the implementation of the Java side of the *SOCS-SI* system, which is Federico Chesani's work. I also wish to thank Tarin Gamberini and Valentina Maraldi, who worked as M.Sc. students on SOCS-related issues at the ENDIF in Ferrara, for their useful feedback on the $\mathcal{S}$CIFF framework.

My work has been supported financially by the European Commission within the SOCS project (IST-2001-32530), part of the Global Computing Programme, and by the Italian MIUR, within the COFIN 2003 project *La Gestione e la negoziazione automatica dei diritti sulle opere dell'ingegno digitali: aspetti giuridici e informatici.*

Abstract

Open agent societies are multiagent systems where no assumptions can be made on the agents that join them. Therefore, when specifying and verifying of open agent societies, it is necessary to abstract away from the internal architecture and policies of the individual agents, and to adopt an external perspective, focusing on the observable agent behaviour.

Such an external (or social) point of view was adopted in $\mathcal{S}$CIFF, the social framework developed in the SOCS (IST-2001-32530) project.

The $\mathcal{S}$CIFF framework is based on Computational Logic, in an attempt to bridge the gap between declarative specification an operational verification. In particular, $\mathcal{S}$CIFF exploits abduction, a concept used in hypothetical reasaoning, which appears an appropriate paradigm to reason about open agent societies, when the agent behaviour cannot be predicted or enforced, but only hypothesised. $\mathcal{S}$CIFF features a language, with an abductive declarative semantics, for the specification of agent interaction; its operational counterpart, an abductive proof procedure, can be used for verifying that the agent behaviour is compliant to the specification. The $\mathcal{S}$CIFF proof procedure has also been extended in order to synthesize a compliant agent behaviour, rather than only checking the compliance of a given behaviour; this extension, called g-$\mathcal{S}$CIFF, can be used for verifying protocol properties.

In this thesis, we describe the $\mathcal{S}$CIFF framework, focusing on the implementation of the $\mathcal{S}$CIFF proof procedure and on the specification and verification of two aspects of agent interaction: the social semantics of Agent Communication Languages, and Interaction Protocols.

Contents

1 Introduction **11**

1.1 Specification and Verification of Open MAS . . . 11

1.1.1 Multiagent Systems . . . 11

1.1.2 Open Agent Societies . . . 12

1.2 Abduction in SOCS . . . 15

1.3 Content of this thesis . . . 16

2 The $\mathcal{S}$CIFF Social Framework **19**

2.1 Preliminaries . . . 19

2.2 Syntax . . . 20

2.2.1 Representation of the agent behaviour . . . 20

2.2.1.1 Events . . . 20

2.2.1.2 Expectations . . . 21

2.2.2 Social specifications . . . 23

2.2.2.1 Social Knowledge Base . . . 23

2.2.2.2 Social Integrity Constraints . . . 24

2.2.2.3 Social Specification . . . 27

2.3 Declarative Semantics . . . 27

2.4 Related work . . . 30

3 The $\mathcal{S}$CIFF proof procedure **33**

3.1 Data Structures . . . 33

3.1.1 Initial Node and Success . . . 34

3.2 Variables . . . 35

3.3 Transitions . . . 37

3.3.1 IFF-like transitions . . . 38
3.3.2 Dynamically growing history . . . 40
3.3.3 Fulfillment and Violation . . . 42
3.3.4 Consistency . . . 44
3.3.5 CLP . . . 44
3.4 $\mathcal{S}$CIFF properties . . . 45
3.4.1 Soundness of $\mathcal{S}$CIFF . . . 46
3.4.2 Termination of $\mathcal{S}$CIFF . . . 46

4 Constraint Handling Rules 47
4.1 Syntax and semantics . . . 47
4.2 The SICStus Prolog *CHR* library . . . 48

5 $\mathcal{S}$CIFF Implementation 51
5.1 Overview . . . 51
5.1.1 Technology . . . 51
5.1.2 Adapting the $\mathcal{S}$CIFF execution to Prolog . . . 53
5.2 Representation of an instance . . . 54
5.3 Data Structures . . . 54
5.3.1 Resolvent . . . 55
5.3.2 Constraint Store . . . 55
5.3.3 Proof sets: PSIC, **HAP**, **PEND**, **FULF**, **VIOL** . . . 55
5.3.3.1 Partially Solved Integrity Constraints . . . 56
5.3.3.2 History . . . 57
5.3.3.3 Pending Expectations . . . 58
5.3.3.4 Fulfilled Expectations . . . 59
5.3.3.5 Violated Expectations . . . 59
5.4 Variables . . . 59
5.5 Transitions . . . 60
5.5.1 IFF-like Transitions. . . . 60
5.5.2 Dynamically Growing History. . . . 64
5.5.3 Fulfillment and violation. . . . 65
5.5.4 Consistency . . . 68

5.6 The SOCS-SI system 69
5.7 Related work 70

6 Applications 73
6.1 Social ACL Semantics 73
6.1.1 Agent Communication Languages 73
6.1.1.1 Mentalistic semantics 74
6.1.1.2 Social semantics 75
6.1.2 Social ACL semantics with $\mathcal{S}$CIFF 77
6.2 Definition of Protocols 82
6.2.1 Formalisms for Agent Interaction Protocols 82
6.2.2 Conventions for describing interactions 84
6.2.3 Semi-open society 85
6.2.4 FIPA Request Interaction Protocol 86
6.2.5 NetBill 89
6.2.6 The Needham-Schroeder Public Key protocol 92
6.2.7 First Price Sealed Bid auction 99
6.2.8 Combinatorial auctions 99
6.2.8.1 Basic Combinatorial Auction 101
6.2.8.2 Double combinatorial auction 103
6.2.8.3 Combinatorial auction with NetBill 104

7 $\mathcal{S}$CIFF performance 107
7.1 $\mathcal{S}$CIFF Complexity 107
7.2 Experimental results 109
7.2.1 The effect of the branching factor 109
7.2.2 The effect of the number of events 112

8 g-$\mathcal{S}$CIFF 117
8.1 The g-$\mathcal{S}$CIFF proof procedure 117
8.1.1 Formal results 119
8.2 g-$\mathcal{S}$CIFF implementation 120

9 Using g-$\mathcal{S}$CIFF **123**

9.1 The g-$\mathcal{S}$CIFF approach 123

9.2 Case studies 125

9.2.1 Verifying the NetBill protocol 125

9.2.2 Verifying the Needham-Schroeder Protocol 127

9.3 Related work. 129

10 Conclusions **131**

10.1 Summary 131

10.2 Future research 132

Chapter 1

Introduction

In this chapter, we define the setting of our work, and we outline the content of the thesis.

1.1 Specification and Verification of Open Multiagent Systems

1.1.1 Multiagent Systems

The research area of Multiagent Systems [Woo02], is aimed at modelling, designing and implementing complex systems of interacting computational entities at a high abstraction level. The high level of abstraction used in these models often leads to viewing such systems as *societies* of *agents*.

The Multiagent Systems (also MAS, for short, in the following) research employs notions and methods borrowed from Distributed Systems, Artificial Intelligence, Economics, Game Theory and Social Sciences, together with newly developed ones.

The current status of the MAS field can be seen as the result of an evolution of concepts and models developed since the late 1970s.

The need for modelling complex systems emerged in the field of Distributed Problem Solving (DPS) where problems are addressed by coordinating several computational entities. Notable examples of this approach are *market models* [Wel93, WW98], mainly applied to resource reallocation, and the *Contract*

Net model [IS00, Smi80], mainly applied to task distribution.

Dependence-based models [SCDC98, Sic01] focus on dependence between agents (i.e., the capability of an agent to facilitate or prevent the achievement of another agent's goals).

A milestone in the modelling of MAS is the *Belief, Desire, Intention* (BDI) model [RG92a], which, in many variations, is still commonly used nowadays. BDI architectures use modal logic to represent agents as having a *mental state*, composed of *beliefs* about the external world (including other agents' mental states), *desires* on goals to achieve and *intentions* on which actions to perform.

Organisational models [DMW02] focus more on the social aspect of the agent interaction, using Deontic Logic [Wri51] to specify norms and their dependence on roles. *Institutions* [EdlCS02, NS02] are entities that facilitate, oversee and enforce commitments among agents.

Despite the considerable research effort put into MAS by academic and industrial entities, and several attempts of a standardisation (FIPA [FIP] being the most notable), even the definition of *agent* is still an open issue. A common perspective is the one found in [Woo02], which characterises an agent as an *autonomous* (capable of acting independently in some environment) entity that is *reactive* (able to respond to changes in the environment), *pro-active* (exhibiting a goal-directed behaviour) and *social* (aware of the existence of other agents, and able to fruitfully interact with them).

However, we will not discuss the essence of the individual agent any further, for two reasons. First, it is beyond the scope of this thesis; second, in this thesis we concentrate on a particular kind of multiagent systems which necessarily abstracts away from the internals of the individual agents and focuses on the agent *externally observable* behaviour and interactions: *open* multiagent systems.

1.1.2 Open Agent Societies

Openness in Agent Societies. In the MAS literature, the term *openness* has been used with several meanings. In the following, we summarise two of the most widely accepted definitions.

According to [Dav01], in an open (artificial) society "there are no restrictions

for agents/processes to join/leave the society". This means that it is possible for any agent to enter the society simply by starting an interaction with a member of it.

Another widely accepted definition of openness in agent societies is that given in [APS02] (derived from [Hew91]) where an agent society is open if three properties hold:

1. the behaviour of members and their interactions are unpredictable (i.e., the execution of the society is non-deterministic);

2. the internal architecture of each member is neither publicly known nor observable (i.e., members may have heterogeneous architectures);

3. members of the society do not necessarily share common goals, desires or intentions (i.e., each member may conflict with others when trying to reach its own purposes).

While these two concepts of openness are in principle independent, it is reasonable for a society that is open in one sense to be open in the other sense, too. If there is no restriction for joining the society, then also unknown (or even malicious) agents could join it; on the other hand, if unknown (or even malicious) agents are eligible members of the society, then it would not be very useful to impose other restrictions to joining the society.

In fact, in the following, we will consider agent societies that are open in both senses.

Specification of Open Agent Societies. The openness of a society impacts on what can be specified of it, and on what needs to be specified.

Since the internal architecture, internal state or policies of the members are not, in general, accessible, it is not possible to give the specification of an open society by constraining the internal state of the members, but it is only possible to constrain their externally observable behaviour, i.e., their actions. A good example of the difference is found in the two most common approaches to the definition of the semantics for Agent Communication Languages (ACLs): the *mentalistic* and the *social* (see Sect. 6.1) .

For the specification of agent interaction from an external perspective, we propose the $\mathcal{S}$CIFF social framework (see Ch. 2).

Verification of Open Agent Societies. In [GP02a, GP02b], F. Guerin and J. Pitt propose a classification of properties that are relevant for e-commerce systems, in particular with respect to properties of protocols and interaction.

Verification of properties is classified into three types, depending on the information available and whether the verification is done at design time or at run time:

Type 1: verify that an agent will always comply;

Type 2: verify compliance by observation;

Type 3: verify protocol properties.

Type 1 verification can be performed at design time. Given a representation of the agent, by means of some proof technique (such as *model checking* [Mer01]) it proves that the agent will always exhibit the desired behaviour. Since we focus on open agent societies, where the internals of the members are in general not accessible, we do not consider *Type 1* verification in this thesis.

Type 2 verification can be performed at runtime. It checks that the *actual* agent behaviour being observed is compliant to some specification. It does not require any knowledge about the agent internals, and is thus viable in open agent societies. For this purpose, the $\mathcal{S}$CIFF proof procedure has been defined (see Ch. 3) and implemented (see Ch. 5).

Type 3 verification can be performed at design time. It proves that some property will hold in the society, provided that the agents follow the interaction protocols (i.e., behave accordingly to the interaction specification). This kind of verification is independent of the agent internals and is thus viable in agent societies. For this purpose, we have developed the g-$\mathcal{S}$CIFF proof procedure (see Ch. 8).

1.2 Abductive Logic-based Specification and Verification in SOCS

Computational Logic is a cumulative term which denotes the applications of Logic to Computer Science. As such, it is a huge research area, applied to virtually all fields of computer science, ranging from the foundational aspects (such as denotational semantics of programming languages) to the practical (such as the verification of the behaviour of concrete systems by model checking).

However, in this thesis, by Computational Logic we will refer to the set of models and techniques originating from the field of Logic Programming [Llo87], and in particular Abductive Logic Programming [KKT93] and Constraint Logic Programming [JM94].

The main practical advantage of the use of Logic Programming and its extensions has been identified in its joining two aspects:

- a clear and simple declarative semantics, which makes programs easier to understand and to reason about (also automatically);
- an operational counterpart (in the form of a proof procedure), which makes the declarative specification directly executable.

This has also been the main motivation for choosing Computational Logic as the paradigm for the specification and verification of open multiagent systems in the context of the European SOCS project [SOC]: to bridge the gap between declarative specification and operational verification, which, in the literature are often kept separate.

In particular, Abductive Logic Programming, which is a powerful technique for hypothetical reasoning, appeared as a natural choice for representing the specification of open societies, where the behaviour of the individual agents cannot be predicted or enforced, but only hypothesised. Constraint Logic Programming let us express quantitative requirements on the agent behaviours (most notably, time deadlines), exploiting the efficient computing machinery available from the literature.

The result of the work done in the context of the SOCS project in the nodes of Bologna and Ferrara is the $\mathcal{S}$CIFF framework, which provides:

- a language equipped with an abductive declarative semantics, for the specification of agent interaction;
- a (theoretical) operational semantics for the language, in the form of an abductive proof procedure;
- the implementation of the proof procedure based on SICStus Prolog [SIC03] and *CHR* [Frü98]: such operational semantics has been used to achieve what is referred to as *type 2* verification in Sect. 1.1.2, i.e, *on-the-fly* verification;
- the integration of the proof procedure in SOCS-SI, a graphical, networked system which has been integrated in multiagent systems;
- several examples and case studies of verification of well known agent interaction protocols;
- an extension of $\mathcal{S}$CIFF which supports the synthesis of an agent behaviour compliant to a specification, which can be used to verify protocol properties.

1.3 Content of this thesis

The main aim of this thesis is to report on the implementation of the $\mathcal{S}$CIFF and g-$\mathcal{S}$CIFF proof procedures, and on the application of the $\mathcal{S}$CIFF framework to the specification and verification of agent interaction.

In Ch. 2, we describe the syntax and declarative semantics of the $\mathcal{S}$CIFF social framework.

The operational semantics of the framework, consisting of the $\mathcal{S}$CIFF Abductive proof procedure, is described in Ch. 3.

The description of the implementation of $\mathcal{S}$CIFF, in Ch. 5, follows a brief introduction to the *Constraint Handling Rules* language, used for the implementation.

In Ch. 6 we show several examples of social specifications by means of the $\mathcal{S}$CIFF framework, focusing on social semantics of agent communication languages and interaction protocols.

Ch. 8 presents g-$\mathcal{S}$CIFF, the extension of $\mathcal{S}$CIFF that can synthesise an agent behaviour compliant to a specification. Ch. 9 shows how g-$\mathcal{S}$CIFF can be used to prove protocol properties (*type 3* verification in Sect. 1.1.2), with some case studies.

Chapter 2

The $\mathcal{S}$CIFF Social Framework

In this chapter, we briefly describe the $\mathcal{S}$CIFF abductive logic framework, developed in the SOCS project [SOC] for the specification of interaction in open multiagent systems. We describe the syntax and the declarative semantics of the language; the operational semantics is described in Ch. 3.

To illustrate the concepts, we take examples from the simple *query_ref* social specification.

A discussion of the motivations behind the language choices made for the framework can be found in [AGL+04b, ACG+05b].

2.1 Preliminaries

In the remainder of the thesis, we assume a basic familiarity with the concepts, results and conventions of Logic Programming. A good introduction is that provided by Lloyd [Llo87].

The words *integer*, *variable*, *term*, *atom* will be used in the following with their usual meaning in Logic Programming [Llo87].

A *restriction* is an atom whose signature belongs to a set $\mathcal{R}$ of *restriction signatures.*

An *abductive logic program* [KKT93] is a triple $\langle P, Ab, IC \rangle$ where:

- P is a (normal) logic program, that is, a set of clauses of the form $A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_{m+n}$, where $m, n \geq 0$, each A_i $(i = 1, \ldots, m+n)$ is an atom, and all variables are implicitly universally quantified with

scope the entire clause. A_0 is called the *head* and $A_1, \ldots, A_m$, *not* $A_{m+1}, \ldots,$ *not* A_{m+n} is called the *body* of any such clause.

- *Ab* a set of *abducible predicates*, p, such that p is a predicate in the language of P which does not occur in the head of any clause of P (without loss of generality, see [KKT93]).

- IC is a set of integrity constraints, that is, a set of sentences in the language of P.

Abducible predicates (or simply abducibles) are the predicates about which assumptions (or abductions) can be made. These predicates carry all the incompleteness of the domain, they can have a partial definition or no definition at all, while all other predicates have a complete definition in the logic program.

Given an abductive logic program $T = \langle P, Ab, IC \rangle$ and a formula G, the goal of abduction is to find a (possibly minimal) set of ground atoms Δ (*abductive explanation*) in predicates in *Ab* which, together with P, "entails" G, i.e., $P \cup \Delta \models G$, and such that $P \cup \Delta$ "satisfies" IC, e.g. $P \cup \Delta \models IC$ (see [KKT93] for other possible notions of integrity constraint "satisfaction"). Here, the notion of "entailment" $\models$ depends on the semantics associated with the logic program P (there are many different choices for such semantics, as it is well-documented in the Computational Logic literature).

2.2 Syntax

In this section, we define the syntax of the logic language used in the $\mathcal{S}$CIFF framework. The language is composed of entities for expressing:

- the actual and expected agent behaviour;

- a specification of the agent behaviour.

2.2.1 Representation of the agent behaviour

2.2.1.1 Events

Events are the abstraction used to represent the actual agent behaviour.

Definition 2.2.1 *An* event *is an atom whose*

- *functor is* **H***;*
- *first argument is a ground term;*
- *second (optional) argument is an integer.*

Intuitively, the first argument is meant to represent the description of the happened event, according to application-specific conventions, and the second argument is meant to represent the time at which the event has happened.

A *negated event* is an event with the unary prefix operator **not** applied to it. As will be clear from the declarative semantics, this type of negation is negation by failure.

We will usually call set of events a *history*, and often denote it with the symbol **HAP**.

Example 2.2.2

$$\mathbf{H}(tell(alice, bob, query_ref(phone_number), dialog_id), 10) \tag{2.2.1}$$

could represent the fact that alice asked bob his phone_number with a query_ref message, in the context identified by dialog_id, at time 10.

2.2.1.2 Expectations

Expectations are the abstraction used to represent the desired agent behaviour. The choice of the term “expectation” is due to the fact that agents are autonomous, and thus their behaviour cannot be enforced, but only expected, to be compliant to a specification. For the formal meaning of expectations, see Sect. 2.3, or [AGL+03a].

Expectations are of two types:

- *positive*: representing some event that is expected to happen;
- *negative*: representing some event that is expected *not* to happen.

Definition 2.2.3 *A* positive expectation *is an atom whose*

- *functor is* **E***;*
- *first argument is a term;*
- *second (optional) argument is a variable or an integer.*

A *negated positive expectation* is a positive expectation with the unary prefix operator $\neg$ applied to it. As will be clear from Def. 2.3.2, the kind of negation used here is *explicit negation*.

Example 2.2.4 *The atom*

$$\mathbf{E}(tell(bob, alice, inform(phone_number, Answer), dialog_id), Ti) \tag{2.2.2}$$

could represent an expectation for bob to inform alice that the value for the piece of information identified by phone_number is Answer, in the context identified by dialog_id, at time Ti.

Definition 2.2.5 *A* negative expectation *is an atom whose*

- *functor is* **EN***;*
- *first argument is a term;*
- *second (optional) argument is a variable or an integer.*

A *negated negative expectation* is a negative expectation with the unary prefix operator $\neg$ applied to it. As in the case of negated positive expectations, the negation is explicit (see Def. 2.3.2).

Example 2.2.6 *The atom*

$$\mathbf{EN}(tell(bob, alice, refuse(phone_number), dialog_id), Tr) \tag{2.2.3}$$

could represent that bob is expected not *to refuse to alice his phone_number, in the context identified by dialog_id, at* any *time.*

As the examples show, expectations can contain variables, as it might be desirable to leave the expected agent behaviour not completely specified.

The syntax of events and expectations is definitions are summarised in Spec. 2.1. The definitions there contained are valid also in Specs. 2.2 and 2.4.

Specification 2.1 Syntactical entities for events and expectations

$$\begin{aligned}
Literal &::= [\mathbf{not}]Atom \\
Event &::= \mathbf{H}(GroundTerm[, Integer]) \\
EventLiteral &::= [\mathbf{not}]Event \\
PosExp &::= \mathbf{E}(Term[, Variable \mid Integer]) \\
NegExp &::= \mathbf{EN}(Term[, Variable \mid Integer]) \\
PosExpLiteral &::= [\neg]PosExp \\
NegExpLiteral &::= [\neg]NegExp \\
ExpLiteral &::= PosExpLiteral \mid NegExpLiteral
\end{aligned}$$

2.2.2 Social specifications

A social specification, i.e, a specification of the agent behaviour in the $\mathcal{S}$CIFF framework, is composed of two elements:

- A *Social Knowledge Base*;
- A set of *Social Integrity Constraints*.

2.2.2.1 Social Knowledge Base

The Social Knowledge Base is a logic program, extended in that the body of the clauses can contain expectation literals.

Intuitively, the social knowledge base can be used to express declarative knowledge about the agent society, ranging from simple information such as the value of time parameters, to complex organisational knowledge such as that regarding roles.

Specification 2.2 Syntax of the Social Knowledge Base

$$\begin{aligned}
KB_S &::= [Clause]^\star \\
Clause &::= Atom \leftarrow Body \\
Body &::= ExtLiteral\ [\ \wedge\ ExtLiteral\ ]^\star \\
ExtLiteral &::= Literal \mid ExpLiteral \mid Restriction
\end{aligned}$$

The syntax of the Social Knowledge Base is given in Fig. 2.2.

Allowedness conditions The syntactic restrictions defined in the following are motivated by the operational semantics of the framework (see Ch. 3), and will be supposed to hold throughout the thesis.

Definition 2.2.7 *A clause Head* ← *Body is* allowed *if every variable that occurs in a negative Literal of a definite predicate in Body, also occurs in at least one positive Literal, or in a PosExpLiteral, or in Head.*

Definition 2.2.8 *A Clause is* restriction allowed *if the variables that are universally quantified with scope the body do not occur in Restrictions, and each variable that occurs in a Restriction also occurs in at least one PosExp in the body.*

Variable quantification and scope The quantification and scope of variables is implicit. In each clause, the variables are quantified as follows:

- universally, if they occur only in negative expectations (and possibly restrictions), with scope the *Body*;
- otherwise universally, with scope the entire *Clause*.

Specification 2.3 Social Knowledge Base for the *query_ref* social specification

$$qr_deadline(10).$$

Example 2.2.9 *Spec. 2.3 shows a simple example of a social knowledge base, which defines the qr_deadline/1 predicate by means of one fact.*

2.2.2.2 Social Integrity Constraints

Social Integrity Constraints (also SICs, for short, in the following) are implications that, operationally, are used as forward rules, as will be explained in Ch. 3.

Specification 2.4 Syntax of the Social Integrity Constraints

$$
\begin{array}{rcl}
\mathcal{IC}_S & ::= & [SIC]^\star \\
SIC & ::= & Body \rightarrow Head \\
Body & ::= & (EventLiteral \mid ExpLiteral)\ [\ \wedge\ BodyLiteral\ ]^\star \\
BodyLiteral & ::= & EventLiteral \mid ExpLiteral \mid Literal \mid Restriction \\
Head & ::= & HeadDisjunct\ [\ \vee HeadDisjunct\ ]^\star \mid false \\
HeadDisjunct & ::= & ExpLiteral\ [\ \wedge(ExpLiteral \mid Restriction)]^\star
\end{array}
$$

Declaratively, their main use is to specify that is some set of events happens, then one of several other sets of events is expected to happen, or not to happen.

The syntax of Social Integrity Constraints is given in Fig. 2.4.

Given a social integrity constraint *Body* → *Head*, we will sometimes call *Body* its *condition*, and *Head* its *conclusion*.

Allowedness conditions As in the case of the Social Knowledge Base syntax, following syntactic restrictions are motivated by the operational semantics, and will be supposed to hold throughout the thesis.

A variable cannot occur in a Social Integrity Constraint only in negative, definite literals, but it must always appear in literals with predicates **H**, **E**, **EN**.

Definition 2.2.10 *A Social Integrity Constraint Body → Head is* quantifier allowed *if*

- *each variable that occurs in a PosExpLiteral in Head does not occur in Body, except possibly in Events or in PosExpLiterals;*
- *each variable that occurs in a negative Literal in Body also occurs in at least one Event or PosExpLiteral in Body.*

Definition 2.2.11 *A social integrity constraint is* restriction allowed *if*

- *all the variables that are universally quantified with scope Body do not occur in Restrictions;*

- *the other variables (that occur only in Head, or both in Head and in the Body) can occur in Restrictions. Each Restriction occurring in the social integrity constraint should:*
  - *either involve only variables that also occur in PosExpLiterals or Events,*
  - *or involve* one *variable that also occurs in at least one NegExpLiteral, and possibly other variables which only occur in Events.*

Variable quantification and scope The rules of scope and quantification for the variables in a social integrity constraint *Body* → *Head* are as follows:

1. Each variable that occurs both in *Body* and in *Head* is quantified universally, with scope the social integrity constraint.
2. Each variable that occurs only in *Head* must occur in at least one *ExpLiteral*, and
   - if it occurs in *PosExpLiteral*s, it is quantified existentially and has as scope the disjunct where it occurs;
   - otherwise it is quantified universally.
3. Each variable that occurs only in *Body* is quantified with scope *Body* as follows:
   (a) universally, if it occurs only in negative *EventLiteral*s, *NegExpLiteral*s or *Restriction*s;
   (b) existentially, otherwise.

Example 2.2.12 *Spec. 2.5 shows the SICs for the query_ref social specification.*

Intuitively, the first SIC means that if agent A sends to agent B a query_ref message, then B is expected to reply with either an inform or a refuse message by TD time units later, where TD is defined in the Social Knowledge Base by the qt_deadline predicate (with the example in Spec. 2.3, the value of TD would be 10*).*

The second SIC means that, if an agent sends an inform message, then it is expected not to send a refuse message at any time.

Specification 2.5 Integrity Constraints for the *query_ref* social specification.

$$
\begin{aligned}
& \mathbf{H}(tell(A, B, query_ref(Info), D), T) \wedge \\
& qr_deadline(TD) \\
\rightarrow\ & \mathbf{E}(tell(B, A, inform(Info, Answer), D), T1) \wedge \\
& T1 < T + TD \\
\vee\ & \mathbf{E}(tell(B, A, refuse(Info), D), T1) \wedge \\
& T1 < T + TD
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{H}(tell(A, B, inform(Info, Answer), D), Ti) \\
\rightarrow\ & \mathbf{EN}(tell(A, B, refuse(Info), D), Tr)
\end{aligned}
$$

2.2.2.3 Social Specification

Given a Social Knowledge Base KB_S and a set of $\mathcal{IC}_S$ of Social Integrity Constraints, we call the pair $\langle KB_S, \mathcal{IC}_S \rangle$ a *Social Specification.* We will often use the symbol $\mathcal{S}$ to denote a social specification.

