SOCS

A COMPUTATIONAL LOGIC MODEL FOR THE DESCRIPTION, ANALYSIS AND VERIFICATION OF GLOBAL AND OPEN SOCIETIES OF HETEROGENEOUS COMPUTEES

IST-2001-32530

D9: A Prototype for the Animation of Societies of Computees

Project number:	IST-2001-32530
Project acronym:	SOCS
Document type:	IN (information note)
Document distribution:	I (internal to SOCS and PO)
CEC Document number:	IST32530/CITY/012/IN/I/a1
File name:	3012-a1[socs-d9].pdf
Editor:	K. Stathis and P. Torroni
Contributing partners:	ALL
Contributing workpackages:	WP4
Estimated person months:	35
Date of completion:	22 December 2003
Date of delivery to the EC:	1 January 2004
Number of pages:	85

ABSTRACT
The development of the prototype demonstrator in SOCS integrates the implementation effort of the logical models developed in deliverables D4 [57] and D5 [66] according to the computational model described in D8 [58]. The combined implementation effort results in PROSOCS, a generic platform whose name stands for **Pro**gramming **S**ocieties **O**f **C**omputee**S**. We describe PROSOCS, we demonstrate its application to implement a series of examples in the context of a global computing application, we evaluate the resulting interactions, and we discuss our plans for future work.

D9: A Prototype for the Animation of Societies of Computees

Marco Alberti[6], Andrea Bracciali[2], Anna Ciampolini[5], Federico Chesani[5], Neophytos Demetriou[4], Ulle Endriss[1], Marco Gavanelli[6], Antonis Kakas[4], Evelina Lamma[6], Wenjin Lu[3], Paolo Mancarella[2], Paola Mello[5], Michela Milano[5], Fabrizio Riguzzi[6], Fariba Sadri[1], Kostas Stathis[3], Francesca Toni[1], Paolo Torroni[5]

[1] Department of Computing, Imperial College London, UK.
Email: {ue,ft,fs}@doc.ic.ac.uk

[2] Dipartimento di Informatica, Università degli Studi di Pisa
Email: {braccia,paolo}@di.unipi.it

[3] Department of Computing, City University London, UK.
Email: {lue,kostas}@soi.city.ac.uk

[4] Department of Computer Science, University of Cyprus
Email: {nkd,antonis}@ucy.ac.cy

[5] DEIS, Università degli Studi di Bologna
Email: {aciampolini,fchesani,pmello,mmilano,ptorroni}@deis.unibo.it

[6] Dipartimento di Ingegnieria, Università degli Studi di Bologna
Email: {malberti,elamma,mgavanelli,friguzzi}@ing.unife.it

ABSTRACT

The development of the prototype demonstrator in SOCS integrates the implementation effort of the logical models developed in deliverables D4 [57] and D5 [66] according to the computational model described in D8 [58]. The combined implementation effort results in PROSOCS, a generic platform whose name stands for **Pro**gramming **S**ocieties **O**f **C**omputee**S**. We describe PROSOCS, we demonstrate its application to implement a series of examples in the context of a global computing application, we evaluate the resulting interactions, and we discuss our plans for future work.

Contents

I OVERVIEW **5**

1 Introduction **5**
1.1 Motivation . . . 5
1.2 Aims . . . 6
1.3 Objectives . . . 6
1.4 Contribution . . . 6
1.5 Structure . . . 7

2 Deliverable Information **8**
2.1 Task Allocation . . . 8
2.2 Components of D9 . . . 8
2.3 Publications . . . 9

II IMPLEMENTATION OF SOCS **10**

3 The PROSOCS Platform **10**
3.1 The PROSOCS Reference Model . . . 10
3.2 The Reference Model of Individual Computees . . . 11
3.3 The Society Reference Model . . . 12
3.4 The Medium . . . 13
3.5 Choice of Technologies for Building PROSOCS . . . 13

4 Implementation of a Computee **15**
4.1 Implementation of the Mind . . . 15
4.1.1 Computee Implementation Architecture . . . 16
4.1.2 The State of a Computee . . . 17
4.1.3 Proof–Procedures . . . 20
4.1.4 Capabilities . . . 25
4.1.5 Transitions . . . 30
4.1.6 Cycle Theory . . . 32
4.2 The Implementation of the Body . . . 37
4.2.1 Body . . . 37
4.2.2 GUI . . . 38
4.3 The Implementation of the Medium . . . 39
4.3.1 The JXTA Project . . . 39
4.3.2 The PROSOCS Medium as an API . . . 40

5 Implementation of Societies **40**
5.1 Overall Architecture . . . 42
5.2 The Society Module . . . 43
5.2.1 The Social Compliance Verifier . . . 44
5.2.2 The History Manager . . . 44
5.2.3 The Event Recorder . . . 45

5.2.4 Message processing . . . 47
5.3 Proof Procedure . . . 47
5.3.1 Overview . . . 48
5.3.2 Variables . . . 48
5.3.3 Data Structures . . . 49
5.3.4 Transitions . . . 50
5.4 GUI . . . 55

III THE DEMONSTRATOR IN PROSOCS 57

6 A Prototype Application 57
6.1 The Application Scenario . . . 57
6.2 Summary of the *Leaving San Vincenzo* scenario . . . 57
6.3 Extending *Leaving San Vincenzo* . . . 58
6.4 Running the SOCSDemo . . . 58
6.5 An Example Run . . . 59

7 Evaluation 60
7.1 Evaluation of WP4 . . . 60
7.2 Mobility in (PRO)SOCS: A Feasibility Study . . . 61
7.2.1 Preliminaries . . . 62
7.2.2 Mobility in PROSOCS: A Sketch . . . 64
7.2.3 Which primitives? . . . 66
7.2.4 Can we take advantage of SOCS to support mobility? . . . 67
7.3 Related work . . . 68
7.3.1 Platforms based on a Directory Facilitator Approach . . . 68
7.3.2 Platforms that treat Agents as First Class Objects . . . 69
7.3.3 Commercial-grade platforms . . . 71
7.3.4 Agent Programming Languages . . . 72
7.3.5 Social Infrastructures . . . 74

IV CONCLUSION 75

8 Summary 75

9 Future work 75

10 Appendices 83
10.1 Protocol definition language . . . 83
10.2 A brief introduction to Constraint Handling Rules . . . 84

Part I
OVERVIEW

1 Introduction

Global Computing(GC) [44] proposes a new vision for information and communication technologies whereby most physical objects that people use will be transformed into electronic devices with computing processors and embedded software designed to perform (or control) a multitude of tasks in people's everyday activities. Many of these devices will also be able to communicate with each other and to interact with the environment in carrying out the tasks they are designed for. The expectation is that the design of computational and information processing systems will rely entirely on the available infrastructure. The challenge this vision poses is huge in that it requires the definition and exploitation of dynamically configured applications of software entities interacting in novel ways with other entities to achieve or control their computational tasks and processing power around us.

In the context of GC environments, the SOCS project [83] investigates the development of computational and logical models of individual and collective behaviour of computational entities - which we refer to as *computees*. Computees are software agents with a strong logic-based component representing the cognitive and social capabilities of such agents in a global and open computing environment. By using these socio-cognitive abilities computees can form complex organisations, which in SOCS we call *Societies of Computees*.

1.1 Motivation

In SOCS we are motivated by the observation that techniques for developing interactions in global computing environments result either in low-level implementations with no obvious logical characterisation, which are, therefore, not verifiable, or in abstract specifications possibly employing expressive modalities, but which are computationally intractable in many cases. In workpackages WP1, WP2 and WP3, we have developed computational logic models [57, 66, 58] that attempt to bridge this gap between specification and implementation. However, we also need to demonstrate their realisation by means of a platform, as well as, their computational viability with experiments. The implementation effort described in this workpackage WP4 seeks to support the activities of the third and final year of the project, in particular, the experimentation process planned be carried out in workpackage WP6.

Another motivation of this work stems from the fact that although a number of Multi-Agent Systems (MAS) platforms and tools are available, for example see [72, 16, 74], often the programmer is left alone with the responsibility to develop the reasoning capabilities of the agent from scratch. Other times, whenever a language with reasoning features is available [75, 45], the programmer is often left alone to deal with with the development of the communication and interaction of the agent in an open and distributed environment. In many cases where both the agent reasoning and the distributed systems infrastructure is in place [1, 64], there is little help with tools that ensure how (or in what sense) the actual behaviour of the artificial agent societies being created deviate from the expected behaviour required by the specification of the application. In some cases [12], it is not that the technologies are not in place, but rather that the construction of the platform is motivated by a specific class of applications, where the conformance to an expected overall behaviour simply plays a less prominent role.

1.2 Aims

Given the original motivation of the SOCS project and the current state of the art of MAS platforms this work aims at:

- identifying the generic components of the logical models to form the basis of the platform;
- developing a platform suitable for building societies of computees based on the computational models developed in the SOCS project (WP1,WP2,WP3);
- providing implementation reference models for organising the interaction of the generic components;
- demonstrating how the generic functionality of the platform can be used to develop applications in the context of GC.

1.3 Objectives

The main objective of this work is to implement a concrete instance of the platform showing how the logical components can be combined in a distributed system implementation to support practical applications. This instance should support a prototype demonstrator showing how examples of GC problems/applications can be tackled using the resulting technology.

The specific objectives of the work is to use the demonstrator to animate the models provided in previous workpackages (WP1,WP2,WP3), in order to support:

- different reasoning capabilities of a computee;
- knowledge, goals, and plans of a computee;
- communication capabilities of a computee;
- protocol-based interaction amongst computees in a society;
- normative control by the society over computees;
- adaptive behaviour of a computee;
- properties of individual computees and societies of computees.

In the longer term the platform seeks to support not only the specification of computees and their societies, but also the use of the platform to verify their properties experimentally (WP6), which can be expressed and verified formally with respect to the SOCS models (WP5). In future experiments, these properties may be local - within a single computing environment, or global - within a dynamic collection of open and connected sub-environments.

1.4 Contribution

The main contribution of this work is that it demonstrates the computational feasibility of the models developed in deliverables D4 [57], D5 [66], and D8 [58]. In this context, it presents an implementation framework exemplifying how the different computational logic techniques developed in the project can be integrated in a single reusable tool.

The specific contribution of this work is that it:

- proposes an agent architecture that uses logic programming to represent the *Mind* of a computee and concurrent, object-oriented programming to represent the *Body* of a computee, in this way separating clearly the development of the reasoning capabilities of the computee from the interaction of the computee with the environment;

- identifies and implements what we call a *Medium*, an Applications Programmer's Interface (API) developed on top of the Peer-to-Peer Computing support of JXTA[56], that allows the body of a computee to sense and affect the state of an open and networked electronic environment;

- proposes a *Social Infrastructure* component that allows a user to specify in a logical form the expected interactions amongst computees using logic programming and object-oriented programming techniques;

- uses the Medium to enable the Social Infrastructure to access the interactions amongst computees and verify their compliance against expected behaviour specified in the form of logical protocols;

- implements an initial set of *Graphical User Interfaces* that allow the user to view the reasoning and the interactions of computees, as well as test the interactions against the expectations of the societies these computees are members of;

- illustrates through concrete examples how the generic functionalities of the proposed system can be used to develop GC applications;

- investigates how the resulting platform can be extended to provide mobility of computees;

- compares and contrasts the approach taken to develop the work with existing approaches in the literature that have achieved the building of noteworthy agent platforms.

1.5 Structure

Apart from this introduction, this overview part I contains in the next section useful information about the deliverable for the reviewers, such as the allocation of tasks, the components of the deliverable, and the published papers that support the work described in this deliverable.

Three additional main parts follow.

Part II describes the implementation of the SOCS models, whose realisation gives rise to the PROSOCS platform. In this part we present the reference models for PROSOCS and we give a justification for the choice of technologies to implement them. We then proceed to discuss in detail the implementation of individual computees and, separately, the implementation of their societies.

The generic implementation of the platform is instantiated in Part III, where we describe a demonstration scenario that will allow us to exemplify the interactions that are currently being supported by the system. We also evaluate the current status of the implementation. The evaluation contains a separate discussion on the feasibility of mobility in SOCS, as well as a section on related work.

The final part, part IV, summarises the work that we have presented, further discussing concrete directions for future work.

2 Deliverable Information

2.1 Task Allocation

The allocation of tasks for WP4 and the way the consortium was organised during the second year to meet this challenge has already been discussed in detail in deliverable D7 [18]. The task allocation is shown in Fig. 1. Each task identified is discussed in detail later in the document.

	ICSTM	DIPISA	CITY	UCY	UNIBO	DIFERRARA
Social Infrastructure					X	X
Society GUI					X	X
Cycle + State + Transitions			X	X		
Planning + Reactivity	X	X				
Preconds + Constraint Solving	X					
Goal Decision				X		
Temporal Reasoning		X		X		
Computee Interface			X			
Sensing Capability			X			
Medium			X			
Platform	X	X	X	X	X	X
Example Documents	X	X	X	X	X	X
Web-Site for the Prototype			X		X	X

Figure 1: Task Allocation for WP4 in the second year

The allocation of these tasks has been done coherently with the WP3 allocation task plan, shown in the companion deliverable D8, as it seemed reasonable for the same partners to take primary responsibility for both the definition and the implementation of each computational model they were allocated.

2.2 Components of D9

Contractually, deliverable D9 should consist of the following components:

- the software deliverable of the prototype demonstrator of SOCS, which we shall refer to in this document as the *SOCSDemo*;
- a web-site dedicated solely on the SOCSDemo explaining how it can be accessed/downloaded;
- a paper deliverable (this document), which should provide an overview of the implementation effort, so that the contribution and the status of the implementation effort is given the right importance.

To demonstrate the significance the WP4 task has in SOCS, however, the consortium decided that we should further complement the contractual obligations for D9 with the following documents:

- two documents with example descriptions, whose aim is to focus the development activity, specifying the concrete interactions characteristic of GC kind of applications that the SOCSDemo should seek to demonstrate;
- a user manual, providing detailed instructions on how to run the SOCSDemo, including a first attempt that discusses how to use the PROSOCS platform for writing a simple application from scratch.

2.3 Publications

The following publications have been produced as a result of the implementation effort:

- a general description of PROSOCS with emphasis on the computee architecture and its implementation has been accepted for presentation in [88];
- a general description of the implementation of the Society Infrastructure and verifier has been submitted and is under review [8];
- a preliminary implementation of the social proof using constraint handling rules, along with some examples including the NetBill and the FIPA-Contract-Net protocols has been presented/accepted for presentation in [9, 10];
- preliminary work discussing the combination of Peer-to-Peer Computing with the Computational Logic we use to build PROSOCS is described in [63];
- a simplified version of the scenario of the SOCSDemo has been presented in [30].

Part II
IMPLEMENTATION OF SOCS

3 The PROSOCS Platform

PROSOCS (**Pro**gramming **So**cieties of **C**omputee**s**) is the generic functionality obtained by implementing in a distributed environment the logical models specified in the SOCS project. In this section we present the organisation of PROSOCS in terms of its constituent components. This is an implementation-independent part of the system, which is intended to describe what we call *reference models* of PROSOCS, viz., the overall architecture and conceptual organisation of the system. In other words, the intention here is to show an intuitive picture of what the prototyping effort has considered as important, before giving the technical details that will follow in the sections after this. We close the section with a discussion of the technologies used to build PROSOCS.

3.1 The PROSOCS Reference Model

PROSOCS is built according to the Reference Model shown in Fig. 2. This figure depicts an

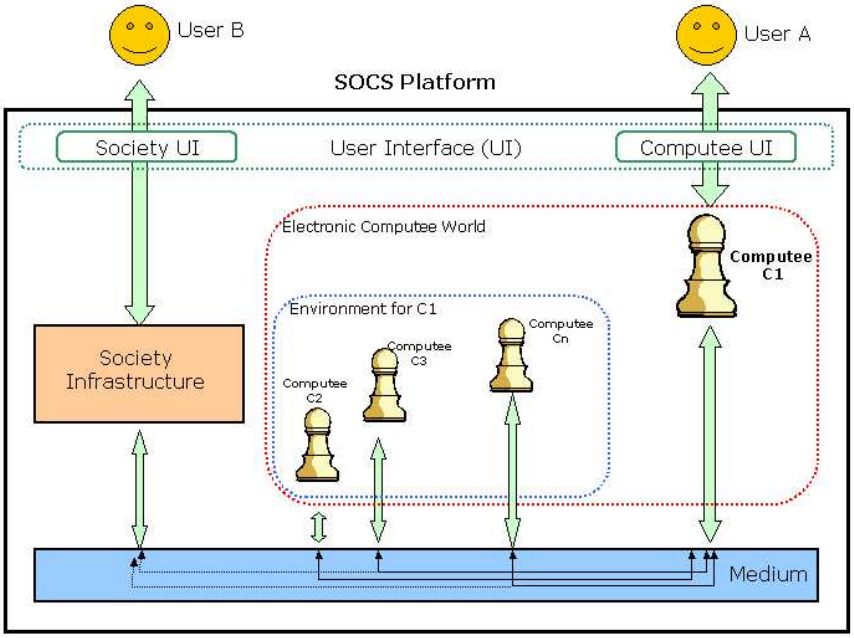

Figure 2: The Reference Model of PROSOCS

electronic world which provides, first of all, a *Medium* that allows entities in that world to interact and communicate with each other, discover each other, and that allows connectivity of

the world with other worlds representing different instances of the same platform. To view the world and communicate their needs, users interact with the world via a *User Interface*.

The reference model also depicts a *computee* C_1 in the world, with the world situating C_1 in an *Environment*, a notional component that conceptually aggregates all the other actual computees in the world except C_1, viz., $C_2, C_3, ..., C_n$. It is important to note that a computee is considered to be an intelligent software component with reasoning capabilities that allow the computee to act autonomously in the world. In this model, objects in the environment can be handled by introducing computees that enforce the rules of the environment to manage and access the functionalities and states of these objects. We will discuss computees in section 3.2.

To regulate the interactions amongst computees in a world, the reference model also relies on a *Social Infrastructure*, a component that enforces *social rules* which in turn specify the ideal functioning of the computees' world. For example, communication protocols and social membership rules are social rules of the kind handled by the Social Infrastructure. We will describe this component in more detail later in section 3.3.

3.2 The Reference Model of Individual Computees

We propose a computee architecture where we interpret a computee being situated in an environment as the computee directly controlling a *body* that allows the computee to access the world that gives rise to that environment. By direct control we mean that the body will execute any action that the computee *mind* selects to execute. However, the results of these actions are mediated by the interaction of the body and the external environment, which are outside of the mind's control.

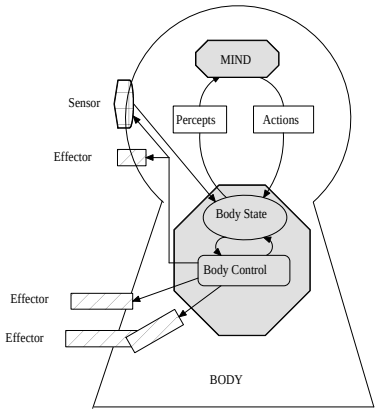

Figure 3: The software architecture of a single computee (defined in [86]), in the special case where the computee's body has three effectors and one sensor.

As depicted in Fig 3, a body senses what is external to it by using *sensors* that can access the current state of the external world. Information from the sensors is then stored in the *body state*, which is a data structure containing all the necessary information that enables the body to act. The body state is accessed by the *mind* which is effectively treated as an action selection process, that can be integrated with the body in order to choose what the body will do next.

In SOCS this action selection process reasons upon a set of extended logic programs consisting of a cycle theory and a number of (sub) knowledge bases, by means of transitions, capabilities, and proof-procedures as specified in the KGP model [57].

Communication between the body and the mind is achieved through the *percepts* that the mind needs to observe in order to select an action. Actions selected by the mind change the state of the body, which is also accessed by the *body control*, a component that mechanically updates the body state and communicates the action to the *effectors*, which in turn influence the environment. With this architecture of computees, we conceptually separate the logical process of reasoning with the physical processes of Action Execution and sensing.

3.3 The Society Reference Model

Within the PROSOCS reference model, a society is an instance of a social framework, where behaviour protocols are coded. In particular, we are interested in the *social* behaviour of computees, i.e., the interactions occurring among members of the society. In the context of the PROSOCS platform, the society collects messages exchanged among computees and reasons upon them. The output of such reasoning activity is part of the SEKB, the dynamic and evolving Social Environment Knowledge Base, as we called it in Deliverable D5 [66]: the (positive or negative) expectations which are pending, fulfilled, or violated. The declarative model of the social framework is defined in Deliverable D5, and its operational semantics is defined in Deliverable D8 [58].

A society can also function as a stand-alone module. In that case, it will not reason upon messages received through a medium, but it will read messages from other media such as an input file, or from the user prompt.

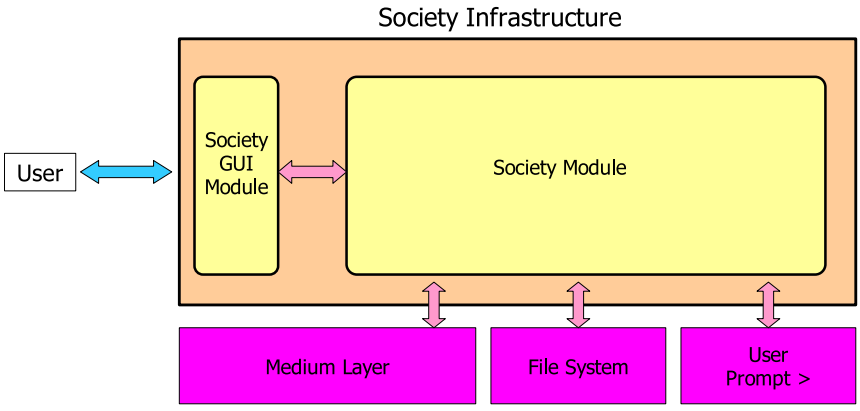

Figure 4: The Society Reference Model

Figure 4 depicts the society reference model, where we present the possible inputs and the output given through a graphical user interface. Later on in this document we will refine this figure and describe how all the various elements have been implemented in detail.

3.4 The Medium

The PROSOCS architecture relies on the premise that the body of an agent is part of the electronic world, thus mediating the interactions between the mind of the computee and the environment. We assume that the body of a computee has access to the medium through modular constructs embedded in the computee's body, that represent specific types of generic sensors and effectors.

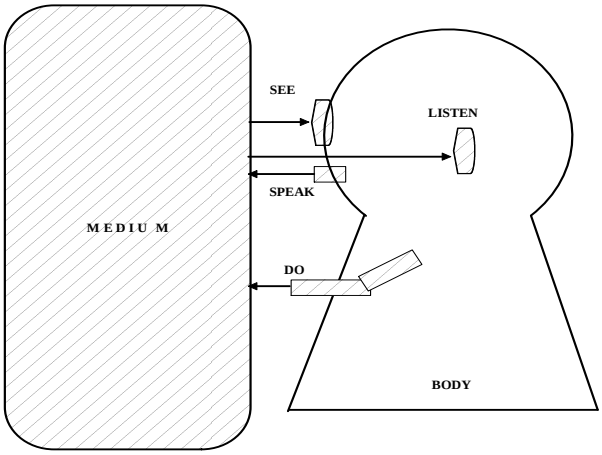

Figure 5: Using the Medium via an Agent Body

As shown in Fig. 5, the reference model of PROSOCS assumes a medium that provides the following sensors and effectors:

- a speak effector that allows the body to utter a statement in a communication language;
- a listen sensor that allows the body to listen to statements uttered by other agents in that language;
- a do effector that can also perform "physical" actions in the environment;
- a see sensor that allows sensing the effects of actions in the environment.

In addition to the above functionality, we also assume that it is part of the implementation of the medium to provide automatic discovery of components such as computees or the society through the support of low-level protocols. We will touch this point in the next section, where we are going to discuss what kind of technology will be used to implement the PROSOCS medium.

3.5 Choice of Technologies for Building PROSOCS

In general, the implementation of the SOCS project poses three main challenging requirements on the technologies available to build agent-based systems. The first requirement is that the technology should allow us to develop the computational logic components, to smoothly support

the detailed reasoning processes based on the models developed in WP1, WP2, WP3, and WP5. The second requirement is that the technology should provide support for an open, distributed and interactive systems infrastructure that will place our approach in the GC context and accommodate the experimentation work in WP6. The third requirement is about combining the reasoning and distributed aspects in an integrated and reusable whole to allows us to differentiate the contribution of this work in the area of Multi-Agent Systems.

The main requirements above have been concrete in deliverable D7 [18] by a set of more specific requirements for building the PROSOCS platform. These can be summarised as follows:

- *R1* – support the ability to deploy computees, rapidly prototype their knowledge bases and reasoning capabilities with logical tools that are easily extensible for this purpose (required by WP1, WP2, WP3, WP5, WP6);

- *R2* – offer the potential of reusing existing computational logic tools already developed by SOCS partners, provided that these tools may sufficiently accelerate the implementation effort without compromising its generality (required by WP4);

- *R3* – provide facilities that support communication and interaction in an open and distributed environment (such as those envisaged by GC) where the discovery of computees and their services in a network is done dynamically and do not rely on solutions provided at the application level (required by GC applications);

- *R4* – make available primitives that allow for multi-threaded, modular and component-based development of applications (required by WP1, WP2, WP3, WP6);

- *R5* – be platform independent and afford libraries of components that provide for the development of user interfaces (required by WP4 and WP6).

At the time when the SOCS project started there was a number of computational logic languages and tools (e.g. SICStus Prolog [80], Qu-Prolog [24], Jinnie [90], and Go! [23]) that could be used to develop the reasoning capabilities of computees and their societies. However, at that time, only a few (e.g. Qu-Prolog [24]) of these tools could easily support reasoning to be executed on a network with little programming effort. None of them provided facilities for managing computees, except perhaps [23] that however failed R5, as these languages have no concept of computee (agent) as a first-class object. Moreover, as these languages do not have consistent product support and facilities for graphical user interfaces, committing to implement in them PROSOCS would have been too risky.

A plethora of languages and systems also exist for building agent-based systems (e.g. JACK [2], AgentBuilder [3], JADE [16], FIPAOS [74], ZEUS [72]), but few were available to express the knowledge and reasoning capabilities of the agent in logic-based formalisms, in particular, the kind of computational logic (and its extensions) that we are trying to explore in SOCS. Although these tools could support some of the requirements that we have for the SOCS demonstrator, most of them would conflict either with the model of individual computees or with the model of the societies or both. Almost all of those that were stable, could not support dynamic discovery of agents in open environments as envisaged in GC (we will return to this point later, when we discuss related work in section 7.3).

To accommodate all the SOCS requirements, we have chosen an integrated approach with the following combination of technologies:

- use SICStus Prolog [80] for the knowledge representation, reasoning, logical and social capabilities of the computee, as well as the reasoning required by the society module; the advantage here being that this Prolog system is extremely stable and good for rapid prototyping, it can support the reasoning capabilities of computees, it supports multi-threading and constraint solving, it has the potential of reusing work already done by partners through its libraries, like CLP(FD) and CHR, and it is extensible with its use of meta-level facilities;
- use the Sun Microsystems Peer-to-Peer platform JXTA [56] to develop the low-level communication requirements of the implementation and handle the openness of the GC environment;
- use XML [94] for exchanging messages between JXTA peers;
- use Java [55] for integrating the various components and building the user interfaces.

Before committing to the above technologies we have investigated alternative approaches. One is reported in [11], but it has been discarded, as it was not sufficient for fulfilling the requirement R3. Another, was attempted in the first year of the project, and involved experimenting with existing Prolog technologies, in particular the Qu-Prolog system [24], used to implement the IFF procedure as the engine underlying communicating agents[33]. This implementation platform was discarded because it did not support requirement R5.

4 Implementation of a Computee

In the previous section we presented the structure of the PROSOCS platform by presenting the parts that described the generic organisation of the system. The resulting reference models, however, were implementation independent. Using the choice of technologies proposed in the previous section, we present here the implementation of PROSOCS, namely, the tools that we have developed and the concrete choices that we have made in order to implement the computee. We discuss first how we have implemented the mind component, then how we have implemented the body component. We also show how the body relies on an *Application Programming Interface* (API) that we have developed to support the functionality of the PROSOCS medium.

4.1 Implementation of the Mind

In order to prototype any software component that is based on a demanding logical model such as KGP [57], a programmer will often have to make a number of pragmatic choices. Stating these choices at the outset is very important for two main reasons. First, these choices give an idea of which parts of the model are straightforward and which ones are complex from an implementation perspective. Secondly, these choices provide a clear idea of what assumptions have been made and, as a result, what extensions need to be incorporated into the system in the future.

In PROSOCS, while building the mind of a computee using the KGP model we have made the following choices:

- the (core) selection functions, for the selection of actions, goals, preconditions and fluents in cycle, return individual items rather than sets of items; as a consequence

- the transitions Action Execution (AE), Plan Introduction (PI), Sensing Introduction (SI), Active Observation Introduction (AOI), which take sets (of actions, goals, preconditions of actions and fluents, respectively) as inputs, are given singleton sets as inputs instead;

- the transition Goal Introduction (GI) returns only one goal as output, rather than a full set;

- the identification of preconditions capability is incorporated within the planning and reactivity capabilities;

- sensing actions are assigned to have no preconditions, represented by *true* in the model.

- we implement the version of operational trace described in D4, rather than its generalisation described in D8.

- the proof-procedure C-IFF, when invoked from within PI, returns partial plans of depth one, rather than any depth;

- temporal constraints are not stored alongside actions in *Plan* and goals in *Goals*, but rather in a global constraint store; we believe this to be not restrictive with respect to the model, and to actually provide a potential improvement upon the model, that might be imported into the model in the third phase of the project;

- KB_0 is stored directly as a set of ground facts, alongside the (existentially quantified) variable substitutions that are implicitly kept within KB_0 in the model;

- the operational trace obtained by reasoning with the cycle is finite rather than infinite (as envisaged by the model); finiteness is achieved via appropriate inputs from the computee's graphical user interface.

4.1.1 Computee Implementation Architecture

Based on the choice of technologies that we have identified earlier for building PROSOCS, the implementation architecture of a computee is shown in Fig. 6. As shown in Fig. 6, the mind of a computee is a SICStus Prolog program implementing the mind's state, the proof-procedures required to build the mind's capabilities, the transitions of the mind based on these capabilities, and the cycle theory of the mind that controls which transitions to execute next, depending on the computation of the selection functions. We use the JASPER interface of SICStus to connect the mind of the computee with its body implemented in Java. We have also used Java to implement a Medium API that uses JXTA to implements the generic effectors (such as the speak and do), sensors (such as see and listen), as well as the functionality of clock. These effectors and sensors allow us to access and affect a whole virtual network of computers via the JXTA P2P infrastructure that will also be used in the same manner to support other computees that are started using PROSOCS.

We shall spend the rest of this section to define the components of the implementation architecture of a computee in more detail.

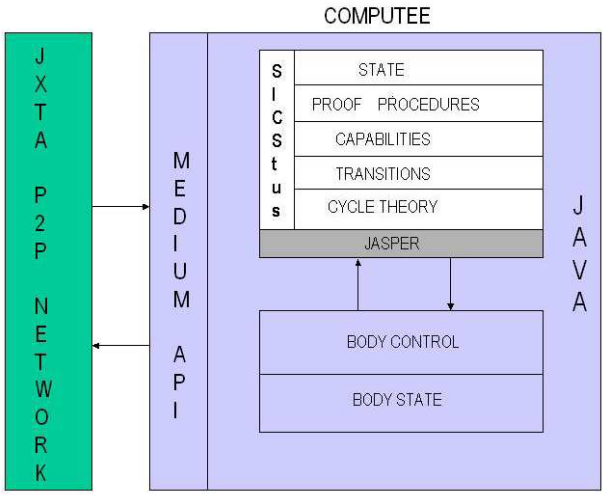

Figure 6: Implementation Architecture of a PROSOCS Computee

4.1.2 The State of a Computee

In KGP the state of a computee is a triple $\langle KB, Goals, Plans\rangle$, where KB is the computee's knowledge base, *Goals* is the set of properties that the computee currently wants to achieve, and *Plans* is a set of concrete actions by which the computee aims at reaching its goals. KB is divided into several (sub)knowledge bases to support different reasoning tasks. These (sub)knowledge bases, except KB_0, in turn may contain different forms of rules and constraints to represent knowledge, which is specific to these modules as we shall see in the next section. We describe next how we have implemented the state components of goals, plans, and the sub-knowledge base KB_0.

Existentially quantified variables. In order to distinguish between existentially quantified variables (as defined in [57]) and normal Prolog variables in the representation of goals and actions, we use terms of the form:

```
eqv(Id)
```

in the representation of the state of a computee. `Id` is a unique identifier generated using the Prolog primitive `gensym/2`. For instance, the term:

```
eqv(t1)
```

is an example of such a generated existentially quantified variable for a time `t1`. We will see shortly how these variables are utilised in the representation of goals and plans.

Temporal Constraints Store. Apart from the representation of existentially quantified variables, at the implementation level we have also decided to maintain temporal constraints

in a single global constraint store (to be shared by all transitions, selection functions, and capabilities) in order to avoid replication. This global constraint store is represented as a list of temporal constraints such as:

```
[egv(t1) #>= 18, ..., eqv(t12) #< 32]
```

We shall see in subsequent sections how this list of temporal constraints is updated by transitions such as Goal Introduction, Plan Introduction, and Reactivity when goals and plans are. For interpreting the store we use the C-IFF proof-procedure whose syntactic features to represent constraints, such as `#<`, will be discussed in in the C-IFF part of section 4.1.3. For the time being, it suffices to say that these constraints are accessible at the implementation level by calling the predicate `get_TCS/1`; TCS stands for *Temporal Constraints Store*.

Goals and Plans. Goals and actions in the state are organised in a hierarchy represented as a tree with goals and actions as nodes. This tree has a root `root_0` representing $\bot$ in D4 [57], and children `root_1` and `root_2` representing $\bot^r$ (reactive sub-tree) and $\bot^{nr}$ (non-reactive sub-tree) respectively. Goals in the tree are represented by assertions of the form:

```
goal(Goal, ParentGoal, TemporalConstraints).
```

`Goal` and `ParentGoal` are both of the form `(Fluent, Time)`, indicating that the computee is trying to achieve the goal specified in the `Fluent` at a `Time` (an existentially quantified variable constrained by a the list of `TemporalConstraints` [1]. For example, the assertion:

```
goal((have(festival_ticket),eqv(t4)), root_1, [eqv(t4)#=<10]).
```

represents a top-level goal (i.e. a goal with the reactive root as the parent) for a computee to have a festival ticket before or at time 10 (note the example shows also an instance of an existentially quantified variable).

Similarly to goals, actions are represented by assertions of the form:

```
action(Action, ParentGoal, Preconditions, TemporalConstraints).
```

`Action` is a pair `(Operator, Time)`, whose `Operator` must be executed at a `Time` (constrained as before by the list of `TemporalConstraints`), provided that the list of `Preconditions` is not violated in the current state of the computee. The example below:

```
action((call_taxi(Taxi), eqv(t17)),
       (get_to_station, eqv(t5)),
       [taxi_is_available(Taxi)],
       [eqv(t17) #=< 13]
).
```

[1] Note that, despite the choice to use a TCS to store all the temporal constraints in the state, we still associate a temporal constraint to goals and actions, as in the KGP model. This is a left over from earlier stages of the development of PROSOCS which aids our presentation here. Note also that in the actual implementation, the temporal constraints of goals and actions will always be empty (represented as the empty list `[]` in Prolog), while the concrete temporal constraints associated with goals and actions exemplified below will always be part of the TCS.

shows how an action whose operation is `call_taxi(Taxi)`, with parent goal `get_to_station`, preconditions `taxi_is_available(Taxi)`, and a temporal constraint on the time of the action to be less than or equal to 13.

In addition to the predicates for representing goals and actions, a set of auxiliary predicates have also been introduced to represent relationships amongst nodes in the goal/action tree, as well as to collect state components, such as plans for goals.

Data structures for KB_0. KB_0 holds the (dynamic) knowledge of the computee about the external world, which are mainly observed facts and happened events represented by assertions of the form:

- `observed(Property, Time)` – represents the fact that `Property` has been observed to hold at `Time`.
- `observed(Agent, Action, Time)` – states the fact that an `Agent` has been observed to execute an `Action` at a specific `Time`.