Definition 2.2.13 *A social specification $\mathcal{S} = \langle KB_S, \mathcal{IC}_S \rangle$ is* quantifier allowed *if all the social integrity constraints in $\mathcal{IC}_S$ are quantifier allowed. $\mathcal{S}$ is* restriction allowed *if all the clauses in KB_S and all the social integrity constraints in $\mathcal{IC}_S$ are restriction allowed. $\mathcal{S}$ is* allowed *if it is quantifier allowed and restriction allowed, and KB_S is allowed.*

2.3 Declarative Semantics

In the following, we briefly summarise the (abductive) declarative semantics of the $\mathcal{S}$CIFF framework, which is inspired by other abductive frameworks, but introduces the concept of fulfillment, used to express a correspondence between the expected and the actual agent behaviour.

A more detailed description of the semantics can be found in [AGL$^+$03a].

Definition 2.3.1 *Given a social specification $\mathcal{S} = \langle KB_S, \mathcal{IC}_S \rangle$ and a history* **HAP***, $\mathcal{S}_{\mathbf{HAP}}$ represents the pair $\langle \mathcal{S}, \mathbf{HAP} \rangle$, called the* **HAP**-instance of $\mathcal{S}$.

The following definition implements explicit negation [AB94] for expectation atoms.

Definition 2.3.2 *A set* **EXP** *of expectations is* ¬-consistent *if and only if for each (ground) term p:*

$$\{\mathbf{E}(p), \neg\mathbf{E}(p)\} \not\subseteq \mathbf{EXP} \qquad \textit{and} \qquad \{\mathbf{EN}(p), \neg\mathbf{EN}(p)\} \not\subseteq \mathbf{EXP}. \tag{2.3.1}$$

The following definition prevents the same event from being both expected to happen and expected not to happen.

Definition 2.3.3 *A set* **EXP***of expectations is* **E**-consistent *if and only if for each (ground) term p:*

$$\{\mathbf{E}(p), \mathbf{EN}(p)\} \not\subseteq \mathbf{EXP} \tag{2.3.2}$$

The following definition establishes a link between the actual and the expected agent behaviour, by requiring positive expectations to be matched by events, and negative expectations not to be matched by events.

Definition 2.3.4 *Given a history* **HAP***, a set* **EXP** *of expectations is* **HAP**-fulfilled *if and only if*

$$Comp(\mathbf{EXP} \cup \mathbf{HAP}) \cup \mathcal{IC}_S \cup \text{CET} \not\models \textit{false} \tag{2.3.3}$$

where Comp represents the completion *of a theory [Kun87], and* CET *is Clark's equational theory [Cla78].*

Otherwise, **EXP** *is* **HAP**-violated*.*

When **HAP** is apparent from the context, we will often omit mentioning it.

The following definition requires consistence of the set of expectations, with respect to an instance of the social specification.

Definition 2.3.5 *Given a social specification* $\mathcal{S} = \langle KB_S, \mathcal{IC}_S \rangle$*, and an instance* $\mathcal{S}_{\mathbf{HAP}}$ *of* $\mathcal{S}$*, a set* **EXP** *of expectations is* $\mathcal{S}_{\mathbf{HAP}}$-consistent *if and only if*

$$Comp(KB_S \cup \mathbf{HAP} \cup \mathbf{EXP}) \cup \text{CET} \models \mathcal{IC}_S \tag{2.3.4}$$

The following definition supports goal-directed social specifications: it requires the instance of the specification to entail a goal, while being consistent with respect to the previous definitions.

Definition 2.3.6 *Given a social specification* $\mathcal{S} = \langle KB_S, \mathcal{IC}_S \rangle$*, and an instance* $\mathcal{S}_{\mathbf{HAP}}$ *of* $\mathcal{S}$*, a goal* $\mathcal{G}$ *is* achieved *in* $\mathcal{S}_{\mathbf{HAP}}$ *if there exists a* $\neg$*-consistent,* **E**-*consistent,* $\mathcal{S}_{\mathbf{HAP}}$*-consistent and* **HAP**-*fulfilled set* **EXP** *of expectations such that*

$$Comp(KB_S \cup \mathbf{EXP}) \cup \text{CET} \models \mathcal{G} \quad (2.3.5)$$

In this case, we write $\mathcal{S}_{\mathbf{HAP}} \vDash_{\mathbf{EXP}} \mathcal{G}$ *and we say that* **HAP** *is* compliant *to* $\mathcal{S}$ *with respect to* $\mathcal{G}$*.*

In the remainder of this thesis, when we simply say that a history **HAP** is compliant to a social specification $\mathcal{S}$, we will mean that **HAP** is compliant to $\mathcal{S}$ with respect to the goal true. This will usually be the case when the specification is used to express an interaction protocol, with no particular social goal. We will often say that a history **HAP** *violates* a specification $\mathcal{S}$ to mean that **HAP** is not compliant to $\mathcal{S}$.

The following definitions identifies ill-defined social specifications, i.e., those for which there is no compliant history, which are obviously undesirable from an agent society designer viewpoint.

Definition 2.3.7 (Well-definedness w.r.t. a goal) *Given a goal* $\mathcal{G}$*, a social specification* $\mathcal{S}$ *is well-defined with respect to* $\mathcal{G}$ *iff there exists at least one history that is compliant to* $\mathcal{S}$ *w.r.t.* $\mathcal{G}$*, i.e., iff:*

$$\exists \mathbf{HAP}\ \exists \mathbf{EXP} \mathcal{S}_{\mathbf{HAP}} \vDash_{\mathbf{EXP}} \mathcal{G} \quad (2.3.6)$$

In the remainder of this thesis, when we simply say that a social specification $\mathcal{S}$ is well defined, we will mean that $\mathcal{S}$ is well defined with respect to the goal true.

Example 2.3.8 *The query_ref social specification* $\mathcal{S} = \langle KB_S, \mathcal{IC}_S \rangle$*, where* KB_S *is defined in Spec. 2.2.9, and* $\mathcal{IC}_S$ *is defined in Spec. 2.5, is well defined. For instance, the history*

$$\begin{array}{l} \{\mathbf{H}(tell(alice, bob, query_ref(phone_number), dialog_id), 10), \\ \ \mathbf{H}(tell(bob, alice, inform(phone_number, 5551234), dialog_id), 12)\} \end{array} \quad (2.3.7)$$

is compliant to $\mathcal{S}$*.*

A note on terminology

As will be shown in Sect. 6.2, one of the main uses of the $\mathcal{S}$CIFF framework is to define Interaction Protocols. For uniformity with the existing literature, we will often use the expression “interaction protocol”, or simply “protocol” as a substitute for “social specification”.

2.4 Related work

In this section, we compare the $\mathcal{S}$CIFF framework with other frameworks, related to it as far as objectives and methodologies.

Several researchers have studied the concepts of norms, commitments and social relations in the context of Multi-Agent Systems [CFS99]. Furthermore, a lot of research has been devoted in proposing architectures for developing agents with social awareness (see, for instance [CDJT99]). Our approach can be conceived as complementary to these efforts, since instead of proposing a specific architecture for designing computees, our work in mainly focused on the definition of a society infrastructure based on Computational Logic for regulating and improving robustness of interaction in an open environment, where the internal architecture of the computees might be unknown.

Our work is very close, as far as the objectives and methodology, to the work on computational societies presented and developed in the context of the ALFEBIITE project [ALF99], and the work by Singh and co-authors [YS02] where a social semantics is exemplified by using a commitment-based approach. With these works we share the same view of an open society as that of [APS02]. However, our work is especially oriented to computational aspects, and it was developed with the purpose of providing a computational framework that can be directly used for automatic verification of properties such as compliance (see Ch. 3).

In [APS02], Artikis et al. present a theoretical framework for providing executable specifications of particular kinds of multi-agent systems, called open computational societies, and present a formal framework for specifying, animating and reasoning about and verifying the properties of systems where the be-

haviour of the members and their interactions cannot be predicted in advance. Three key components of computational systems are specified, namely the social constraints, social roles and social states. The specifications of these concepts is based on and motivated by the formal study of legal and social systems (a goal of the ALFEBIITE [ALF99] project), and therefore operators of Deontic Logic are used for expressing legal social behaviour of agents [Wri51, van03]. ALFEBIITE has investigated the application of formal models of norm-governed activity to the definition, management and regulation of interactions between info-habitants in the information society. Their logical framework comprises a set of building blocks (including doxastic, deontic and praxeologic notions) as well as composite notions (including deontic right, power, trust, role and signalling acts). Intuitively, a correspondence can be established our expectation abducibles (positive expectation **E** and negative expectation **EN**) and the operators of Deontic Logic (obligation $\mathcal{O}$ and prohibition $\mathcal{F}$). As suggested in [AGL$^+$05], this correspondence lets logical relations between Deontic Logic operators be expressed as abductive integrity constraints. However, the semantics in the two cases are different: abductive for expectations, modal for deontic operators. Another notable difference with [APS02] is that we do not explicitly represent the institutional power of the members and the concept of valid action. Permitted are all social events that do not determine a violation, i.e., in deontic terms, all events that are not explicitly forbidden are allowed. Differently, permission, when it needs to be explicitly expressed, can be mapped to a negated negative expectation ($\neg$**EN**).

Chapter 3

The $\mathcal{S}$CIFF proof procedure

The operational semantics of the $\mathcal{S}$CIFF proof procedure is given by an abductive proof procedure.

Since the language and declarative semantics of the $\mathcal{S}$CIFF framework are closely related with the IFF abductive framework by Fung and Kowalski [FK97], the $\mathcal{S}$CIFF proof procedure has been inspired by the IFF proof procedure. However, some modifications were necessary, due to the following differences between the frameworks:

- $\mathcal{S}$CIFF requires support for the dynamical happening of events, i.e., the insertion of new facts in the knowledge base during the computation;
- $\mathcal{S}$CIFF requires universally quantified variables in abducibles;
- $\mathcal{S}$CIFF needs support for quantifier restrictions;
- $\mathcal{S}$CIFF needs support for the concepts of fulfillment and violation (see Def. 2.3.4).

3.1 Data Structures

The $\mathcal{S}$CIFF proof procedure is based on a rewriting system transforming one node to another (or to others). In this way, starting from an initial node, the proof tree is defined.

A node can be either the special node *false*, or defined by the following tuple

$$T \equiv \langle R, CS, PSIC, \mathbf{PEND}, \mathbf{HAP}, \mathbf{FULF}, \mathbf{VIOL} \rangle$$

where

- R is the resolvent: a conjunction, whose conjuncts can be atoms or disjunctions of conjunctions of atoms
- CS is the constraint store
- $PSIC$ is the set of partially solved integrity constraints
- **PEND** is the set of (pending) expectations
- **HAP** is the history of happened events, represented by a set of events, plus a *open/closed* attribute
- **FULF** is a set of fulfilled expectations
- **VIOL** is a set of violated expectations

If one of the elements of the tuple is *false*, then the whole tuple is the special node *false*, which cannot have successors.

3.1.1 Initial Node and Success

A derivation D is a sequence of nodes

$$T_0 \rightarrow T_1 \rightarrow \cdots \rightarrow T_{n-1} \rightarrow T_n.$$

Given a goal $\mathcal{G}$ and a set of social integrity constraints $\mathcal{IC}_S$, we build the first node in the following way:

$$T_0 \equiv \langle \{\mathcal{G}\}, \emptyset, \mathcal{IC}_S, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

i.e., the resolvent R is initially the query ($R_0 = \{G\}$) and the partially solved integrity constraints *PSIC* is the set of integrity constraints ($PSIC_0 = \mathcal{IC}_S$).

The other nodes $T_j, j > 0$, are obtained by applying the transitions that we will define in the next section, until no further transition can be applied (we call this last condition *quiescence*).

Definition 3.1.1 *Given an instance* $\mathcal{S}_{\mathbf{HAP}^i}$ *of a social specification* $\mathcal{S} = \langle KB_S, \mathcal{IC}_S \rangle$ *and a set* $\mathbf{HAP}^f \supseteq \mathbf{HAP}^i$ *there exists a* successful derivation *for a goal* G *iff the proof tree with root node* $\langle \{G\}, \emptyset, \mathcal{IC}_S, \emptyset, \mathbf{HAP}^i, \emptyset, \emptyset \rangle$ *has at least one leaf node*

$$\langle \emptyset, CS, PSIC, \mathbf{PEND}, \mathbf{HAP}^f, \mathbf{FULF}, \emptyset \rangle$$

where CS *is consistent, and* $\mathbf{PEND}$ *contains only negative literals* $\neg\mathbf{E}$ *and* $\neg\mathbf{EN}$*. In such a case, we write:*

$$\mathcal{S}_{\mathbf{HAP}^i} \vdash^{\mathbf{HAP}^f}_{\mathbf{PEND} \cup \mathbf{FULF}} G.$$

From a non-failure leaf node N, answers can be extracted in a very similar way to the IFF proof procedure. Answers of the $\mathcal{S}$CIFF proof procedure are called *expectation answers.* To compute an expectation answer, first, a substitution σ' is computed such that

- σ' replaces all variables in N that are not universally quantified by a ground term
- σ' satisfies all the constraints in the store CS_N.

If the constraint solver is (theory) complete [JMMS98] (i.e., for each set of constraints c, the solver always returns *true* or *false*, and never *unknown*), then there will always exist a substitution σ' for each non-failure leaf node N. Otherwise, if the solver is incomplete, σ' may not exist. The non-existence of σ' is discovered during the answer extraction phase. In such a case, the node N will be marked as a failure node, and another success node can be selected (if there is one).

Definition 3.1.2 *Let* $\sigma = \sigma'|_{vars(G)}$ *be the restriction of* σ' *to the variables occurring in the initial goal* G*. Let* $\Delta = (\mathbf{FULF}_N \cup \mathbf{PEND}_N)\sigma'$*. The pair* (Δ, σ) *is the* expectation answer *obtained from the node* N*.*

3.2 Variables

Quantification. Concerning variable quantification, $\mathcal{S}$CIFF differs from IFF in the following aspects:

- in IFF, all the variables that occur in the resolvent or in abduced literals are existentially quantified, while the others (that occur only in implications) are universally quantified; in $\mathcal{S}$CIFF, variables that occur in the resolvent or in abducibles can be universally quantified;

- in IFF, variables in an implication are existentially quantified if they also occur in an abducible or in the resolvent, while in $\mathcal{S}$CIFF variables in implications can be universally quantified even if they do not occur elsewhere.

For these reasons, in the $\mathcal{S}$CIFF proof procedure the quantification of variables is explicit.

Scope. The scope of the variables differs depending on where they occur:

- if they occur in the resolvent or in abducibles, their scope is the whole tuple representing the node (see Sect. 3.1);

- if they occur in an implication, their scope is the implication in which they occur.

In the first case, we say that the variable is *flagged*. The *flagging* status of a variable (i.e., its being flagged or not) influences how the term in which the variable appears is copied (see Def. 3.2.1).

In the following, when we want to make explicit the fact that a variable X is flagged (when it is not clear from the context), it will be indicated with $\hat{X}$, while if we want to highlight that it is not flagged, it will be indicated with $\check{X}$.

Quantifier restrictions. Variables can be associated with *quantifier restrictions* [Bür94]. Quantifier restrictions restrict the applicability of a quantifier.

The semantics of quantifier restrictions is different for the case of universally quantified and existentially quantified variables, as follows:

$$\begin{array}{lcl} \forall_{X:c(X)} p(X) & \iff & \forall X c(X) \rightarrow p(X) \\ \exists_{X:c(X)} p(X) & \iff & \exists X c(X) \wedge p(X) \end{array} \tag{3.2.1}$$

For existentially quantified variables, quantifier restrictions have the same meaning of constraints.

Given a variable X, with $QR(X)$ we will denote the quantifier restrictions on X. If X is universally quantified, we restrict ourselves to quantifier restrictions that are *unary*, meaning that they involve only X.

In the tuple, the quantifier restrictions on variables are recorded in the constraint store CS, and will be handled by the constraint solver.

Copy of a formula. When making a copy of a formula, we keep into account the scope of the variables in it by means of their flagging status, as follows.

Definition 3.2.1 *Given a formula F, we call* copy *of F a formula*

$$F' = copy(F)$$

where the universally quantified variables and the non flagged variables are renamed.

For example,

$$\exists_{\hat{Y}} \forall_{\hat{X'} > 50} \forall_{\check{Z'}} \mathbf{E}(p(\hat{Y})) \land \mathbf{EN}(q(\hat{X'}, \hat{Y})) \land [\mathbf{EN}(r(\hat{Y}, \check{Z'})) \rightarrow \exists_{\check{K'}} \mathbf{E}(p(\check{K'}))]$$

is a copy of the formula:

$$\exists_{\hat{Y}} \forall_{\hat{X} > 50} \forall_{\check{Z}} \mathbf{E}(p(\hat{Y})) \land \mathbf{EN}(q(\hat{X}, \hat{Y})) \land [\mathbf{EN}(r(\hat{Y}, \check{Z})) \rightarrow \exists_{\check{K}} \mathbf{E}(p(\check{K}))]$$

Notice that, by Definition 3.2.1, if F contains only flagged existentially quantified variables, then $copy(F) \equiv F$.

3.3 Transitions

The transitions are based on the transitions of the IFF proof procedure, enlarged with those of CLP [JM94], and with specific transitions accommodating the concepts of fulfillment, dynamically growing history and consistency of the set of expectations with respect to the given definitions (Defs. 2.3.2, 2.3.3 and 2.3.5).

3.3.1 IFF-like transitions

Unfolding Since the variables in the head of a clause in the KB_S are all universally quantified with scope the entire clause, the unfolding step is basically the same as in many abductive proof procedures. It is defined as follows.

Let L_i be the selected literal in the resolvent $R_k = L_1, \ldots, L_r$. Let it (L_i) be a predicate defined in the KB_S of the social specification. Unfolding generates a child node for each of the definitions of L_i; in each node, L_i is replaced with its definition.

Moreover, as in the IFF proof procedure, unfolding is also applied to a defined atom in the body of an implication. In this case, only one child node is generated, which contains a new implication for each definition of the atom.

Abduction Since the $\mathcal{S}$CIFF proof procedure (differently from the IFF) keeps the set of abducibles separate from the resolvent, a transition has been introduced for abduction which, intuitively, moves an abducible from the resolvent to the set of abduced atoms.

More precisely:

- if $R_k = L_1, \ldots, L_r$, and the selected literal L_i is of type **E**, **EN**, $\neg$**E**, or $\neg$**EN**,
- then $R_{k+1} = L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_r$ and $\mathbf{PEND}_{k+1} \equiv \mathbf{PEND}_k \cup \{L_i\}$.

Propagation Let $L_1, \ldots, L_n \rightarrow H_1 \vee \cdots \vee H_j$ be an integrity constraint, belonging to the set PSIC, and let A be

- either an event belonging to $\mathbf{HAP}_k$ (in which case A is an **H** event),
- or an expectation belonging to $\mathbf{PEND}_k$, $\mathbf{FULF}_k$ or $\mathbf{VIOL}_k$,

Then, by *Propagation*, we perform the following steps:

- we make a copy of A, $copy(A)$; the new atom is inserted in the same element of the tuple where the original atom occurs.
- $PSIC_{k+1} = PSIC_k \cup \{A = L'_i, L'_1, \ldots, L'_{i-1}, L'_{i+1} \ldots, L'_n \rightarrow H'_1 \vee \cdots \vee H'_j\}$, where $L'_1, \ldots, L'_n \rightarrow H'_1 \vee \cdots \vee H'_j = copy(L_1, \ldots, L_n \rightarrow H_1 \vee \cdots \vee H_j)$

This transition does not have any effect on the constraint store, since *case analysis* will take care of the equality in the body of the implication.

Splitting Given a node with

- $R_k = L_1, \ldots, L_{i-1}, (L_i \vee L_{i+1}), L_{i+2}, \ldots, L_r$

splitting produces two nodes, N^1 and N^2 such that in node N^1

- $R^1_{k+1} = L_1, \ldots, L_i, L_{i+2} \ldots, L_r$

and in node N^2

- $R^2_{k+1} = L_1, \ldots, L_{i-1}, , L_{i+1}, \ldots, L_r$

In the $\mathcal{S}$CIFF proof procedure, disjunctions may appear also in the constraint store. In the following, we will assume that disjunctions are dealt with by the constraint solver itself (e.g., by means of constructive disjunction [vSD93] or cardinality operator [vD91]).

Case Analysis Given a node with an implication

$$PSIC_k = PSIC' \cup \{A = B, L_1, \ldots, L_n \rightarrow H_1 \vee \cdots \vee H_j\}$$

the node is replaced by two identical nodes, except for the following.
In Node 1 we hypothesise that the equality $A = B$ holds:

- $PSIC^1_{k+1} = PSIC' \cup \{L_1, \ldots, L_n \rightarrow H_1 \vee \cdots \vee H_j\}$
- $CS^1_{k+1} = CS_k \cup \{A = B\}$

In Node 2, we hypothesise the opposite:

- $PSIC^2_{k+1} = PSIC'$
- $CS^2_{k+1} = CS_k \cup \{A \neq B\}$

where $\neq$ stands for the constraint of non-unification.

Factoring *Factoring* can be applied in a node N_k, in which:

- $\mathbf{PEND}_k \cup \mathbf{FULF}_k \cup \mathbf{VIOL}_k \supseteq \{A_1, A_2\}$

where A_1 and A_2 are (abducible) atoms in which all the variables are existentially quantified (and, of course, flagged). Factoring generates two children nodes, N^1 and N^2. In N^1:

- $CS^1_{k+1} = CS_k \cup \{A_1 = A_2\}$

and in N^2:

- $CS^1_{k+1} = CS_k \cup \{A_1 \neq A_2\}$

Equivalence Rewriting The equivalence rewriting operations are delegated to the constraint solver. Note that a constraint solver works on a constraint domain which has an associated interpretation. In addition, the constraint solver should handle the constraints among terms derived from the unification. Therefore, beside the specific constraint propagation on the constraint domain, we hypothesise that the constraint solver is equipped with further inference rules for coping with the unification.

Logical Equivalence The rule

$$\text{“}true \rightarrow A \text{ is equivalent to } A\text{”}$$

of the IFF proof procedure is translated as follows. If $PSIC_k = PSIC' \cup \{true \rightarrow A\}$, we generate a new node such that:

- $PSIC_{k+1} = PSIC'$
- $R_{k+1} = R_k, A'$

where A' is obtained from A by flagging all the variables that were not already flagged.

3.3.2 Dynamically growing history

A set of transitions deals with a dynamically growing history **HAP**. The transitions are used to reason upon the happening (or non-happening) of events.

Closure The transition *closure* informs the proof procedure that no more events will happen, i.e., the set **HAP** will not grow any more. It switches the *open/closed* attribute of **HAP** to *closed*.

Transition *Closure* is only applicable when no other transition is applicable. In other words, it is only applicable at the quiescence of the set of the other transitions.

Given a state where:

- closed($\mathbf{HAP}_k$) = *false*

in which no other transition is applicable, transition *Closure* produces two nodes . Node N^1 is the following:

- closed($\mathbf{HAP}_k$) = true

and node N^2 is identical to its father. In order to avoid infinite loops, transition *Closure* cannot be again applied to the node N^2 before a Happening transition has been applied.

Happening of Events The happening of events is considered by a transition *Happening*. This transition takes an event $\mathbf{H}(Event)$ from an external queue and puts it in the history **HAP**; the transition *Happening* is applicable only if an *Event* such that $\mathbf{H}(Event) \notin \mathbf{HAP}$ is in the external queue.

Given a state in which

- closed($\mathbf{HAP}_k$) = $false$

the transition *Happening* produces a single successor

$$\mathbf{HAP}_{k+1} = \mathbf{HAP}_k \cup \{\mathbf{H}(Event)\}.$$

Otherwise, given a state in which

- closed($\mathbf{HAP}_k$) = $true$

the transition *Happening* produces a single successor

$$false$$

(which means that happening is not possible with a closed history).

Non-happening The *Non-happening* transition that can be considered an application of *constructive negation*. Constructive negation is a powerful inference that is particularly well suited in CLP [Stu95].

Rule *non-happening* applies when the history is closed and a literal **not H** is in the body of a PSIC.

Given a node where:

- $PSIC_k = \{\mathbf{not\ H}(E_1), L_2, \ldots, L_n \rightarrow H_1 \vee \cdots \vee H_m\} \cup PSIC'$
- closed$(\mathbf{HAP}_k) = true$

non-happening produces a new node.

Let E_1' be a renaming of E_1 (i.e., all the variables in E_1 are substituted with fresh new variables). Let all the new variables in E_1' be universally quantified and flagged. For each variable $X_j \in vars(E_1)$, let $ren(X_j)$ be the corresponding, renamed variable in $vars(E_1')$. For all atoms $\mathbf{H}(E) \in \overline{\mathbf{HAP}_k}$ that unify with $\mathbf{H}(E_1')$, we impose the quantifier restrictions on the variables in E_1' given by the following disjunction:

$$\bigwedge_{\substack{\mathbf{H}(E) \in \mathbf{HAP}_k \\ s.t. unifies(E, E_1')}} \left(\bigvee_{X_j \in vars(E_1)} ren(X_j) \neq t_j \right)$$

where t_j is the term in E corresponding to X_j in E_1.

The child node, $k+1$, is then defined by:

- $PSIC_{k+1} = \{E_1 = E_1', L_2, \ldots, L_n \rightarrow H_1 \vee \cdots \vee H_m\} \cup PSIC'$

3.3.3 Fulfillment and Violation

Violation EN Given a node N with the following situation:

- $\mathbf{PEND}_k = \mathbf{PEND}' \cup \{\mathbf{EN}(E_1)\}$
- $\mathbf{HAP}_k = \mathbf{HAP}' \cup \{\mathbf{H}(E_2)\}$

violation **EN** produces two nodes N^1 and N^2, where N^1 is as follows:

- $\mathbf{VIOL}^1_{k+1} = \mathbf{VIOL}_k \cup \{\mathbf{EN}(E_1)\}$
- $CS^1_{k+1} = CS_k \cup \{E_1 = E_2\}$

and N^2 is as follows:

- $\mathbf{VIOL}^2_{k+1} = \mathbf{VIOL}_k$
- $CS^2_{k+1} = CS_k \cup \{E_1 \neq E_2\}$

Fulfillment E Starting from a node N as follows:

- $\mathbf{PEND}_k = \mathbf{PEND}' \cup \{\mathbf{E}(Event_1)\}$
- $\mathbf{HAP}_k = \mathbf{HAP}' \cup \{\mathbf{H}(Event_2)\}$

Fulfillment **E** builds two nodes, N^1 and N^2, that are identical to their father except for the following.

In node N^1 we hypothesise that the expectation and the happened event unify:

- $\mathbf{PEND}^1_{k+1} = \mathbf{PEND}'$
- $\mathbf{FULF}^1_{k+1} = \mathbf{FULF}_k \cup \{\mathbf{E}(Event_1)\}$
- $CS^1_{k+1} = CS_k \cup \{Event_1 = Event_2\}$

In node N^2 we hypothesise that the two will not unify:

- $\mathbf{PEND}^2_{k+1} = \mathbf{PEND}_k$
- $\mathbf{FULF}^2_{k+1} = \mathbf{FULF}_k$
- $CS^2_{k+1} = CS_k \cup \{Event_1 \neq Event_2\}$

Violation E Given a state where

- $closed(\mathbf{HAP}_k) = true$
- $\mathbf{PEND}_k = \mathbf{PEND}' \cup \{\mathbf{E}(Event_1)\}$

transition *Violation* **E** creates a successor node in which

- $\mathbf{VIOL}_{k+1} = \mathbf{VIOL}_k \cup \{\mathbf{E}(Event_1)\}$.
- $\mathbf{PEND}_{k+1} = \mathbf{PEND}'$

Fulfillment EN Given a state

- $closed(\mathbf{HAP}_k) = true$
- $\mathbf{PEND}_k = \mathbf{PEND}' \cup \{\mathbf{EN}(Event_1)\}$ does not unify with $Event_1$

transition *Fulfillment* **EN** creates a successor node in which

- $\mathbf{FULF}_{k+1} = \mathbf{FULF}_k \cup \{\mathbf{EN}(Event_1)\}$
- $\mathbf{PEND}_{k+1} = \mathbf{PEND}'$.