The assertion below is an example of an observation illustrating the fact that the computee whose KB_0 we are modeling has observed at time `7` that a computee `c1` has informed a computee `c2` at time `5` within a dialogue `d` about the fact that there are `no_member_seats`.

```
observed(c1, tell(c1, c2, inform(no_member_seats), d, 5), 7).
```

The knowledge base KB_0 also contains assertions of the form:

```
executed(Action, Time).
```

recording the fact that `Action` has been executed at `Time` by the computee "owning" KB_0. For instance, the assertion:

```
executed((call_taxi(taxi12), eqv(t17)), 6).
```

is an example of a concrete instance of such recording. It is important to note that when an action is executed the system instantiates the time of the existentially quantified variable of the action. A list of variable instantiations, which we call *Sigma*, represents what we have called Σ in deliverable D4 [57]. This list contains equalities for existentially quantified variables that result from the execution of actions and observations of fluents. For example, if the action:

```
(call_taxi(taxi12),eqv(t17))
```

is executed at time 6, then *Sigma* is updated by appending the equality constraint:

```
[eqv(t17)#=6]
```

to it. We call these instantiations of variables in Sigma *Equality Temporal Constraints*. Such constraints are then used together with the temporal constraints store in transitions and selection functions that rely on the planning, reactivity, and temporal reasoning capabilities, as well as the constraint solving underlying the computee models in D4 [57] and D8 [58]. For the time being it suffices to say that in order to access the data structure representing *Sigma*, we have implemented a predicate called `get_Sigma/1`. We will see examples of how this predicate is used when we discuss the implementation of transitions.

4.1.3 Proof–Procedures

Gorgias: Implementing $LPwNF$

A proof-procedure for LPwNF (and its integration with abduction) has been implemented as a Prolog meta-interpreter, named *Gorgias*[48]. Gorgias is a system for argumentation in LPwNF that combines preference reasoning and abduction. It can form the basis for reasoning about adaptable preference policies in the face of incomplete information from dynamic and evolving environments. Gorgias has been ported to work both with SICStus and SWI-Prolog. One can run user defined domain descriptions by loading the system itself and a knowledge base of rules that specify the preference policy at hand.

A Gorgias program consists of Prolog labelled rules of the form:

$$rule(Label, Head, Arg\_Body) :- Body.$$

where, the $Head$ is a literal, the $Body$ is a list of literals referring to auxiliary predicates defined by ordinary Prolog rules, and $Label$ is a compound term composed of a rule name and selected variables from the $Head$, Arg_Body, and $Body$. Arg_Body is a list of literals referring to predicates on which we carry out preference reasoning. Negative literals are terms of the form $neg(L)$.

Priority rules are again described using the rule syntax given above together with the special predicate $prefer(Label_1, Label_2)$ in the head of the rule which means that the rule with name $Label_1$ has higher priority than the rule with name $Label_2$ if the body of this priority rule holds. The role of these priority rules is to encode locally the relative strength of rules in the theory, typically between contradictory rules.

The statement $conflict(Label_1, Label_2)$ indicates that the rules with names $Label_1$ and $Label_2$ are conflicting. In many cases $conflict(Label_1, Label_2)$ will be true iff the heads of the rules are contrary literals, but other rules can also be declared as conflicting by the designer of the domain description. In addition, the predicate $conflict/2$ is derived by statements of the form of $complement(Literal_1, Literal_2)$ which means that rules with heads $Literal_1$ and $Literal_2$ are conflicting. A literal and its negation are always considered to be complements of each other. Other possibilities for *complement* are derived through *incompatibility* statements in the domain dependent part of a given theory.

The computation of a query in Gorgias consists of two interleaving phases, called resolution and qualification (or argumentation) phases. In the resolution phase, the query is reduced to an initial set of rules identified by their labels and a partial valuation of the variables for each label together with ground hypotheses on the open abducible predicates. The top-level rule of the Gorgias meta interpreter is:

```
prove(Query, Delta) :-
    resolve(Query, Delta0),
    isconsistent(Delta0),
    extend(Delta0, [], Delta).
```

The qualification phase extends the initial set of rules (an initial argument) to an admissible set, i.e. to a set that can remain consistent (not self-attacking) and that there exists an admissible defense against each (minimal) attack. The existence of several answers reflects itself on the existence of several defenses for a given set, and imposes a non-deterministic choice among

defenses in the proof-procedure. However, not every potential defense can be promoted to the answer set and thus this affects back the overall computation.

Proof procedures for Gorgias are obtained by adopting specific ways of computing attacks in the particular framework in such a way that it permits the monotonic growth of the answer set, Δ, during the computation. This is captured by the following definition:

```
extend([], DeltaAcc, DeltaAcc).
extend(Delta0, DeltaAcc, Delta) :-
       isconsistent(Delta0),
       findall(AttackNode, (attacks(_, 'A', Delta0, AttackNode)),AttackNodes),
       union(Delta0, DeltaAcc, NewDeltaAcc),
       counterattack(AttackNodes, NewDeltaAcc, Delta).

counterattack([], DeltaThis, DeltaThis).
counterattack([AttackNode|Rest], DeltaThis, Delta) :-
        counterattackone(AttackNode, DeltaThis, NewDeltaThis),
        counterattack(Rest, NewDeltaThis, Delta).

counterattackone([], DeltaThis, DeltaThis).
counterattackone(AttackNode, DeltaThis, Delta) :-
        findall(DefenceNode, (attacks(_, 'D', AttackNode, DefenceNode),
        isconsistent(DeltaThis, DefenceNode)), DefenceNodes),
        member(DefenceNode, DefenceNodes),
        counterattackoneaux(DefenceNode, DeltaThis, Delta).
```

The predicate $counterattackoneaux/3$ will first check whether the $DefenceNode$ is already subsumed by the current answer set or otherwise extend the answer set, appropriately.

The attacking relation, namely $attacks/4$, used in this employs a qualification relation during argumentation that encodes the relative strength of arguments. One simple example of a qualification relation is "prefer rules of the theory" over (abductive) assumptions of the negative conclusion.

A qualification relation is specified via rules of the form

```
attacks0(?QR, ?LT, +Culprit, +Argument, ?CounterArgument)
```

where QR is the name of a concrete instance of a qualification relation, LT is the level type that the relation may be applied (either $'A'$ or $'D'$ for attack and defense level, respectively), $Culprit$ is a rule label or assumption in the given $Argument$ that forms the basis on which the QR constructs the $CounterArgument$. For instance,

```
attacks0('GEN', _, ass(L), _, CA) :-
    complement(L, NL),
    rule(Sig, NL, ArgBody),
    resolve(ArgBody, Resolvent),
    CA = [Sig|Resolvent].
```

denotes the qualification relation GEN (generation) that may be applied either in an attack level or a defense level, i.e. no constraints on the level type, and it constructs a counterargument CA provided that the given argument contains an assumption (the culprit in our case) $ass(L)$.

Here, the predicate *complement*/2 was used in order to find contrary, with respect to negation, literals.

There are cases of qualification relations in which the qualification is concluded based on the rule labels only and thus the predicate *conflict*/2 is used instead of *complement*/2 as seen below.

```
attacks0('DYN_GEQ', _, Culprit, _, CA) :-
        conflict(Culprit, CC),
        rule(SigHP, prefer(CC, Culprit), BodyHP),
        resolve(BodyHP, ResolventHP),
        rule(CC, _, BodyCC),
        resolve(BodyCC, ResolventCC),
        union([nott(SigHP), CC|ResolventCC], [SigHP|ResolventHP], CA).
```

Here the counter argument CA consists of the rules $[CC|ResolventCC]$ that derive the conflict to the culprit and the rules $[SigHP|ResolventHP]$ that derive that at least one rule in the counter argument has higher priority than the rule of the culprit.

Gorgias can also be extended with domain-specific qualification relations.

We have tested the Gorgias system with examples from various domains like legal reasoning, inheritance theory, and autonomous agent deliberation. The experiments have shown the flexibility of the system, and its ability to be specialised to additional or changing needs. It forms the basis for the implementation of the Goal Decision capability and the cycle theories of computees.

C-IFF

The C-IFF proof-procedure for abductive logic programming with constraints forms the basis of the implementation of the planning, reactivity, and temporal reasoning capabilities described later in this document, as well as the constraint solver. This proof-procedure is both an extension and a refinement of the IFF proof-procedure proposed by Fung and Kowalski [43] and is described in deliverable D8 [58].

The procedure has been implemented in SICStus Prolog [80]. Most of the code could very easily be ported to any other Prolog system conforming to standard Edinburgh syntax.[2] The main predicate of our implementation of the C-IFF proof-procedure is `ciff/4`:

```
ciff( +Defs, +ICs, +Query, -Answer).
```

The first argument is a list of iff-definitions, the second is a list of integrity constraints, and the third is the list of conjuncts in the query. The answer consists of three parts: a list of abducible atoms, a list of restrictions on the answer substitution, and a list of (temporal or arithmetic) constraints.

[2] A small exception is the module concerned with constraint solving as it relies on SICStus' built-in constraint logic programming solver over finite domains CLP(FD) [20]. However, the modularity of our implementation would also make it relatively easy to integrate a different constraint solver into the system. The only changes required would be an appropriate re-implementation of a handful of simple predicates providing a wrapper around the constraint solver chosen for the current implementation.

Syntax of abductive logic programs. The C-IFF procedure is defined over *completed* logic programs, i.e. the logic program in the input is required to be a list of predicate definitions in iff-form rather than a list of rules (in if-form) and facts. As these definitions can become rather long and difficult to read, our implementation includes a simple module that translates logic programs into completed logic programs which may be used as input to the C-IFF procedure. Being able to complete logic programs on the fly also allows us to spread the definition of a particular predicate over different knowledge bases. The syntax for facts and rules of a logic program is taken from Prolog. In addition, we also allow for (temporal) constraints as subgoals to a rule. Admissible constraints are terms such as `T1 #< T2+5`. The available constraint predicates are `#=`, `#\=`, `#<`, `#=<`, `#>`, and `#>=`, each of which take two arguments that may be any arithmetic expressions over variables and integers (using operators such as addition, subtraction, and multiplication). Note that for equalities over terms that are not arithmetic terms, the usual equality predicate `=` should be used (e.g. `C = francisco`). Furthermore, we use the predicate `not/1` for the negation of subgoals in a rule.

Integrity constraints are expressions of the form A `implies` B where A is a list of literals (representing a conjunction) and B is a list of atoms (representing a disjunction). Atoms are atomic formulas, including temporal constraints and equalities, but expressions using `not/1` are not allowed. For the list of literals, on the other hand, also negated atoms are admissible. We should stress here that not every logic program with integrity constraints following the syntax definitions given here constitutes an *allowed* abductive logic program according to the original IFF. Additional restrictions are required to be able to avoid the explicit representation of quantifiers. Appropriate allowedness conditions are given in deliverable D8 [58].

Data structure and proof rules. We are now going to turn our attention to the actual implementation of the C-IFF proof-procedure. The procedure manipulates, essentially, a set of formulas that are either atoms or implications (the latter coming from the integrity constraints). The set of definitions of the abductive logic program is kept in the background and is only used to *unfold* defined predicates as they are being encountered. In addition to atoms and implications the aforementioned set of formulas may contain disjunctions of atoms and implications to which the *splitting* rule may be applied, i.e. which give rise to different branches in the proof search tree. In the terminology of C-IFF, the sets of formulas manipulated by the procedure are called *nodes*. A node is a set (representing a conjunction) of formulas (atoms, implications, or disjunctions thereof) which are called *goals*.

A proof is initialised with the node containing the integrity constraints in the program and the literals of the query. The proof-procedure then repeatedly manipulates the current node of goals by rewriting goals in the node, adding new goals to it, or deleting superfluous goals from it. These *proof rules* used to manipulate the current node and to derive its successor node are described in detail in deliverable D8 [58]. Whenever a disjunction is encountered, the current node is split into a set of successor nodes (one for each disjunct). The procedure then picks one of these successor nodes to continue the proof search and backtracking over this choice-point results in all possible successor nodes being explored. In theory, the choice of which successor node to explore next is taken nondeterministically; in practice we simply move through nodes from left to right. The procedure terminates when no more proof rules apply (to the current node) and finishes by extracting an answer from the final node. Enforced backtracking will result in the next branch of the proof tree being explored, i.e. in the remaining abductive answers being enumerated.

The Prolog predicate implementing proof rules is called `sat/7` (as it is used to *saturate* a

set of formulas):

```
sat( +Node, +EV, +CL, +LM, +Defs, +FreeVars, -Answer).
```

`Node` is a list of goals, representing a conjunction. Given our earlier discussion, from a syntactic point of view, we can distinguish three kinds of goals: (1) Prolog terms representing atoms; (2) Prolog terms of the form A `implies` B representing implications (residues of integrity constraints) whose antecedents are conjunctions of atoms and whose consequents are disjunctions of literals; and (3) lists of lists of either of the above, representing disjunctions of conjunctions of formulas.

`EV`, the second argument of `sat/7`, is used to keep track of existentially quantified variables in the node. This set is relevant to assess the applicability of some of the proof rules. `CL` (for constraint list) is used to store the (temporal) constraints that have been accumulated so far. The next argument, `LM` (for loop management) is a list of expressions of the form $A:B$ recording pairs of formulas that have already been used with particular proof rules, thereby allowing us to avoid loops by applying these rules over and over to the same arguments (this is necessary for both the *propagation* and the *factoring* rule). `Defs` is a list of iff-definitions. They represent the completed logic program with respect to which we are evaluating the query. The penultimate argument, `FreeVars`, is used to store the list of free variables, i.e. the list of variables appearing in the original query. Finally, running `sat/7` will result in the variable `Answer` given in the final argument position to be instantiated with a representation of the abductive answer found by the procedure (consisting of a list of abducible atoms, a list of restrictions on the answer substitution, and a list of constraints).

An example. Let us now look at an example of how the proof rules have been implemented. Each rules corresponds to a Prolog clause in the implementation of `sat/7`. The *unfolding rule for atoms* is used to replace an atom in a node with its definition according to the abductive logic program in question. This rule has been implemented as follows:

```
sat( Node, EV, CL, LM, Defs, FreeVars, Answer) :-
  member( A, Node),
  is_atom( A),
  get_def( A, Defs, Ds), !,
  delete( Node, A, Node1),
  NewNode = [Ds|Node1],
  inform( 'unfold atom', A iff Ds, NewNode),
  sat( NewNode, EV, CL, LM, Defs, FreeVars, Answer).
```

The auxiliary predicate `is_atom/1` will succeed whenever the argument represents an atom (i.e., in particular, whenever it is not an implication). Furthermore, `get_def( A, Defs, Ds)`, with the first two arguments being instantiated at the time of calling the predicate, will instantiate `Ds` with the list of lists representing the disjunction that defines the atom `A` according to the iff-definitions given in `Defs` whenever there *is* such a definition (i.e. the predicate will fail for abducible predicates). The appropriate substitutions in `Ds` are also made within `get_def/3`. Once `get_def( A, Defs, Ds)` succeeds we definitely know that the unfolding rule is applicable: there exists and atom `A` in the current `Node` and it is not abducible. Therefore, this is the right point to insert a cut into the clause as we do not want to allow any backtracking over the *order* in which rules are being applied. After we are certain that this rule should be applied we manipulate the current `Node` and generate its successor `NewNode`. We first delete the atom

`A` and then replace it with the disjunction `Ds`. The predicate `sat/7` then recursively calls itself with the new node.[3]

Testing. The Prolog clauses in the implementation of `sat/7` may be reordered almost arbitrarily (the only requirement is that the clause used to implement answer extraction is listed last). Each order of clauses corresponds to a different proof strategy, as it implicitly assigns different priorities to the different proof rules. This feature of our implementation would, in principle, allow for an experimental study of which strategies yield the fastest derivations (although the systematic evaluation of the efficiency of a proof-procedure such as C-IFF does not fall within the scope of SOCS). The order in which proof rules are applied in the current implementation follows some simple heuristics. For instance, logical simplification rules as well as rules to rewrite equality atoms are always applied first. Splitting, on the other hand, is one of the last rules to be applied.

The implementation of the C-IFF proof-procedure has been tested successfully on a number of different examples. Most of these examples are taken from applications of C-IFF within SOCS, e.g. examples to demonstrate different features of the planning and the reactivity capabilities, both of which have been implemented using C-IFF.

4.1.4 Capabilities

To give examples of how capabilities are represented in the mind of a PROSOCS agent, we show here how the capabilities of Goal Decision, Planning, Reactivity, Temporal Reasoning, and Constraint Checking are implemented.

Goal Decision. The Goal Decision (GD) capability is called through the Goal Introduction (GI) transition in order to decide the current preferred goals of the computee. The specification of goal decision and its computational model are described in D4 and D8. The derivability relation $\vdash_{GD}$ of goal decision is implemented in Gorgias by the following top-level rule:

```
self__goal_decision(OGDs) :-
    findall(GD, self__prove(gd(GD)), GDs),
    filter_incompatible(GDs, OGDs).
```

The goal decision capability returns a set of goals by finding first the list of goals using the proof-procedure of Gorgias, and then filters out incompatible goals in the returned answer set via means of the *filter_incompatible*/2 predicate.

During its computation, Gorgias uses the predicate *complement*/2 to identify conflicting goals in terms of user-defined incompatibilities specified via the predicate *kb_gd__incompatible*/2 in KB_{GD}. The link between the aforementioned predicates is coded as follows:

```
complement(gd(X),gd(Y)) :-
    kb_gd__incompatible(X,Y).
```

The predicate *filter_incompatible*/2 is coded as follows:

[3]The predicate `inform/3` is used by the debugger to allow for the name of the rule that has been applied, the formulas involved, and the new node to be displayed on demand.

```
filter_incompatible([], []).
filter_incompatible([G|Gs], Result) :-
    findall_incompatible_to(G, Gs, IGs),
    difference(Gs, IGs, Rest),
    (IGs = [] -> Result = [G|Rest] ; Result = Rest).

findall_incompatible_to(G, Gs, IGs) :-
    findall(IG, (member(IG, Gs), kb_gd__incompatible(G,IG)), IGs).
```

For the generation of non-ground goals, Gorgias needs an extension of its *complement*/2 predicate as follows:

```
complement(gd((L1,T1,TC1)), gd(L2,T2,TC2)) :-
    kb_gd__incompatible(L1,L2),
    overlap(TC1,TC2).
```

where $TC1$ and $TC2$ are of the form $T_{low} < T < T_{high}$ with T_{low} and T_{high} ground timepoints, and *overlap*/2 holds when these two time intervals overlap.

The goal decision capability is called by the GI transition to decide the preferred goals at a given current time, namely T_{now}. This time is used in the rules of the KB_{GD} so that these conditions are evaluated accordingly (see the Users' Manual entry for Goal Decision). Conditions of the goal decision rules of the form $holds\_at(P, T)$ are evaluated using the temporal reasoning capability. The SOCS library arranges for this through the rule

```
holds_at(P,T) :- self__tr(P,T).
```

The GI transition calls the goal decision capability by using the predicate $self\_\_goal\_decision/1$ directly. For example, the call $self\_\_goal\_decision(Goals)$ when the current time $T_{now} = 2$ for the first example given in the Goal Decision section of the users' manual returns:

```
Goals = [(lba,4)]
```

when the conditions $holds\_at(finished\_work, 2)$ and $holds\_at(low\_battery, 2)$ are evaluated true by the temporal reasoning capability.

Planning. Given the computational model for planning put forward in deliverable D8 [58] and the implementation of the C-IFF proof-procedure described earlier in this document, the implementation of the planning capability has been straightforward. In essence, it consists of only a single Prolog clause:

```
plan( KB0, Assumptions, SelectedGoal, TCs, Answer) :-
  switch_triggering( on),
  kb( plan, PlanDefs, ICs),
  close_pred( executed/2, KB0, EX),
  close_pred( observed/2, KB0, OB),
  close_pred( observed/3, KB0, OA),
  close_pred( time_now/1, KB0, TN),
  Defs = [EX,OB,OA,TN|PlanDefs],
  append( [SelectedGoal|Assumptions], TCs, Query),
```

```
ciff( Defs, ICs, Query, Plan:Substitution:NewTCs),
delete( Plan, Assumptions, NewPlan),
Answer = NewPlan:Substitution:NewTCs.
```

`KB0` is a list of (ground) terms of the form `executed(Action,T)`, `observed(Fluent,T)`, and `observed(Computee,Action,T)` as well as a single term of the form `time_now(N)` to communicate the current time (`N` is an integer). `Assumptions` is a list of terms of the form `assume_holds(Goal,T)` and `assume_happens(Action,T)` encoding the goals and actions in the current state. `SelectedGoal`, the goal to plan for, is a term of the form `holds(Goal,T)` and `TCs` is a list of temporal constraints. The variable `Answer` will be instantiated with a representation of the chosen plan if there exists one; otherwise `plan/5` fails.

The first subgoal sets up the C-IFF proof-procedure in such a way as not to unfold certain implications before having applied the propagation rule. This variant of the search strategy is required to handle the "non-allowedness" of parts of the theory used for planning (see [58] for details on this issue). Then the iff-definitions (`PlanDefs`) and the integrity constraints (`ICs`) of KB_{plan} are retrieved. The predicate `close_pred/3` is used to generated iff-definitions for the predicates occurring in `KB0` and these are appended to the list of definitions to obtain `Defs`. Then the C-IFF proof-procedure is called with `Defs` as the background theory, `ICs` as the integrity constraints, and the list of all other relevant terms as the query. The first component of the answer consists of a list of abducible predicate encoding the plan (using `assume_holds/2` for subgoals and `assume_happens/2` for actions). The assumptions (goals and actions) already present in the input need to be deleted from this list. Furthermore, the answer may also include a list of variable substitutions and an updated constraint store `NewTCs`.

An additional predicate (`set_kbplan/1`) is used to read the definition of a planning knowledge base from a file (or possibly a list of files). After completing the respective abductive logic program, iff-definitions and integrity constraints can be retrieved using `kb/3` with the first argument being `plan`. The remaining implementation effort has been spent on providing tools to translate between the representation formalism used within the main computee module on the one hand and the module implementing the C-IFF proof-procedure on the other. For instance, while the former employs a ground representation of existentially quantified variables, the latter relies on actual Prolog variables. Also, only the latter uses the predicates familiar form the abductive event calculus to represent goals and actions. Finally, the translation process also includes the identification of any preconditions of actions generated by the planning capability.

Reactivity. The planning and reactivity capabilities are closely related; the reactivity knowledge base is an extension of the planning knowledge base and both capabilities have been implemented in terms of the C-IFF proof-procedure. Consequently, the implementation of the reactivity capability is very similar to that of planning.

In fact, only three predicates needed to be implemented specifically for reactivity. The central one is `react/4` which is almost identical to the predicate `plan/5` described earlier. The only difference is that there is no selected goal passed to reactivity and hence the query submitted to C-IFF only consists of the list of assumptions (goals and actions in the current state) and the list of temporal constraints. The second predicate provides the interface to the main computee module and handles the translation between the different representation formalism used in exactly the same way as this is done in the case of planning. The third of the aforementioned predicates is `set_kbreact/1` to load the reactive knowledge base KB_{react}, which should include the planning knowledge base KB_{plan}.

Temporal Reasoning. The Temporal Reasoning computational model is based on the "standard" computational model of abductive reasoning in ALP. The current implementation within the SOCSDemo runs over the C-IFF proof-procedure for an appropriately restricted class of temporal reasoning knowledge bases. In this subsection, the current implementation is illustrated with reference to the formal model and the computational model developed in D4 and D8, respectively. Some considerations about how this implementation can be extended in order to cover more general forms of temporal reasoning are given.

How Temporal Reasoning (KB_{TR}) works: Some assumptions have been introduced in order to restrict the general problem of reasoning with time, see D8. Basically, the core framework which has been implemented aims at proving whether a fluent literal, i.e. a property of the world holds namely, whether the goal `holds_at(Fl, T)` is entailed by the temporal reasoning knowledge base, with `Fl` a fluent literal, and `T` a time point. The main restrictions adopted are

- KB_{TR} must provide a possibly partial, but consistent, view of what has occurred in the world, i.e. it can not entail that a fluent holds and does not hold at the same time.
- all the observed facts and executed actions have a ground associated time, and queries are ground (or quantified, in a first extended version) with respect to time.

KB_{TR} consists of

1. a *domain independent* theory, based on Abductive Event Calculus [78], which explains how events cause fluents and how fluents persist in time. For instance, the domain independent rule

   ```
   holds_at(F, T) :-
        happens(A,T'),
        T'  < T,
        initiates(A, T', F),
        not clipped(T', F, T).
   ```

   states that a fluent `F` holds at time `T` if an action `A` has occurred at the previous time `T'`, it causes the fluent and the fluent has not been clipped in the meantime.

2. a *domain dependent* theory, specific of the domain at hand, like

   ```
   initiates(switch_on, T, light) :-
        holds_at(neg(broken_bulb), T).
   ```

   putting the switch on causes light if the bulb is not broken.

3. a *narration*, contained in KB_0, representing facts that are known to be occurred, like

   ```
   executed(switch_on, 10).
   ```

4. the *consistency* integrity constraint

   ```
   [holds_at(F, T), holds_at(neg(F), T)] implies false.
   ```

which is a C-IFF implication preventing a fluent and its negation to hold at the same time.

Implementation of the basic computational model: With the above mentioned simplifying assumptions, the so-called "deserts and oases" approach has been adopted in order to overcome some difficulties with reasoning with universal quantification of time variables, as presented in D8. Moreover, the approach has revealed to be suitable for supporting the extensions of the computational model obtained by relaxing some of the initial assumptions. The implementation described here refers to the basic cases of reasoning (*credulous* and *skeptical*), on which the current implementation of the SOCSDemo is based.

According to the "deserts and oases" approach, the consistency integrity constraint is grounded into the significant points of the time line, called oasis, i.e. all the points where something significant has occurred. Results from D8 guarantee that this is a sound and complete transformation. Then, the so constructed theory is processed by the C-IFF proof-procedure.

Let $< \mathtt{P_{TR}}, \mathtt{A_{TR}}, \mathtt{I_{TR}} >$ be the abductive logic program, where $\mathtt{P_{TR}}$ is the domain dependent and independent part of KB_{TR}(in the C-IFF syntax), $\mathtt{A_{TR}}$ consists of the abducible `assume_holds(F,O)` and $\mathtt{IC_{TR}}$ of the consistency integrity constraint, $\mathtt{KB_0}$ is the current narration and `holds_at(F,T)`. is the query fluent `F` which is required to hold at time `T`.

The general schema of the main predicate implementing the Temporal Reasoning capability is as follows:

```
query_credulous_TR((PTR,ATR,ITR),KB0,holds_at(F,T),Answer) :-
      extract_oases(KB0,O),
      instantiate_constraints(ITR,O,OITR),
      prove_credulously_TR((PTR,ATR,OITR),KB0,holds_at(F,T),Answer).

prove_credulously_TR((PTR,ATR,OITR),KB0,holds_at(F,T),Answer) :-
      set_up_theory((PTR,ATR,OITR),KB0,Defs,ICs),
      ciff(Defs,ICs,holds_at(F,T),Answer).
```

The `extract_oases(KB0,O)` predicate extracts the significant points, the oases, from the narration, according to the definition given in D8. The predicate `instantiate_constraints(ITR,O,OITR)`, makes the grounding of the (universally quantified) consistency integrity constraint on such significant points. The predicate `set_up_theory((PTR,ATR,OITR),KB0,Defs,ICs)` compiles the theory in a form that can be passed to the C-IFF proof-procedure. Actually, differently from this abstract schema, only the narrative KB_0 is passed as a parameter to each call of TR. The rest of the TR theory is pre-consulted and dynamically accessed. Finally, the predicate `ciff(Defs,ICs,holds_at(F,T),Answer)` calls the C-IFF proof-procedure.

The call schema for skeptical reasoning, follows straightforwardly from its definition in D4:

```
query_skeptically_TR(KBTR,KB0,Goal,Answer) :-
      query_credulously_TR(KBTR,KB0,Goal,Answer),
      negate_goal(Goal,NegGoal),
      not query_credulously_TR(KBTR,KB0,NegGoal,_)
```

where `NegGoal` is the goal relative to the fluent that is the negation of the fluent in `Goal`.

How Temporal Reasoning is called: The core functioning of TR is exploited within a computee by other capabilities, transitions and cycle theory. TR exports two different main predicates that can be used by them, according to their needs. Basically, those which do not require TR to return an answer. e.g. the current implementation of Goal Decision and the preferential theory of the main Cycle Theory, access TR by means of the following call to skeptical reasoning

```
ground_temporal_reasoning_skeptical(KB0, Goal).
```

Note that only KB_0 and a ground fluent as a Goal are passed to TR. No answer is required, since it is only necessary to check whether the goal is or is not entailed by the theory and the current narration. In the first case the call simply succeeds, otherwise it fails.

On the other hand, some components of the computee model need to ask TR whether a fluent holds at an existentially quantified time point within an interval, possibly subject to further temporal constraints from the state of the computee. In this case, TR provides the call

```
temporal_reasoning_skeptical(KB0, EqvGoal, EqvTCs, EqvSigma).
```

where `KB0` is the narrative, `EqvGoal` is the goal, `EqvTCs` is a set of temporal constraints existential variables, and `EqvSigma` is an assignment for these variables, as they are represented in the state of the computee (TR performs an appropriate translation into its own syntax). Note that the present call differs from the basic temporal reasoning in that the query is not necessarily relative to a ground time point, and, also, in that the temporal reasoning is intertwined with the satisfaction of constraint sets (dealt with by exploiting, in a way compatible with the temporal reasoning, the C-IFF constraint solver).

Constraint Checking. This module is implemented in a straightforward call to the C-IFF system as follows:

```
check_constraints( EqvTCs ) :-
   eqvs_to_vars( EqvTCs, TCs, _ ),
   ciff( [], [], TCs, _ ).
```

In other words, to check the consistency of a list of constraints in the form of existentially quantified variables (described in section 4.1.2) we transform them to C-IFF constraints first and then we use C-IFF to solve them. The predicate `check_constraints/1` succeeds if the list of constraints is consistent.

4.1.5 Transitions

We use capabilities and the proof-procedures they rely upon to write state transition rules for the computee. Transitions are implemented with rules of the form:

```
<nameOfTransition>(<Parameters>) :- Conditions.
```

To give an example of how a transition is implemented, we present here the definitions of the Goal Revision (GR) transition. Whenever this transition is called, it enables the computee to revise its goals. This is done by building the new goals for both sub–trees (the one holding the reactive goals and the one holding the non-reactive goals) by following conditions (i), (ii), and (iii) of its specification in D4 [57], further updating the state with these new goals at the current `TimeNow` of the system, which is supplied as input. The following Prolog code implements this transition:

```
goal_revision(TimeNow) :-
        reactive_root(RootR),
        non_reactive_root(RootNR),
        select_children([RootNR,RootR], TopLevel),
        revised_goals(TimeNow, TopLevel, [], NewGoals),
        update_goals(NewGoals).
```

In the definition above, the first two predicates access a description of the roots for the sub-trees of goals (`reactive_root/1` and `non_reactive_root/1`), while the third predicate evaluates the `TopLevel` goals by looking at the children of the roots using `select_children/2`. The `TopLevel` goals are then revised in order to build up the `NewGoals`. This is achieved by using the predicate:

```
revised_goals(Now, Tocheck, Goals, Goals) :-
        persist_goals(Now, Tocheck, [], []), !.
revised_goals(Now, Tocheck, GoalsSoFar, NewGoals) :-
        persist_goals(Now, Tocheck, [], Persistent),
        append(GoalsSoFar, Persistent, IntermGoals),
        select_children(Persistent, Children),
        revised_goals(Now, Children, IntermGoals, NewGoals).
```

`revised_goals/4` builds the new revised goals from the current state incrementally (i.e. only those that whose parent is in the list of the new goals persist). In addition, they should also still have not been achieved, and they should not be timed out. These checks are carried out by the predicate :

```
persist_goals(_, [], Goals, Goals).
persist_goals(Now, [G|Rest], Goals, NewGoals) :-
        goal_not_achieved(Now, G),
        goal_not_timed_out(Now, G), !,
        persist_goals(Now, Rest, [G|Goals], NewGoals).
persist_goals(Now, [_|Rest], Goals, NewGoals) :-
        persist_goals(Now, Rest, Goals, NewGoals).
```

In other words, a goal persists if it has not been achieved or if its time has not run out; otherwise, the goal is ignored in the new state (this is the implementation of the condition (i) of the specification of the GR transition in D4). We then define a goal as not achieved as follows:

```
goal_not_achieved(Now, Goal):-
        \+ goal_achieved(Now, Goal).

goal_achieved(Now, Goal):-
        time_of(Goal, T),
        get_TCS(TCS),
        append(TCS, [T #< Now], TCS_U_Now),
        kb_0(KB0),
        get_Sigma(EqvSigma),
        temporal_reasoning_skeptical(KB0, Goal, TCS_U_Now, EqvSigma).
```

The definition of how a goal is achieved (second predicate above) shows how we call the capability of temporal reasoning to check the constraints (this is in fact how the implementation ensures that the condition (ii) of the specification of the GR transition as defined in D4).

Similarly, the representation of a goal that is not considered timed out is given by the definition:

```
goal_not_timed_out(Now, Goal):-
         time_of(Goal, T),
         get_TCS(TCS),
         append(TCS, [T #> Now], Constraints),
         check_constraints(Constraints).
```

The definition above ensures, using the constraint solver of the system (`check_constraints/1` uses C-IFF to access the SICStus constraint solver), that the global constraint store (TCS), together with the temporal constraints of the goal are satisfiable (this is in fact how the implementation of condition (iii) of the GR transition specification is represented).

Once all the new goals have been generated, the system updates the new goals in the state using the definition:

```
update_goals(NewGoals):-
        retractall(goal(_,_,_)),
        assert_list(NewGoals).

assert_list([]).
assert_list([One|Rest]):-
        assert(One),!,
        assert_list(Rest).
```

In other words, we first clear all the goals from the state of the computee and then we add the new goals.

The rest of the transition are implemented in a similar manner, following the specification of D4 and the computational model of D8.

4.1.6 Cycle Theory

Cycle Theories for Computees. The operation of a computee is controlled by its cycle theory. Cycle theories define in a declarative way the possible alternative operational traces of the internal transition of a computee depending on the particular circumstances of the external environment at the time of the operation. Their role is not to provide absolute control on the operation but to regulate the operation and to provide a desired pattern of behaviour.

A cycle theory is a logic program with priorities in the $LPwNF$ framework that specifies a preference policy on how to choose the next transition under the current circumstances. Hence the reasoning of a cycle theory is the same preference reasoning as in other parts of the model, e.g. in the goal decision capability. But now this is carried out at the meta-level, reasoning on the whole state of the computee and using other capabilities, such as temporal reasoning, in an auxiliary way.

The cycle theory is called through the outer computee shell and is executed by Gorgias to give the next internal transition of the computee. This is done through the top-level rule of:

```
self__next(step(Transition,Input)) :-
        self__prove(step(Transition,Input)).

self__prove(X) :-
        gorgias__prove(X).
```

The execution of this query, namely $step(Transition, Input)$, depends on the given cycle theory of the computee and its current state. Note that this is a simplified execution where we choose the first transition that can be proved admissibly by Gorgias from the cycle theory. In general, we can find all the admissible next transitions and choose randomly one from them, as specified by the computational model of cycle theories in [58].

In this execution of Gorgias for cycle theories, the notion of conflicting rules is defined via means of a specialised $complement/2$ predicate that states that any two transitions are incompatible. Specifically,

```
complement(step(X,_),step(Y,_)) :-
        ct__incompatible(X,Y).

complement(step(X,Input_1),step(X,Input_2)) :-
        ct__istransition(X),
        ct__ec(X,Input_1),
        ct__ec(X,Input_2),
        Input_1 \= Input_2.
```

Note that, if we want to extend the computational model of cycle theories to allow concurrent execution of different transitions, we need to adapt appropriately the definition of this predicate.

A cycle theory consists of three components:

- A *basic* part that determines the basic steps of operation by specifying the allowed unitary cycle-steps from one transition to the next one.
- An *interrupt* part that specifies the cycle-steps that can follow a passive observation introduction, i.e. an interrupt with new information. These are viewed as (possible) re-initialization steps for the cycle operation.
- A *behaviour* part specifies priority rules on the alternatives given in the basic and interrupt parts, and thus specifies the special characteristics of the operation of the computee[4].