3.3.4 Consistency

E-Consistency In order to ensure **E**-consistency (see Def. 2.3.3) of the set of expectations, we impose the following integrity constraint:

$$\mathbf{E}(T) \wedge \mathbf{EN}(T) \rightarrow \mathit{false} \tag{3.3.1}$$

¬-Consistency In order to ensure ¬-consistency (see Def. 2.3.2) of the set of expectations, we impose the following integrity constraints:

$$\begin{array}{rcccl} \mathbf{E}(T) & \wedge & \neg\mathbf{E}(T) & \rightarrow & \mathit{false} \\ \mathbf{EN}(T) & \wedge & \neg\mathbf{EN}(T) & \rightarrow & \mathit{false} \end{array} \tag{3.3.2}$$

3.3.5 CLP

Here we suppose to have the same transitions as in CLP [JM94].

The constraint solver deals also with quantifier restrictions. If a quantifier restriction (due to unification) gets all the variables existentially quantified, then we replace it with the corresponding constraint. E.g., if in the tuple we have two variables $\hat{X}$ and $\hat{Y}$ quantified as follows:

$$\exists \hat{Y}, \forall_{\hat{X} \neq 1},$$

and variable $\hat{X}$ is unified with $\hat{Y}$, we obtain that $\exists \hat{Y}$, $\hat{Y} \neq 1$ (the quantifier restriction $\hat{X} \neq 1$ becomes a constraint on the variable $\hat{Y}$).

Constrain Given a node with

- $R_k = L_1, \ldots, L_r$

and the selected literal, L_i is a constraint , *constrain* produces a node with

- $R_{k+1} = L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_r$
- $CS_{k+1} = CS_k \cup \{L_i\}$

Infer Given a node, the transition *Infer* modifies the constraint store by means of a function *infer(CS)*. This function is typical of the adopted constraint sort. E.g., the function *infer* in a FD (Finite Domain) sort will typically compute (generalised) arc-consistency.

- $CS_{k+1} = \mathit{infer}(CS_k)$

Consistent Given a node, the transition *Consistent* will check the consistency of the constraint store (by means of a solver of the domain) and will generate a new node. The new node can either be a special node *fail* or a node identical to its father.

If $consistent(CS_k)$ then

- $T_{k+1} = T_k$

If $\neg consistent(CS_k)$ then

- $T_{k+1} = fail$

3.4 $\mathcal{S}$CIFF properties

In general, the *soundness* of a proof procedure is defined as follows: if there exists a successful derivation from a set of formulae $\mathcal{F}_1$ to a set of formulae $\mathcal{F}_2$, then $\mathcal{F}_2$ is a logical consequence of $\mathcal{F}_1$. In symbols:

$$\mathcal{F}_1 \;\vdash\; \mathcal{F}_2 \;\rightarrow\; \mathcal{F}_1 \;\models\; \mathcal{F}_2$$

Completeness is the reverse property: if a set of formulae $\mathcal{F}_2$ is a logical consequence of a set of formulae $\mathcal{F}_1$, then there exists a successful derivation of the proof from $\mathcal{F}_1$ to $\mathcal{F}_2$. In symbols:

$$\mathcal{F}_1 \models \mathcal{F}_2 \rightarrow \mathcal{F}_1 \vdash \mathcal{F}_2$$

For $\mathcal{S}$CIFF, at the time of writing, soundness has been proved, but completeness has not.

Moreover, a termination result for $\mathcal{S}$CIFF has been proved.

In the following, we only give the statements of the results. Proofs can be found in [GLMT04] and [GLM05].

3.4.1 Soundness of $\mathcal{S}$CIFF

Theorem 3.4.1 (Soundness of $\mathcal{S}$CIFF) *Given a society instance* $\mathcal{S}_{\overline{\mathbf{HAP}^f}}$*, if*

$$\mathcal{S}_{\mathbf{HAP}^i} \vdash^{\mathbf{HAP}^f}_{\mathbf{PEND} \cup \mathbf{FULF}} G$$

with expectation answer $(\mathbf{PEND} \cup \mathbf{FULF}, \sigma)$ *then*

$$\mathcal{S}_{\mathbf{HAP}^f} \models_{(\mathbf{PEND} \cup \mathbf{FULF})\sigma} G\sigma$$

3.4.2 Termination of $\mathcal{S}$CIFF

Termination is proven, as for SLD resolution, for *acyclic* knowledge bases and *bounded* goals and implications. For definitions of boundedness and acyclicity for the society Knowledge Bases, the reader can refer to [Xan03].

Theorem 3.4.2 (Termination of $\mathcal{S}$CIFF) *Let $\mathcal{G}$ be a query to a society $\mathcal{S} = \langle KB_S, \mathcal{IC}_S \rangle$, where KB_S, $\mathcal{IC}_S$ and $\mathcal{G}$ are acyclic w.r.t. some level mapping, and $\mathcal{G}$ and all implications in $\mathcal{IC}_S$ are bounded w.r.t. the level-mapping. Then, every $\mathcal{S}$CIFF derivation for $\mathcal{G}$ for each instance of $\mathcal{G}$ is finite.*

Chapter 4

Constraint Handling Rules

Constraint Handling Rules [Frü98] (*CHR* for brevity hereafter) are essentially a committed-choice language consisting of guarded rules that rewrite constraints in a store into simpler ones until they are solved. *CHR* define both *simplification* (replacing constraints by simpler constraints while preserving logical equivalence) and *propagation* (adding new, logically redundant but computationally useful, constraints) over user-defined constraints.

The main intended use for *CHR* is to write constraint solvers, or to extend existing ones. However, although ours is not a classic constraint programming setting, the computational model of *CHR* presents features that make it a useful tool for the implementation of the $\mathcal{S}$CIFF proof-procedure, as will be explained in Ch. 5.

In the following, we briefly introduce the *CHR* language. The interested reader can refer to [Frü98] for a complete introduction.

4.1 Syntax and semantics

CHR rules are of three types: *simplification*, *propagation*, and *simpagation*.

Simplification CHRs. Simplification rules are of the form

$$H_1, \ldots, H_i \Longleftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \tag{4.1.1}$$

with $i > 0$, $j \geq 0$, $k \geq 0$ and where the multi-head $H_1, \ldots, H_i$ is a nonempty sequence of *CHR* constraints, the guard $G_1, \ldots, G_j$ is a sequence of built-in con-

straints, and the body $B_1, \ldots, B_k$ is a sequence of built-in and *CHR* constraints.

Declaratively, a simplification rule is a logical equivalence, provided that the guard is true. Operationally, when constraints $H_1, \ldots, H_i$ in the head are in the store and the guard $G_1, \ldots, G_j$ is true, they are replaced by constraints $B_1, \ldots, B_k$ in the body.

Propagation CHRs. Propagation rules have the form

$$H_1, \ldots, H_i \Longrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \qquad (4.1.2)$$

where the symbols have the same meaning and constraints of those in the simplification rules (4.1.1).

Declaratively, a propagation rule is an implication, provided that the guard is true. Operationally, when the constraints in the head are in the store, and the guard is true, the constraints in the body are added to the store.

Simpagation CHRs. Simpagation rules have the form

$$H_1, \ldots, H_l \backslash H_{l+1}, \ldots, H_i \Longleftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \qquad (4.1.3)$$

where $l > 0$ and the other symbols have the same meaning and constraints of those of simplification *CHR*s (4.1.1).

Declaratively, the rule of Eq. (4.1.3) is equivalent to

$$H_1, \ldots, H_l, H_{l+1}, \ldots, H_i \Longleftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k, H_1, \ldots, H_l \qquad (4.1.4)$$

Operationally, when the constraints in the head are in the store and the guard is true, $H_1, \ldots, H_l$ remain in the store, and $H_{l+1}, \ldots, H_i$ are replaced by $B_1, \ldots, B_k$.

4.2 The SICStus Prolog *CHR* library

The reference implementation of *CHR* is provided with SICStus Prolog [SIC03]. The SICStus *CHR* library offers implementation-specific operational features which have been used in the $\mathcal{S}$CIFF implementation. In particular, we have made extensive use of the following (for more details, see the SICStus Manual [SIC03]):

- `passive` pragma: this directive lets the programmer declare a constraint as passive in the head of a rule. In this way, no code will be generated for the constraint. In practice, the rule will not be activated because of that constraint; in some cases, in this way efficiency is improved, but completeness may be lost;

- `remove_constraint` built-in predicate: this predicate non-declaratively removes a constraint from the *CHR* store, given the internal constraint identifier.

Chapter 5

Implementation of the $\mathcal{S}$CIFF proof procedure

In this chapter, we describe the implementation of the $\mathcal{S}$CIFF proof procedure described in Ch. 3.

5.1 Overview of the implementation

5.1.1 Technology

In choosing the technology (programming languages, systems and techniques) to be used, we had to take into account the requirements posed by the implementation of $\mathcal{S}$CIFF, some common to other proof procedures, some peculiar to $\mathcal{S}$CIFF.

The choice of the Prolog programming language was motivated by the following main reasons:

- While no quantitative results about the computational complexity of the $\mathcal{S}$CIFF proof procedure have been proved yet, a consideration of its behaviour with some sample specification suggests an high time complexity of the exploration of the proof tree. In this perspective, the constant-factor advantage that could be achieved by an imperative language such as C when compared to Prolog is less significant.

- Prolog supports dynamic data structures representing symbolic information (which is required by $\mathcal{S}$CIFF) in a simple and natural way.
- Thanks to its operational semantics (which could informally be described as a depth-first exploration of a proof tree), Prolog is a natural candidate for the implementation of a proof procedure.
- Many Prolog systems are extended to efficient Constraint Logic Programming [JM94] systems, and offer constraint solving facilities which are needed by $\mathcal{S}$CIFF.

The most usual technique for implementing (abductive) proof procedures has probably been meta-interpretation, which lets the programmer adjust the built-in search strategy of Prolog to application-specific requirements in a compact (if computationally expensive) way. However, the common understanding of abducible and constraints suggested by Kowalski *et al.* [KTW98] paved the way for some authors to implement abduction in the *Constraint Handling Rules* (see Ch. 4 and [Frü98]) language [AC00, CD04, GLM+03], with advantages in execution time with respect to meta-interpretation. Besides, a *CHR*-based implementation offers, as a byproduct, the possibility of the seamless integration of a high-level implementation of constraint solvers (see, for instance, [AL02] or [AGL+04a]), which $\mathcal{S}$CIFF needs to manage quantifier restrictions. Moreover, we had already fruitfully used *CHR* for the implementation of previous of $\mathcal{S}$CIFF-like verification procedures [AGL+03b, ADG+04]. For these reasons, *CHR* was chosen as the language for implementing the abduction-related parts of $\mathcal{S}$CIFF.

One further $\mathcal{S}$CIFF requirement is the possibility to represent the variable quantification (see Sect. 3.2) and to adjust the behaviour of unification to keep it into account. Attributed variables [Hol90] provide a good language abstraction for this purpose.

Concerning the choice of the Prolog system, SICStus Prolog [SIC03] appeared as the natural choice for the following reasons:

- it is a stable, fast, well documented and well supported Prolog system;
- it provides state-of-the-art CLP solvers: notably, on finite domains variables (CLPFD) and boolean variables (CLPB);

- it provides the reference implementation of *CHR*;

- it supports attributed variables.

5.1.2 Adapting the $\mathcal{S}$CIFF execution to Prolog

As the IFF proof-procedure [FK97], the $\mathcal{S}$CIFF proof procedure specifies the proof tree, leaving the search strategy to be defined at the implementation level. The two most obvious possibilities are the depth-first and the breadth-first strategies.

The implementation described here is based on a depth-first strategy. This choice enabled us to tailor the implementation upon the operational semantics of Prolog: in particular, the resolvent of the proof is represented by the Prolog resolvent (see Sect. 5.3), and thus the Prolog stack is used directly for chronological backtracking.

Depth-first exploration has its drawbacks: probably, the most notable is the possibility of infinite loops in case of cyclic programs (the termination result for $\mathcal{S}$CIFF in Sect. 3.4.2 is relative to acyclic programs), although, considering the applications of $\mathcal{S}$CIFF it is also unfortunate to lose a complete representation of the proof tree frontier that a breadth-first strategy would allow for. However, we believe that the advantages in terms of execution time, memory management and implementation simplicity granted by the depth-first strategy more than compensate for the disadvantages.

Success and failure of the implementation map directly the corresponding notions of $\mathcal{S}$CIFF. In particular, the implementation returns success when a state of goal achievement is found; instead, all the failure conditions, such as inconsistency (both with respect to **E**-consistency and $\neg$-consistency, see Defs. 2.3.3 and 2.3.2), inconsistent constraint store and violation generate a failure, and cause (chronological) backtracking.

5.2 Representation of the Social Specification instance.

The inputs to the $\mathcal{S}$CIFF implementation are those defining an instance of the social specification (see Ch. 2), i.e.:

- The Social Knowledge Base
- the set of SICs;
- the history **HAP**.

The Social Knowledge Base is contained in a text file with the syntax of Spec. 2.2. Internally, it is represented as part of the Prolog database, after having been modified to support $\mathcal{S}$CIFF features such as, for instance, variable quantification.

The set of SICs is also represented in the Prolog database. When the computation starts, SICs are imposed as partially solved integrity constraints (see Sect. 5.3.3).

The representation of the history is application specific. When using $\mathcal{S}$CIFF for offline verification, the history is part of the Prolog database; but events can also be received from an external queue, when $\mathcal{S}$CIFF is integrated in a multiagent system (as described in Sect. 5.6).

5.3 Data Structures

Each state of the proof (as specified in Sect. 3.1) is represented by a tuple with the following structure:

$$T \equiv \langle R, CS, PSIC, \mathbf{PEND}, \mathbf{HAP}, \mathbf{FULF}, \mathbf{VIOL} \rangle$$

The data structures are implemented by means of Prolog built-in structures and the *CHR* constraint store. In the following, we describe the implementation of each element of the tuple.

Set	***CHR* Constraint**	**Atom**	**Meaning**
PSIC	`psic/2`	`psic(B,H)`	$B \rightarrow H \in PSIC$
HAP	`h/2`	`h(D,T)`	$\mathbf{H}(D,T) \in \mathbf{HAP}$
PEND	`pending/1`	`pending(e(D,T))`	$\mathbf{E}(D,T) \in \mathbf{PEND}$
FULF	`fulf/1`	`fulf(e(D,T))`	$\mathbf{E}(D,T) \in \mathbf{FULF}$
VIOL	`viol/1`	`viol(e(D,T))`	$\mathbf{E}(D,T) \in \mathbf{VIOL}$

Table 5.1: $\mathcal{S}$CIFF sets and *CHR* constraints

5.3.1 Resolvent

The resolvent of the proof is implemented by the Prolog resolvent. This allows us to exploit the Prolog stack for depth-first exploration of the tree of states.

At the beginning of the compuation, the resolvent is initialised with the goal; in a success node, the resolvent is true.

5.3.2 Constraint Store

The constraint store of the proof[1] is represented as the union of the CLP constraint stores. For the implementation of the proof, the CLPFD and CLPB libraries of SICStus Prolog, a *CHR*-based solver on finite and infinite domains, and an *ad-hoc* solver for reified unification have been used. However, in principle, it should be possible to integrate with the proof any constraint solver available for SICStus Prolog.

5.3.3 Proof sets: PSIC, HAP, PEND, FULF, VIOL

The sets PSIC, **HAP**, **PEND**, **FULF**, and **VIOL** are represented by means of *CHR* constraints, by exploiting the set semantics that can be given to constraints, as follows. To a set I a *CHR* constraint `i` is associated; the intended semantics is that the constraint is imposed on an atom A (`i(A)` is in the *CHR* store) if and only if $A \in I$.

[1] This constraint store, which contains CLP constraints over variables, should not be confused with the *CHR* constraint store, which is used for the implementation of the other data structures.

The associations between sets and *CHR* constraints is shown in Tab. 5.1.

This representation makes the *CHR*-based implementation of many $\mathcal{S}$CIFF transitions straightforward. In fact, many of the $\mathcal{S}$CIFF transitions can be expressed with the following pattern:

- if $A_1 \in I_1, \ldots, A_k \in I_k, A_{k+1} \in I_{k+1}, \ldots, A_n \in I_n$
- then impose $A_{n+1} \in I_{n+1}, \ldots, A_m \in I_m$
- and remove A_1 from I_1, ..., A_k from I_k

which maps directly to the following simpagation rule (where *CHR* constraints have the aforementioned intended set semantics):

$$\begin{array}{l} i_1(A_1), \ldots, i_k(A_k) \\ \backslash \\ i_{k+1}(A_{k+1}), \ldots, i_n(A_n) \\ \Longleftrightarrow \\ i_{n+1}(A_{n+1}), \ldots, i_m(A_m) \end{array} \tag{5.3.1}$$

Moreover, inserting an element in a set is as simple as imposing the corresponding *CHR* constraint, which in SICStus Prolog amounts to calling it.

5.3.3.1 Partially Solved Integrity Constraints

Partially solved integrity constraints are formulae derived from social integrity constraints by means of transitions such as *propagation* (see Sect. 5.5.1).

Each partially solved integrity constraint is represented by means of a `psic/2` *CHR* constraint, which has as two arguments:

- the first argument is a list of lists representing the body of the partially solved integrity constraint. Each sub-list contains terms of type:
  1. **H** (events);
  2. **not H** (negated events);
  3. **E** (positive expectations);

4. ¬**E** (negated positive expectations);
5. **EN** (negative expectations);
6. ¬**EN** (negated negative expectations);
7. constraints and defined predicates.

The reason for each sub-list to contain predicates of the same type is to make the *propagation* (see Sect. 5.5.1) transition more efficient;

- the second argument is a list of lists representing the head of the partially solved integrity constraint. Each sub-list represents one disjunct of the head, and each element of each sub-list (a Prolog term which can represent an expectation or a constraint) is a conjunct.

For instance, the following partially solved integrity constraint :

$$\begin{aligned} & \mathbf{H}(tell(A, B, query_ref(Info), D), T) \wedge \\ & qr_deadline(TD) \\ \rightarrow\; & \mathbf{E}(tell(B, A, inform(Info, Answer), D), T1) \wedge \\ & T1 \; < \; T \; + \; TD \\ \vee\; & \mathbf{E}(tell(B, A, refuse(Info), D), T1) \wedge \\ & T1 \; < \; T \; + \; TD \end{aligned} \tag{5.3.2}$$

would be represented by the following *CHR* constraint (except, of course, for a renaming of the variables, and where the CLP constraints are represented in the notation of the SICStus Prolog CLPFD solver):

```
psic([[h(tell(A,B,query_ref(Info),D),T1)],[],[],[],[],[],
       [qr_deadline(TD)]],
      [[e(tell(B,A,inform(Info,Answer),D),T2),T2#<T1+TD],
       [e(tell(B,A,refuse(Info),D),T3),T3#<T1+TD]])
```

5.3.3.2 History

Each event is represented by means of a `h/2` *CHR* constraint, whose (ground) arguments are the content and the time of the event. For instance, the event

$$\mathbf{H}(tell(alice, bob, query_ref(phone_number), dialog_id), 10) \tag{5.3.3}$$

would be represented as:

```
h(tell(alice,bob,query_ref(phone_number),dialog_id),10)
```

5.3.3.3 Pending Expectations

Expectations that are neither fulfilled nor violated (i.e., belonging the set **PEND**) are represented by means of a `pending/1` *CHR* constraint, whose content is a term (with functor `e` for **E** expectations and `en` for **EN** expectations) representing the pending expectations. The `pending/1` constraint, obviously, does not apply to ¬**E** or ¬**EN**. For example, if the expectation

$$\mathbf{E}(tell(bob, alice, inform(phone_number, Answer), dialog_id), Ti) \quad (5.3.4)$$

were pending, a *CHR* constraint

```
pending(e(tell(bob,alice,inform(phone_number,Answer),dialog_id),Ti))
```

would be in the *CHR* store, except for a renaming of the variables. The reader should note that the representation of CLP constraints on variable `Ti`, such as `Ti#<20`, are represented in the CLP constraint store, rather than in the expectation itself.

Additionally, *CHR* constraints are used to represent all expectations, either pending, fulfilled or violated: this is needed because transitions such as propagation apply to pending, fulfilled or violated expectations in the same way. These constraints are `e/2`, `en/2`, `note/2` or `noten/2`, for **E**, **EN**, ¬**E** or ¬**EN** expectations, respectively.

The two arguments of these *CHR* constraints are the content and the time of the expectation.

For instance, in the case above mentioned, a *CHR* constraint

```
e(tell(bob,alice,inform(phone_number,Answer),dialog_id),Ti)
```

would be in the store.

5.3.3.4 Fulfilled Expectations

Each fulfilled expectation is represented by a `fulf/1` *CHR* constraint, whose argument is a term representing the fulfilled expectation. For instance, if the expectation

$$\mathbf{E}(\mathit{tell}(\mathit{bob}, \mathit{alice}, \mathit{inform}(\mathit{phone_number}, \mathit{5551235}), \mathit{dialog_id}), \mathit{12}) \qquad (5.3.5)$$

were fulfilled, the *CHR* constraint

```
fulf(e(tell(bob,alice,inform(phone_number,5551234),dialog_id),12))
```

would be in the *CHR* store.

5.3.3.5 Violated Expectations

Each violated expectation is represented by a `viol/1` *CHR* constraint, whose argument is a term representing the violated expectation.

5.4 Variables

Variables are represented by means of attributed SICStus Prolog variables [Hol90, SIC03]. Attributes are used to express the quantification of variables, to mark flagged variables and to impose quantifier restrictions on universally quantified variables.

Flagging, Quantification and Quantifier Restrictions. As explained in Sect. 3.2, variables in the resolvent and in abduced atoms are *flagged.* The flagging of a variable determines whether it is copied when a new copy of a term in which the variable occurs is made: in particular, existentially quantified, flagged variables are not copied.

The quantification of variables, combined with their flagging, is represented by a `quant/1` attribute, whose argument can assume one of the following values :

- `exists`, for existentially quantified, non-flagged variables;
- `existsf`, for existentially quantified, flagged variables;

- `forall`, for universally quantified, non-flagged variables;
- `forallf`, for universally quantified, flagged variables.

Quantifier restrictions for universally quantified variables are expressed by means of attribute `restrictions/1`, whose argument is the expression representing the quantifier restriction.

Constraints for existentially quantified variables are implemented by means of external CLP solvers: in particular, by the CLPFD solver of SICStus Prolog, and by an *ad hoc* constraint solver implemented in CHR (an adaptation of the `domain` solver distributed with the CHR library).

Unification. Unification between terms is implemented as reified unification by means of a *CHR* constraint solver. The *CHR* constraint `reif_unify(T1,T2,B)` means that the terms `T1` and `T2` unify if and only if `B=1`.

5.5 Transitions

The implementation of transitions has been designed so to exploit the built-in Prolog and *CHR* mechanisms whenever possible, both for simplicity and for efficiency.

This has been made possible by the choice of a depth-first strategy for the exploration of the proof tree, and by the representation of data structures described in Sect. 5.3.

5.5.1 IFF-like Transitions.

Unfolding. As explained in Sect. 3.3.1, unfolding applies to defined literals in the resolvent and to defined atoms in the body of social integrity constraints. At the implementation level, we use two different mechanisms to handle the two cases:

- unfolding for a defined literal in the resolvent is achieved by mere Prolog resolution (i.e., by calling the literal);

- unfolding for defined atoms in the body of ICs is achieved by replacing the atom with its definition (by means of the Prolog `clause/2` built-in predicate).

Abduction. Abducible sets (**PEND**, **FULF**, **VIOL**, and their union) are represented as *CHR* constraints, as explained in Sect. 5.3.3.

When the selected literal of the resolvent is an abducible (i.e., a term of functor `e`, `en`, `note`, `noten`), abduction can thus be obtained by:

1. Flagging the variables in the abducible (see Sect. 3.2);
2. Calling the abducible.

This is achieved by the following predicate:

```
abduce(Abducible):-
    term_variables(Abducible,Variables),
    flag_variables(Variables),
    call(Abducible).
```

When the abducible is of type **E** or **EN**, it is also inserted into the **PEND** set of pending expectations, by means of the following propagation *CHR*s:

```
pending_e @
    e(Event,Time)
    ==>
    pending(e(Event,Time)).


pending_en @
    en(Event,Time)
    ==>
    pending(en(Event,Time)).
```

Example 5.5.1 *If the atom* `e(p(X),T)` *were selected in the resolvent, after abduction and the application of the* `pending_e` *rule the* CHR *constraints* `pending(e(p(X),T)` *and* `e(p(X),T)` *would be in the* CHR *store, and the variable* `X` *and* `T` *would be flagged.*

Propagation. Propagation of events and expectations with partially solved integrity constraints exploits the *CHR*-based representation of **HAP**, **PEND** and PSIC explained in Sect. 5.3.3; in this way, propagation of a given kind of atom can be achieved by means of one *CHR* rule. For instance, the following rule implements propagation (and the subsequent case analysis) for **H** atoms:

```
propagation_h @
    h(Event1,Time1),
    psic([[h(Event2,Time2)|MoreH],NotH,E,NotE,En,NotEn,A],Head)
    ==>
    fn_ok(Event1,Event2)
    |
    ccopy(p([[h(Event2,Time2)|MoreH],NotH,E,NotE,En,NotEn,A],
            Head),
          p([[h(Event2a,Time2a)|MoreHa],NotHa,Ea,NotEa,Ena,NotEna,Aa],
            Heada)),
    (reif_unify(p(Event1,Time1),p(Event2a,Time2a),1)->
        psic([MoreHa,NotHa,Ea,NotEa,Ena,NotEna,Aa],Heada);
        reif_unify(p(Event1,Time1),p(Event2a,Time2a),0)).
```

The predicate `fn_ok/2`, called in the rule guard, recursively checks that two terms are compatible as for functor and arity. For instance, `fn_ok(p(a,f(Y)),p(X,f(b)))` holds, `fn_ok(p(A,f(B)),p(A,g(B)))` does not.

The rule is activated each time a new `h/2` or `psic/2` constraint is imposed (i.e., each time a new element is put into **HAP** or PSIC).

The guard checks functor/arity compatibility between the event and the head of the event sub-list in the body of the integrity constraint. If the guard succeeds, a copy is made of the integrity constraint, and unification is attempted between the event and the head of the event sublist of the body of the newly copied integrity constraint. If the unification succeeds, the new integrity constraint (without the head of the event sub-list of the body) is added to PSIC(i.e., the `psic/2` constraint is imposed on it); otherwise, the dis-unification constraint is imposed.

By only trying propagation when the event matches the head of the event sub-list of the integrity constraint, we avoid duplicating the generated expectations.

The rules for propagation of **E**, **EN**, ¬**E** and ¬**EN** atoms are analogous.

Splitting. The depth-first strategy of the implementation allows for dealing with disjunctions according to the following (very common in Prolog practice) schema:

```
split([Disjunct|_]):-
        call(Disjunct).
split([_|MoreDisjuncts]):-
        split(MoreDisjuncts).
```

This schema is applied to disjunctions in the head of integrity constraints. Multiple clauses in predicate definitions in the social knowledge base replace the IFF disjunctions in predicate definitions, and the Prolog resolution deals with them.