The basic part of any cycle theory consists of rules of the form:

```
ct__rule(step(Transition, Input), step(Transition, Input), []) :-
        ct__ec(Transition, Input).
```

where the first argument $ct\_rule/3$ is a label that names this rule which can be chosen by the user as any Prolog term. In practice though it is important to have in this the name of the transition and its parameters chosen by this rule. Hence we are going to adopt the convention to use the head of the basic cycle step rule, which is the second argument, also as the name of the rule so that we carry all relevant information of the rule in its name. One example of such rule that states that GI is a possible next transition step when currently there are no unplanned goals is given below:

[4] At the moment a computee has a fixed cycle theory given to it at design time but we envisage that computees will be able to change their cycle theory during their operation.

```
ct__rule(step('GI', []), step('GI', []), []) :-
        ct__ec('GI', []).
```

The enabling conditions *ct__ec*/2 in this case check that there are no unplanned goals. Its definition is given by:

```
ct__ec('GI',[]) :-
        ct__ec_aux_p('GI',[]).
```

where *ct__ec_aux_p*/2 is a general predicate that checks whether the selected step is allowed based on a given previous transition. This is coded as follows:

```
ct__ec_aux_p(Transition,Input) :-
        self__timestamp_current(T1),
        T0 is T1 - 1,
        self__history(T0,step(Prev_Transition,Prev_Input)),
        ct__ec_user_p(Prev_Transition, Prev_Input, Transition, Input).
```

where it first retrieves the previous transition, namely *Prev_Transition*, from its internal history and then checks that *Transition* is allowed to follow *Prev_Transition* via the user-defined predicate *ct__ec_user_p*/4. In our example, we have the rule

```
ct__ec_user_p(_, 'GI') :-
        findall(G, fun__goal_selection(G),[]).
```

which states that GI may follow any transition provided that there are no unplanned goals as mentioned earlier.

Another example of a basic cycle-step rule that enables Plan Introduction as the next transition is given by:

```
ct__rule(step('PI', Gs), step('PI', Gs), []) :-
        ct__ec('PI', Gs).


ct__ec('PI',Gs) :-
        fun__goal_selection(Gs),
        ct__ec_aux_p('PI',Gs).
```

where *fun__goal_selection*/1 is a core selection function that selects from the current state an unplanned goal or subgoal that is used as an input to the PI transition. It is defined as follows:

```
fun__goal_selection(((Goal,GT),Parent_Goal,TC)) :-
        self__goal((Goal,GT),Parent_Goal,TC),
        self__temporal_constraints_validate(GT,TC),
        self__goal_ancestors_eq(Parent_Goal,Ancestors),
        self__goal_check_ancestors_nopass(Ancestors).
```

where (a) the first conditions picks a goal from the state of the computee via the predicate *self__goal*/3, then (b) checks that the temporal constraints are still satisfied, and (c) the last two conditions check that none of the goal's ancestors have been satisfied already.

The interrupt rules of the cycle theory are analogous, in syntax, to the basic rules. However, each interrupt rule specifies what might follow a PO transition, which acts as an interrupt. This is again accomplished as follows:

```
ct__ec_user_p('PO','GI').
ct__ec_user_p('PO','RE').
ct__ec_user_p('PO','GR').
```

The Behaviour Part of Cycle Theory. The behaviour part of the cycle theory consists of priority rules whose role is to encode locally the relative strength of the rules in the other components of the cycle theory of the computee. These are then used to determine, amongst all the enabled cycle-steps, which ones are preferred under the current circumstances. Through the behaviour part of the cycle theory we can encode different patterns of operation.

Behaviour rules are Gorgias rules of the following form:

```
patt__rule(prefer(Pattern_Name,step(Tr1,I1),step(Tr2,I2)),
    prefer(step(Tr1,I1),step(Tr2,I2)), []) :-
        behaviour_conditions(Tr1,I1,Tr2,I2).
```

where $prefer(Pattern\_Name, step(Tr1, I1), step(Tr2, I2))$ is the label that names this rule which can be chosen by the user as any Prolog term. As above for basic rules, it is important to have in this the name of the transitions involved and their parameters. Hence we are going to adopt the convention to use in this name both steps involved together with their parameters and in addition use the pattern name in order to be able to define general higher-order priority rules.

The behaviour conditions are heuristic conditions (e.g. heuristic selection functions) under which the cycle step $step(Tr1, I1)$ is preferred over $step(Tr2, I2)$.

For example, the careful pattern of behaviour which gives priority to Plan Revision over any other transition when the current state contains timed out actions will have the following priority in its behaviour pattern:

```
patt__rule(prefer(careful,step('PR',[]),step(Z,X)),
    prefer(step('PR',[]),step(Z,X)), []) :-
        fun__action_timeout,
        Z \= 'PR'.
```

Similarly, by giving priority to cycle steps of Active Observation on the effect of an action after an Action Execution transition we have a cautious pattern of behaviour where the computee attempts to get explicit confirmation of the successful execution of its actions. This latter preference is captured by the behaviour rule:

```
patt__rule(prefer(cautious,step('AO',Fs),step(Z,X)),
    prefer(step('AO',Fs),step(Z,X)), []) :-
        self__last_transition('AE',As),
        fun__action_effect(As,Fs),
        Z \= 'AO'.
```

where $fun\_action\_effect/2$ returns a set of fluents which are the desired effects of the last executed action. This condition can be restricted further to apply only for some types of actions which are typically unreliable under the present conditions when the action was executed.

A pattern of operation that can be taken as an underlying basis on which we can build different and additional patterns of behaviour is what we can call the *normal* pattern of behaviour. This specifies a pattern of operation where the computee prefers to follow a sequence

of transitions that allows it to achieve its goals in a way that matches an expected "normal" behaviour. Basically, it introduces goals , then plans for them, executes a plan, revises the state, executes another plan until all goals are dealt with (successfully completed or revised away) and then returns to introduce new goals. See deliverable D8 for its full definition.

This pattern contains behaviour rules to capture that after Goal Introduction we plan for the goal introduced via Plan Introduction and that after the last possible Action Execution we prefer to do a revision of the state.

These rules are coded respectively as follows:

```
patt__rule(prefer(normal,step('PI',Gs), step(Z,X)),
    prefer(step('PI',Gs), step(Z,X)), []) :-
        self__last_transition('GI'),
        Z \= 'PI'.

patt__rule([refer(normal,step('GR',_), step(_,_)),
    prefer(step('GR',_), step(_,_)), []):-
        self__last_transition('AE'),
        empty_plan.
```

The condition, *empty_plan*/0, used in this example is a behaviour condition that is true whenever there are no actions in the state of the computee, i.e.:

```
empty_plan :-
    findall(A, self__action(A), As),
    As = [].
```

Under the normal pattern of behaviour we also give preference to responding to communication messages received by a computee as passive observations via the PO transition. This is captured by the behaviour rule:

```
patt__rule(prefer(normal,step('GI',_), step(_,_)),
    prefer(step('GI',_), step(_,_)), []) :-
        self__last_transition('PO', Obs),
        comm_msg(Obs).
```

which gives preference to the Goal Introduction transition that through its goal decision capability will decide the response to *Obs*. In addition, in the normal pattern of behaviour the rule

```
patt__rule(prefer(normal,step('AE',As), step(Z,X)),
    prefer(step('AE',As), step(Z,X)), []) :-
        self__last_transition('PI'),
        Z \= 'AE'.
```

gives preference to Action Execution (AE) after a Plan Introduction (PI) transition while the rule

```
patt__rule(prefer(normal,step('AE',As), step('AE',As_2)),
    prefer(step('AE',As), step('AE',As_2)), []) :-
        comm_action_selection(As).
```

states that amongst possible actions to be executed, the communication actions are those preferred. Furthermore, the higher-order priority

```
patt__rule(prefer(normal_ho_pref), prefer(Pref_1, Pref_2)) :-
    Pref_1 = prefer(normal,step('AE',As_1), _),
    Pref_2 = prefer(normal,step('AE',As_2), _),
    more_urgent(As_1, As_2).
```

ensures that the more urgent communication action is preferred.

Here *comm_action_selection* and *more_urgent* are heuristic selection functions that contribute to the behaviour conditions of the pattern specified by the cycle theory. These are auxiliary predicates defined by Prolog rules. For example, *comm_action_selection* is defined by:

```
comm_action_selection(A) :-
    self__action((OP,T),_,_,_),
    OP = tell(_,_,_,_).
```

4.2 The Implementation of the Body

The body of a computee in PROSOCS is implemented as a Java process. This process connects the mind of the computee with the electronic environment that the computee is situated in. The connection of the body with the environment is achieved by importing the Medium API that we shall describe in section4.3. We outline in this section how the body is implemented and we concentrate on the representation of the control loop used by the body. We also discuss how a graphical user interface is used to animate useful information to a user that uses the platform.

4.2.1 Body

When a computee is created in PROSOCS the system creates a generic body which uses the Medium API to import the functions of sensors/effectors. The system also creates a mind, which is a Prolog process, appropriately instantiated with a mind state. Creating a body also implies creating a body state, containing information about the mind of the computee, structures that contain information for the interfacing of the mind with the body, as well as configuration information (such as the name of the file containing the KB underlying the mind). The body also contains a new thread of control represented by the pseudo-code:

```
private void bodyControl() {
    do {
        BodyAction nextAction = askActionFromMind();
          if (nextAction != null)
             switch (isOfActionType(nextAction)){
                 case SENSING: doSee(nextAction);break;
                 case COMMUNICATIVE: doSpeak(nextAction);break;
                 case PHYSICAL: doEffectors(nextAction);break;
             }
        Percepts nextPercepts = sensors.passiveObservation();
          if (nextPercepts != null) tellMind(nextPercepts);
```

```
   } while (!stopped);
}
```

The pseudo-code above shows how the body control of a computee interfaces the mind with the environment. The loop starts by asking the mind for the next action to be executed. This causes the mind to call the cycle theory for the next transition to be executed. If such a transition exists, then this transition will be called to change the state of the mind. The body then checks to see if the transition has generated a new action. If it has, the body will carry it out by invoking the appropriate effector. In any case, the body will also perform a passive observation to see if there are any events that the sensors have recorded as a result of events happening in the environment. If any, the body will inform the mind about the events through its percepts. This process will go on forever, until the computee is stopped, via the user interface.

4.2.2 GUI

The graphical user interface is designed to animate useful information about the body and the mind of a computee in the platform. It provides functions to create and manage a particular computee, as well as animate the operational status of other computees in the environment. A screen-shot of the current status of a computee's graphical user interface is shown in Figure 7.

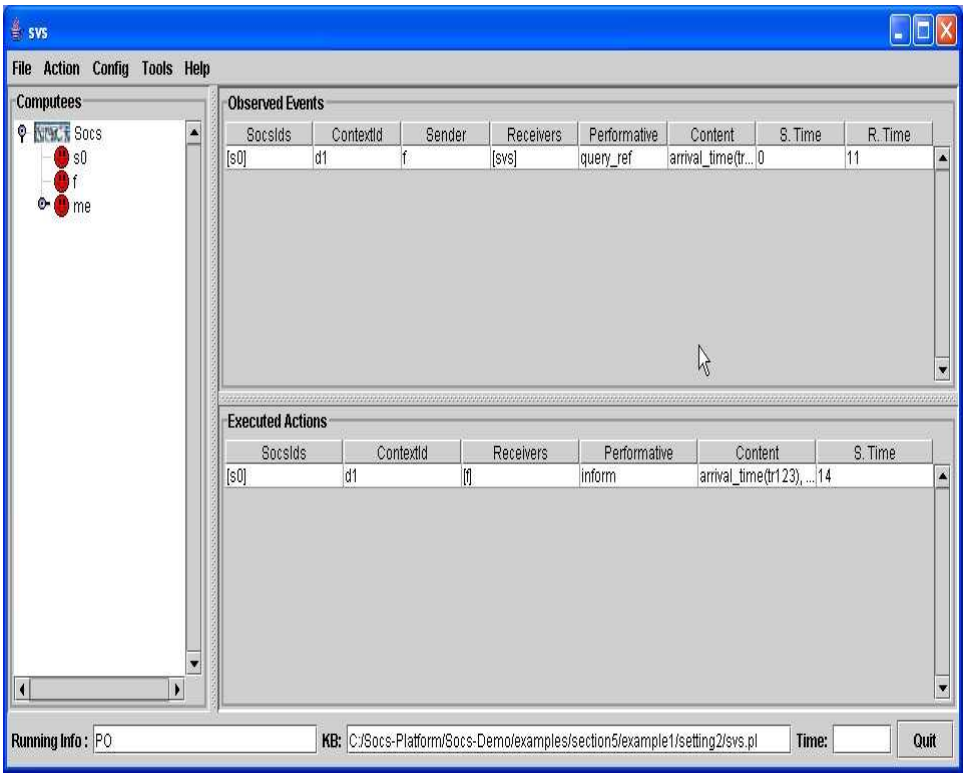

Figure 7: The GUI of a Computee

To facilitate users create a computee, the interface provides a configure function that allows a user to select a knowledge base. Once the mind (knowledge and cycle theory) is selected, a computee is created. The computee can then be started, and subsequently suspended, resumed and stopped through the menus provided by the interface.

The computees present in the environment and status information are animated in the top left part of the interface through icons labelled with their name. Different icons represent different statuses for a computee. When a computee's status changes, its status icon will automatically change on the other computees' interface.

To help the user to trace computees interactions, messages exchanged between the computee and other computees in the SOCS group are also displayed. The top right part of the window displays the messages observed by the computee `svs` (in this case that a computee called `f` has requested from `svs` information about the arrival time of a train `arrival_time(tr123)`). The bottom right part of the window displays actions executed by the computee (in this case that `svs` has sent a message to inform `f` about the arrival time of the train).

4.3 The Implementation of the Medium

As we discussed in section 3.4, the medium provides functionality that allows a computee to access other computees in a networked and distributed environment. In order to support the implementation of the body with the environment, we have developed an API that supports the functions discussed in section 3.4, viz., `speak`, `listen`, `do` and `see`, by building on top of the Peer-to-Peer (P2P) platform JXTA [56, 93]. In this section we first provide a brief introduction to the JXTA project and we then illustrate how the Medium API is built on top of this global platform .

4.3.1 The JXTA Project

The open-source Project JXTA [56, 37, 93] is the industry leading peer-to-peer (P2P) platform originally conceived by Sun Microsystems Inc. and designed with the participation of a small but growing number of experts from academic institutions and industry. The Project JXTA protocols establish a virtual network overlay on top of the Internet and non-IP networks, allowing peers to directly interact and self-organize independently of their network connectivity.

Project JXTA standardizes a common set of protocols for building P2P virtual networks. The JXTA protocols defines the minimum required network semantic for peers to form and join a virtual network. The Project JXTA protocols define a generic network substrate usable to build a wide variety of P2P networks. Project JXTA enables application developers, not just network administrators to design network topology that best match their application requirements. Multiple ad hoc virtual networks can be created and dynamically mapped into one single physical network.

In PROSOCS we use the Project JXTA 2.0 implementation that builds upon the virtual network abstractions introduced in JXTA 1.0 [37]. A *peer* is any network device that implements one or more of the JXTA protocols. Each peer exists independently and asynchronously from all other peers, and uniquely identified by a *peer ID*. The notion of *peergroups* let peers dynamically self-organize into protected virtual domains. Peers in peergroups use *advertisements* to publish peer resources (peer, peergroup, endpoint, service, content). In addition, a universal binding mechanism, called the *resolver* performs all binding operations required in a distributed system.

Finally, the notion of *pipes* is used as virtual communication channels enabling applications to communicate between each other.

4.3.2 The PROSOCS Medium as an API

In the current prototype of PROSOCS, all interactions amongst computees are implemented via JXTA communication. The messages that are exchanged for this kind of communication are represented as XML documents. The example below shows a message for communicative actions in PROSOCS:

```
<?xml version=''1.0'' ?>
<socsmsg>
    <socsId> society0 </socsId>
    <contextId> dialogue14 </contextId>
    <sender> computee5 </sender>
    <receivers> computee7 </receivers>
    <performative> request </performative>
    <content> tel(john, X) </content>
    <time> 1 </time>
</socsmsg>
```

A `socsId` holds the society for which this communicative act is valid. The medium will convey the message to this society only; if there is no society specified, the message is delivered as a private message between computees. `ContextId` is an identifier that represents the context of the communication. `sender` is a unique identifier representing the computee that performs the act, while `receivers` is a list of intended receivers that can listen to the act. `performative` is the performative that characterises the act, whose content kept in the `content` has been performed at a specific time held in the `time` attribute.

To guarantee the safe interactions among computees a default PROSOCS group has also been created, forming a logical region whose boundaries limit access to non-PROSOCS peers. The resulting Medium API is then implemented as a new service called `SocsService` in the PROSOCS group to support sensing and action execution via effectors, see Fig. 8. For instance, the effector `speak` is implemented by a method `speak(SocsMsg msg)`, which takes an instance of a `SocsMsg` as a parameter and sends it to the computee(s) whose name(s) appear(s) in the receiver field.

5 Implementation of Societies

As with Section 4.1, we will start this section by presenting some design choices that we made when implementing the models of D8.

In D5 [66] and subsequent documents, we assume that the society will be aware of "socially relevant events", which in particular could be communicative actions that unify with atoms in the body of integrity constraints. An issue is how to decide which events are relevant, and how can the society be aware of such events, without being intrusive in the computees' private behaviour. Also, a computee being in some physical environment does not necessarily mean it abiding to some social rules (or it being expected to do so). And, in principle, computees might be into several virtual societies at the same time. Some of them could be interested in only a subset of its messages exchanged with other computees.

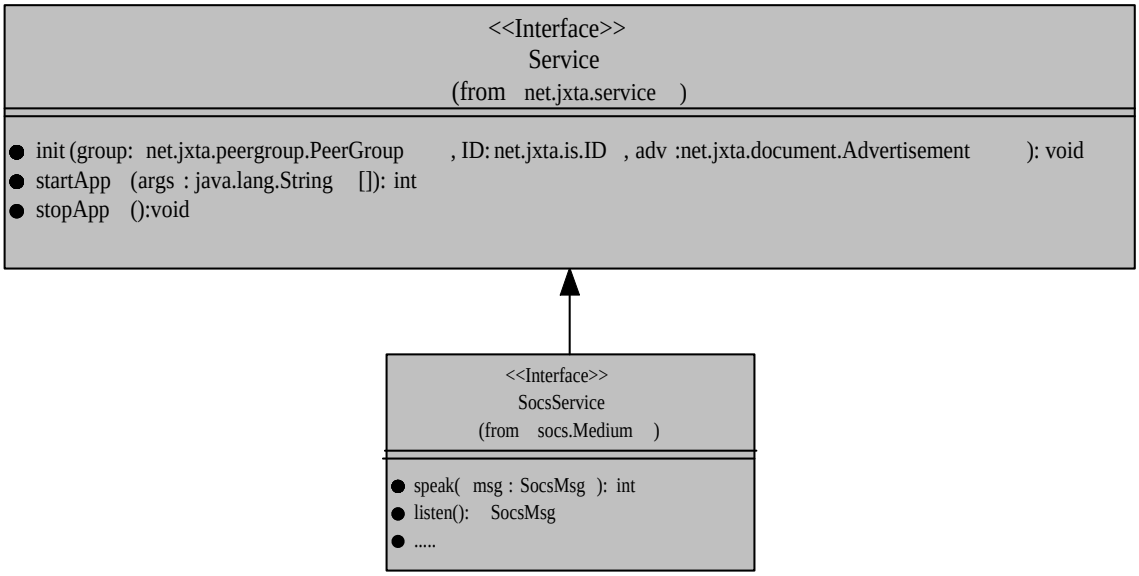

Figure 8: PROSOCS Service Interface

The solution that we adopted in the implementation is: computees will explicit direct the relevance of messages to some society. That is, there is an element in a communication act, which specifies a society identifier. Messages are dispatched through the medium: the medium will be in charge of sending the message both to the addressees (computees) and possibly to the specified society.

In WP2 and subsequent documents, we address the issue of membership and openness of societies. We report a classification of open societies, due to Paul Davidsson [29], where four degrees of openness are identified. In PROSOCS, we made the choice of implementing open societies, which have no intermediate steps that computees require to do before they become member. In fact, when running the software, the society will consider new members those who utter messages that are socially relevant for some specific society. We did not implement other kinds of openness, but we show in the examples document how to implement semi-open societies, by way of a gatekeeper computee and a protocol to enter the society.

As far as the time, we are aware that it is impossible to implement a global time in a distributed system, but we make the assumption that some 'small' delay in the delivery of messages does not affect the correct behaviour of the society. For this reason, in the examples that we test that contain deadlines, we consider large enough time periods. We do not propose a precise threshold for this.

A couple of noteworthy design choices are about the implementation of the proof. Firstly, in the current implementation it is not possible to define social integrity constraints with a negated **H** predicate in the body, as they are not supported. Such constraints are not used in the examples of the demo. Secondly, for efficiency reasons, the *propagation* step of the implemented *SCIFF* makes a choice on the constraints containing two atoms that can unify with each other. For instance, if both $\mathbf{H}(p(X), T_1)$ and $\mathbf{H}(p(Y), T_2)$ are in the body of a social

integrity constraint, during the propagation step the only branch which is considered is that resulting by the unification $X/Y, T_1/T_2$. Again, this limitation does not affect the currently implemented examples. Finally, the *SCIFF* (as well as the IFF), does not open a unification branch in the unfolding of existentially quantified variables.

5.1 Overall Architecture

The *Society Infrastructure* software application is realised as a set of modules and sub-modules; each module is characterised by a specific Application Program Interface (API). All the software components are implemented by using the Java language. Every module and sub-module is realised as a set of one or more Java classes.

Only one component is not implemented as a Java class: this module, named "Proof Procedure", is written in the Prolog language. Due to its features, this module is better implemented by using a logic programming language. To execute the component, a "run-time" version of a SICStus Prolog [80] is embedded into the societies.

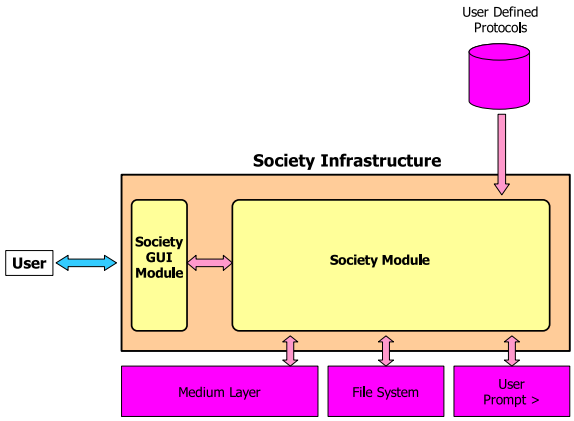

Figure 9: Overview of the *Society Infrastructure*

During the initialization the software prototype receives as input a set of protocols. The protocols are defined by the user, which specifies them using a protocol definition language. The proof-procedure later uses the protocols in order to check the compliance of computees to the society rules. During the initialization the user can also select a different proof procedure, in order to obtain different features and characteristics.

At runtime, the prototype receives messages from a selected source, it elaborates them and publishes the results of such elaboration through a Graphic User Interface (GUI).

All the software components are members of two main modules, depicted in Figure 9. The Society module is the component responsible for elaborating the messages, and for evaluating the compliance of the computees to the social protocols. The GUI module instead is devoted to visualizing the results of such elaboration. It also provides the user a suitable way to interact with the application and to control it.

Thanks to this clear and neat separation between the two parts, it is possible to implement

new interfaces as well as new elaboration modules, by implementing the interfaces between them. Components don't need to have any knowledge about the internal architecture/implementation of each other, as long as they implement the interfaces.

5.2 The Society Module

The Society module is realised as a set of independent and collaborating software components. Some of these components are implemented as "active" components (here by "active" we simply mean that they are implemented as Java threads). Other modules are instead "passive", in the sense that they offer services to components but they are not independent threads.

Each module is coded into one or more Java classes, but in this section we will describe the software components from the functionality view point. Technical details are reported in the SOCSDemo application manual, attached to the present document.

The core of the Society module is composed by three main sub-modules (see fig. 10), namely:

- *Event Recorder* (fetches messages from different sources);
- *History Manager* (receives events from the *Event Recorder* and composes them into a "history");
- *Social Compliance Verifier* (checks for compliance of the history of events to social integrity constraints);

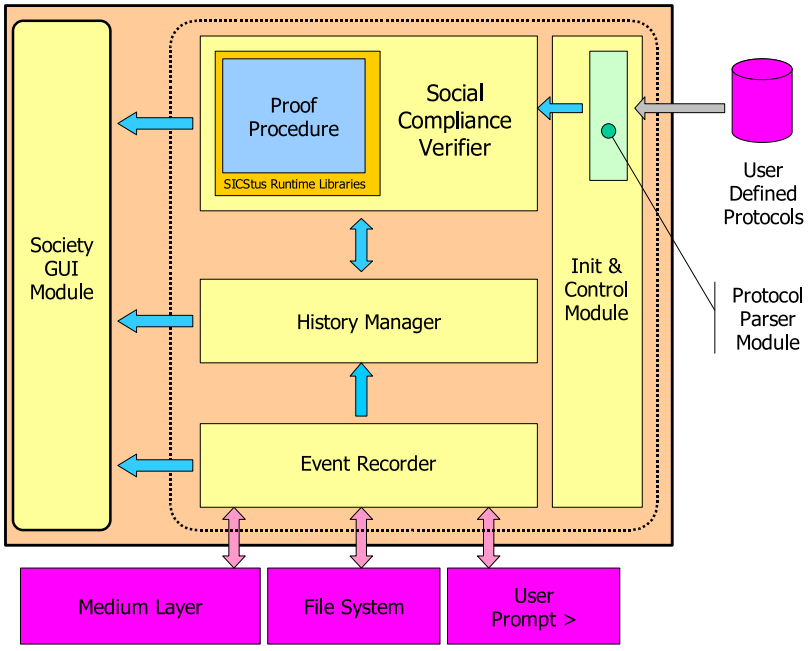

Figure 10: Internal composition of the Society module

Init & Control is a fourth module, devoted to initializing the other components in the right order. It also parses the society protocols defined by the users and stored in a file, by using the protocol parser module. The protocols are specified using an easily readable protocol definition language, adapted from D5; they are coded in an internal format. In Appendix 10.1 we describe the protocol definition language using an EBNF syntax notation.

The *Init & Control* module provides also some control methods and act as a proxy object, called whenever a GUI component needs to communicate to one of the Society module components.

The biggest advantage of using this approach is that we can simply use different GUIs without changing the other modules. Symmetrically, we can realize different proofs and modules without changing the GUI. To keep things simple, this organization has not been depicted in Figure 10, where the communications between the internal modules of the Society component and the GUI have been drawn as "direct" arrows, by hiding the proxies.

5.2.1 The Social Compliance Verifier

The *Social Compliance Verifier* is the software component responsible for the interaction between all the Java components and the proof-procedure. The proof-procedure is realised as a program written in SICStus Prolog [80], and it is executed through the SICStus Runtime libraries. The main task of the *Social Compliance Verifier* consists on interacting with these libraries, as it is shown in more detail below. Other tasks of this component are: *a*) to allow the proof-procedure to retrieve new happened events; *b*) to allow the proof to print out the computational state whenever desired.

The *Social Compliance Verifier* is implemented as a Java thread: first of all it initialises the SICStus runtime engine, then it sees to loading and correctly "starting" the proof procedure. The last step means also to "load" the user-defined protocols (already parsed by the Init&Control sub-module).

From this moment the control of the execution is taken over by the proof-procedure. The logic program calls back the *Social Compliance Verifier* for two different services: *a*) to fetch new events to elaborate, and *b*) to communicate through the GUI useful information about the computation.

All the data exchanged between the *Social Compliance Verifier* and the proof-procedure are type String. Regular expressions have been used in order to effectively treat these data on the Java side. These expressions are stored in a configuration file. It is indeed possible to change the proof procedure and the data exchanged, provided that the regular expressions are re-defined properly. No other assumptions are done about the data exchanged between the *Social Compliance Verifier* and the proof-procedure.

5.2.2 The History Manager

The *History Manager* is the software component in charge of keeping track of all the happened events. It stores internally the set of all the events received by the *Society Infrastructure*. It also provides access to this set through different methods. These methods are called by two other blocks, the *Social Compliance Verifier* and the *Event Recorder*. The *History Manager* is mainly a passive block, in the sense that it is not realised as a Java thread; it simply provides services for storing and for retrieving happened events. It can be viewed as a buffer where all the events are stored.

Every time a new happened event is received, the *Event Recorder* sees to it that it gets stored in the *History Manager*. As soon as a new event is stored, the GUI module gets notified, and the graphic interface is updated accordingly.

Similarly, the *Social Compliance Verifier* fetches the events (one at a time) whenever it finishes the elaboration of the previous event; again the GUI is properly updated. Please note that the *History Manager* contains all the events, both the ones already processed by the *Social Compliance Verifier*, and the events still to be elaborated. The messages window in the GUI shows the content of the *History Manager*.

The history manager has been introduced because the *Social Compliance Verifier* has a very different performance (in terms of execution time) from the *Event Recorder*. To speed up the execution and taking advantage of multi-threading environment we needed to have two asynchronous components. The synchronization with the *History Manager* is determined by the availability of events in the buffer.

5.2.3 The Event Recorder

The *Event Recorder* is the software component that interacts with the external message sources. This interaction consists of fetching the messages when they arrive and translating them as events; then each event is stored inside the *History Manager*, where it becomes available to the *Social Compliance Verifier* in order to be processed.

Outside of the *Society Infrastructure* all the information is exchanged using the concept of "message" (see Section 4.3). Inside the Society Infrastructure each message is translated in the concept of "event".

There is a subtle distinction between message and event: the message represents the information exchanged between two computees, while the event represents the fact that a communication act happened between these two computees.

Whenever a new message arrives (or a new message is read from a file, or the user types a new message at the prompt), the *Event Recorder* does the following:

a) it fetches the message,

b) it translates the message into an event object and,

c) it records the event object in the *History Manager*.

It is possible to configure the *Event Recorder* so that it checks if the arrival order of the messages respects the sending time of the messages itself. In fact, due to the "distributed" nature of the prototype, it is possible that some message are received in a different order from the order in which they have been sent. In the current implementation, messages arriving in the wrong order are discarded.

The*Event Recorder* uses specific software modules to interact with different message sources. Until now we have identified and implemented three different sources of the messages: the first source is the communication medium. The second source is a text file with all the events registered, while the third source (mainly for debugging purposes) is the standard input (the user keyboard). For each one of these possible sources a suitable and coherent implementation is provided. Referring to the Java terminology, we can say that the *Event Recorder* interacts with the different sources through a common interface (the *Recorder Interface* of Figure 11).

It is simple to extend the prototype with new message sources, by creating a new class that implements the specified interface.

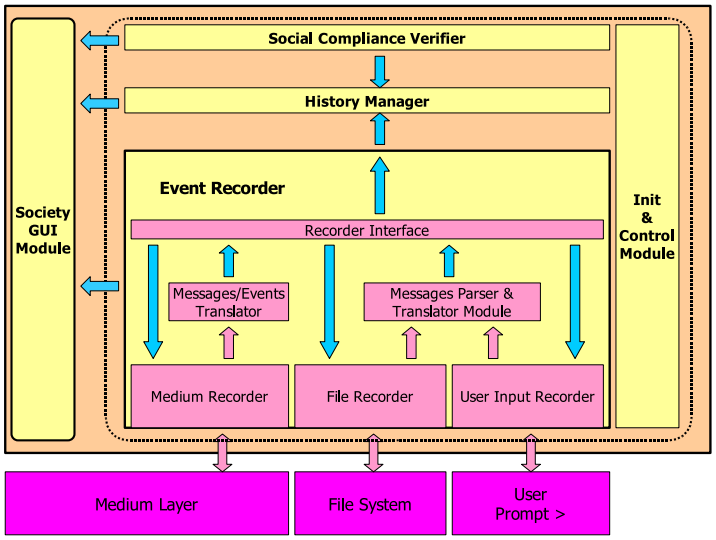

Figure 11: Internal composition of the Event Recorder sub-module

Note that the different sources are interchangeable, but it is not possible to use two different sources at the same time. Until one source is selected, the *Society Infrastructure* will reason about the messages fetched from that source. A brief description of each different source is given below:

i) The **Medium Recorder** is the most important source. All the computees communicate through the communication medium; also the messages exchanged are fetched by the *Society Infrastructure* through the same medium. The messages fetched by the *Medium Recorder* are already in a "structured" form, and they need only to be translated into events. This is not true for the other sources, that return a string description of the messages. For that case a parser component has been added in order to extract the data from the string representation.

ii) The **File Recorder** represents another important messages source. An important feature of the *Society Infrastructure*, along with the ability to react to dynamic environments, is the ability to process a static given history of events. This feature implies reading history data from a text file, where all the messages have been previously saved. So the *Event Recorder* must be able to retrieve the messages from a simple text file. Note that in this scenario all the messages are already available, while using the previous source there is the idea of "suspension" until a new message arrives.

iii) The **User Input Recorder** is especially important for the early stages of the prototype development, since it lets the user type in all the desired messages. The same result can be achieved also through the "file source" (for instance, if we want to repeat the same experiment twice).

5.2.4 Message processing

Computees communicate by exchanging messages. As we mentioned at the beginning of Section 5, in this implementation of SOCSDemo we made the assumption that computees know what events are socially relevant. Those messages will be visible to both the intended recipient and the society. A socially relevant message is fetched by the *Event Recorder* through the specialised module. Then the message is translated into an event object, and it is registered in the *History Manager*. As soon as the event is registered, the GUI is notified and updated. From this moment the event is available to the proof-procedure.

Until it terminates, the proof-procedure can be in one among several different states: it could be *blocked* waiting an acknowledge from the user, or it could be *busy* processing a previous event, or it could be *waiting* for new events.

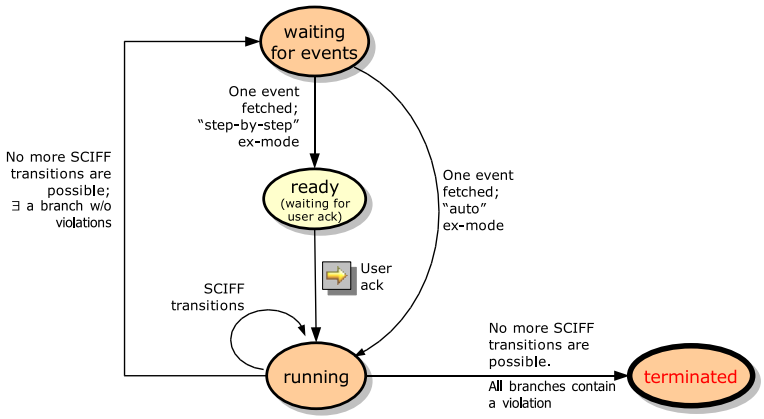

Figure 12: Possible states of the proof

Starting from either state, at some point the proof-procedure will ask the *Social Compliance Verifier* to fetch a new event. The *Social Compliance Verifier* will forward the request directly to the *History Manager*, that will answer with the event and will update the GUI. The proof-procedure will process the event. Then, the proof-procedure will notify the GUI of the results of the elaboration (again by using the services offered by the *Social Compliance Verifier*). The proof-procedure will then try to fetch a new event through a call to Java. The processing of the message terminates as a copy of the event is permanently stored by the *History Manager*. An exception to this behavior can happen if the elaboration of the event leads to a failure state. This could be caused by a violation of the society protocols. Depending on the implementation of the proof-procedure, it is possible that a backtracking is operated. In such a case, the event can be "re-processed" in order to reach a new success state, if there exists any.

5.3 Proof Procedure

In this section, we describe the implementation of the proof procedure specified in Deliverable D8 [58].

5.3.1 Overview

Technology. In the implementation of the society proof-procedure, we have found the choice of SICStus Prolog [80] very useful, for the following reasons:

- the Prolog language offers built-in facilities for the implementation of dynamic data structures and (customizable) search strategies;
- SICStus Prolog allows for state-of-the-art Constraint Logic Programming [54]; in particular, the CLPB, CLPFD and *CHR* libraries have been exploited (a brief introduction to *CHR* can be found in Sect. 10.2);
- SICStus Prolog features a bidirectional Java-Prolog interface (Jasper), which we need to interface the proof-procedure with the other modules of the social demonstrator (see Sect. 5.1).

Search strategy. As the IFF proof-procedure [43], the social proof procedure described in Deliverable D8 [58] specifies the proof tree, leaving the search strategy to be defined at the implementation level. The implementation described here is based on a depth-first strategy. This choice enables us to tailor the implementation upon the computational model of Prolog: in particular, the resolvent of the proof is represented by the Prolog resolvent (see Sect. 5.3.3), and thus the Prolog stack is used directly for backtracking. However, in this way, only one node of the proof tree is examined at each computation step, instead of the full frontier of the proof tree (see [58]).

Success and failure. The proof-procedure returns success when a state of closed fulfillment is found. In this perspective, not only inconsistency (both with respect to **E**-Consistency and $\neg$-Consistency, see [58]), but also violation generates a failure, and causes backtracking.

5.3.2 Variables

Variables are represented by attributed SICStus Prolog variables [50, 80]. 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 Deliverable D8 [58], variables in the resolvent and in abduced atoms are *flagged*. 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.

Quantification of variables is represented by a `quant/1` attribute, whose attribute 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`,which has, as argument, 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`.