Disjunctions in the constraint store are handled by the constraint solver(s).

Case Analysis. Case analysis is not implemented as an independent transition, but its implementation is integrated in the transitions that can lead to case an analysis (namely propagation, fulfillment and violation).

Equivalence Rewriting. As specified in Sect. 3.3.1, equivalence rewriting is delegated to the constraint solver(s).

Logical Equivalence. Logical equivalence replaces a partially solved integrity constraint whose body is *true* with its head. This is implemented by the following *CHR* rule:

```
logical_eq @
    psic([[],[],[],[],[],[],[]],Head)
    <=>
    impose_head(Head).
```

The rule is activated when a partially solved integrity constraints whose body is empty is added to the *CHR* store: simply, the head of the partially solved integrity constraint is imposed (which will usually involve splitting).

5.5.2 Dynamically Growing History.

Happening. Happening of events (i.e., insertion of a **H** atom in **HAP**, see Sect. 3.3.2) is achieved by imposing a `h/2` *CHR* constraint, whose (ground) arguments are the content and the time of the event.

For instance, the insertion into **HAP** of the following event

$$\mathbf{H}(\mathit{tell}(\mathit{alice}, \mathit{bob}, \mathit{query_ref}(\mathit{phone_number}), \mathit{dialog_id}), \mathit{10})$$

is achieved by imposing (calling) the following *CHR* constraint:

```
h(tell(alice,bob,query_ref(phone_number),dialog_id),10)
```

Closure. Closure of the history of the society is achieved by imposing a `close_history/0` *CHR* constraint. The presence of this constraint in the store will be checked by other transitions such as fulfillment of **EN** expectations, or propagation of **not H** atoms.

Propagation of not H atoms. The propagation of **not H** atoms is an application of constructive negation (see Sect. 3.3.2), by means of the following rule:

```
propagation_noth @
    close_history,
    psic([H,[NotH|MoreNotH],E,NotE,EN,NotEN,A],Head) # _psic
    ==>
    true
    &
    Body=[H,[NotH|MoreNotH],E,NotE,EN,NotEN,A],
    ccopy(p(Body,Head),p(Body1,Head1)),
    propagate_noth(Body1,Body2)
```

```
    |
    psic(Body2,Head1)
    pragma
    passive(_psic).
```

The rule is activated when the history is closed by the `close_history` constraint. For each partially solved integrity constraint with `noth` atoms in its body, a copy is made of the integrity constraint. A new version `Body2` of the body `Body1` is obtained by the `propagate_noth/2` predicate, which

1. collects all `h/2` *CHR* constraints whose arguments may unify with those of the `noth` atom;

2. imposes dis-unification between the arguments;

3. removes the `noth` atom from the body.

The partially solved integrity constraint with the new body is then inserted into PSIC (i.e., the `psic/2` constraint is imposed on it).

5.5.3 Fulfillment and violation.

E Fulfillment and EN Violation Fulfillment of **E** and violation of **EN** can be detected while the history is still open. The following *CHR*s implements fulfillment of **E** expectations:

```
fulfillment @
    h(HEvent,HTime),
    pending(e(EEvent,ETime)) # _pending
    ==>
    fn_ok(HEvent,EEvent)
    |
    ccopy(p(EEvent,ETime),p(EEvent1,ETime1)),
    case_analysis_fulfillment(HEvent,HTime,EEvent,ETime,
                  EEvent1,ETime1,_pending).
```

The rule is applied when an event and a pending expectation whose content have recursively the same functor and arity (this is checked by the fn/2 predicate in the guard of the rule) are in the *CHR* store. In this case, a copy is made of the expectation[2] and the case_analysis_fulfillment/7 predicate is called.

```
case_analysis_fulfillment(HEvent,HTime,EEvent,ETime,
              EEvent1,ETime1,_pending):-
    remove_constraint(_pending),
    reif_unify(p(HEvent,HTime),p(EEvent1,ETime1),1),
    fulf(e(EEvent,ETime)).
case_analysis_fulfillment(HEvent,HTime,_,_,EEvent1,ETime1,_):-
    reif_unify(p(HEvent,HTime),p(EEvent1,ETime1),0).
```

The arguments of this predicate represent, respectively, the content of the event, the time of the event, the content of the expectation, the time of the expectation, a copy of the content of the expectation, a copy of the time of the expectation, and the internal constant representing the pending/1 constraint for the expectation. Two nodes are created by case_analysis_fulfillment/7:

- one where unification is imposed between the expectation and the event, the pending/1 constraint for the expectation is removed and fulf/1 *CHR* constraint for the expectation is imposed (which means that the expectation is moved from **PEND** to **FULF**, as explained in Sect. 5.3.3);
- one where non-unification between the expectation and the event is imposed.

Example 5.5.2 *Let the following two* CHR *constraints be in the store:*

```
h(tell(alice,bob,refuse(phone_number),dialog_id),13)
pending(e(tell(alice,bob,refuse(phone_number),dialog_id),T))
```

Then, by rule fulfillment*, in one node the constraint* T=13 *would be imposed, the* CHR *constraint*

[2] As specified in [GLT+03]: this allows for universally quantified variables in **EN** expectations to remain unbound.

```
pending(e(tell(alice,bob,refuse(phone_number),dialog_id),T))
```

would be removed, and the CHR *constraint*

```
fulf(e(tell(alice,bob,refuse(phone_number),dialog_id),T))
```

would be imposed; in the other node, the CHR *constraints would remain the same, but the constraint* `T#\=13` *would be imposed.*

E Violation and EN Fulfillment. When the history of the society is closed (by means of the closure transitions), all **E** expectations in **PEND** are moved to **VIOL**, and all **EN** in **PEND** are moved to **FULF**. This is achieved by the following two rules:

```
closure_e @
        (close_history)
        \
        (pending(e(Event,Time)) # _pending)
        <=>
        viol(e(Event,Time))
        pragma
        passive(_pending).

closure_en @
        (close_history)
        \
        (pending(en(Event,Time)) # _pending)
        <=>
        fulf(en(Event,Time))
        pragma
        passive(_pending).
```

In these two rules, the `pending/1` constraint for the expectation is declared to be passive: thus, the two rules are activated only when the `close_history/0` constraint is imposed.

5.5.4 Consistency

In Sect. 3.3.4, **E**-consistency and ¬-consistency are achieved by imposing additional integrity constraints to the social specification. However, since such integrity constraints would be applied many times during the computation, in order to improve the performance we have preferred to implement **E**-consistency and ¬-consistency by means of a specialised mechanism; precisely, by means of three on-purpose *CHR* rules, as follows.

E-consistency. **E**-consistency is implemented by imposing non-unification on the (*Content*,*Time*) pairs of **E** and **EN** expectations in the store:

```
e_consistency @
        e(EEvent,ETime),
        en(ENEvent,ENTime)
        ==>
        reif_unify(p(EEvent,ETime),p(ENEvent,ENTime),0).
```

Example 5.5.3 *Given the expectations* `e(a,T)` *and* `en(a,3)`*,* **E***-consistency would impose* `reif_unify(p(a,T),p(a,3),0)`*, which would propagate to the CLP constraint* $T \neq 3$ *(represented as* `T#\=3` *in the CLPFD solver available for SICStus Prolog).*

¬-consistency. Analogously to **E**-Consistency, ¬-Consistency is implemented by by imposing non-unification on the (*Content*,*Time*) pairs of **E** and ¬**E** (or **EN** and ¬**EN**) expectations in the store:

```
not_consistency_e @
        e(EEvent,ETime),
        note(NotEEvent,NotETime)
        ==>
        reif_unify(p(EEvent,ETime),p(NotEEvent,NotETime),0).


not_consistency_en @
        en(EnEvent,EnTime),
```

```
noten(NotEnEvent,NotEnTime)
==>
reif_unify(p(EnEvent,EnTime),p(NotEnEvent,NotEnTime),0).
```

5.6 The SOCS-SI system

In order to use $\mathcal{S}$CIFF for actual verification of interaction in multiagent systems, it has been integrated in *SOCS-SI* [ACG+04], a Java system equipped with a Graphical User Interface and interfaces for observing agent interaction (i.e., collecting events) from difference sources.

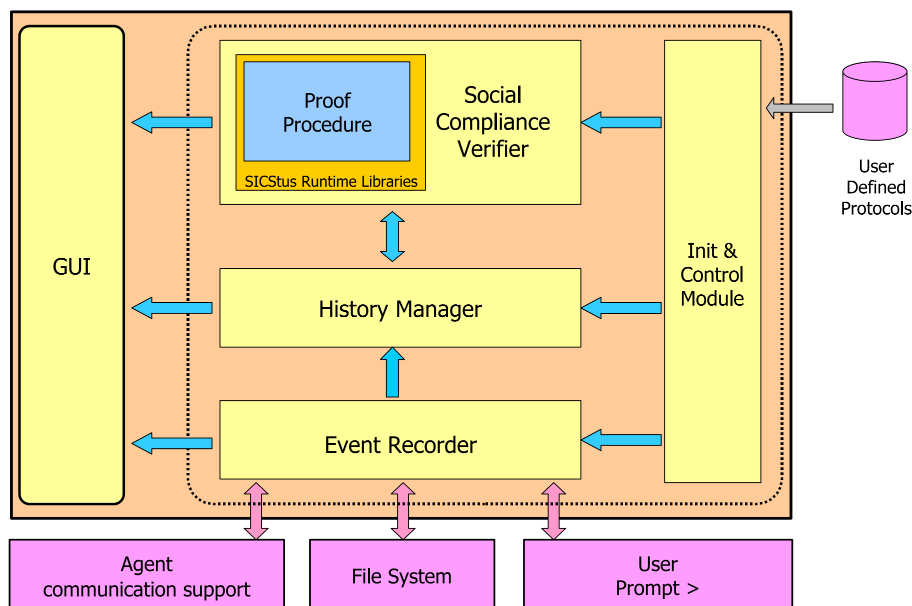


Figure 5.1: The SOCS-SI architecture

The core of *SOCS-SI* is composed of three main modules (see Fig. 5.1), namely:

- *Event Recorder*: fetches events from different sources and stores them inside the *History Manager*.
- *History Manager*: receives events from the *Event Recorder* and composes them into a history.
- *Social Compliance Verifier* (SCV): fetches events from the *History Manager* and passes them on to $\mathcal{S}$CIFF in order to check the compliance of the history

to the specification. It receives the expectations from $\mathcal{S}$CIFF and visualises them in the GUI.

All the modules, except $\mathcal{S}$CIFF, are implemented in Java language. The $\mathcal{S}$CIFF implementation is interfaced to the SCV by means of the Jasper Java-Prolog interface available in SICStus Prolog [SIC03].

The GUI lets the user select:

- the social specification (social knowledge base and social integrity constraints) to be used and the goal to be achieved;
- the source of the events to compose the history, which can be one of the following:
  - a networked source;
  - a text file;
  - the user prompt.

Among the supported networked sources, the first was the communication layer of the PROSOCS [SKL+04] platform, an agent platform developed during the SOCS project. However, *SOCS-SI* has also been integrated with the agent platform JADE [JAD], the coordination platform TuCSoN [OZ99], and an email system.

The GUI (see Fig. 5.2) displays the results of the computation after each event. In particular, it shows the expectation in a non-failure node, if there exists one, or the result of failure. The computation nodes are arranged in a tree structure, which can be viewed and recalled (possibly to understand the cause of a failure in the computation).

5.7 Related work

Before $\mathcal{S}$CIFF, other authors proposed the implementation of abduction in *CHR*, thanks to the common understanding of abducibles and constraints proposed by Kowalski *et al.* [KTW98].

Abdennadher and Christiansen [AC00] characterise abduction in the $\text{CHR}^\vee$ language, which extends *CHR* with disjunctions in the body of its rules. Christiansen and Dahl [CD04], who propose to exploit the *CHR* language to extend

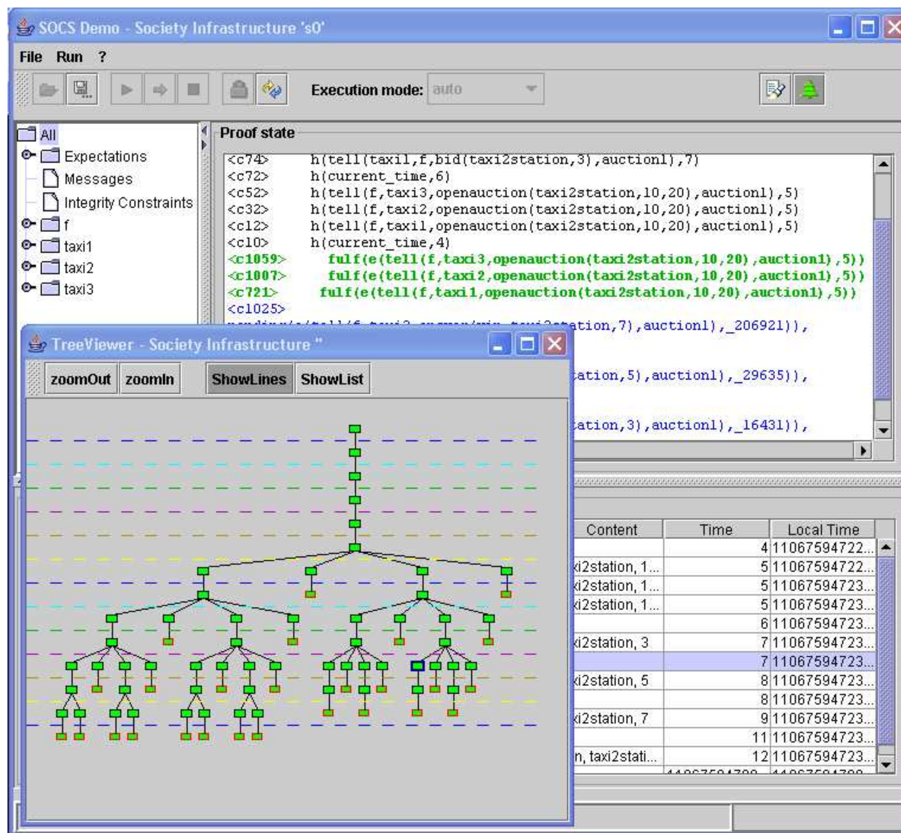


Figure 5.2: A screenshot of SOCS-SI

SICStus Prolog to support abduction more efficiently than with metainterpretation-based solutions. The authors of both papers represent abducibles as *CHR* constraints, and they represent integrity constraints directly as *CHR* propagation rules, using the built-in *CHR* matching mechanism. The same perspective is adopted by Gavanelli *et al.* in the first implementation proposed in [GLM+03].

This technique has the advantage of exploiting directly the existent *CHR* implementation based on Prolog. However, this technique does not seem viable for the implementation of the $\mathcal{S}$CIFF proof procedure, which also needs to handle universally quantified variables and CLP constraints. For this reason, our approach is more similar to the second proposed in [GLM+03], where integrity constraints are implemented ad CLP constraints, rather than as rules.

Chapter 6

Applications

In this chapter, we show some examples of specifications of agent interaction in open agent societies by means of the $\mathcal{S}$CIFF framework. We demonstrate two level of specification: the semantics of Agent Communication Languages, and Interaction Protocols.

6.1 Social ACL Semantics

In this section, we show how the $\mathcal{S}$CIFF framework can be used to give a social semantics to Agent Communication Languages. Further discussion and examples can be found in [ACG+03].

6.1.1 Agent Communication Languages

In most proposals for Multiagent Systems, knowledge exchange between agents is achieved by communication in an *Agent Communication Language* (also ACL, for short, in the following). An ACL provides language primitives (also called *communicative acts*, *utterances*, and in certain contexts *performatives*, *illocutions*, or *dialogue moves*) which agents can use to convey meaning to other agents, according to some predefined semantics.

In most proposals, an ACL is defined by two languages:

- the *communication* language, represented by a set of communication performatives, each corresponding to a different illocution;

- the *content* language which expresses the information to be transferred.

In recent years, much effort has been devoted to the definition of a standard ACL (for instance, by the FIPA consortium [FIP01]); however, the definition of ACL semantics is still an open issue.

Currently, in the international agent community, the most common approaches for the definition of ACL semantics could be denoted by the terms *mentalistic* [FLM97, FIP01] and *social* [Sin98]. In the following, we briefly illustrate the differences between the two approaches.

6.1.1.1 Mentalistic semantics

The mentalistic semantics of an ACL is given by defining which *mental states* lead to the utterance of a communicative act by the speaker agent, and which are the effects of the communicative acts on the mental state of the hearer agent. Obviously, such a semantics strictly constrains the internal architecture of agents to support mental states.

For instance, FIPA ACL assumes a BDI (Belief, Desire, Intention) model for the agents [RG92b], and relies on it for defining the semantics of communicative acts in terms of Feasibility Preconditions (i.e., the conditions that have to be satisfied for the communicative act to be planned) and Rational Effects (i.e., the expected effect that the communicative act would have).

As an example, the definition of the FIPA ACL performative *request* is as follows:

<Sender, REQUEST (Receiver,a)>

$$
\begin{array}{ll}
\mathtt{FP}: & \mathtt{FP(a)[Sender\backslash Receiver]}\wedge \\
 & \mathtt{B_{Sender}Agent(Receiver, a)}\wedge \\
 & \mathtt{B_{Sender}\neg PG_{Receiver}Done(a)} \\
\mathtt{RE}: & \mathtt{Done(a)}
\end{array}
$$

where

- FP denotes the *feasibility preconditions* of the act;
- RE denotes the *rational effect* of the act;

- $\texttt{FP}(\texttt{a})[\texttt{Sender} \setminus \texttt{Receiver}]$ denotes the part of the FPs of a, which are mental attitudes of the Sender;

- $B_{\texttt{Sender}}\texttt{Agent}(\texttt{Receiver}, \texttt{a})$ means that Sender believes that Receiver can perform a;

- $B_{\texttt{Sender}} \neg PG_{\texttt{Receiver}}\ \texttt{Done(a)}$ means that Sender believes that Receiver does not (yet) intend to perform a.

- $\texttt{Done}(\texttt{a})$ means that a has "just" taken place.

It is worth noticing that, according to this definition, the agent Sender should not only be aware of its own mental state, but also have beliefs about the agent Receiver's mental state.

The mentalistic approach to the ACL semantics has been much criticized mainly because its underlying assumptions regarding agents' internals are not realistic in open societies of heterogeneous agents. As Singh stated [Sin98], emphasizing mental agency leads to the supposition that agents should be primarily understood in terms of mental concepts, such as beliefs and intentions: this approach supposes, in essence, that agents can read each other's minds. Whenever agents' mental states are not accessible, which is reasonably the case if agents operate in open and heterogeneous environments, it is impossible to verify semantic compliance of communicative acts.

6.1.1.2 Social semantics

The *social* approach defines ACL semantics in terms of the effects of the communicative acts on the agent society as a whole. Following this approach, even if the agents' mental state cannot be accessed, it is possible to verify whether communicating agents in a society comply to some social laws which regulate the interactions.

Notable proposals for a social semantics are commitment-based [Sin00, FC02]. A *social commitment* is an obligation which binds an agent (usually the speaker in a communicative act) to the society. So, each social commitment refers to a *content* (i.e., the action, or the proposition to be made true), a *debtor* (i.e., the

agent engaged to make the content true) and a *creditor* (i.e., the agent relative to which the commitment is made).

In particular, in Singh's work [Sin00] three levels of semantics for each communication performative are defined: the objective claim (that the subject of the communication is true), the subjective claim (that the communication is sincere) and the practical claim (that the speaker is justified in making the communication).

For instance, the semantics of the performative `inform(s, h, p)` ("`s` `informs` `h` of `p`") is:

- `s` is committed towards `h` that `p` holds (objective claim);
- `s` is committed towards `h` that `s` believes `p` (subjective claim);
- `s` is committed towards the society that he has reasons to believe `p` (practical claim).

In this way, the mentalistic approach is adopted only at the subjective level, while at the practical level a commitment towards the agent society is used.

The social approach is applied to the definition of ACL semantics in [FC02], where an operational specification of an ACL is given in an object-oriented framework by means of the *commitment* class. A commitment represents an obligation for its *debtor* towards its *creditor*. A commitment is described by a finite state automaton, whose states (which can take the values of *empty*, *pre-commitment*, *cancelled*, *conditional*, *active*, *fulfilled* and *violated*) can change by application of methods of the commitment class, or of rules triggered by external conditions. Semantics of communicative acts is specified in terms of methods to be applied to a commitment when a communicative act is issued.

Within this framework, for instance, the semantics of the assertive `inform` performative is given as follows:

$$\mathtt{inform}(\mathtt{Sender}, \mathtt{Receiver}, \mathtt{P}) = \mathtt{make}\ \mathtt{C}(\mathtt{Sender}, \mathtt{Receiver}, \mathtt{P}, \mathtt{true}, \mathtt{CC})$$

where `C(Sender,Receiver,P,true,CC)` is a (*conditional*) commitment (with the condition already *true*) made by `Sender`, to agent `Receiver` (the "creditor" of the commitment) that `P` (the content) will be satisfied. The effect of this utterance

will define a commitment initially in the transitory state CC, that will immediately move (due to the *true* condition) in the state A. It might be later either fulfilled (if P becomes *true*), or violated (if P becomes *false*).

6.1.2 Social ACL semantics with $\mathcal{S}$CIFF

As the authors cited in Sect. 6.1.1.2, and for the same reasons, we believe that, in open societies of possibly heterogeneous agents, a social semantics of communicative acts is more appropriate than a mentalistic one. This does not mean that we oppose the definition of a mentalistic semantics: in fact, the mentalistic semantics is probably the best way to implement the communication features of agents which have a mental structure. However, in open societies, there is no guarantee that agents will have a prescribed mental structure and will communicate accordingly; thus an external perspective on ACL semantics is necessary, if the semantics is to be verifiable.

A social semantics of communicative acts can be given in the $\mathcal{S}$CIFF framework in a way similar to the commitment-based, by establishing a correspondence between commitments and expectations. The social integrity constraints can be seen as forward rules which, given the occurrence of a communicative act (represented by an event), produce an expectation on the future behaviour of the agents involved in the communication, which is related to the concept of commitment.

What follows should not be intended as a proposal for a set of communication performatives (we believe that the set proposed by FIPA [FIP01], for instance, is comprehensive and covers adequately the communication needs of agents), but rather as a demonstration of how a given set of communication performatives can be given a social semantics.

As an example, in the following we map into our framework some of the linguistic primitives defined in [FC02].

In the following, unless otherwise specified, we will express the description of an agent's communicative act (the first argument of the **H** atom, see Def. 2.2.1) as follows:

$$CommunicativeActId(Speaker, Recipient, Content, Context)$$

where *Speaker* is the speaker agent, *Recipient* is the intended recipient, *Content* is the content of the message and *Context* is an identifier of the interaction context between X and Y. In the case of dialogues, *Context* can be a dialogue identifier, set by the agent who initiates the dialogue. A possible event is:

$$\mathbf{H}(\mathit{request}(\mathit{alice}, \mathit{bob}, \mathit{give}(\mathit{scooter}), \mathit{evening_dialog}), 21),$$

meaning the agent *alice* issued a *request* to agent *bob* to *give* a *scooter*, in the context of *evening_dialog*, at time 21.

Assertives: inform - Intuitively, an *inform* communicative act is used by an agent to assert the truth of the content to another agent. In a commitment-based setting like that of [FC02], this equates to the speaker agent to *commit* to the truth of the content to the hearer agent.

In our framework, a possible definition of the semantics of *inform* is as follows:

$$\begin{aligned} &\mathbf{H}(\mathit{inform}(A, B, P, D), T) \\ \rightarrow &\mathbf{E}(\mathit{true}(A, B, P)) \end{aligned} \tag{6.1.1}$$

where, with $\mathbf{E}(\mathit{true}(A, B, P))$, we mean that A is responsible towards B with respect to the truth of P; in other words, if P is proved false, then A has violated a commitment towards B.

We are aware that verifiability is an issue here: who is supposed to verify the truth of P? According to our approach, built on the principle that it is not acceptable to make assumptions about agents (and, therefore, about their truthfulness) there should be a *super partes* entity in the society, equipped with a knowledge base allowing it to decide the truth value of the content of a message. If this is not the case, the only way is probably to associate no expectations to an *inform* act, and let the hearer agent decide about the trustworthiness of the speaker.

Commissives: promise and conditionalPromise - A *promise*, like an *inform*, commits the speaker to the truth of the content, but for the former the speaker is responsible for fulfilling it by means of a physical action.

$$\begin{aligned} &\mathbf{H}(\mathit{promise}(A, B, P, D), T_p) \\ \rightarrow &\mathbf{E}(\mathit{do}(A, B, P, D), T_d) \ \wedge \ T_d \leq T_p + \tau \end{aligned} \tag{6.1.2}$$

where *do* is the action that should make P true. The restriction $T_d \leq T_p + \tau$, where τ is some constant, expresses that the expectation will be fulfilled only if the *do* event happens by the prescribed deadline $T_p + \tau$.

The expectation in a *conditionalPromise* becomes effective only when an event (which plays the role of a condition[1] that is external to the dialogue and thus, intuitively, is supposed not to be an action performed by the speaker) happens:

$$\begin{aligned} &\mathbf{H}(conditionalPromise(A, B, cond(P, Q), D), T_c) \\ &\wedge \mathbf{H}(Q, T_Q) \\ &\rightarrow \mathbf{E}(do(A, B, P, D), T_d) \ \wedge \ T_d \leq max(T_c, T_Q) + \tau \end{aligned} \tag{6.1.3}$$

where Q is a term describing an event and, as usual, $\mathbf{H}(Q, T_Q)$ expresses that Q happens at time T_Q.

Example 6.1.1 *Given the events*

$$\mathbf{H}(conditionalPromise(alice, bob, cond(give(umbrella), start_raining), a_dialog), 10)$$

and

$$\mathbf{H}(start_raining, 15)$$

and assuming $\tau = 10$*, the following expectation would be generated by* $\mathcal{S}$*CIFF:*

$$\mathbf{E}(do(alice, bob, give(umbrella), a_dialog), T_d),$$

with the restriction $T_d \leq 25$*, which could be fulfilled, for example, by the following event:*

$$\mathbf{H}(do(alice, bob, give(umbrella), a_dialog), 22).$$

Directives: request and conditionalRequest - A *request* does not, by itself, generate any expectation. The hearer agent can either *accept* or *reject* the content

[1] In [FC02], the condition is expressed as a *temporal proposition* object. Temporal propositions express the truth value (true, false or undefined) of a statement (about some state of affairs holding, or some action having been performed, or commitment having been created) in a given time interval, with existential or universal temporal quantification. Thus, Fornara and Colombetti's framework can express a broader set of conditions than ours.

of the *request*, by the corresponding communicative acts. Only in case of an *accept* the content of the *request* becomes expected:

$$\begin{aligned} &\mathbf{H}(request(A, B, P, D), T_r) \\ \wedge &\mathbf{H}(accept(B, A, P, D), T_a) \\ \wedge &T_r < T_a \\ \rightarrow &\mathbf{E}(do(B, A, P, D), T_d) \ \wedge \ T_d \leq T_a + \tau \end{aligned} \tag{6.1.4}$$

where $T_r < T_a$ means that the expectation will be raised only if the *request* happens before the *accept*.

A *conditionalRequest* is different from a *request* in that its content becomes the content of an expectation only once the hearer has *accept*ed it *and* an event, specified as a condition in the content of the *conditionalRequest*, has happened.

$$\begin{aligned} &\mathbf{H}(conditionalRequest(A, B, cond(P, Q), D), T_r) \\ \wedge &\mathbf{H}(accept(B, A, cond(P, Q), D), T_a) \\ \wedge &T_r < T_a \\ \wedge &\mathbf{H}(Q, T_Q) \\ \rightarrow &\mathbf{E}(do(B, A, P, D), T_d) \ \wedge \ T_d \leq max(T_a, T_Q) + \tau \end{aligned} \tag{6.1.5}$$

There is no need to express the semantics of a *reject* by a SIC, because a *reject*ed *request* (or *conditionalRequest*) generates no expectations.

Example 6.1.2 *Given the events*

$$\mathbf{H}(conditionalRequest(alice, bob, cond(give(umbrella), start_raining), a_dialog), 10),$$

$$\mathbf{H}(accept(bob, alice, cond(give(umbrella), start_raining), a_dialog), 12),$$

and

$$\mathbf{H}(start_raining, 18)$$

and assuming $\tau = 10$, the following expectation would be generated by $\mathcal{S}$CIFF:

$$\mathbf{E}(do(bob, alice, give(umbrella), a_dialog), T_d),$$

with the restriction $T_d \leq 28$, which could be fulfilled, for example, by the following event:

$$\mathbf{H}(do(bob, alice, give(umbrella), a_dialog), 23).$$

Proposals: propose - A *propose* is similar to a *conditionalRequest*, with the difference that for the former the speaker is able by itself to fulfill the condition by a *do* action.

As *conditionalRequest*, however, *propose* does not, by itself, generate any expectation. It is with *accept* that both the speaker and the hearer become committed to their respective expectations. We assume that the hearer and the speaker can have different time limits for fulfilling the expectations on their behaviour.

$$
\begin{aligned}
&\mathbf{H}(propose(A,B,prop(P_A,P_B),D),T_p)\\
\wedge&\mathbf{H}(accept(B,A,prop(P_A,P_B),D),T_a)\\
\wedge&T_p < T_a\\
\rightarrow&\mathbf{E}(do(A,B,P_A,D),T_{d_A}) \;\wedge\; T_{d_A} \leq T_a + \tau_A\\
\wedge&\mathbf{E}(do(B,A,P_B,D),T_{d_B}) \;\wedge\; T_{d_B} \leq T_a + \tau_B
\end{aligned}
\tag{6.1.6}
$$

Example 6.1.3 *Given the events*

$$\mathbf{H}(propose(alice, bob, prop(give(fight_club), give(the_game)), a_dialog), 10)$$

and

$$\mathbf{H}(accept(bob, alice, prop(give(fight_club), give(the_game)), a_dialog), 13)$$

and assuming $\tau_A = \tau_B = 10$*, the following expectations would be generated by* $\mathcal{S}$*CIFF:*

$$\mathbf{E}(do(alice, bob, give(fight_club), a_dialog), T_{d_A})$$

$$\mathbf{E}(do(bob, alice, give(the_game), a_dialog), T_{d_B})$$

with the restrictions $T_{d_A} \leq 23$ *and* $T_{d_B} \leq 23$*; the expectations could be fulfilled by events as*

$$\mathbf{H}(do(alice, bob, give(fight_club), a_dialog), 18)$$

$$\mathbf{H}(do(bob, alice, give(the_game), a_dialog), 19)$$

6.2 Definition of Agent Interaction Protocols

An *Interaction Protocol* specifies the “rules of encounter” governing a dialogue between agents [RZ94, MPW02]. It specifies which agent is allowed to say what in a given situation. It will usually allow for several alternative utterances in every situation and the agent in question has to choose one according to its private *policy*. From the individual agent perspective, protocols are practically important because they may help to select the adequate answer to an incoming utterance, thus reducing the complexity of this task for an agent ([EMST03b]).

In the following, we briefly review some of the formalisms used for specifying Interaction Protocols and motivate the use of the $\mathcal{S}$CIFF formalism for this purpose. Then, we provide several examples of definitions of Interaction Protocols in the $\mathcal{S}$CIFF framework.

6.2.1 Formalisms for Agent Interaction Protocols

Until recently, the research on multiagent systems viewed Interaction Protocols (or IPs, for short) as a practical matter as far as agent communication theory was concerned, thus remaining disconnected from the large amount of existing work on IPs [BHS93, Dem95]. It was indeed assumed that conversations structure should emerge as a consequence of the semantics of individual messages.

This position has raised many critics, especially in the context of open systems, and IPs are now considered as structures of theoretical importance when one tries to model agent interactions.

So far, no formalism has been accepted as a standard for expressing IPs in multiagent systems. In fact, the literature on IPs offers a number of different formalisms, the most commonly used being *Finite State Machines* and *AUML Diagrams*.

Finite State Machines (FSMs) are arguably the most adequate (and popular) formalism to account for *sequential* interactions. The state of the automaton describes the state of the conversation. Carefully designed FSMs have been implemented in real application,s see for instance COOL [BF95]. However, because it is necessary to specify all the local states of the interaction, it is clear that

designers face a practical specification problem and consequently tend to over-simplify the protocols.

AUML Protocol Diagrams rely on an extension of the classical UML formalism specially dedicated to agents [BMO01]. Protocol diagrams introduce a number of new features: most notably, concurrent messages are allowed, and the cardinality of messages is not restricted to the one-to-one case. The notion of role is central: protocol diagrams typically represents the lifelines of agents using defined roles, and the steps in which the communicative acts are sent between these agents. AUML supports partial or complete reuse of protocols. There is still ongoing research trying to enhance the formalism with useful notions (e.g., synchronisation, exception handling, see [Hug02]). However, it should be kept in mind that AUML remains a semi-formal specification.

In fact, in its 1999 specifications, FIPA [FIP] used a finite state machine representation of its interaction protocols; but, as a consequence of the collaboration between FIPA and OMG (Object Management Group), the 2001 specification has adopted the new Agent UML standard [BMO01] and thus uses Protocol Diagrams to describe interaction (see, for instance, the specification of the FIPA Request IP in Fig. 6.1).

Most of the times, however, protocol designers use the simplest formalism which meet their requirement for a given application.

We believe that logic-based approaches can be fruitfully exploited for the definition of IPs.

For instance, in [YS02], a variant of the event calculus [KS86] is applied to commitment-based protocol specification. The semantics of messages (i.e., their effect on commitments) is described by a set of *operations* whose semantics, in turn, is described by *predicates* on *events* and *fluents*; in addition, commitments can evolve, independently of communicative acts, in relation to *events* and *fluents* as prescribed by a set of *postulates*. This way of specifying protocols is more flexible than traditional approaches specifying protocols as action sequences in that it prescribes no initial and final states or transitions explicitly, but allows any possible protocol to run with the only condition that, at the end of a protocol run, no commitment must be pending; agents with reasoning capabilities can

themselves plan an execution path suitable for their purposes (which, in that work, is implemented by an abductive event calculus planner).

Our motivations for adopting the $\mathcal{S}$CIFF framework for defining IPs are the same supporting commitments and committed-based semantics in [YS02]. The idea is also in a way similar to that of *conversation policies*, defined as *"general constraints on the sequences of semantically coherent messages leading to a goal"* [GHB00], but with a more flexible approach.

In particular, we identify the main strengths of our approach in:

Flexibility Most of the formal approaches to model protocols require that each state of the interaction be described. This can be practically tedious and motivate designers to over-constrain protocols, affecting in turn the flexibility of the interactions and the autonomy of the computees. Instead, *"participants must be constrained in their interactions only to the extent necessary to carry out the given protocol and no more"* [YS02].

Expressiveness - Sometimes, it can be necessary to include extra integrity constraints left implicit in semi-formal models, as shown in [EMST03a]. For example, the explicit representation of the time parameter within the constraints allows to handle time deadlines and synchronisation easily.

Properties - Being based on Computational Logic, the $\mathcal{S}$CIFF framework lends itself well to studying and verifying properties of protocols.

Agent Autonomy - Our approach also guarantees autonomy, in that agents are not constrained in their behaviour but they can act as they planned to do. The outcome of their actions will depend, from a social perspective, from the fact that they obey or not to the protocol definition.

6.2.2 Conventions for describing interactions

In the following protocol definitions, all events will be assumed to be communicative actions. The *Description* of an event (the first argument of **H** atoms) will have the following format:

$$\mathit{tell}(\mathit{Performer}, \mathit{Addressee}, \mathit{Content}, \mathit{Context}),$$

where *Performer* is the agent performing the action, *Addressee* is the agent towards which the action is addressed, *Content* is the content of the action and *Context* is an identifier of the dialogue or interaction in which the action takes place.

6.2.3 Semi-open society

According to [Dav01], societies can be classified into 4 groups, each characterised by a different degree of openness. In the following, we give an example of how our framework can model a semi-open society, i.e., a society that can be joined by an agent by executing an entering protocol.

We suppose that a particular agent (indicated by the constant identifier *gatekeeper*) is in charge of receiving joining requests, and requires, for agents to enter, some form to be filled. In detail, a protocol run is as follows:

1. The candidate agent C sends a registration request to *gatekeeper*;
2. *gatekeeper* asks C to submit a registration form;
3. C submits the registration form;
4. *gatekeeper* either accepts or rejects the registration request.

The entering protocol is defined by the SICs in Spec. 6.1.

The first SIC imposes that *gatekeeper* reply to a registration request from an agent C by asking C for a registration form. The second SIC imposes that, after requesting the registration and having received the request for the registration form, C send the registration form. The third SIC imposes that *gatekeeper*, once received from C the requested registration form, reply to C by either accepting or rejecting the registration request.

By means of the restriction on the time variable in all the generated expectations, each message is imposed to follow the previous message by at most 10 time units.

Specification 6.1 Semi-open society: SICs for the entering phase.

$$\mathbf{H}(tell(C, gatekeeper, ask(register), D), T)$$
$$\rightarrow \mathbf{E}(tell(gatekeeper, C, ask(form), D), T1) \ \wedge \ T1 \ < \ T \ + \ 10$$

$$\mathbf{H}(tell(C, gatekeeper, ask(register), D), T) \ \wedge$$
$$\mathbf{H}(tell(gatekeeper, C, ask(form), D), T1) \ \wedge \ T \ < \ T1$$
$$\rightarrow \mathbf{E}(tell(C, gatekeeper, send(form, F), D), T2) \ \wedge \ T2 \ < \ T1 \ + \ 10$$

$$\mathbf{H}(tell(gatekeeper, C, ask(form), D), T1) \ \wedge$$
$$\mathbf{H}(tell(C, gatekeeper, send(form, F), D), T2) \ \wedge \ T1 \ < \ T2$$
$$\rightarrow \mathbf{E}(tell(gatekeeper, C, accept(register), D), T3) \ \wedge \ T3 \ < \ T2 \ + \ 10$$
$$\vee \ \mathbf{E}(tell(gatekeeper, C, reject(register), D), T3) \ \wedge \ T3 \ < \ T2 \ + \ 10$$

Once the protocol has been completed, the agent is a "full member" of the society. In this perspective, the presence in the history of an event of type:

$H(tell(gatekeeper, C, accept(register), D), T)$

represents the "full membership" of agent C in the society, and can be used in SICs as a condition for generating expectations.

For instance, the SICs of the *query_ref* protocol (see Spec. 2.5) can be modified, in order to take membership into account, as in Spec. 6.2.

In the new version, the events that represent the full membership of the agents to the society appear in the body of all SICs. In this way, *query_ref* messages do not generate any expectation, unless both the sender and the receiver are full members of the society, in the sense explained above.

It is apparent that this mechanism makes SICs bigger and less readable; however, it is quite easy to devise some kind of syntactic sugar to avoid the problem.

6.2.4 FIPA Request Interaction Protocol

The FIPA Request Interaction Protocol [FIP02], depicted in Fig. 6.1 allows one agent to request another to perform some action. The normal protocol flow is

Specification 6.2 Semi-open society: SICs for the *query_ref* protocol with "full membership" condition.

$$
\begin{aligned}
& \mathbf{H}(\mathit{tell}(A, B, \mathit{query_ref}(\mathit{Info}), D), T) \wedge \\
& \mathbf{H}(\mathit{tell}(\mathit{gatekeeper}, A, \mathit{accept}(\mathit{register}), D1), Ta) \ \wedge \ Ta \ < \ T \wedge \\
& \mathbf{H}(\mathit{tell}(\mathit{gatekeeper}, B, \mathit{accept}(\mathit{register}), D1), Tb) \ \wedge \ Tb \ < \ T \\
\rightarrow\ & \mathbf{E}(\mathit{tell}(B, A, \mathit{inform}(\mathit{Info}, \mathit{Answer}), D), T1) \ \wedge \ T1 \ < \ T \\
\vee\ & \mathbf{E}(\mathit{tell}(B, A, \mathit{refuse}(\mathit{Info}), D), T1) \ \wedge \ T1 \ < \ T
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{H}(\mathit{tell}(A, B, \mathit{inform}(\mathit{Info}, \mathit{Answer}), D), T) \wedge \\
& \mathbf{H}(\mathit{tell}(\mathit{gatekeeper}, A, \mathit{accept}(\mathit{register}), D1), Ta) \ \wedge \ Ta \ < \ T \wedge \\
& \mathbf{H}(\mathit{tell}(\mathit{gatekeeper}, B, \mathit{accept}(\mathit{register}), D1), Tb) \ \wedge \ Tb \ < \ T \\
\rightarrow\ & \mathbf{EN}(\mathit{tell}(A, B, \mathit{refuse}(\mathit{Info}), D), T1)
\end{aligned}
$$

composed of the following steps:

1. The *Initiator* agent issues a *request* to a *Participant* agent to perform an action P.
2. *Participant* can either
   - *refuse* to perform P, in which case the protocol ends; or
   - *accept* to perform P; in this case, after performing the action,
3. *Participant* will issue to *Initiator* one of the following:
   - *inform_done*(P), which simply tells *Initiator* that P has been performed;
   - *inform_result*(P,R), which also contains, in R, some information about the result of performing the action;
   - *failure*(P), which reports a failure;

The $\mathcal{S}$CIFF-based specification of the protocol is shown in Spec. 6.3 (the social knowledge base is empty). The first SIC imposes to a *Participant* who has received a *request* to perform an action, to reply with either *accept* or *refuse*. The

Specification 6.3 SICs for the FIPA Request interaction protocol.

$$\begin{aligned}
&\mathbf{H}(\mathit{tell}(\mathit{Initiator}, \mathit{Participant}, \mathit{request}(P), D), T)\\
\rightarrow\ &\mathbf{E}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{agree}(P), D), \mathit{T1})\ \wedge\ T < \mathit{T1}\\
\vee\ &\mathbf{E}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{refuse}(P), D), \mathit{T1})\ \wedge\ T < \mathit{T1}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{H}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{agree}(P), D), \mathit{T1})\\
\rightarrow\ &\mathbf{EN}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{refuse}(P), D), \mathit{T2})\ \wedge\ \mathit{T1} < \mathit{T2}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{H}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{refuse}(P), D), \mathit{T1})\\
\rightarrow\ &\mathbf{EN}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{agree}(P), D), \mathit{T2})\ \wedge\ \mathit{T1} < \mathit{T2}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{H}(\mathit{tell}(\mathit{Initiator}, \mathit{Participant}, \mathit{request}(P), D), T)\ \wedge\\
&\mathbf{H}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{agree}(P), D), \mathit{T1})\ \wedge\ T < \mathit{T1}\\
\rightarrow\ &\mathbf{E}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{failure}(P), D), \mathit{T2})\ \wedge\ \mathit{T1} < \mathit{T2}\\
\vee\ &\mathbf{E}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{inform_done}(P), D), \mathit{T2})\ \wedge\ \mathit{T1} < \mathit{T2}\\
\vee\ &\mathbf{E}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{inform_result}(P, R), D), \mathit{T2})\ \wedge\ \mathit{T1} < \mathit{T2}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{H}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{failure}(P), D), T)\\
\rightarrow\ &\mathbf{EN}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{inform_done}(P), D), \mathit{T1})\ \wedge\ T < \mathit{T1}\ \wedge\\
&\mathbf{EN}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{inform_result}(P, R), D), \mathit{T2})\ \wedge\ T < \mathit{T2}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{H}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{inform_done}(P), D), T)\\
\rightarrow\ &\mathbf{EN}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{failure}(P), D), \mathit{T1})\ \wedge\ T < \mathit{T1}\ \wedge\\
&\mathbf{EN}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{inform_result}(P, R), D), \mathit{T2})\ \wedge\ T < \mathit{T2}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{H}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{inform_result}(P, R), D), T)\\
\rightarrow\ &\mathbf{EN}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{inform_done}(P), D), \mathit{T1})\ \wedge\ T < \mathit{T1}\ \wedge\\
&\mathbf{EN}(\mathit{tell}(\mathit{Participant}, \mathit{Initiator}, \mathit{failure}(P), D), \mathit{T2})\ \wedge\ T < \mathit{T2}
\end{aligned}$$

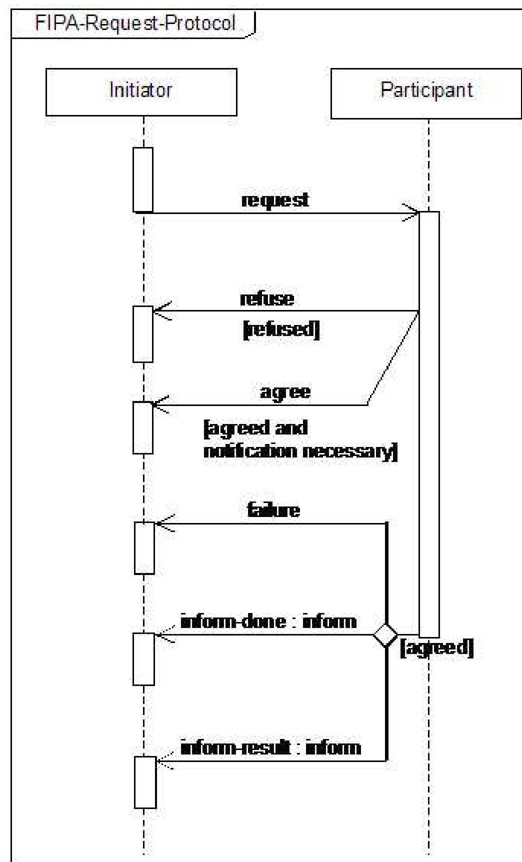


Figure 6.1: FIPA Request Interaction Protocol (from [FIP02])

second and third SICs impose mutual exclusiveness between *accept* and *refuse*: if *Participant* has *accept*ed, it cannot *refuse* later, and vice-versa.

The fourth SIC imposes *request* and *agree* to be followed by one of *inform_done*, *inform_result*, and *failure*. The last three SICs impose mutual exclusiveness among the three.

It can be noted that, in this case, time deadlines have not been specified, but restrictions on the time variables are used to imposed the temporal order of events.

6.2.5 NetBill

NetBill (see [CTS95]) is a security and transaction protocol optimised for the selling and delivery of low-priced information goods, like software or journal articles. The protocol rules transactions between two agents: *merchant* and *customer*. A

NetBill server is used to deal with financial issues such as those related to credit card accounts of customer and merchant.

In the following, we focus on the type of the NetBill protocol designed for non zero-priced goods, and do not consider the variants that deal with zero-priced goods.

The typical protocol flow is composed of three phases:

1. *price negotiation.* The customer requests a quote for a good identified by *PrId* (*priceRequest(PrId)*), and the merchant replies with (*priceQuote(PrId,Quote)*).

2. *good delivery.* The customer requests the good (*goodRequest(PrId,Quote)*) and the merchant delivers it in an encrypted format (*deliver(crypt(PrId,Key),Quote)*).

3. *payment.* The customer issues an Electronic Payment Order (EPO) to the merchant, for the amount agreed for the good (*payment(epo(C,crypt(PrId,K),Quote))*); the merchant appends the decryption key for the good to the EPO, signs the pair and forwards it to the NetBill server (*endorsedEPO(epo(C,crypt(PrId,K),Quote),M)*); the NetBill server deals with the actual money transfer and returns the result to the merchant (*signedResult(C,PrID,Price,K)*), who will, in her turn, send a receipt for the good and the decryption key to the customer (*receipt(PrId,Price,K)*).

The customer can withdraw from the transaction until she has issued the *EPO* message; the merchant until she has issued the *endorsedEPO* message.

The NetBill protocol is implemented in the $\mathcal{S}$CIFF framework by means of SICs which, conceptually, are of two types:

- *backward integrity constraints* (Spec. 6.4), i.e., integrity constraints that state that if some set of event happens, then some other set of event is expected to have happened before.

  For instance, the first backward integrity constraints imposes that, if *M* has sent a *priceQuote* message to *C*, stating that *M*'s quote for the good

Specification 6.4 NetBill protocol: backward SICs.

$$\mathbf{H}(tell(M, C, priceQuote(PrId, Quote), Id), T)$$
$$\rightarrow \mathbf{E}(tell(C, M, priceRequest(PrId), Id), T2) \ \wedge \ T2 \ < \ T$$

$$\mathbf{H}(tell(C, M, goodRequest(PrId, Quote), Id), T)$$
$$\rightarrow \mathbf{E}(tell(M, C, priceQuote(PrId, Quote), Id), Tpri) \ \wedge \ Tpri \ < \ T$$

$$\mathbf{H}(tell(M, C, goodDelivery(crypt(PrId, K), Quote), Id), T)$$
$$\rightarrow \mathbf{E}(tell(C, M, goodRequest(PrId, Quote), Id), Treq) \ \wedge \ Treq \ < \ T$$

$$\mathbf{H}(tell(C, M, payment(C, crypt(PrId, K), Quote), Id), T)$$
$$\rightarrow \mathbf{E}(tell(M, C, goodDelivery(crypt(PrId, K), Quote), Id), Tdel) \ \wedge \ Tdel \ < \ T$$

$$\mathbf{H}(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign) \ \wedge \ M \neq netbill$$
$$\rightarrow \mathbf{E}(tell(M, netbill, endorsedEPO(epo(C, PrId, Quote), K, M), Id), T) \ \wedge \ T \ < \ Tsign$$

$$\mathbf{H}(tell(M, C, receipt(PrId, Quote, K), Id), Ts)$$
$$\rightarrow \mathbf{E}(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign) \ \wedge \ Tsign \ < \ Ts$$

identified by *PrId* is *Quote*, in the interaction identified by *Id*, then *C* is expected to have sent to *M* a *priceRequest* message for the same good, in the same interaction, at an earlier time;

Specification 6.5 NetBill protocol: forward SICs.

$$\mathbf{H}(tell(M, netbill, endorsedEPO(epo(C, PrId, Quote), K, M), Id), T)$$
$$\rightarrow \mathbf{E}(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign) \ \wedge \ T \ < \ Tsign$$

$$\mathbf{H}(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign)$$
$$\rightarrow \mathbf{E}(tell(M, C, receipt(PrId, Quote, K), Id), Ts) \ \wedge \ Tsign \ < \ Ts$$

- *forward integrity constraints* (Spec. 6.5), i.e., constraints that state that if some conjunction of event has happened, then some other set of event is expected to happen in the future.

  For instance, the first forward integrity constraint in Fig. 6.5 imposes that an *endorsedEPO* message from *M* to the *netbill* server be followed by a *signedResult* message, with the corresponding parameters.

  We only impose forward constraints from the *endorsedEPO* message onwards, because both parties (merchant and customer) can withdraw from the transaction at the previous steps.

6.2.6 The Needham-Schroeder Public Key protocol

Protocol description. The Needham-Schroeder protocol has been presented in [NS78], where the authors discuss a way for ensuring the mutual exchange of a secret (a pair of numbers, called *nonces*) between two peers over an *insecure network connection.* The purpose of the protocol is to ensure mutual authentication while maintaining secrecy. In other words, once agents *A* and *B* have successfully completed a run of the protocol, *A* should believe his partner to be *B* if and only if *B* believes his partner to be *A*.

The condition of *insecure network connection* can be stated as follows:

1. when a peer sends a message to another peer, the sender has no way to know if the message has been received or not;

2. when a peer receives a message, there is no way to be sure about the sender, unless this information is somehow coded into the payload;

3. the content of a message could be compromised someway.

In support of the authentication procedure, agents rely on the well-known public key encryption technology. By following the protocol, the two agents involved in a communication session (conversation) challenge each other to make sure that each one's partner in the conversation is actually the holder of the private key associated with his public key.

The protocol consists of seven steps, but, as other authors [DFFv04], we focus on a simplified version of it, consisting of only three steps, which are the kernel of the protocol. The simplification means that we assume that all the agents know the public key of the other agents. The protocol flow can be represented as in Spec. 6.6, where $\langle M \rangle_{PK}$ means that M is encrypted with public key PK.

Specification 6.6 The Needham-Schroeder protocol (simplified version)

(1) $A \rightarrow B : \langle N_A, A \rangle_{pub_key(B)}$

(2) $B \rightarrow A : \langle N_A, N_B \rangle_{pub_key(A)}$

(3) $A \rightarrow B : \langle N_B \rangle_{pub_key(B)}$

By message (1), A challenges B to decrypt his nonce N_A encrypted using B's public key. By message (2), B responds to A's challenge, by attaching to N_A a new nonce N_B, which he generated himself, and encrypting the whole set of two nonces using A's public key, thus challenging A to decrypt N_B and prove to be the holder of A's private key. At this point of interaction, A believes that he is speaking with B, since the latter proved to be able to decrypt the message (1) and answering back the N_A. Of course, this is reasonable under the assumption that it is extremely improbable that an agent could guess the nonce N_A. By message (3), A responds to B's challenge, giving a proof (the N_B sent in message

(2)) of being A. In similar way to what happens after messages (1) and (2), B believes his fellow is A upon receiving message (3).

As for the agents' abilities, we refer to the Dolev-Yao model, which relies on the perfect cryptography assumption (nothing can be learned on a plain text from its encrypted version, without knowing the decryption key). In particular, if we want to define the perfect cryptography assumption in terms of exchanged messages, we say that an agent can:

- decrypt messages encrypted with his own public key;
- generate messages with nonces that (*i*) have never been generated by other agents, or (*ii*) that he received in a message encrypted with his own public key;
- forward messages.

Specification 6.7 Lowe's attack on the Needham-Schroeder protocol

(1) $A \rightarrow I : \langle N_A, A \rangle_{pub_key(I)}$

(2) $I(A) \rightarrow B : \langle N_A, A \rangle_{pub_key(B)}$

(3) $B \rightarrow I(A) : \langle N_A, N_B \rangle_{pub_key(A)}$

(4) $I \rightarrow A : \langle N_A, N_B \rangle_{pub_key(A)}$

(5) $A \rightarrow I : \langle N_B \rangle_{pub_key(I)}$

(6) $I(A) \rightarrow B : \langle N_B \rangle_{pub_key(B)}$

Lowe's attack on the protocol. Eighteen years after the publication of the NSPK protocol, Lowe [Low96] proved it to be prone to security attack. Lowe's attack on the protocol is shown in Spec. 6.7, where a third agent I (standing for *intruder*) manages to successfully authenticate himself as agent A with a third agent B. Although the protocol is correctly followed, B believes he is communicating with A, while instead he is communicating with I.

The messages composing the attack belong to two different dialogues, A with I and I with B. Each dialogue follows the protocol, but I exploits the information of the first dialogue to deceive B in the second dialogue.