5.3.3 Data Structures

Each state of the proof (as specified in [58]) is represented by a tuple with the following structure:

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

The data structures are implemented by means of Prolog built-in structures and the *CHR* constraint store. In particular, the *CHR*-based representation of PSIC, **EXP**, **HAP**, **FULF**, and **VIOL** allowed us to exploit the computational model of *CHR* (see Sect. 10.2) in order to apply the appropriate transitions (implemented by means of *CHR* rules) whenever one of these data structures changes (which, due to their *CHR* representation, amounts to insertion or removal of constraints in the store). In the following, we describe the implementation of each element of the tuple.

Resolvent R. The resolvent of the proof is represented as the Prolog resolvent. This allows us to exploit the Prolog stack for depth-first exploration of the tree of states.

Constraint Store CS. The constraint store of the proof[5] 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 that works on top of SICStus Prolog.

Partially Solved Integrity Constraints $PSIC$. 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. Elements of the lists are Prolog terms representing events, expectations, constraints or predicates. Each sub-list contains predicates of the same type (e.g., only events or only expectations) in order to make the propagation 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.

[5]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.

For instance, the following partially solved integrity constraint:

$$\begin{array}{l} \mathbf{H}(\mathit{tell}(A, B, \mathit{request}(P), D), T_1) \\ \rightarrow \mathbf{E}(\mathit{tell}(B, A, \mathit{accept}(P), D), T_2) \ \wedge \ T_2 < T_1 + 10 \\ \vee \mathbf{E}(\mathit{tell}(B, A, \mathit{refuse}(P), D), T_2) \ \wedge \ T_2 < T_1 + 10 \end{array} \quad (1)$$

would be represented by the following *CHR* constraint (where the CLP constraints are represented in the SICStus Prolog CLPFD notation):

```
psic([[h(tell(A,B,request(P),D),T1)],[],[],[],[],[],[]],
     [[e(tell(B,A,accept(P),D),T2),T2#<T1+10],
      [e(tell(B,A,refuse(P),D),T2),T2#<T1+10]])
```

History HAP. Each event is represented by means of a `h/2` *CHR* constraint, whose (ground) arguments are the content and the time of the event. An example of event is:

```
h(tell(yves,thomas,request(scooter),a_dialogue),10)
```

Expectations EXP. Expectations that are neither fulfilled nor violated are represented by means of a `pending/1` *CHR* constraint, whose content is a term (with functor `e` for **E** expectations and `en` for **NE** expectations) representing the pending expectations. The `pending/1` constraint,obviously, does not apply to $\neg$**E** or $\neg$**NE**. An example of pending expectation is:

```
pending(e(tell(thomas,yves,accept(scooter),a_dialogue),T))
```

The reader should note that the representation of CLP constraints on variable `T`, such as `T#<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**, **NE**, $\neg$**E** or $\neg$**NE** expectations, respectively. The two arguments of these *CHR* constraints are the content and the time of the expectation.

Fulfilled Expectations FULF. Each fulfilled expectation is represented by a `fulf/1` *CHR* constraint, whose argument is a term representing the fulfilled expectation.

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

5.3.4 Transitions

The implementation of transitions has been designed so to exploit the built-in Prolog 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.

IFF-like Transitions.

1. Unfolding

   According to Deliverable D8 [58], 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;
   - 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).

2. Abduction

   Abducibles (**E**, **NE**, ¬**E**, ¬**NE**) are represented as *CHR* constraints; thus, abduction can simply be achieved by calling them.

3. Propagation

   Propagation of events and expectations with partially solved integrity constraints exploits the *CHR*-based representation of **HAP**, **EXP** and PSIC; 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 of **H** atoms:

```
propagation_h @
    h(Event,Time),
    psic(Body,Head) # _psic
    ==>
    Body=[H,_,_,_,_,_,_,_],
    find_candidate(H,h(Event,Time),1,N)
    |
    ccopy(p(Body,Head),p(Body1,Head1)),
    sub_body_prop(h,Body1,Head1,h(Event,Time),N)
    pragma
    passive(_psic).
```

   The rule is activated each time a new `h/2` is added to the *CHR* store; the rule is activated for all `psic/2` *CHR* constraints present in the store.

   The guard of the rule checks whether the body of the partially solved integrity constraints contains at least a term that can be propagated with the `h/2` atom. This is achieved by isolating the sublist of the body that contains `h` terms (`H`) and applying the `find_candidate/4` predicate which will succeed if the `N`-th element of `H` is a candidate for propagation (if particular, in the functors and arities of the term and the atom are the same).

   If the guard succeeds, a copy is made of the partially solved integrity constraint, which will be actually propagated: the original partially solved integrity constraint is left unchanged in the *CHR* store in order to be possibly propagated with other atoms.

   Actual propagation is performed by the `sub_body_prop/5` predicate, which will replace each element of the sublist of the body that can propagate with the `h/2` atom with the

unification constraint. At the end, the new partially solved integrity constraint is added to the *CHR* store.

The `psic/2` constraint in the head of the rule is declared as *passive*, i.e., the rule is not activated when a `psic/2` constraint is added to the *CHR* store. This is needed to avoid multiple propagations of the same partially solved integrity constraint with the same set of atoms.

Rules for propagation of **E**, **NE**, ¬**E** and ¬**NE** atoms are analogous.

4. 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. Disjunctions in the definitions of atoms (as in IFF) are expressed by writing more than one clause for each atom, which is dealt with by Prolog. Disjunctions in the constraint store are handled by the constraint solver(s).

5. 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).

6. Equivalence Rewriting

   As explained in Deliverable D8 [58], equivalence rewriting is delegated to the constraint solver(s).

7. 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:

```
trigger_psic @
    psic([[],[],[],[],[],[],Atoms],Head)
    <=>
    true
    &
    call_list(Atoms)
    |
    impose_head(Head).
```

   The rule is activated when a partially solved integrity constraints that does not contain events or expectations in the body is added to the *CHR* store: the guard imposes `Atoms` (constraints or defined atoms) by calling each of them, and if success is returned imposes the head of the partially solved integrity constraint.

Dynamically Growing History.

1. Happening

   Happening of events is achieved by imposing a `h/2` *CHR* constraint, whose (ground) arguments are the content and the time of the event.

2. 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 **NE** expectations.

Fulfillment.

1. **E** Fulfillment and **NE** Violation

   Fulfillment of **E** and violation of **NE** 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 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[6] and the `case_analysis_fulfillment/7` predicate is called.

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

[6] As specified in [58]: this allows for universally quantified variables in **NE** expectations to remain unbound.

- 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;
- one where non-unification between the expectation and the event is imposed.

2. **E** Violation and **NE** Fulfillment (closed history)

   When the history of the society is closed (by means of a closure transitions), all pending **E** are marked as violated and all pending **NE** are declared fulfilled. 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.