$\mathcal{S}$CIFF-based specification of the protocol. The possible formats of the events that represent the messages are the following:

- $\mathbf{H}(send(A, B, content(key(K), agent(A), nonce(N_A))), T)$

  (agent A has sent to agent B its own identifier and a nonce, encrypted with a key K, at time T)

- $\mathbf{H}(send(A, B, content(key(K), nonce(N_A), nonce(N_B))), T)$

  (agent A has sent to agent B two nonces, encrypted with a key K, at time T)

- $\mathbf{H}(send(A, B, content(key(K), nonce(N_A), empty(0))), T)$

  (agent A has sent to agent B a nonce, encrypted with a key K, at time T)

The SICs used for defining the protocol are of two types:

Specification 6.8 SICs for the NSPK protocol.

$$
\begin{aligned}
& \mathbf{H}(send(X, B, content(key(KB), agent(A), nonce(NA))), T1) \\
\rightarrow\ & \mathbf{E}(send(B, X, content(key(KA), nonce(NA), nonce(NB))), T2) \wedge \\
& NA \neq NB \wedge isPublicKey(A, KA) \wedge isNonce(NB) \wedge \\
& isMaxTime(TMax) \wedge T2 > T1 \wedge T2 < TMax
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{H}(send(X, B, content(key(KB), agent(A), nonce(NA))), T1) \wedge \\
& \mathbf{H}(send(B, X, content(key(KA), nonce(NA), nonce(NB))), T2) \wedge T2 > T1 \\
\rightarrow\ & \mathbf{E}(send(X, B, content(key(KB), nonce(NB), empty(0))), T3) \wedge \\
& isMaxTime(TMax) \wedge T3 > T2 \wedge T3 < TMax
\end{aligned}
$$

- A first group of SICs, depicted in Spec. 6.8, defines the protocol itself or, more precisely, the simplified version of the protocol we are modelling. A

second group of SICs has been introduced to define the "contour" conditions applied to the protocol.

The first SIC of Spec. 6.8 states that, whenever an agent B receives a message from agent X, and this message contains the name of some agent A (possibly the name of X himself), some nonce N_A, encrypted with some public key K_B, then a message is expected to be sent at a later time (and by some deadline T_{Max}) from B to X, containing the original nonce N_A and a new nonce N_B, encrypted with the public key of A.

The second SIC of Spec. 6.8 expresses that if two messages have been sent, with the characteristics that: a) a first message has been sent at the instant T_1, from X to B, containing the name of some agent A and some nonce N_A, encrypted with some public key K_B; and b) a second message has been sent at a later instant T_2, from B to X, containing the original nonce N_A and a new nonce N_B, encrypted with the public key of A; then a third message is expected to be sent from X to B, containing N_B, and encrypted with the public key of B.

- A second group of SICs is needed in order to impose the condition that an agent is not able to guess another agent's *nonce*, neither a private key that he does not own. In Spec. 6.9 it is shown how this condition has been translated. Intuitively, we can say that an agent X can send to another agent Y a message containing a nonce N_X which he does not initially know only if one of the following two cases hold: either (i) X received N_X from another agent, encrypted in X's own public key, or (ii) X received a message containing N_X and encrypted with a public key K_Y, in which case X can forward exactly the same message, without operating any modification on it.

The predicates used in the SICs are defined in the social knowledge base in Spec. 6.10.

Specification 6.9 SICs to to express that an agent cannot guess a nonce in the NSPK protocol.

$$\begin{aligned}
& \mathbf{H}(send(X, Y, content(key(KY), agent(W), nonce(NX))), T1) \wedge \\
& X \neq Y \wedge notIsNonce(X, NX) \\
\rightarrow\ & \mathbf{E}(send(V, X, content(key(KX), agent(V), nonce(NX))), T0) \wedge \\
& X \neq V \wedge isNonce(V, NX) \wedge isPublicKey(X, KX) \wedge \\
& isAgent(V) \wedge T0 < T1 \wedge T0 > 0
\end{aligned}$$

$$\begin{aligned}
& \mathbf{H}(send(X, Y, content(key(KY), nonce(NX), nonce(NY))), T1) \wedge \\
& X \neq Y \wedge notIsNonce(X, NX) \\
\rightarrow\ & \mathbf{E}(send(Z, X, content(key(KX), agent(V), nonce(NX))), T0) \wedge \\
& X \neq V \wedge Z \neq X \wedge isPublicKey(X, KX) \wedge \\
& isAgent(V) \wedge isAgent(Z) \wedge T0 < T1 \wedge T0 > 0
\end{aligned}$$

$$\begin{aligned}
& \mathbf{H}(send(X, Y, content(key(KY), nonce(NX), empty(0))), T1) \wedge \\
& X \neq Y \wedge notIsNonce(X, NX) \\
\rightarrow\ & \mathbf{E}(send(Y, X, content(key(KX), nonce(NW), nonce(NX))), T0) \wedge \\
& isPublicKey(X, KX) \wedge isNonce(NW) \wedge NW \neq NX \wedge \\
& T0 < T1 \wedge T0 > 0 \\
\vee\ & \mathbf{E}(send(Z, X, content(key(KX), nonce(NX), empty(0))), T0) \wedge \\
& isPublicKey(X, KX) \wedge X \neq Z \wedge Y \neq Z \wedge \\
& isAgent(Z) \wedge T0 < T1 \wedge T0 > 0
\end{aligned}$$

Specification 6.10 Social knowledge base for the NSPK protocol.

$$isPublicKey(PK) \leftarrow isPublicKey(_, PK).$$

$$isPublicKey(i, ki).$$
$$isPublicKey(b, kb).$$
$$isPublicKey(a, ka).$$

$$isMaxTime(7).$$

$$isAgent(i).$$
$$isAgent(a).$$
$$isAgent(b).$$

$$isNonce(N) \leftarrow isNonce(_, N).$$

$$isNonce(b, nb).$$
$$isNonce(i, ni).$$
$$isNonce(a, na).$$

6.2.7 First Price Sealed Bid auction

The *First Price Sealed Bid* is a simple auction, where the bidders can make at most one offer.

The protocol flow is as follows:

1. an *Auctioneer* agent opens the auction with an *openauction* message, in which the *Item* being sold (or bought), the auction closing time *TEnd* and the deadline *TDeadline* for winner declaration are specified;

2. the interested agents bid for *Item* with a *Quote*, by *TEnd*;

3. by *TDeadline*, each bid is declared by the auctioneer as winning or losing, but not both.

The protocol regulating the "first price sealed bid" auction can be represented by the SICs in Spec. 6.11.

The first SIC is a backward one, which checks that, for each *bid*, an *openauction* for the correct *Item* and with correct time parameters have been issued. The second SIC imposes that each *bid*, if preceded by a correspondent *openauction*, receive either a *win* or a *lose* reply. The last two SICs impose mutual exclusiveness between *win* and *lose* replies.

6.2.8 Combinatorial auctions

Combinatorial auctions are a type of auction where the auctioneer intends to buy (or to sell) a set I of items, and bidders bid for subsets of I. Among the several types of combinatorial auction, in the following, we focus on *single unit, reverse auctions*, where the auctioneer is a customer which attempts to buy a set of distinguishable items at the minimum cost.

Although combinatorial auctions are a powerful sale mechanism, in that they let bidders bid for sets of items, possibly expressing complementarity or substitutability among items [Nis00], their use in real world E-Commerce has been prevented until recently by the NP-hard complexity of the *Winner Determination Problem* (WDP). However, the availability of efficient solvers has now made combinatorial auctions viable.

Specification 6.11 SICs for the first price sealed bid auction protocol.

$$
\begin{aligned}
& \mathbf{H}(\mathit{tell}(B, A, \mathit{bid}(\mathit{Item}, \mathit{Quote}), \mathit{AuctionId}), \mathit{TBid}) \\
\rightarrow\ & \mathbf{E}(\mathit{tell}(A, B, \mathit{openauction}(\mathit{Item}, \mathit{TEnd}, \mathit{TDeadline}), \mathit{AuctionId}), \mathit{TOpen}) \wedge \\
& \mathit{TOpen} < \mathit{TBid} \wedge \mathit{TBid} \leq \mathit{TEnd} \wedge \mathit{TEnd} < \mathit{TDeadline}
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{H}(\mathit{tell}(A, B, \mathit{openauction}(\mathit{Item}, \mathit{TEnd}, \mathit{TDeadline}), \mathit{AuctionId}), \mathit{TOpen}) \wedge \\
& \mathbf{H}(\mathit{tell}(B, A, \mathit{bid}(\mathit{Item}, \mathit{Quote}), \mathit{AuctionId}), \mathit{TBid}) \wedge \\
& \mathit{TOpen} < \mathit{TBid} \wedge \mathit{TOpen} \leq \mathit{TEnd} \wedge \mathit{TEnd} < \mathit{TDeadline} \\
\rightarrow\ & \mathbf{E}(\mathit{tell}(A, B, \mathit{answer}(\mathit{win}, \mathit{Item}, \mathit{Quote}), \mathit{AuctionId}), \mathit{TWin}) \wedge \\
& \mathit{TWin} \leq \mathit{TDeadline} \wedge \mathit{TEnd} < \mathit{TWin} \\
\vee\ & \mathbf{E}(\mathit{tell}(A, B, \mathit{answer}(\mathit{lose}, \mathit{Item}, \mathit{Quote}), \mathit{AuctionId}), \mathit{TLose}) \wedge \\
& \mathit{TLose} \leq \mathit{TDeadline} \wedge \mathit{TEnd} < \mathit{TLose}
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{H}(\mathit{tell}(A, B, \mathit{answer}(\mathit{win}, \mathit{Item}, \mathit{Quote}), \mathit{AuctionId}), \mathit{TWin}) \\
\rightarrow\ & \mathbf{EN}(\mathit{tell}(A, B, \mathit{answer}(\mathit{lose}, \mathit{Item}, \mathit{Quote}), \mathit{AuctionId}), \mathit{TLose}) \wedge \mathit{TLose} > \mathit{TWin}
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{H}(\mathit{tell}(A, B, \mathit{answer}(\mathit{lose}, \mathit{Item}, \mathit{Quote}), \mathit{AuctionId}), \mathit{TLose}) \\
\rightarrow\ & \mathbf{EN}(\mathit{tell}(A, B, \mathit{answer}(\mathit{win}, \mathit{Item}, \mathit{Quote}), \mathit{AuctionId}), \mathit{TWin}) \wedge \mathit{TWin} > \mathit{TLose}
\end{aligned}
$$

In defining the versions of the protocols for combinatorial auctions, we focus on the communicative aspects of the auctions, assuming that the WDP is solved by a trusted, external entity .

6.2.8.1 Basic Combinatorial Auction

The protocol flow of the basic combinatorial auction is as follows:

1. the auctioneer agent opens the auction with an *openauction* message to a set of potential bidders, specifying the *Items* object of the auction, the time *TEnd* at which the auction will end, and the deadline *TDeadline* for notification to bidders;

2. by *TEnd*, the interested agents *bid* for a subset of *Items*;

3. at *TEnd*, the auctioneer closes the auction with the *closeauction* message;

4. each bid receives either a *win* or a *lose* notification.

The SICs for the basic combinatorial auction protocol are shown in Spec. 6.12.

The first SIC is a backward one which requires a correct *openauction* to have happened before each *bid*.

The second SIC prescribes that all incorrect bids (i.e., bids for items not present in the auction) should be notified of losing. The correctness of a bid is evaluated using the *included/2* predicate, defined in the Social Knowledge Base as in Spec. 6.13.

The third SIC requires for the auctioneer to close each opened auction with a *closeauction* message.

The fourth SIC requires for the auctioneer to reply to each bid with a notification of winning or losing. Differently from the case of the first price sealed bid auction (see Spec. 6.11), in this case the alternative is expressed by representing the answer with a variable (*Answer*) whose domain is the set *[win,lose]*, rather than with a disjunction.

The last two SICs express mutual exclusiveness between the *win* and *lose* answers.

Specification 6.12 SICs for the basic combinatorial auction protocol.

$$\mathbf{H}(\mathit{tell}(S, R, \mathit{bid}(\mathit{ItemList}, P), \mathit{Anumber}), \mathit{Tbid})$$
$$\rightarrow \mathbf{E}(\mathit{tell}(R, S, \mathit{openauction}(\mathit{Items}, \mathit{Tend}, \mathit{Tdeadline}), \mathit{Anumber}), \mathit{Topen}) \wedge$$
$$\mathit{Topen} < \mathit{Tbid} \wedge \mathit{Tbid} \leq \mathit{Tend}$$

$$\mathbf{H}(\mathit{tell}(A, B, \mathit{openauction}(\mathit{Items}, \mathit{Tend}, \mathit{Tdeadline}), \mathit{Anumber}), \mathit{Topen}) \wedge$$
$$\mathbf{H}(\mathit{tell}(B, A, \mathit{bid}(\mathit{Itembid}, P), \mathit{Anumber}), \mathit{Tbid}) \wedge$$
$$\mathit{not\ included}(\mathit{Itembid}, \mathit{Items})$$
$$\rightarrow \mathbf{E}(\mathit{tell}(A, B, \mathit{answer}(\mathit{lose}, \mathit{Bidder}, \mathit{Itembids}, P), \mathit{Anumber}), T)$$

$$\mathbf{H}(\mathit{tell}(A, B, \mathit{openauction}(\mathit{Items}, \mathit{Tend}, \mathit{Tdeadline}), \mathit{Anumber}), \mathit{Topen})$$
$$\rightarrow \mathbf{E}(\mathit{tell}(A, B, \mathit{closeauction}, \mathit{Anumber}), \mathit{Tend})$$

$$\mathbf{H}(\mathit{tell}(B, A, \mathit{bid}(\mathit{ItemList}, P), \mathit{Anumber}), \mathit{Tbid}) \wedge$$
$$\mathbf{H}(\mathit{tell}(A, B, \mathit{openauction}(\mathit{Items}, \mathit{Tend}, \mathit{Tdeadline}), \mathit{Anumber}), \mathit{Topen})$$
$$\rightarrow \mathbf{E}(\mathit{tell}(A, B, \mathit{answer}(\mathit{Answer}, B, \mathit{Itemlist}, P), \mathit{Anumber}), \mathit{Tanswer}) \wedge$$
$$\mathit{Tanswer} \geq \mathit{Tend} \wedge \mathit{Tanswer} \leq \mathit{Tdeadline} \wedge \mathit{Answer} :: [\mathit{win}, \mathit{lose}]$$

$$\mathbf{H}(\mathit{tell}(A, B, \mathit{answer}(\mathit{lose}, B, \mathit{Itemlist}, P), \mathit{Anumber}), \mathit{T1})$$
$$\rightarrow \mathbf{EN}(\mathit{tell}(A, B, \mathit{answer}(\mathit{win}, B, \mathit{Itemlist}, P), \mathit{Anumber}), \mathit{T2})$$

$$\mathbf{H}(\mathit{tell}(A, B, \mathit{answer}(\mathit{win}, B, \mathit{Itemlist}, P), \mathit{Anumber}), \mathit{T1})$$
$$\rightarrow \mathbf{EN}(\mathit{tell}(A, B, \mathit{answer}(\mathit{lose}, B, \mathit{Itemlist}, P), \mathit{Anumber}), \mathit{T2})$$

Specification 6.13 Social knowledge base for the basic combinatorial auction protocol.

$$
\begin{aligned}
&included([], _). \\
&included(.(H, T), L) \leftarrow member(H, L), \\
&\qquad included(T, L).
\end{aligned}
$$

6.2.8.2 Double combinatorial auction

This protocol extends the previous basic combinatorial auction protocol to support those cases where, for instance, a bidder opens an auction for buying items he needs for posting a more appealing bid. In this cases, conflicts between the two auctions may arise.

Specification 6.14 Additional SICs for the double combinatorial auction.

$$
\begin{aligned}
&\mathbf{H}(tell(A, B, openauction(Items1, Tend1, Tdeadline1), A1), Topen1) \land \\
&\mathbf{H}(tell(B, C, openauction(Items2, Tend2, Tdeadline2), A2), Topen2) \land \\
&intersect(Items1, Items2) \land Tdeadline2 \geq Tend1 \\
\rightarrow\ &false
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{H}(tell(B, A1, bid(ItemList1, P1), Anumber1), Tbid1) \land \\
&\mathbf{H}(tell(B, A2, bid(ItemList2, P2), Anumber2), Tbid2) \land \\
&Anumber1 \neq Anumber2 \land intersect(ItemList1, ItemList2) \\
\rightarrow\ &false
\end{aligned}
$$

The protocol still comprises the SICs shown in Spec. 6.12; two further SICs, shown in Spec. 6.14, are used to prevent conflicts between two auctions.

The first SIC that if a first auction has been opened and one of the potential bidders has opened another auctions referring to a set of items involved in the first auction, then the second should be closed and winning bids decided before the first closes.

The second SIC prevents a bidder from opening two distinct auctions for the

same items.

Specification 6.15 Social knowledge base for the double combinatorial auction protocol.

$$
\begin{aligned}
&\mathit{included}([], _). \\
\mathit{included}(.(H, T), L) &\leftarrow \mathit{member}(H, L), \\
&\quad\ \mathit{included}(T, L). \\
\\
\mathit{intersect}(A, B) &\leftarrow \mathit{member}(X, A), \\
&\quad\ \mathit{member}(X, B).
\end{aligned}
$$

In both SICs, the presence of the same items in two auctions is checked by means of the *intersect/2* predicate defined in the social knowledge base as in Spec. 6.15.

6.2.8.3 Combinatorial auction with NetBill

This protocol extends the combinatorial auction protocol with a delivery and payment phase according to the NetBill protocol, already described in Sect. 6.2.5 (with minor differences in the format of messages). Of course, this is only applicable to auctions regarding information goods (due to the necessity to deliver the items in encrypted form).

The delivery and payment phase is specified by the additional SICs shown in Specs. 6.16 (backward SICs) and 6.17 (forward SICs).

The delivery and payment phase is started, for winning bids, by the first forward SIC in Spec. 6.17, which requires that a winning bidder deliver the goods in encrypted form (since we are considering a *reverse* auction, the bidder is the seller). After the delivery, the protocol flow proceeds as described in Sect. 6.2.5. Since neither the auctioneer nor the bidder can now withdraw from the transaction, each pair of consecutive protocol steps is linked by a backward SIC (in Spec. 6.16) and a forward one (in Spec. 6.17).

Specification 6.16 Additional backward SICs for the combinatorial auction protocol with NetBill delivery phase.

$$\mathbf{H}(\mathit{tell}(B, A, \mathit{deliver}(\mathit{ItemList}, \mathit{Price}), \mathit{Anumber}), \mathit{T3})$$
$$\rightarrow \mathbf{E}(\mathit{tell}(A, B, \mathit{answer}(\mathit{win}, B, \mathit{ItemList}, P), \mathit{Anumber}), \mathit{T1}) \ \wedge \ \mathit{T1} \ < \ \mathit{T3}$$

$$\mathbf{H}(\mathit{tell}(A, B, \mathit{payment}(\mathit{ItemList}, \mathit{Price}, \mathit{EPOSign}), \mathit{Anumber}), \mathit{T4})$$
$$\rightarrow \mathbf{E}(\mathit{tell}(B, A, \mathit{deliver}(\mathit{ItemList}, \mathit{Price}), \mathit{Anumber}), \mathit{T3}) \ \wedge \ \mathit{T3} \ < \ \mathit{T4}$$

$$\mathbf{H}(\mathit{tell}(B, \mathit{netbill}, \mathit{endorsedEpo}(\mathit{ItemList}, \mathit{Price}, A, \mathit{EPOSign}, \mathit{Key}), \mathit{Anumber}), \mathit{T5})$$
$$\rightarrow \mathbf{E}(\mathit{tell}(A, B, \mathit{payment}(\mathit{ItemList}, \mathit{Price}, \mathit{EPOSign}), \mathit{Anumber}), \mathit{T4}) \ \wedge \ \mathit{T4} \ < \ \mathit{T5}$$

$$\mathbf{H}(\mathit{tell}(\mathit{netbill}, B, \mathit{signedResult}(\mathit{ItemList}, \mathit{Price}, A, \mathit{Result}, \mathit{Key}), \mathit{Anumber}), \mathit{T6})$$
$$\rightarrow \mathbf{E}(\mathit{tell}(B, \mathit{netbill}, \mathit{endorsedEpo}(\mathit{ItemList}, \mathit{Price}, A, \mathit{EPOSign}, \mathit{Key}), \mathit{Anumber}), \mathit{T5}) \ \wedge$$
$$\mathit{T5} \ < \ \mathit{T6}$$

$$\mathbf{H}(\mathit{tell}(B, A, \mathit{signedResult}(\mathit{ItemList}, \mathit{Price}, \mathit{Result}, \mathit{Key}), \mathit{Anumber}), \mathit{T7})$$
$$\rightarrow \mathbf{E}(\mathit{tell}(\mathit{netbill}, B, \mathit{signedResult}(\mathit{ItemList}, \mathit{Price}, A, \mathit{Result}, \mathit{Key}), \mathit{Anumber}), \mathit{T6}) \ \wedge$$
$$\mathit{T6} \ < \ \mathit{T7}$$

Specification 6.17 Additional forward SICs for the combinatorial auction protocol with NetBill delivery phase.

$$\mathbf{H}(tell(B, A, bid(ItemList, P), Anumber), Tbid) \wedge$$
$$\mathbf{H}(tell(A, B, answer(win, B, ItemList, P), Anumber), T1) \wedge Tbid < T1$$
$$\rightarrow \mathbf{E}(tell(B, A, deliver(ItemList, P), Anumber), T3) \wedge T3 > T1$$

$$\mathbf{H}(tell(B, A, deliver(ItemList, Price), Anumber), T3)$$
$$\rightarrow \mathbf{E}(tell(A, B, payment(ItemList, Price, EPOSign), Anumber), T4) \wedge T4 > T3$$

$$\mathbf{H}(tell(A, B, payment(ItemList, Price, EPOSign), Anumber), T4)$$
$$\rightarrow \mathbf{E}(tell(B, netbill, endorsedEpo(ItemList, Price, A, EPOSign, Key), Anumber), T5) \wedge$$
$$T5 > T4$$

$$\mathbf{H}(tell(B, netbill, endorsedEpo(ItemList, Price, A, EPOSign, Key), Anumber), T5)$$
$$\rightarrow \mathbf{E}(tell(netbill, B, signedResult(ItemList, Price, A, Result, Key), Anumber), T6) \wedge$$
$$T6 > T5$$

$$\mathbf{H}(tell(netbill, B, signedResult(ItemList, Price, A, Result, Key), Anumber), T6)$$
$$\rightarrow \mathbf{E}(tell(B, A, signedResult(ItemList, Price, Result, Key), Anumber), T7) \wedge T7 > T6$$

Chapter 7

$\mathcal{S}$CIFF performance

No formal result have been proved about the computational complexity of $\mathcal{S}$CIFF. However, in this chapter, we first make some qualitative considerations on the factors that contribute to the complexity, and then show the results of practical tests.

7.1 Considerations on the $\mathcal{S}$CIFF computational complexity

The $\mathcal{S}$CIFF proof procedure was designed to be used for *on-the-fly* compliance check, while the interaction is taking place, as well as for *a-posteriori* check of the interactions. The computation time becomes a critical issue in the case of *on-the-fly* verification. In the latter case, SOCS-SI should behave like a *real-time* tool, that acts as quickly as possible whenever the interaction between computees evolves.

Memory requirements are also important in order to determine the (maximum) dimension of societies (in terms of participants and/or interactions) that the implementation of $\mathcal{S}$CIFF can support.

Each $\mathcal{S}$CIFF computation produces a search tree whose *depth* and *breadth* determine the total number of nodes, and thus the time needed to explore the (whole) tree. As the proof tree is explored by $\mathcal{S}$CIFF in a depth-first fashion, the depth of the tree, together with the *size* of a single node, also impacts on space

requirements. For both time and space, the worst case is when each branch leads to violation, because in this case the whole tree is explored in search of a success node.

In the following, we identify the features of social specifications that have greater influence on the three factors above: namely, *depth*, *breadth* of the search tree, and *size* of nodes.

Intuitively, the *depth* of the search tree depends on the total *number of messages exchanged* within a society. This parameter can be varied, and incremented in particular, by (*i*) having the same computees repeat more times the same interaction, or (*ii*) increasing the number of computees participating a society, or (*iii*) having "longer" interactions, where each compliant run is made up of different messages, increasing in number.

The *breadth* of the search tree, instead, is influenced by both the number of disjuncts in the head of social integrity constraints, and the alternative branches arising in several of the $\mathcal{S}$CIFF transitions (see Sect. 3.3).

For instance, in the *Fulfillment* transition, the fulfillment of an **E** expectation leaves a choice open for non-unification between the expectation and the event that would fulfill it. In many cases, avoiding this branching does not change the behaviour of $\mathcal{S}$CIFF with respect to fulfillment and violation : for instance, when a positive expectation can be fulfilled by only one event, the non-unification branch cannot be one of success. In such cases, cutting the non-unification branch is safe, and saves a considerable amount of computation space and time. We call such a $\mathcal{S}$CIFF behaviour *f-deterministic* meaning that it behaves deterministically on *Fulfillment*.

We let the user decide for which expectations $\mathcal{S}$CIFF should adopt the *f-deterministic* behaviour (or, in other words, which expectations will be *f-deterministic*), by means of one or more `fdet/1` directives, whose argument is a term T: all expectations that are instances of T will be as *f-deterministic*. Of course, it is possible to make all expectations *f-deterministic* by a directive such as `fdet(e(X,T))`. For simplicity, we will refer to $\mathcal{S}$CIFF with this directive as the *f-deterministic* version of $\mathcal{S}$CIFF; the *f-non-deterministic* version of $\mathcal{S}$CIFF will denote $\mathcal{S}$CIFF with no `fdet/1` directive.

7.2 Experimental results

In this section, we show some experimental results obtained applying $\mathcal{S}$CIFF to the verification of compliance to the combinatorial auction protocols described in Sect. 6.2.8. While not being an exhaustive experimentation, the results show the effect on the time complexity of $\mathcal{S}$CIFF of the breadth and depth of the search tree.

7.2.1 The effect of the branching factor

The branching factor of the proof tree obviously impacts heavily on the computational complexity of $\mathcal{S}$CIFF. In order to show its effect, we have performed some experiments on the combinatorial auction scenario (Sect. 6.2.8.1) varying two parameters which contribute to determine the breadth of the proof tree:

1. $\mathcal{S}$CIFF version (*f-non-deterministic* vs. *f-deterministic*, as defined in Sect. 7.1);
2. social specification (one version with a disjunction in the head of a SIC vs. one version with no disjunctions, with alternative expressed by means of variables with domain).

In particular, we measure the computation time for sequences of auctions with different numbers of bidders in the two following implementations of the protocol:

1. *f-non-deterministic* $\mathcal{S}$CIFF, protocol with disjunction (which we call the *first setup* of $\mathcal{S}$CIFF and protocol);
2. *f-deterministic* $\mathcal{S}$CIFF, protocol with no disjunction (which we call the *second setup* of $\mathcal{S}$CIFF and protocol).

In both cases, the goal is $true$, and the $SOKB$ is that reported in Spec. 6.13.

The SICs are those reported in Spec. 6.12, apart from the fourth one which, in the first setup, is replaced by the one in Spec. 7.1.

The protocols have been run by varying the number N of bidders, in two different cases.

Specification 7.1 Replacement for the fourth SIC in Spec. 6.12.

$$
\begin{aligned}
&\mathbf{H}(tell(B, A, bid(ItemList, P), Anumber), Tbid) \wedge \\
&\mathbf{H}(tell(A, B, openauction(Items, Tend, Tdeadline), Anumber), Topen) \\
\rightarrow\ &\mathbf{E}(tell(A, B, answer(win, B, Itemlist, P), Anumber), Tanswer) \wedge \\
&Tanswer > Tend \wedge Tanswer < Tdeadline \\
\vee\ &\mathbf{E}(tell(A, B, answer(lose, B, Itemlist, P), Anumber), Tanswer) \wedge \\
&Tanswer > Tend \wedge Tanswer < Tdeadline
\end{aligned}
$$

- In each run of the first case:

  1. the auctioneer sends an *openauction* message to each of the N bidders;
  2. each of the N bidders places a *bid*;
  3. the auctioneer issues a *closeauction* message to each of the N bidders;
  4. the auctioneer notifies each of the N bidders with either a *win* or a *lose* message,

  thus resulting in $4N$ total messages exchanged.

- In each run of the second case, the last notification to one of the bidders is missing, thus resulting in a violation of the protocol and $4N - 1$ total messages.

The experiments were run on a PC with a 2 GHz Pentium IV CPU, 512 MB of RAM, Linux 2.4.18, glibc 2.2.5 and SICStus Prolog 3.10.1. Reported times are in seconds.

In case of fulfillment (see Table 7.1), the first setup of $\mathcal{S}$CIFF and protocol seems to scale well with the number of bidders and, in fact, it achieves better execution timing than the second. This is basically due to the fact that the chosen setup of interactions directly leads to a successful $\mathcal{S}$CIFF derivation, and only one branch of the tree is explored.

In the case of violation (see Table 7.2), however, the first setup of $\mathcal{S}$CIFF and protocol explodes for a very small number of bidders. The experiment with

f-non-deterministic,disjunction		*f-deterministic,domain*	
Bidders	Time(sec.)	Bidders	Time(sec.)
5	1	5	1
10	1	10	1
15	2	15	2
20	3	20	6
25	4	25	8
30	6	30	10
35	9	35	15
40	10	40	18
45	12	45	23
50	21	50	30

Table 7.1: Combinatorial Auction case 1: Fulfillment

f-non-deterministic,disjunction		*f-deterministic,domain*	
Bidders	Time(sec.)	Bidders	Time(sec.)
3	7	3	0
4	55	4	0
5	?	5	0
10	?	10	1
15	?	15	3
20	?	20	4
25	?	25	7
30	?	30	10
35	?	35	14
40	?	40	17
45	?	45	22
50	?	50	26

Table 7.2: Combinatorial Auction case 2: Violation

5 bidders was suspended since this did not reach the answer of violation after several minutes of computing time; no experiments were performed with a higher number of computees, which would have made things even worse. The second setup, instead, scales very well also in case of violation. In this case, a CLP(FD) solver, written in CHR, directly manages the two alternative values for variable `Answer`.

The difference between the two setups of $\mathcal{S}$CIFF and protocol becomes apparent in the worst case (i.e., the case of violation) when the whole tree is explored. With the first setup, the choice points left open in case of fulfillment and the disjunctions in the head of the integrity constraint make the number of nodes in the proof tree explode even for small number of bidders. With the second setup, instead, the tree has only one branch, and is thus explored in a reasonable time when the number of bidders increases.

7.2.2 The effect of the number of events

In this section, we report on the experimental results on compliance checking for the three versions of the combinatorial auction scenario described in Sects. 6.2.8.1, 6.2.8.2, and 6.2.8.3.

The aim of these experiments is to evaluate the scalability of $\mathcal{S}$CIFF with respect to the number of exchanged events in an agent society.

For each protocol, we have performed two series of protocol runs, varying the number of bidders: one compliant, one violating.

In all three cases, we have used the *f-deterministic* version of $\mathcal{S}$CIFF.

The setting for the basic auction scenario is the same described in Sect. 7.2.1: i.e., if N is the number of bidders, $4N$ is the number of messages exchanged in the compliant run, and $4N - 1$ is the number of messages exchanged in the non compliant run.

The results for the compliant and non compliant runs are shown in Figs. 7.1 and 7.2, respectively.

In the experiments with the double auction protocols, one of the N bidders of the original auction opens a second auction, to which N bidders participate. In this way, the total number of messages is $8N$ for the compliant run, and $8N - 1$

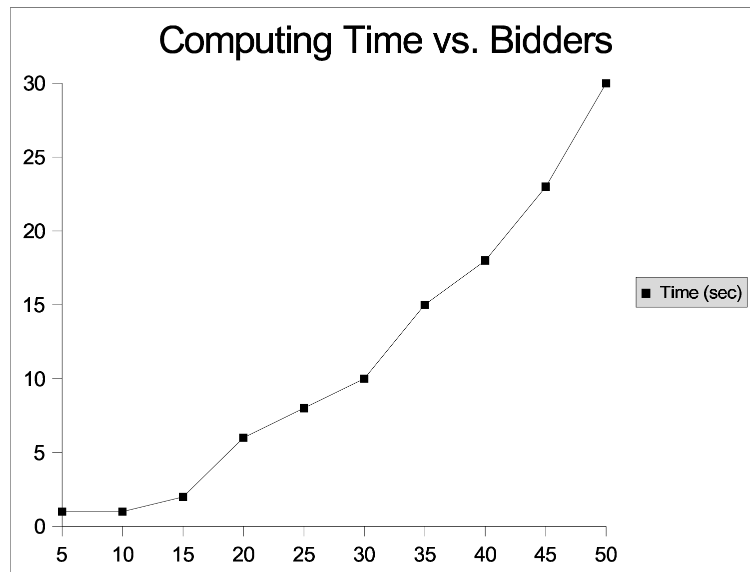


Figure 7.1: Proof performance on a basic auction (compliant)

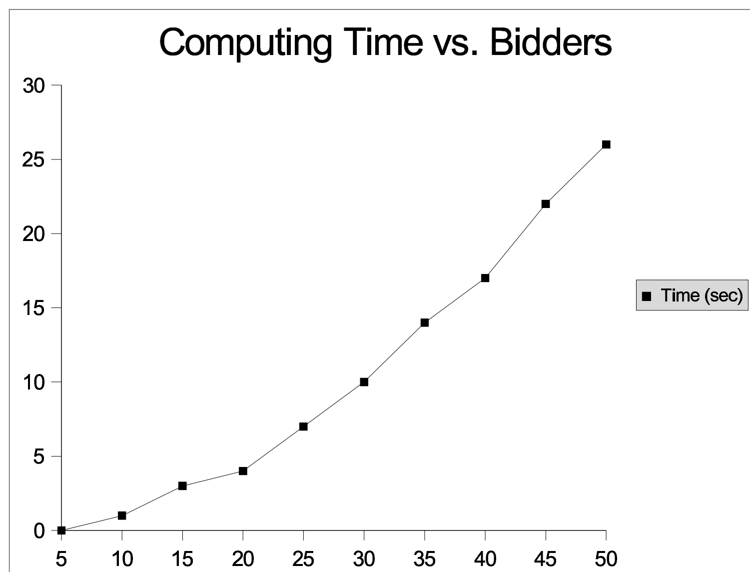


Figure 7.2: Proof performance on a basic auction (non compliant)

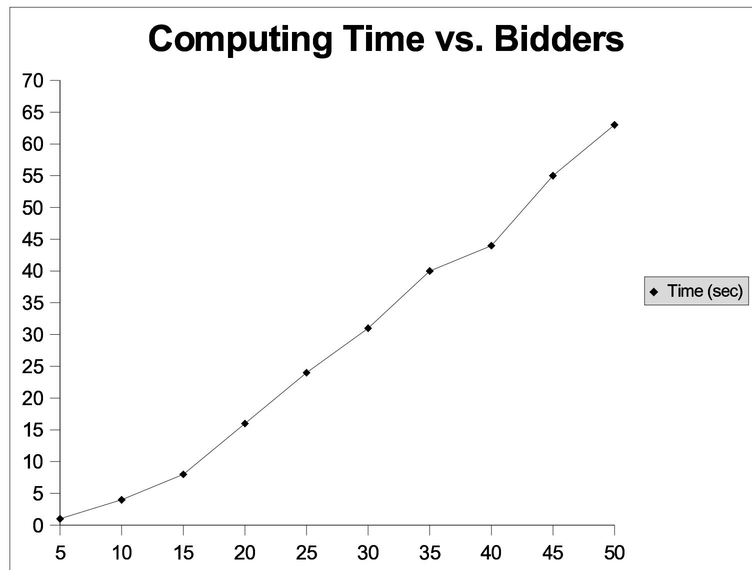


Figure 7.3: Proof performance on a double auction (compliant)

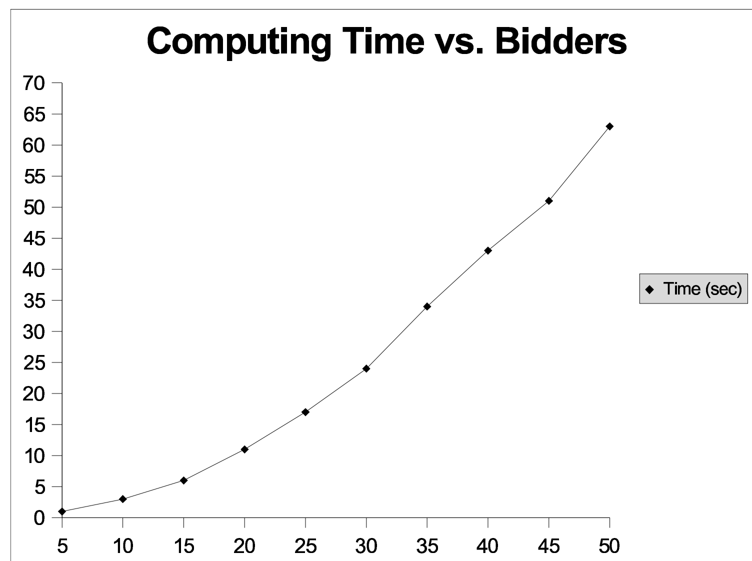


Figure 7.4: Proof performance on a double auction (non compliant)

for the non compliant run.

The results are shown in Figs. 7.3 and 7.4, respectively. To a roughly doubled number of events with respect to the basic auction case, corresponds a roughly doubled computing time.

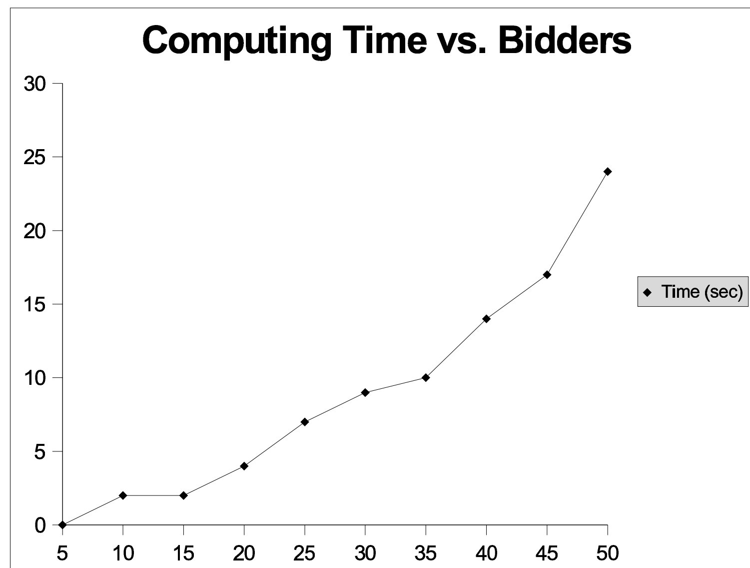


Figure 7.5: Proof performance on an auction plus NetBill (compliant)

In the experiments with the combinatorial auction with NetBill delivery and payment, only one of the N bidders is the winner, and thus the total number of messages exchanged is $4N + 5$ in the compliant case, and $4N + 4$ in the non-compliant case.

The results for the compliant and the non-compliant cases are shown in Figs. 7.5 and 7.6, respectively. As is expected, the results are very similar to those of the basic combinatorial auction, as the number of events is almost the same.

The overall evaluation of these experiments suggests that if it is possible to keep the branching of the proof tree limited, the performance of $\mathcal{S}$CIFF scales reasonably well with the number of events in the agent society, making $\mathcal{S}$CIFF applicable to practical cases.

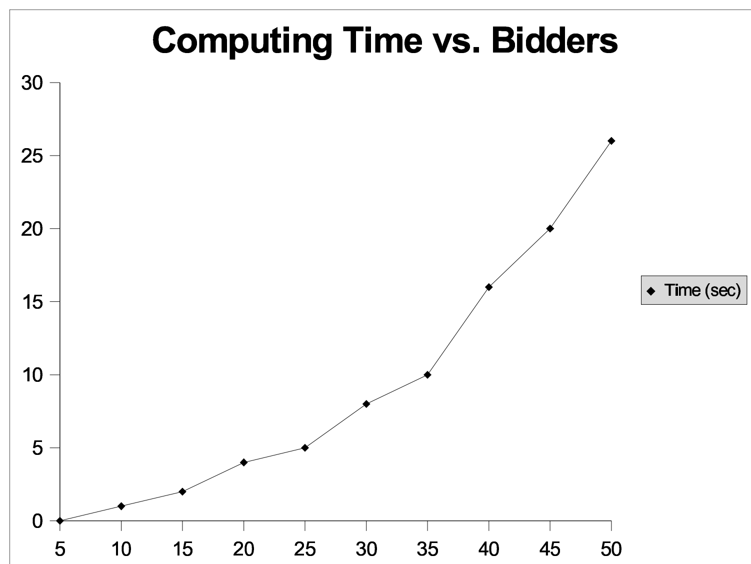


Figure 7.6: Proof performance on an auction plus NetBill (compliant)

Chapter 8

Extending $\mathcal{S}$CIFF for automatic proof of properties

In this chapter, we describe an extension of the $\mathcal{S}$CIFF proof procedure, called g-$\mathcal{S}$CIFF, which is able to generate histories that are compliant to a protocol. We describe both the definition and the implementation of the extension.

The intended use of g-$\mathcal{S}$CIFF is the automatic proof of protocol properties, as will be shown in Ch. 9.

8.1 The g-$\mathcal{S}$CIFF proof procedure

The $\mathcal{S}$CIFF proof procedure has been designed to take the history of the social interaction as an input, to check if it is compliant to a given social specification.

One further step is to try to generate a compliant history, rather than just checking if a given one is compliant. For instance, a protocol designer might want to be sure that the protocol that he or she is designing actually has a compliant history (or, in other words, is well-defined, see Def. 2.3.7); or finding out a compliant history with some undesirable feature might lead the designer to reconsider the definition of the protocol. More in general, an ability to generate compliant history can be used to verify protocol properties, as will be shown in Ch. 9.

It is apparent that most protocols, even very simple ones, can have an infinite number of compliant histories. One can restrict himself to only finite universe

situations: for example, one where the number of involved agents is finite, the set of possible utterances is given, and the language is function-free. In such cases, the number of compliant histories is finite, if big, and one can consider generating all the compliant histories, possibly pruning the search space by means of some efficient technique.

Another way (which we follow) is to let the events in the history contain variables, that can possibly concisely express a number of different instantiations. Thus, the generated atoms will have the same syntax as the happened events of the $\mathcal{S}$CIFF framework (see Sect. 2.2.1.1), but will not be required to be ground. Variables in **H** atoms will be considered existentially quantified, as we search for at least one compliant history violating the property.

Since **H** literals will be generated, it is quite natural to map them onto abducible literals. In fact, from a declarative viewpoint, such a choice is rather harmless, as in all the formulas that define the declarative semantics, (in particular, Def. 2.3.5 and 2.3.6), the history (**HAP**) occurs on the left hand side of the entailment symbol, just like the sets of the abducible atoms (**EXP**).

Operationally, it turns out that a proof procedure able to generate compliant histories can be obtained by means of a simple modification to the $\mathcal{S}$CIFF proof procedure. We call the new proof procedure g-$\mathcal{S}$CIFF (*generative* $\mathcal{S}$CIFF).

In the operational semantics, the transition *Happening* (which inserts a new event in the current history, see Sec. 3.3) is no longer needed (as this is not on-the-fly verification), and it is replaced by the following transition.

Fulfiller. Given a node N_k in which

- $\mathbf{PEND}_k = \mathbf{PEND}' \cup \{\mathbf{E}(E,T)\}$

and transitions of fulfillment are not applicable, transition *Fulfiller* is applicable and generates a node N_{k+1} identical to N_k except:

- $\mathbf{PEND}_{k+1} = \mathbf{PEND}'$
- $\mathbf{FULF}_{k+1} = \mathbf{FULF}_k \cup \{\mathbf{E}(E,T)\}$
- $\mathbf{HAP}_{k+1} = \mathbf{HAP}_k \cup \{\mathbf{H}(E,T)\}$

i.e., a new event is inserted in the history fulfilling the expectation.

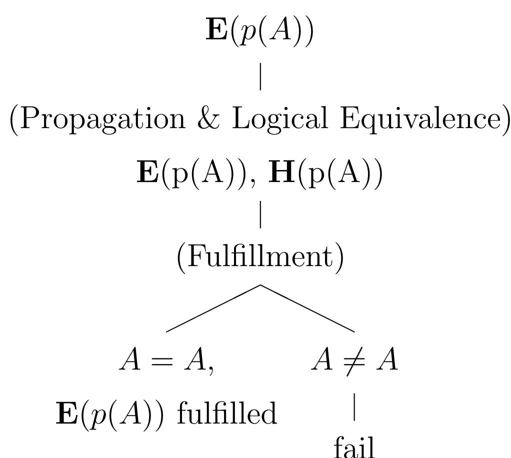
We will use the symbol $\overset{g}{\vdash}$ to indicate a derivation in g-$\mathcal{S}$CIFF, as opposed to one in $\mathcal{S}$CIFF. The generative version, as the non generative one, takes as input an initial history $\mathbf{HAP}^i$ (that will be typically empty) and provides a possibly extended history $\mathbf{HAP}^f$:

$$\mathcal{S}_{\mathbf{HAP}^i} \overset{g}{\vdash}{}^{\mathbf{HAP}^f}_{\mathbf{EXP}} G$$

Note that we would have obtained a semantically equivalent result by imposing the integrity constraint

$$\mathbf{E}(X,T) \rightarrow \mathbf{H}(X,T) \tag{8.1.1}$$

In fact, the derivation tree for a single abduced atom $\mathbf{E}(p(A))$, would have been the following:



Despite their semantical equivalence, transition *Fulfiller* is more efficient, as it avoids generating the right failure branch.

8.1.1 Formal results

The formal properties of g-$\mathcal{S}$CIFF are a consequence of those of $\mathcal{S}$CIFF.

Theorem 8.1.1 (Soundness of g-$\mathcal{S}$CIFF). *Given a society instance $\mathcal{S}_{\mathbf{HAP}^f}$, if*

$$\mathcal{S}_{\mathbf{HAP}^i} \overset{g}{\vdash}{}^{\mathbf{HAP}^f}_{\mathbf{EXP}} G$$

with expectation answer $(\mathbf{EXP}, \sigma)$*, then*

$$\mathcal{S}_{\mathbf{HAP}^f} \models_{\mathbf{EXP}\sigma} G\sigma$$

The proof can be found in [ACG+05a].

Concerning termination, g-$\mathcal{S}$CIFF guarantees termination if the society *together* with the SIC (8.1.1) is acyclic. However, it turns out that some of the protocols that can be expressed through an acyclic social specification may become cyclic when adding rule (8.1.1), and consequently the proof of termination is no longer valid. In such a case, we can rely on the idea of iterative deepening, as many other approaches do (one of them is bounded model checking). For instance, in the context of verification of a security protocol against possible attacks, if we implement iterative deepening we search for attacks of increasing length. We first focus only on attacks of length 1, and if none exists, we look for attacks of length 2, and so on. This method is, of course, unable to prove that a protocol is secure, but it can prove that no attack exists up to any given length n (provided that we have enough time). However, other protocols will retain the acyclicity even with respect to the society knowledge base of (8.1.1), so in such cases we have a greater expressive power than bounded methods.

8.2 g-$\mathcal{S}$CIFF implementation

The implementation of g-$\mathcal{S}$CIFF is very simply obtained from that of $\mathcal{S}$CIFF, by adding the following *CHR* simpagation rule:

```
fulfiller @
    (close_history)
    \
    (pending(e(Event,Time)))
    <=>
    fulf(e(Event,Time)),
    h(Event,Time).
```

Operationally, the rule will be fired after closure, and will generate an event for each pending expectation.

However, this implementation conflicts with the implementation of **not H** by means of constructive negation (see Sect. 5.5.2) . For this reason, the implementation of g-$\mathcal{S}$CIFF here described can only be applied to social specification

where **not H** literals do not occur. Based on the case studies in Ch. 6, this does not appear to be a strict limitation, as **not H** literals do not appear in any of them.

Chapter 9

Automatically proving properties with g-$\mathcal{S}$CIFF

In this chapter, we show how the g-$\mathcal{S}$CIFF extension, described in Ch. 8, of the $\mathcal{S}$CIFF proof procedure can be applied to *Type 3* verification (see Sect. 1.1.2), i.e., verification of protocol properties. We first introduce our approach, and then present two case studies.

9.1 The g-$\mathcal{S}$CIFF approach

We aim at verifying protocol properties that can be expressed by formulae and, in particular:

- *existential* properties, i.e, formulae that hold for at least one history compliant to the protocol;
- *universal* properties, i.e., formulae that hold for all the histories compliant to a protocol.

Formally, our definition of protocol properties is as follows.

In the following definitions, let a protocol $\mathcal{S}$ be defined by KB_S and $\mathcal{IC}_S$, and let $\mathcal{S}$ be well defined (see Def. 2.3.7) with respect to the goal true.

Definition 9.1.1 *A formula f is an existential property of $\mathcal{S}$ iff:*

$$\exists_{\mathbf{HAP}} \exists_{\mathbf{EXP}} \mathcal{S}_{\mathbf{HAP}} \models_{\mathbf{EXP}} f \tag{9.1.1}$$

Definition 9.1.2 *A formula f is a universal property of the protocol $\mathcal{S}$ iff:*

$$\forall \mathbf{HAP} \forall \mathbf{EXP} (\mathcal{S}_{\mathbf{HAP}} \models_{\mathbf{EXP}} true \rightarrow \mathcal{S}_{\mathbf{HAP}} \models_{\mathbf{EXP}} f) \tag{9.1.2}$$

In the definitions above $\mathcal{S}_{\mathbf{HAP}} \models_{\mathbf{EXP}} f$ has the meaning explained in Def. 2.3.6 (goal achievement).

The g-$\mathcal{S}$CIFF proof procedure can be used for verifying both existential and universal properties, as follows.

- An existential property f can be verified by:
  1. expressing f as a $\mathcal{S}$CIFF goal, and
  2. running g-$\mathcal{S}$CIFF. Two cases are possible:
     - g-$\mathcal{S}$CIFF returns failure: f is not an existential property of protocol $\mathcal{P}$;
     - g-$\mathcal{S}$CIFF returns success, with a history $\mathbf{HAP}$: f is an existential property of $\mathcal{P}$, and $\mathbf{HAP}$ is an example instantiation of a history that satisfies f.
- A universal property f can be verified by:
  1. expressing $\neg f$ as a $\mathcal{S}$CIFF goal, and
  2. running g-$\mathcal{S}$CIFF. Two cases are possible:
     - g-$\mathcal{S}$CIFF returns failure: f is a universal property of protocol $\mathcal{P}$;
     - g-$\mathcal{S}$CIFF returns success, with a history $\mathbf{HAP}$: f is not a universal property of $\mathcal{P}$, and $\mathbf{HAP}$ is an example history for which f does not hold.

At the time of writing, our technique can be applied subject to the following restrictions:

- The only properties that we can verify are
  - existential properties that can be expressed as a $\mathcal{S}$CIFF goal;
  - universal properties whose negation can be expressed as a $\mathcal{S}$CIFF goal.

- Soundness of g-$\mathcal{S}$CIFF has been proven, but completeness has not. Thus, our approach is provably effective for proving existential properties and refuting universal properties, but not yet for proving universal properties and refuting existential properties.

9.2 Case studies

In this section, we exemplify the use of g-$\mathcal{S}$CIFF for the verification of protocol properties, focusing on two well known interaction protocols: the NetBill transaction protocol and the Needham-Schroeder security protocol, defined in Sect. 6.2.6.

9.2.1 Verifying the NetBill protocol

In this section, we show how a simple property of the NetBill protocol can be expressed, and verified, with $\mathcal{S}$CIFF.

We want to verify the following property: *the merchant receives the payment for a good G if and only if the customer receives the good G*, as long as the protocol is respected.

Since the $\mathcal{S}$CIFF deals with (communicative) events and not with the states of the agents, we need to express the properties in terms of happened events. To this purpose, we can assume that merchant has received the payment once the NetBill server has issued the *signedResult* message, and that the the customer has received the good if she has received the encrypted good (with a *deliver* message) and the encryption key (with a *receipt* message).

Thus, the property we want to verify can be expressed as

$$\begin{aligned} & H(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign) \\ \Longleftrightarrow\ & H(tell(M, C, goodDelivery(crypt(PrId, K), Quote), Id), T) \\ & \wedge H(tell(M, C, receipt(PrId, Quote, K), Id), Ts) \end{aligned} \tag{9.2.1}$$

whose negation is

$$
\begin{aligned}
&(\neg H(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign) \\
&\wedge H(tell(M, C, goodDelivery(crypt(PrId, K), Quote), Id), T) \\
&\wedge H(tell(M, C, receipt(PrId, Quote, K), Id), Ts)) \\
&\vee \\
&(H(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign) \\
&\wedge \neg H(tell(M, C, goodDelivery(crypt(PrId, K), Quote), Id), T) \\
&\vee \\
&(H(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign) \\
&\wedge \neg H(tell(M, C, goodDelivery(crypt(PrId, K), Quote), Id), T))
\end{aligned}
\qquad (9.2.2)
$$

In other words, an history that entails Eq. (9.2.2) is a counterexample of the property that we want to prove. In order to search for such a history, we define the g-$\mathcal{S}$CIFF goal as follows:

$$
\begin{aligned}
g \leftarrow & EN(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign), \\
& E(tell(M, C, goodDelivery(crypt(PrId, K), Quote), Id), T), \\
& E(tell(M, C, receipt(PrId, Quote, K), Id), Ts)). \\
g \leftarrow & E(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign), \\
& EN(tell(M, C, goodDelivery(crypt(PrId, K), Quote), Id), T). \\
g \leftarrow & E(tell(netbill, M, signedResult(C, PrId, Quote, K), Id), Tsign), \\
& EN(tell(M, C, goodDelivery(crypt(PrId, K), Quote), Id), T))
\end{aligned}
\qquad (9.2.3)
$$

and run g-$\mathcal{S}$CIFF, and the integrity constraints that define the NetBill protocol.

The result of the call is a failure. This suggests that there is no history that entails the negation of the property while respecting the protocol, i.e., the property is likely to hold if the protocol is respected. However, since g-$\mathcal{S}$CIFF has not been proven complete, the failure does not count as a proof of the property.

If we remove the second forward integrity constraints of Spec. 6.5 (which imposes that a `signedResult` message be followed by a `receipt` message), then the following history is generated:

```
h(tell(_E,_F,priceRequest(_D),_C),_M),
h(tell(_F,_E,priceQuote(_D,_B),_C),_L),
h(tell(_E,_F,goodRequest(_D,_B),_C),_K),
h(tell(_F,_E,goodDelivery(crypt(_D,_A),_B),_C),_J),
h(tell(_E,_F,payment(_E,crypt(_D,_A),_B),_C),_I),
h(tell(_F,netbill,endorsedEPO(epo(_E,_D,_B),_A,_F),_C),_H),
h(tell(netbill,_F,signedResult(_E,_D,_B,_A),_C),_G),
_I<_H, _H<_G,
_L>_M, _K>_L, _I>_J, _J>_K,
```

The *receipt* event is missing, which would violate the integrity constraint that has been removed. The generated history is compliant to the protocol while negating the property and, thanks to the soundness of g-$\mathcal{S}$CIFF, is a valid counterexample of the property.

In this way, a protocol designer can make sure that an integrity constraint is not redundant with respect to a desired property of the protocol.

9.2.2 Verifying the Needham-Schroeder Protocol

In the idea of the Needham-Schroeder protocol, an agent trusts the identity of the agent with whom he is communicating by associating his name with his public key and receiving back a nonce that he forged, encrypted in his own public key. If we had to define the idea of an agent B 'trusting' that he is communicating with A, we could do it by using a combination of messages in which an agents responds to a challenge posed by another agent and successfully decrypts a nonce.

Definition 9.2.1 *We say that B trusts that the agent X he is communicating with is A,*[1] *and we write $trust_B(X, A)$ once two messages have been exchanged at times T_1 and T_2, $T_1 < T_2$, having the following sender, recipient, and content:*

$(T_1)\ B \rightarrow X : \{N_B, \dots\}_{pub_key(A)}$

$(T_2)\ X \rightarrow B : \{N_B, \dots\}_{pub_key(B)}$

[1] We restrict ourselves to only one communication session, all the definitions will therefore have as a scope the session.

where N_B is a nonce generated by B.

Note that B is unable to judge whether N_A is a nonce actually generated by X or not, therefore no condition is posed on the origin of such nonce.

Symmetrically, we can consider, from A's viewpoint, messages (1) and (2) as those that prove the identity of B. We therefore implement Def. 9.2.1 in Def. 9.2.2, where messages are expressed using the notation of the $\mathcal{S}$CIFF, namely as events which are part of some "history" **HAP**. The content of messages will be composed of three parts, the first showing the public key used to encrypt it, the second and third containing agent names or nonces or nothing (in particular, the last part may be empty).

Definition 9.2.2 *Let A, B and X be agents, K_A and K_B respectively A's and B's public key, N_B a nonce produced by B, and let $\mathbf{HAP}_1$ and $\mathbf{HAP}_2$ be two sets of events each composed of two elements, namely:*
$\mathbf{HAP}_1 = \{$
$\mathbf{H}(send(B, X, content(key(K_A), agent(B), nonce(N_B))), T_1),$
$\mathbf{H}(send(X, B, content(key(K_B), nonce(N_B), nonce(...))), T_2)$
$\}$, *and*
$\mathbf{HAP}_2 = \{$
$\mathbf{H}(send(B, X, content(key(K_A), nonce(...), nonce(N_B))), T_1),$
$\mathbf{H}(send(X, B, content(key(K_B), nonce(N_B), empty(0))), T_2)$
$\}$.

Then, $trust_B(X, A)$ holds if and only if $\mathbf{HAP}_1 \subseteq \mathbf{HAP}$ or $\mathbf{HAP}_2 \subseteq \mathbf{HAP}$.

The property that we want to disprove is $\mathcal{P}_{trust}$ defined as $trust_B(X, A) \rightarrow X = A$, i.e., if B trusts that he is communicating with A, then he is indeed communicating with A. We obtain a problem which is symmetric in the variables A, B, and X. In order to check if we have a solution we can ground $\mathcal{P}_{trust}$ and define its negation $\neg\mathcal{P}_{trust}$ as a goal, $g3$, where we choose to assign to A, B, and X the values a, b and i:

$g3 \leftarrow isNonce(NA), NA \neq nb,$
$\mathbf{E}(send(b, i, content(key(ka), nonce(NA), nonce(nb))), 3),$
$\mathbf{E}(send(i, b, content(key(kb), nonce(nb), empty(0))), 6).$

Besides defining $g3$ for three specific agents, we also assign definite time points (3 and 6) in order to improve the efficiency of the proof.

Running the g-$\mathcal{S}$CIFF on $g3$ results in a compliant history:

```
h(send(a,i,content(key(ki),agent(a),nonce(na))),1),
h(send(i,b,content(key(kb),agent(a),nonce(na))),2),
h(send(b,i,content(key(ka),nonce(na),nonce(nb))),3),
h(send(i,a,content(key(ka),nonce(na),nonce(nb))),4),
h(send(a,i,content(key(ki),nonce(nb),empty(0))),5),
h(send(i,b,content(key(kb),nonce(nb),empty(0))),6).
```

which is indeed Lowe's attack on the protocol. $\mathbf{HAP}_{gL}$ represents a counterexample of the property $\mathcal{P}_{trust}$ while being compliant to the protocol; which, thanks to the soundness of g-$\mathcal{S}$CIFF, proves that $\mathcal{P}_{trust}$ is not a property of the protocol.

9.3 Related work.

In recent years the provability of properties for communication protocols has received a lot of attention; this holds even more for security protocols. Various techniques have been adopted for the task of automatic verification of properties.

One way to prove/disprove protocol properties, in the security domain, is the cryptographic approach, used for proofs by hand [GMR89] or, more recently, automatically [BP03]. Theorem provers, such as Isabelle/HOL [NPW02] have also been applied to this task, together with tools for graphically representing and defining the protocols [vOL02]. Another viewpoint is to embody a possible intruder and plan for an attack [AM02].

Dixon et al. [DFFv04] specify security protocols in $KL_{(n)}$, a language for representing the *Temporal Logic of Knowledge*. Raimondi and Lomuscio [RL04] also use a temporal logic enriched with epistemic connectives for representing the agents' knowledge, but exploit efficient data structures (namely, Ordered Binary Decision Diagrams) to improve the efficiency of the model checking algorithm.

Armando et al. [ACL04] compile a security program into a logic program with choice lp-rules with answer set semantics. Among other approaches to security protocol verification we cite those developed using hereditary Harrop

formulas [Del01], process-algebraic languages [Pan02], model checking with pre-configuration [KAB04], and proof theory [DE01].

Several other frameworks in the literature aim at verifying properties about the behaviour of social agents at design time. Often, such frameworks define structured hierarchies, roles, and deontic concepts such as norms and obligations as first class entities. Notably, ISLANDER [EdlCS02] is a tool for the specification and verification of interaction in complex social infrastructures, such as electronic institutions. ISLANDER allows for the analysis of situations, called scenes, and visualise liveness or safety properties in some specific settings. The kind of verification involved is static and is used to help designing institutions.

Chapter 10

Conclusions

In this thesis, we have presented the $\mathcal{S}$CIFF abductive framework for the specification and verification of interaction in open agent societies.

In this chapter, we summarise and discuss the results, and propose directions for future research.

10.1 Summary

As far as expressiveness is concerned, we believe that the $\mathcal{S}$CIFF framework (described in Ch. 2) is satisfactory. In particular, as shown in Ch. 6, it is able to express both a social semantics of Agent Communication Languages, and commonly used Agent Interaction Protocols. The $\mathcal{S}$CIFF framework lets the agent society designer specify the agent interaction in a way that follows the recent trends in the multiagent community, i.e., constraining the agent interaction as little as necessary, so to support open agent societies, composed of autonomous and heterogeneous agents.

The declarative framework has an operational counterpart: the $\mathcal{S}$CIFF abductive proof procedure (recalled in Ch. 3), which can be used directly for verification of specifications given in the declarative framework. The proof has been proved sound and terminating.

The implementation of the $\mathcal{S}$CIFF proof procedure, achieved using state-of-the-art (constraint) logic programming technology and described in Ch. 5), makes $\mathcal{S}$CIFF an actually usable tool. In fact, it has already been integrated into a sys-

tem which can be interfaced to existent multiagent platforms. The performance of the proof, discussed in Ch. 7, while largely dependent on the specification being verified, has been found acceptable in the experiments performed on practical cases.

An extension of the $\mathcal{S}$CIFF proof procedure, called g-$\mathcal{S}$CIFF, which is able to generate compliant histories, rather than only checking for compliance of given histories, can be used to verify protocol properties, and its implementation has already been used for this purpose, as shown in Ch. 9. However, the lack of a result of completeness for g-$\mathcal{S}$CIFF limits, at the time of writing, its applicability to real cases.

10.2 Future research

There are many possible developments to the work presented in this thesis.

As far as the formal framework is concerned, in its present state it is only able to specify what the compliant agent behaviour should be, and to detect violations. It would certainly be useful to extend it in order to enable it to also *manage* violations; this could be done by imposing sanctions on agents that to not comply to the specification. Such an extension, however raises theoretical questions: for example, in our framework, it is not obvious to identify the "culprit" of a violation.

A promising ongoing research is the investigation of the link between the $\mathcal{S}$CIFF framework and Deontic Logic or, more in general, deontic notions; once the relation has been established at the theoretical level, it would be possible to exploit the $\mathcal{S}$CIFF computational machinery for the verification of many normative multiagent systems, whose specification is based on deontic concepts.

In the perspective of practical applications, the $\mathcal{S}$CIFF framework would also greatly benefit from improvements of the performance of the $\mathcal{S}$CIFF proof procedure. While improvements are certainly possible at the programming level, the fact remains that the number of nodes of the theoretical $\mathcal{S}$CIFF proof tree can explode, depending on the social specification to be verified. In order to (partly) alleviate this problem, the architecture of the $\mathcal{S}$CIFF proof procedure could be revised, so as to have not one single big proof tree, but a number of smaller trees

(intuitively, one tree for one partially solved integrity constraint).

Another possibility is to specialise the implementation to restricted version of the language, which might also make it possible to prove more formal results, such as completeness of the proof procedure.

Bibliography

[AB94] Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[AC00] Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In H.L. Larsen, J. Kacprzyk, S. Zadrozny, T. Andreasen, and H. Christiansen, editors, *FQAS, Flexible Query Answering Systems*, LNCS, pages 141–152, Warsaw, Poland, October 25–28 2000. Springer-Verlag.

[ACG+03] M. Alberti, A. Ciampolini, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. A social ACL semantics by deontic constraints. In V. Mařík, J. Müller, and M. Pěchouček, editors, *Multi-Agent Systems and Applications III. Proceedings of the 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003*, volume 2691 of *Lecture Notes in Artificial Intelligence*, pages 204–213, Prague, Czech Republic, June 16–18 2003. Springer-Verlag.

[ACG+04] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Compliance verification of agent interaction: a logic-based tool. In Trappl [Tra04], pages 570–575. Extended version to appear in a special issue of Applied Artificial Intelligence, Taylor & Francis, 2005.

[ACG+05a] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. On the automatic verification of

interaction protocols using g-$\mathcal{S}$CIFF. Technical Report DEIS-LIA-04-004, University of Bologna (Italy), 2005. LIA Series no. 72.

[ACG+05b] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. The SOCS computational logic approach for the specification and verification of agent societies. In *Global Computing Workshop, Rovereto, Italy, March 2004*, number 3267 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 2005. To appear.

[ACL04] Alessandro Armando, Luca Compagna, and Yuliya Lierler. Automatic compilation of protocol insecurity problems into logic programming. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Artificial Intelligence*, pages 617–627. Springer-Verlag, 2004.

[ADG+04] M. Alberti, D. Daolio, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and verification of agent interaction protocols in a logic-based system. In Hisham M. Haddad, Andrea Omicini, and Roger L. Wainwright, editors, *Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004). Special Track on Agents, Interactions, Mobility, and Systems (AIMS)*, pages 72–78, Nicosia, Cyprus, March 14–17 2004. ACM Press.

[AGL+03a] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. An Abductive Interpretation for Open Societies. In A. Cappelli and F. Turini, editors, *AI*IA 2003: Advances in Artificial Intelligence, Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence, Pisa*, volume 2829 of *Lecture Notes in Artificial Intelligence*, pages 287–299. Springer-Verlag, September 23–26 2003.

[AGL+03b] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and verification of agent interactions using social integrity

constraints. *Electronic Notes in Theoretical Computer Science*, 85(2), 2003.

[AGL+04a] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Michela Milano. A chr-based implementation of known arc-consistency. *CoRR*, cs.LO/0408056, 2004. To appear in Theory and Practice of Logic Programming.

[AGL+04b] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Modeling interactions using *Social Integrity Constraints*: A resource sharing case study. In João Alexandre Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies*, volume 2990 of *Lecture Notes in Artificial Intelligence*, pages 243–262. Springer-Verlag, May 2004. First International Workshop, DALT 2003. Melbourne, Australia, July 2003. Revised Selected and Invited Papers.

[AGL+05] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, Giovanni Sartor, and Paolo Torroni. Mapping deontic operators to abductive expectations. In *Proceedings of the AISB 2005 Convention, First International Symposium on Normative Multiagent Systems (NorMAS2005)*, April 2005. To appear.

[AL02] M. Alberti and E. Lamma. Synthesis of object models from partial models: a csp perspective. In F. van Harmelen, editor, *Proceedings of the Fifteenth European Conference on Artificial Intelligence, Lyon, France (ECAI 2002)*, volume 77 of *Frontiers in Artificial Intelligence and Applications*, pages 116–120. IOS Press, July 2002.

[ALF99] ALFEBIITE: A Logical Framework for Ethical Behaviour between Infohabitants in the Information Trading Economy of the universal information ecosystem. IST-1999-10298, 1999. Home Page: `http://www.iis.ee.ic.ac.uk/~alfebiite/ab-home.htm`.

[AM02] Luigia Carlucci Aiello and Fabio Massacci. Planning attacks to security protocols: Case studies in logic programming. In Antonis C.

Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *Lecture Notes in Computer Science*, pages 533–560. Springer-Verlag, 2002.

[APS02] A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part III*, pages 1053–1061, Bologna, Italy, July 15–19 2002. ACM Press.

[BF95] M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multi-agent systems. In V. Lesser, editor, *Proceedings of the First Intl. Conference on Multi-Agent Systems*, pages 17–25. AAAI Press/The MIT Press, Menlo Park, CA, June 1995.

[BHS93] B. Burmeister, A. Haddadi, and K. Sundermeyer. Generic, configurable, cooperation protocols for multi-agent systems. In Cristiano Castelfranchi and Jean-Pierre Müller, editors, *From Reaction to Cognition, 5th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'93*, number 957 in Lecture Notes in Computer Science, pages 157–171, Neuchatel, Switzerland, August 1993. Springer-Verlag.

[BMO01] B. Bauer, J. P. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent interaction. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, pages 91–103. Springer-Verlag, 2001.

[BP03] Michael Backes and Birgit Pfitzmann. A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. In Paritosh K. Pandya and Jaikumar Radhakrishnan, editors, *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference, Mumbai, India, December 15-17,*

2003, Proceedings, volume 2914 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2003.

[Bür94] H.J. Bürckert. A resolution principle for constrained logics. *Artificial Intelligence*, 66:235–271, 1994.

[CD04] Henning Christiansen and Veronica Dahl. Assumptions and abduction in prolog. In Susana Muñoz-Hernández, José Manuel Gómez-Perez, and Petra Hofstedt, editors, *Workshop on Multiparadigm Constraint Programming Languages (MultiCPL'04)*, Saint-Malo, France, September 2004. Workshop notes.

[CDJT99] C. Castelfranchi, F. Dignum, C.M. Jonker, and J. Treur. Deliberative normative agents: Principles and architecture. In Nicholas R. Jennings and Yves Lespérance, editors, *Intelligent Agents VI, Agent Theories, Architectures, and Languages, 6th International Workshop, ATAL '99, Orlando, Florida, USA, Proceedings*, number 1757 in Lecture Notes in Computer Science, pages 364–378. Springer-Verlag, 1999.

[CFS99] R. Conte, R. Falcone, and G. Sartor. Special issue on agents and norms. *Artificial Intelligence and Law*, 1(7), March 1999.

[Cla78] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[CTS95] B. Cox, J.C. Tygar, and M. Sirbu. Netbill security and transaction protocol. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, New York, July 1995.

[Dav01] P. Davidsson. Categories of artificial societies. In A. Omicini, P. Petta, and R. Tolksdorf, editors, *Engineering Societies in the Agents World II*, volume 2203 of *Lecture Notes in Artificial Intelligence*, pages 1–9. Springer-Verlag, December 2001. 2nd International Workshop (ESAW'01), Prague, Czech Republic, July 7, 2001, Revised Papers.

[DE01] Giorgio Delzanno and Sandro Etalle. Proof theory, transformations, and logic programming for debugging security protocols. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation : 11th International Workshop, (LOPSTR 2001). Selected papers.*, volume 2372 of *Lecture Notes in Computer Science*, pages 76–90, Paphos, Cyprus, November 2001. Springer Verlag.

[Del01] Giorgio Delzanno. Specifying and debugging security protocols via hereditary Harrop formulas and λProlog - a case-study -. In Herbert Kuchen and Kazunori Ueda, editors, *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings*, volume 2024 of *Lecture Notes in Computer Science*, pages 123–137. Springer-Verlag, 2001.

[Dem95] Y. Demazeau. From interactions to collective behaviour in agent-based systems. In *European Conference on Cognitive Sciences*, 1995.

[DFFv04] Claire Dixon, Mari-Carmen Fernández Gago, Michael Fisher, and Wiebe van der Hoek. Using temporal logics of knowledge in the formal verification of security protocols. In *Procedings of the Eleventh International Workshop on Temporal Representation and Reasoning (TIME'04)*, 2004.

[DMW02] V. Dignum, J. J. Meyer, and H. Weigand. Towards an organizational model for agent societies using contracts. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II*, pages 694–695, Bologna, Italy, July 15–19 2002. ACM Press.

[EdlCS02] M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part III*, pages 1045–1052, Bologna, Italy, July 15–19 2002. ACM Press.

[EMST03a] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Aspects of Protocol Conformance in InterAgent Dialogue. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2003)*, pages 982–983, Melbourne, Victoria, July 14–18 2003. ACM Press.

[EMST03b] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Protocol conformance for logic-based agents. In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico (IJCAI-03)*. Morgan Kaufmann Publishers, August 2003.

[FC02] N. Fornara and M. Colombetti. Operational specification of a commitment-based agent communication language. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II*, pages 535–542, Bologna, Italy, July 15–19 2002. ACM Press.

[FIP] FIPA: Foundation for Intelligent Physical Agents. Home Page: `http://www.fipa.org/`.

[FIP01] FIPA Communicative Act Library Specification, August 2001. Published on August 10th, 2001, available for download from the FIPA website, `http://www.fipa.org`.

[FIP02] FIPA Request Interaction Protocol Specification. Standard SC00026H, Foundation for Intelligent Physical Agents, December 2002. Published on December 3, 2002, available for download from the FIPA website.

[FK97] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, November 1997.

[FLM97] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In J. Bradshaw, editor, *Software Agents*. MIT Press, Cambridge, MA, 1997.

[Frü98] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.

[GHB00] M. Greaves, H. Holmback, and J. Bradshaw. What is a conversation policy? In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, number 1916 in Lecture Notes in Computer Science, pages 118–131. Springer-Verlag, 2000.

[GLM+03] Marco Gavanelli, Evelina Lamma, Paola Mello, Michela Milano, and Paolo Torroni. Interpreting abduction in CLP. In Francesco Buccafurri, editor, *APPIA-GULP-PRODE Joint Conference on Declarative Programming*, pages 25–35, Reggio Calabria, Italy, September 3–5 2003. Università Mediterranea di Reggio Calabria.

[GLM05] Marco Gavanelli, Evelina Lamma, and Paola Mello. Proof of properties of the sciff proof-procedure. Technical Report CS-2005-01, Dipartimento di Ingegneria, Universitá di Ferrara, 2005. Available at `http://www.ing.unife.it/aree_ricerca/informazione/cs/technical_reports/%CS-2005-01.pdf`.

[GLMT04] Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. SCIFF: Full proof of soundness. Deliverable IST32530/DIFERRARA/401/D/I/b1, SOCS Consortium, Jun 2004.

[GLT+03] M. Gavanelli, E. Lamma, P. Torroni, P. Mello, K. Stathis, P. Moraïtis, A. C. Kakas, N. Demetriou, G. Terreni, P. Mancarella, A. Bracciali, F. Toni, F. Sadri, and U. Endriss. Computational model for computees and societies of computees. Technical report, SOCS Consortium, 2003. Deliverable D8.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–207, 1989.

[GP02a] F. Guerin and J. Pitt. Proving properties of open agent systems. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II*, pages 557–558, Bologna, Italy, July 15–19 2002. ACM Press.

[GP02b] Frank Guerin and Jeremy Pitt. Guaranteeing properties for e-commerce systems. In Julian A. Padget, Onn Shehory, David C. Parkes, Norman M. Sadeh, and William E. Walsh, editors, *Agent-Mediated Electronic Commerce IV, Designing Mechanisms and Systems, AAMAS 2002 Workshop on Agent Mediated Electronic Commerce, Bologna, Italy, July 16, 2002, Revised Papers*, pages 253–272. Springer-Verlag, 2002.

[Hew91] C. Hewitt. Open information systems semantics for distributed artificial intelligence. *Artificial Intelligence*, 47(1-3):79–106, 1991.

[Hol90] C. Holzbaur. Specification of constraint based inference mechanism through extended unification. Dissertation, Dept. of Medical Cybernetics & AI, University of Vienna, 1990.

[Hug02] M. Huget. Agent uml class diagrams revisited, 2002.

[IS00] M. Ito and J.S. Sichman. Dependence based coalitions and contract net: A comparative analysis. In *Pacific Rim International Conference on Artificial Intelligence*, page 812, 2000.

[JAD] Java Agent DEvelopment framework. Home Page: `http://sharon.cselt.it/projects/jade/`.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[JMMS98] J. Jaffar, M.J. Maher, K. Marriott, and P.J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

[KAB04] Kyoil Kim, Jacob A. Abraham, and Jayanta Bhadra. Model checking of security protocols with pre-configuration. In Kijoon Chae and Moti Yung, editors, *Information Security Applications, 4th International Workshop, WISA 2003, Jeju Island, Korea, August 25-27, 2003, Revised Papers*, volume 2908 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2004.

[KKT93] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[KS86] R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[KTW98] R.A. Kowalski, F. Toni, and G. Wetzel. Executing suspended logic programs. *Fundamenta Informaticae*, 34:203–224, 1998.

[Kun87] K. Kunen. Negation in logic programming. In *Journal of Logic Programming*, volume 4, pages 289–308, 1987.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd extended edition, 1987.

[Low96] G. Lowe. Breaking and fixing the Needham-Shroeder public-key protocol using CSP and FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems: Second International Workshop, TACAS'96*, volume 1055 of *Lecture Notes in Artificial Intelligence*, pages 147–166. Springer-Verlag, 1996.

[Mer01] S. Merz. Model checking: A tutorial overview. In F. Cassez, C. Jard, B. Rozoy, and M.D.Ryan, editors, *Modeling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science, pages 3–38. Springer-Verlag, 2001.

[MPW02] P. McBurney, S. Parsons, and M. Wooldridge. Desiderata for agent argumentation protocols. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on*

Autonomous Agents and Multiagent Systems (AAMAS-2002), Part I, pages 402–409, Bologna, Italy, July 15–19 2002. ACM Press.

[Nis00] Noam Nisan. Bidding and allocation in combinatorial auction. In *Proceedings of the International Conference on Electronic Commerce (EC-00)*, pages 1–12, 2000.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[NS78] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[NS02] P. Noriega and C. Sierra. Institutions in perspective: An extended abstract. In *Sixth International Workshop CIA-2002 on Cooperative Information Agents*, volume 2446 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[OZ99] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999. Special Issue: Coordination Mechanisms for Web Agents.

[Pan02] Jun Pang. Analysis of a security protocol in μCRL. In Chris George and Huaikou Miao, editors, *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings*, volume 2495 of *Lecture Notes in Computer Science*, pages 396–400. Springer-Verlag, 2002.

[RG92a] A. Rao and M. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of the International Workshop on Knowledge Representation, KR'92*, pages 439–449, 1992.

[RG92b] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KRR92)*, Boston, MA, 1992.

[RL04] Franco Raimondi and Alessio Lomuscio. Verification of multiagent systems via ordered binary decision diagrams: An algorithm and its implementation. In N. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2004)*, pages 630–637, Columbia University, New York City, July 2004. ACM Press.

[RZ94] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers*. MIT Press, Cambridge, MA, 1994.

[SCDC98] J.S. Sichman, R. Conte, Y. Demazeau, and C. Castelfranchi. A social reasoning mechanism based on dependence networks. In M. Huhns and M. Singh, editors, *Readings in Agents*, pages 416–420. Morgan Kaufmann Publishers, 1998.

[Sic01] J.S. Sichman. A dependence-based model for social reasoning in multi-agent systems. Technical Report BT/PCS/0108, Escola Politécnica da Universidade de São Paulo, 2001.

[SIC03] SICStus prolog user manual, release 3.11.0, October 2003. `http://www.sics.se/isl/sicstus/`.

[Sin98] M. Singh. Agent communication language: rethinking the principles. *IEEE Computer*, pages 40–47, December 1998.

[Sin00] M. P. Singh. A social semantics for agent communication languages. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, pages 31–45. Springer-Verlag, 2000.

[SKL$^+$04] Kostas Stathis, Antonis C. Kakas, Wenjin Lu, Neophytos Demetriou, Ulle Endriss, and Andrea Bracciali. PROSOCS: a platform for programming software agents in computational logic. In Trappl [Tra04], pages 523–528. Extended version to appear in a special issue of Applied Artificial Intelligence, Taylor & Francis, 2005.

[Smi80] R.G. Smith. The contract net protocol: High level communication and control in a distributed porblem solver. *IEEE Transaction on Computers*, C-29(12):1104–1113, 1980.

[SOC] Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. IST-2001-32530. Home Page: `http://lia.deis.unibo.it/Research/SOCS/`.

[Stu95] P.J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.

[Tra04] Robert Trappl, editor. *Proceedings of the 17th European Meeting on Cybernetics and Systems Research, Vol. II, Symposium "From Agent Theory to Agent Implementation" (AT2AI-4)*. Austrian Society for Cybernetic Studies, Vienna, Austria, April 13-16 2004.

[van03] L. van der Torre. Contextual deontic logic: Normative agents, violations and independence. *Annals of Mathematics and Artificial Intelligence*, 37(1):33–63, 2003.

[vD91] P. van Hentenryck and Y. Deville. The Cardinality Operator: A new Logical Connective for Constraint Logic Programming. In K. Furukawa, editor, *Logic Programming, Proceedings of the Eigth International Conference, Paris, France*, volume 2, pages 745–759, 1991.

[vOL02] David von Oheimb and Volkmar Lotz. Formal security analysis with interacting state machines. In Dieter Gollmann, Günter Karjoth, and Michael Waidner, editors, *Computer Security - ESORICS 2002, 7th European Symposium on Research in Computer Security, Zurich,*

Switzerland, October 14-16, 2002, Proceedings, volume 2502 of *Lecture Notes in Computer Science*, pages 212–229. Springer-Verlag, 2002.

[vSD93] P. van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). Technical Report CS-93-02, Department of Computer Sciences, Brown University, January 1993.

[Wel93] M. P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[Woo02] M. Wooldridge. *Introduction to Multi-Agent Systems.* John Wiley & Sons, Ltd., 2002.

[Wri51] G.H. Wright. Deontic logic. *Mind*, 60:1–15, 1951.

[WW98] M. P. Wellman and P. R. Wurman. Market-aware agents for a multiagent world. *Robotics and Autonomous Systems*, 24:115–125, 1998.

[Xan03] I. Xanthakos. *Semantic Integration of Information by Abduction.* PhD thesis, Imperial College London, 2003.

[YS02] P. Yolum and M.P. Singh. Flexible protocol specification and execution: applying event calculus planning using commitments. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II*, pages 527–534, Bologna, Italy, July 15–19 2002. ACM Press.