3. **E**-Consistency

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

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

4. ¬-Consistency

   Analogously to **E**-Consistency, ¬-Consistency is implemented by by imposing non-unification on the (*Content*, *Time*) pairs of **E** and ¬**E** (or **NE** and ¬**NE**) 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).
```

5.4 GUI

The graphic user interface is implemented by using the Swing Java classes: this set of libraries has been designed by following the "Model View Control" design pattern (MVC). Once we decided to adopt this set of libraries, we also decided to take advantage of the design pattern implicitly proposed by the Swing collection. As it is possible to note in fig. 13, three different groups of components have been created.

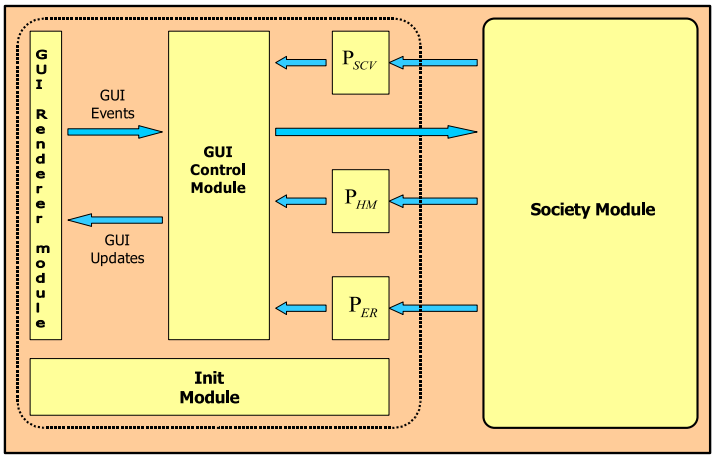

Figure 13: The GUI Module inside

Three components (*Presentation Modules*) are labelled with a bold P: they are the part corresponding to the "Model" in the MVC pattern. The Presentation Modules are responsible for keeping track of the data. For each one of the *Social Compliance Verifier*, the *History Manager* and the *Event Recorder*, there is a Presentation Module devoted to treating their data. These components also see to notifying the GUI Control Module whenever the data are changed or updated.

The GUI Control Module is the component devoted to controlling the GUI and to manage the data part. More precisely, this component has a double role: (*i*) it sees to properly updating the graphic part whenever it is necessary, and (*ii*) it sees to managing the user events on the GUI, such mouse clicks and selections and button pressed events. These graphics events can trigger some responses and changes in the graphics. All the elaboration process of these graphic events is managed by the Init&Control module. This component corresponds to the "Model" part in the MVC pattern.

The "View" role of the MVC design pattern is taken over by the Graphic Renderer. This module is implemented as a collection of Java classes; most of them directly extend and inherit the Swing core classes. A fourth module is implemented, devoted to the initialization of all the GUI components. It is depicted at the bottom of the GUI module. The main task of this component is to initialise all the other sub-modules. Symmetrically to the Init module of the elaboration part, also this module acts as proxy for all the other GUI components. The interested reader is referred to the SOCSDemo user manual [6] for a detailed description of the graphic interface, and on how to use it.

Part III
THE DEMONSTRATOR IN PROSOCS

6 A Prototype Application

6.1 The Application Scenario

We use the *Leaving San Vincenzo* scenario to provide the context of the examples implemented in the demonstrator. To establish the link of these examples to Global Computing, we have discussed in [7] the relevance of the Leaving San Vincenzo scenario with the context of the Global Computing programme. We also integrate in the San Vincenzo scenario the Resource Allocation and Combinatorial Auctions scenaria proposed previously for SOCS, giving rise to a new and extended San Vincenzo scenario. The simplifying assumptions of the extensions are further discussed, however, only some of the identified extensions are outlined in this section, as most of the extensions are discussed in the sections that follow, in the form of concrete examples. The material presented in section 6.2 and 6.3 is adopted from the examples document [7].

6.2 Summary of the *Leaving San Vincenzo* scenario

The original *Leaving San Vincenzo* scenario [85] can be summarised as follows. A Spanish businessman, called Francisco Martinez, travels for work purposes to Italy and, in order to make his trip easier, carries a personal communicator, namely a device that is a hybrid between a mobile phone and a PDA. The application running on this personal communicator provides the environment for a computee, treated for the purposes of the scenario as a piece of software that augments the direct manipulation interface of the device with implicit management. By implicit management we mean that the computee is a personal service agent that provides proactive information management within the device [87] and flexible connectivity to smart services available in the global environment the businessman travels within.

Most of the original scenario unfolds in San Vincenzo, an Italian holiday resort, where Francisco spends the last weekend of his away trip. In this context, the scenario describes a series of events illustrating the kind of interactions (both physical and virtual) that Francisco's computee facilitates via the personal communicator of Francisco. Francisco's communicator connects Francisco's computee with a series of other devices that we assume are available in the physical environment so that Francisco can access smart electronic services. Here are the main examples used in the scenario:

- Before leaving Spain, at the airport gate the computee registered Francisco with a location-independent smart-service provider in Italy.
- When Francisco arrived in Italy (Rome) it was weekend, but as Francisco was uncertain whether to stay there or carry on to his destination, the computee advised Francisco to stay, as there was a jazz festival.
- At the hotel Francisco stayed in the last two days of his visit to Italy, in San Vincenzo, the computee payed Francisco's bill, ordered a taxi, and checked for train information.

- At the train station of San Vincenzo, the computee advised Francisco how to successfully buy a train ticket, despite the difficult conditions (ticket office closed, ticket machine out of order).

Based on the above examples, the scenario seeked to demonstrate how technology can be used to help people with what they do in their ordinary lives, from going to a business trip, to staying in a hotel, travelling by train, and ordering a taxi.

6.3 Extending *Leaving San Vincenzo*

For the purposes of this implementation we extend the original San Vincenzo scenario in two ways: (a) we make Leaving San Vincenzo the "umbrella" scenario for SOCS by integrating into it the other two SOCS scenarios ([35, 67]), and (b) we specialise and augment the interactions of the original scenario so that they can provide examples suitable for the implementation effort.

To achieve (a) we have made the simplified assumption that a piece of information is like a resource (or in some cases like a product[26]) which can be exchanged between computees. By making this assumption we can treat interactions supporting the exchange of information similarly to the negotiations often taking place in resource allocation problems (in this way we incorporate in the San Vincenzo scenario the resource allocation scenario presented in the context of SOCS in [35]), with the additional advantage of contextualising these interactions and present solutions that can be immediately useful to what people do in their everyday activities.

Also for (a) we have interpreted connectivity to smart services as *access to resources* available in and managed by societies of computees that "run" on top of a global computing environment such as the one assumed by the scenario. For this purpose we extend the original scenario, where Francisco' computee was ordering a taxi, with the computee now been engaged in a combinatorial auction [67] but now in order to book taxis for the whole of the return trip, from San Vincenzo in Italy to Francisco's home in Madrid.

Finally, we will present (b) by introducing examples of interactions in the Leaving San Vincenzo context as we go along, illustrating how can the D3 criteria these interactions exemplify be met. As we shall see next, most of these new interactions are specialisations of the interactions presented in the original Leaving San Vincenzo scenario.

6.4 Running the SOCSDemo

A demonstrator of the scenario discussed in the previous section form part of the prototype demonstrator of the SOCS project. This demonstrator is based on a set of concrete examples that have been developed especially to demonstrate the generic functionality of the PROSOCS platform [7]. In particular, the demonstrator implements a list of representative examples, which a user can download from the SOCS web-site:

`http://www.lia.deis.unibo.it/research/projects/socs/`

Apart from the implemented examples the web-site provides additional information about the system, including the manual of the prototype [6]. The main purpose of the manual is to provide precise instructions on how to download the demonstrator, how to install it on a host computer, how to run the implemented examples, and - provided that the user is a logic programmer - how to write new examples. We give next a flavour of how the prototype runs, by presenting how the system operates using one of the demonstrating examples.

6.5 An Example Run

We show an instance of the San Vincenzo scenario demonstrating the integration of the computee prototype and the society prototype within PROSOCS, to verify the expected behaviour of computees. The example that we are going to illustrate is the example presented in section 5.1.3 of the examples document [7]. There are two computees in this example setting, Francisco's computee f and the San Vincenzo train station manager svs. f requests information about the arrival of the train to Rome from svs. This message is received by the svs computee who responds first with a reply to inform f about the arrival of the train, and immediately afterwords replies refusing to provide this information (that it has just provided). This violates the protocol of the society, that one cannot give two answers to the same query. Fig.14 shows the GUI of the svs computee.

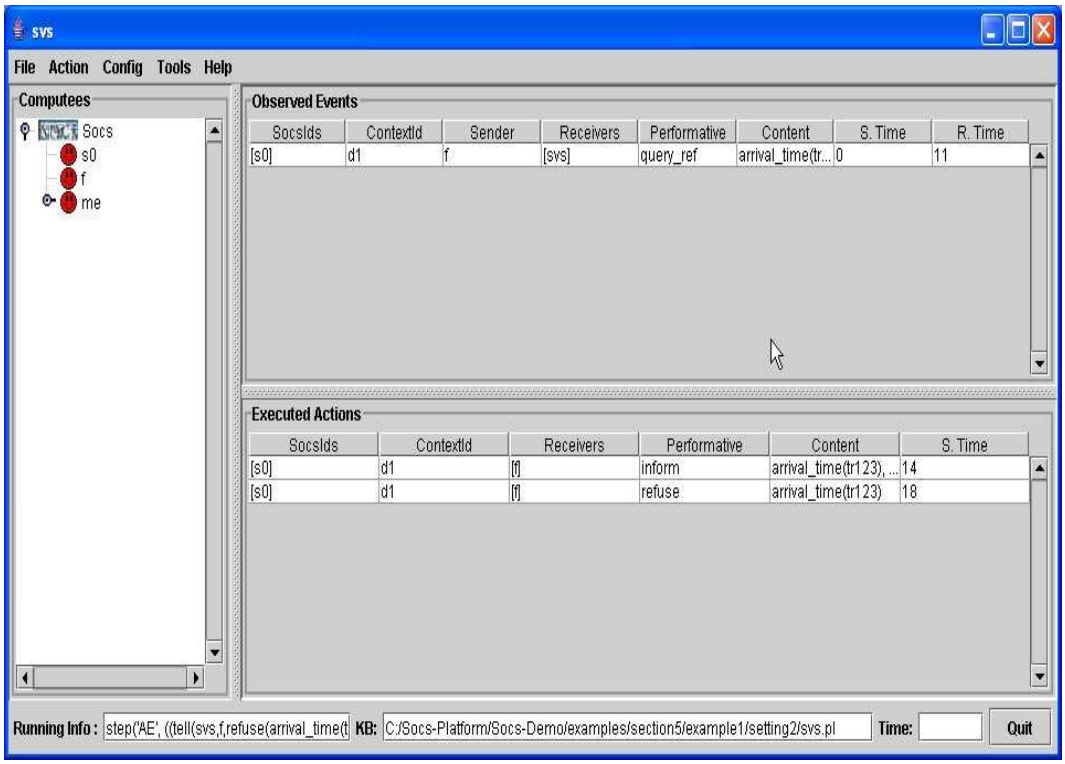

Figure 14: The GUI of the svs computee

The left part of the picture shows the other components that are running in the platform from the point of view of svs, i.e. the computee f and the society component $s0$. On the top right part of Fig.14, under the title `Observed Events`, the details of the received message are being depicted. In the lower right part of Fig.14, the communicative actions executed by the computee are shown.

The behaviour of svs and the communication between computees f and svs can be checked by the society infrastructure of the platform. The user of the society can examine that state of

the society via the society's user interface as depicted in Fig.15.

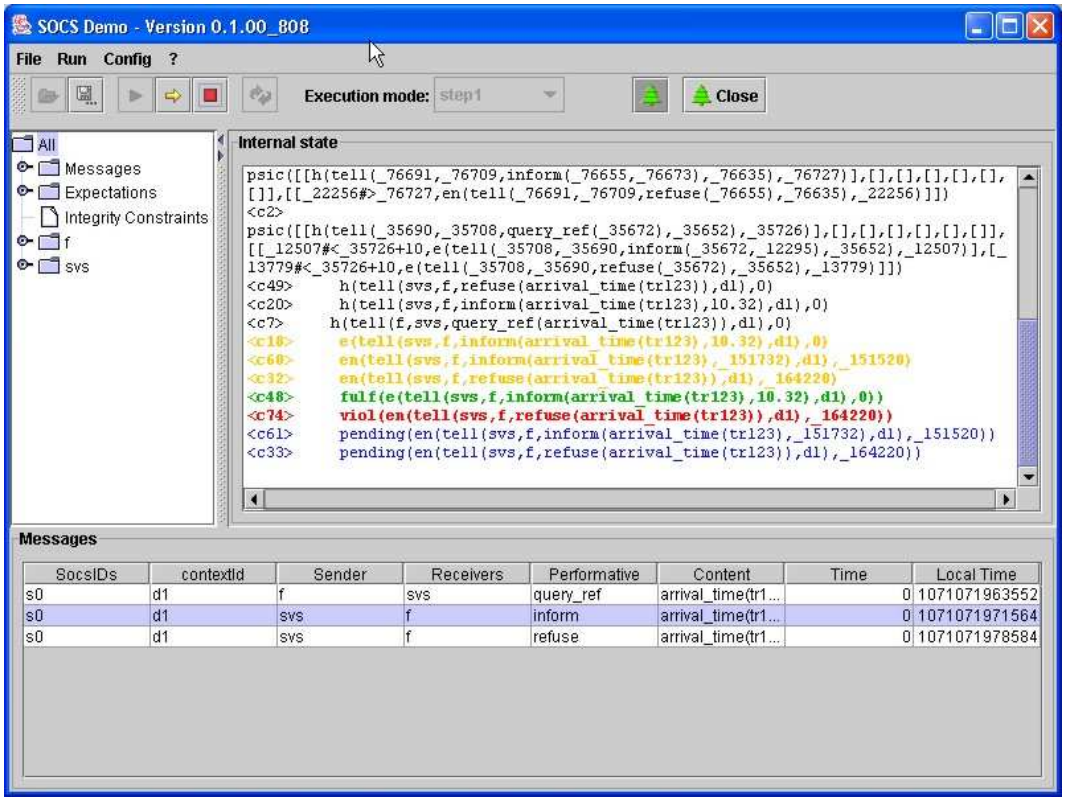

Figure 15: The GUI of the Society Infrastructure

In Fig.15 the lower part of the society infrastructure shows all the messages that have been exchanged by the computees in that society. At the left-hand side of the figure, the presence of the computees f and svs is also depicted, including interface information about the messages and the expectations the society has for different stages of their interaction. In the middle of the society GUI the user can see the details about the expectations and possible violations, as it is the case with this example. The society infrastructure can also show to the user the tree of the interactions between the computees. For more details on how to use the Society interface the interested reader is referred to the manual [6].

7 Evaluation

7.1 Evaluation of WP4

A detailed evaluation of WP4 in the context of the criteria set out in deliverable D3 [60] is provided in deliverable D11 [17], where the results of the work are evaluated in the context of GC and SOCS, and the achievements, together with the weaknesses are discussed in detail.

We will not reevaluate the work of WP4 here again, in order to avoid unnecessary duplication in deliverables. Instead, in this section we want to discuss briefly some important aspects of the system, including the experimentation that we have carried out so far. We also use this space as the right place to discuss any additional assumptions that were not identified in the description of the prototype presented in the previous sections.

The experimentation context that we described via the application scenario and the example run of part III has been demonstrated using controlled experiments involving three laptops running PROSOCS, all connected via a LAN (Local Area Network). We have used different variations of this arrangement. At one end of the spectrum, all the computees and the Society Infrastructure run on one laptop. At the other extreme we have tried to distribute the components of the system by running each of the computees on separate laptops, while on the third laptop the Society Infrastructure.

Our experiments so far, using the different network arrangements just described, have been successful in that they have demonstrated the expected functionalities within controlled settings. Apart from testing the models, a lot of emphasis has been put on the testing of the message transportation ability of the system. For this purpose, as we have also foreseen in D3, our experimentation has concentrated on using communicative actions between computees only. That is, we have not tried physical actions on the environment, a feature which we plan to investigate in the third year.

Moreover, we have experimented with LANs only. In other words, we have not experimented with other kinds of networks such as wide area networks or global networks. For this kind of networks we expect to face the problem of maintaining a global time in network topologies that are more complex than three laptops connected in a LAN. We also expect to find problems on the network due to firewalls, or other security measures that may restrict communication and interaction. Again, these are the kind of issues that we want to explore within the experimentation (WP6) in the final year.

Through the experimentation we have illustrated how to specify computees and how to prototype their states, including the part that the computee will use to interact with other computees. A summary of how a user can specify new problems has also been outlined in the user manual [6]. Although specification of problems has been the primary goal of PROSOCS at this stage of development (it is certainly a very important consideration from a software engineering perspective), we have not yet developed any methodologies for building PROSOCS computees. In particular, we have not adopted the game-based development approach we had foreseen to use within the Technical Annex. As we state in D11 [17] the reasons why we did not pursue the game-based methodology was to allow partner sites to focus on functionality. In addition, in the context of software engineering we have not pursued the idea of ontologies for building specific kinds of applications, as the focus of this work has been mainly the functional integration of the models.

We have already stated in D11 [17] that mobility has not been prototyped in PROSOCS. As mobility is an important aspect of GC, and as the SOCS consortium has foreseen in the technical annex and D3 [60] to investigate mobility within WP4 in SOCS, to some extent, we discuss in the next section this topic separately.

7.2 Mobility in (PRO)SOCS: A Feasibility Study

In what follows we briefly discuss what we consider as the main issues related to mobility and SOCS, which could serve as a starting point for tackling this problem in PROSOCS. To this

purpose, we will consider mainly two aspects. First, we will investigate what kind of mobility is most appropriate for SOCS computees and the Society Infrastructure. Then, we will discuss how mobility could be embedded in SOCS, and also how mobile agent systems could possibly take advantage from such an extended SOCS model. In this context, we hope to address typical problems raised by mobility of software components, such as, for instance, security problems.

From a general perspective, in SOCS we could distinguish between two different forms of mobility. As a first form, mobility could be intended as the capability of moving computees/societies between physical nodes in the network. As a second form, another more abstract notion of mobility could be given: it could be viewed as the computee capability to access and exit to/from societies, i.e., virtually moving across societies. This second interpretation of mobility, can be identified with the openness property of the SOCS computees and society, that we already discussed in the deliverables D3 [60], D5 [66], and D11 [17]. As explained in [66], in fact, SOCS deals with open societies of computees: according to [29], in an open 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. Moreover, each computee is free to move across societies.

In this section we will distinguish between "physical mobility", referring to the first form of mobility, and "logical mobility", referring to societies access. We will concentrate on physical mobility and only touch on logical mobility, which is already documented elsewhere, as explained above.

7.2.1 Preliminaries

It is increasingly recognised that software mobility is a fundamental issue in the development of distributed systems with requirements of efficiency and fault tolerance. This has caused, in recent years, several proposals of innovative programming paradigms based on mobility, in particular, that of Mobile Agents (MA) technology [59].

From a general point of view, several different paradigms based on the possibility of dynamic software migration have been identified and developed (see [42] for a general introduction). Among these, Client/Server (CS) are perhaps the first and simplest forms of mobility, where the client asks the server to execute the computation and possibly send back an answer. This simple case, in which *data* migrate, illustrates the basic mechanisms and motivation for mobility: computational resources are not uniformly distributed, indeed the server is able to process a request on behalf of the client, and data are exchanged from the client to the server, and backward.

More complex forms of mobility focus on the mobility not only of *data*, but also of *code*, and even of whole *computational environments*, according to different paradigms that differ in the distribution of know-how, and resources among the different locations.

For instance, *code* is daily downloaded in the form of the so-called plug-ins in order to update capabilities of locally resident programs. Code can also move around to process data that are locally kept, for instance because of their size or their confidentiality. This kind of code mobility encompasses the Remote Evaluation (REV) and Code on Demand (CoD) paradigms.

In the REV paradigm [84], a component A exploits the computational resources of another component B, typically residing on a different location, by sending instructions (code) specifying how B should perform a service on behalf of A, using its own resources. The component B then executes the received code.

In the CoD paradigm, instead, component A has resources located in its execution envi-

ronment but lacks the proper code to process data and access resources, and hence it needs to obtain the required code from component B. These code-mobility principles are closely related to Component Based Software Engineering, where components in the form of independent pieces of software are deployed by third parties to form dynamically assembled (distributed) systems.

The MA paradigm is an extension of the previous paradigms, where a whole *computational environment moves*. This means that the state of the computation has to be somehow recorded and transfered to the new location. For instance, a component of a distributed system that migrates from one location to another, carries its code and its current execution state, and it may need to keep trace of all the connections, e.g. communications channels, which it had in the original location, and relocate them in the new one. A mobile agent can hence migrate autonomously to a different computing node that can offer the required resources, and it is capable of resuming its execution seamlessly, because it preserves its execution state. It is quite clear that this process poses difficult problems of modelling and verification, as well as of domain administration and security.

In its first stages, code relocation was mainly motivated by performance issues, like load balancing. MA technology for distributed system management has a wider range of motivations, which encourage its utilization, in a Global Computing setting, by showing some significant deriving benefits. The most recognised advantages span from the overall reduction of network traffic by exploiting resource co-locality, to the flexibility of distributing software components at runtime, from the full decentralization of the monitoring, control and management of networks, systems and services, to the increased robustness stemming from decoupling tasks into distributed autonomous activities that can overcome temporary network/resource unavailability.

However, the deployment of MA in GC application domains can be accelerated as soon as MA systems can provide solutions that respect the *opening* and *closing* properties. The opening property permits to overcome the system boundaries in order to inter-operate with any necessary external component and to allow any external recognised usage, while the closing property is the possibility of constraining the system in such a way to identify and exclude any malicious intrusion. The opening property is granted by interoperability considerations and the closing property by security mechanisms and policies.

Interoperability is an important property for MA systems and requires to identify the aspects of the MA technology candidate to become standard. The MA research has promoted interoperable and standard interfaces to interact with resources and service components available in statically unknown hosting environments (compliance with CORBA and MA-specific standards such as OMG MASIF and FIPA [39] [73]); these interoperability features can help in supporting the internetworking of mobile users/terminals with previously unknown local resources.

With regard to security, MAs have fostered even more the traditional security issues to the limit. Indeed, compared to the Client/Server model, the MA paradigm offers greater opportunities for attacks to take place because MA systems provide a distributed computing infrastructure on which applications belonging to different (usually untrusted) users can execute concurrently. Additionally, the execution sites hosting MAs may be managed by different authorities with different and possibly conflicting objectives and may communicate across untrusted communication infrastructures, such as the Internet.

Another relevant issue for mobility is the level of the so-called *local-awareness*. The cases of processes relocated for performance reasons by a distributed operating system, or plug-ins downloaded for enhancing a local application, are quite different from the case of a mobile agent

that autonomously moves from one location to another. In principle, processes and plug-ins do not need to be aware of where they are executed, and do not autonomously decide to move. These functionalities are provided by the underlying operating system.

A more complex framework of distributed interaction is the case of the above cited CORBA: this is a middle-ware which allows distributed components to interact with each other by referencing a common ORB (Object Request Broker). Such broker provides an abstraction of the locality topology in the network, by suitably redirecting requests to the correct service providers, which may be located anywhere, and even move. In this case, components need only a limited view of locations. The case of MAs is different. Here, in order to autonomously move through a network of possible execution sites, we need to have both a "correct" representation of the network, and to be equipped with suitable linguistic constructs for expressing movement statements, treating locations as first order objects. In other words, mobility must be under *program control.*

7.2.2 Mobility in PROSOCS: A Sketch

Currently, PROSOCS does not support mobility. However, many aspects of the SOCS model and several implementation choices that we have made, are suitable to support a future extension of the platform with mobile computees. In the rest of this section we will illustrate the features of the SOCS approach (both at the formal, computational and implementation level) that can support mobility or need to be extended in order to do so. We will also discuss the forms of mobility that could be suitable for SOCS computees (and also societies). Informally speaking, we distinguish between "physical mobility", when we address the issue of moving software from one execution site to another, and "logical mobility", in the context of a more abstract topology of societies and computees that belong to them.

Physical mobility. In this part we refer to mobility as the physical migration of data, code, or software, from an execution environment or site to another.

- A first and simple case of mobility for computees consists of code motion, and resembles the Code-On-Demand or REV scenarios previously introduced. Consider for example a light-weight computee that requires from a trusted repository some set of capabilities/functionalities it needs at a certain moment. For example, a computee might decide to use a plan library rather than planning from first principles.

  Thanks to the modular design of computees, this could be easily achieved, practically without any need of moving the state of the computation, since the computee does not migrate, nor a detailed awareness of the network topology is necessary, but only a reference to the code repository required. What is needed is a *re-loading* (or reconsulting in a more Logic Programming jargon) mechanism.

- A case in between simple code down-loading and proper MA mobility, could be the case of a computee that is dynamically relocated in a different execution environment, but this is not under the program control of the computee itself. Consider, for instance, the case of a user who wants to carry a personal assistant (a computee) from a desktop computer to a mobile phone, as the user leaves the office and wants to be assisted in a journey by the computee.

  Even if the migration is decided and executed by the user, with an external command, the execution state of the computee may be required to migrate together with the computee.

While the current knowledge KB_0, the domain dependent and the domain independent knowledge, which are in general textual information, can be easily moved, the support interface must be able to resolve the redirection of all the references, like communication channels, resource accesses, etc., the computee was counting on. Note that this is already partially supported by means of the underlying JXTA communication infrastructure, which is able to identify, by means of a unique name, a communication partner, such as a computee or a society, independently of its physical location on the network. Clearly, this kind of mobility could be coupled with the previous one, for example because the mobile phone of the user can only support light-weight computees, that hence have to often download, and then discard, needed functionalities in terms of code.

- The most general and appropriate form of mobility in our context seems to be represented by the MA paradigm. In this case, a computee *autonomously* decides to move from one location to another, for example because it is searching for new resources, or for accessing data, or for interacting locally with other computees or resources, or for carrying out a confidential computation within a trusted firewall. In order to achieve this more complete form of location-aware program-controlled mobility, the computee model needs a substantial enhancement, in order to embed location as first order objects of the knowledge base of a computee. Such an extended computee, for instance, could be able to reason about a property (fluent) holding not only at a given point in time, as it is currently able to do, but at a given point in space. By space we mean the execution environment; typically we would want this environment to be a complex structure such as the Internet.

In principle all these forms of mobility could be achieved in SOCS, because the architecture of each computee and society infrastructure both provide a clear separation between state and code, and both state and code are well defined by the computational models in terms of tuple states of a transition system (see D8 [58]). From a practical point of view, the adopted formal model greatly facilitates the task of moving the state/code of a computee/society to a remote node.

Note that the previous scenarios, which have been presented for the perhaps more intuitive case of computees, may apply to the case of societies as well, in the sense that also societies could exhibit mobility features (both as code, data and execution environment mobility that can be location-aware and program-control, or not).

Logical mobility Mobility can be interpreted in the SOCS context, at a logical level, as the possibility for the computees to access societies. Note that this does not necessarily require any relocation, since, as already discussed, the underlying JXTA communication infrastructure may let computee access societies distributed anywhere over the Internet. However, at a different level, the problems of openness and closure are still valid. Indeed, a computee must be able to inter-operate with an external society, and the society must be able to preserve itself from the possibly malicious behaviour of the computee.

In order to accomplish this kind of logical mobility, computees must be equipped with abstract location-aware capabilities, which for instance allow them to maintain a representation of the society topology and to locate the more suitable society for their purposes (we have explored this to some extend in [92]). Societies, on the other hand, must be equipped with access control mechanisms and policies, which implement some form of suitable closure.

Finally, it is worth noting that SOCS could take great advantage from work on mobility done by related GC projects:

- AGILE [5], where, in particular, a set of language primitives with formal semantics for expressing code mobility has been defined; these primitives could be incorporated into computees.
- MIKADO [68] where new formal models for both the specification and programming of large-scale, highly distributed and mobile systems has been defined; this formal programming model, could be possibly exploited to implement a mobility-enabled version of SOCS.

7.2.3 Which primitives?

As argued above, even if mobility is not currently implemented in SOCS, there is no assumption in our models that prevents computees to move from one node to another. Moreover, at the platform level, there is no assumption that restricts computees to reside on specific hardware, so that they can reside and migrate on mobile devices (such as cellular telephones, palm computers, etc.).[7] Future extensions encompassing mobility need to choose the linguistic constructs to be added to computees and societies. As we have seen, for the complete, location-aware scenario, locations must become first order objects of the language, let them be physical or logical locations. We briefly discuss how the main approaches to mobility, currently in literature, might contribute in this sense to SOCS.

There are several well known approaches to process mobility at different levels. Among formal approaches to mobility, for instance, the one proposed by Gilbert and Palamidessi [47] is based on an extension of concurrent constraint programming . Another approach is based on Millner's pi-calculus [69], which is a formalism for dealing with concurrency and mobility; this is the approach followed, for instance, in GC projects MIKADO [68] and MyThS [71]. Among practical approaches, several mobile systems providing a wide range of support facilities have been developed. In this area, relevant work is available, for instance, at the University of Bologna [15, 27] and at Imperial College [25].

Most of these mobility frameworks provide a *move* primitive which can be called by an agent/process to request its migration to a new node. Using this *move* call, a computee could move from a node to another by means of a physical action (consisting of a call to *move*). This migration could happen in a totally transparent way for the (external) society, and requires a proper support from the underlying execution platform. Moreover, the computee could be equipped with a dedicated capability in order to reason about spatial locality, and, for instance, deduce from its current knowledge the most suitable location where it has the right to move in order to accomplish its tasks.

A different technique is the one adopted in the FIPA ACL specification for mobility [40]. Following this approach, the ACL language for interaction between computees could be extended in order to allow a computee to explicitly express its intention of migrating from a place to another one. In our case, a performative request of *move* could be inserted into a tell predicate, as in [40].

It is worth noticing that this technique would be less invasive in the current SOCS models, since it could be solely realised by means of new communicative acts (to some appropriate

[7] We are aware that up to now such hardware platforms are subject to strong limitations that would impose to implement *lighter* versions of computees and societies especially tailored to be loaded on these devices.

receivers). Moreover, mapping mobility into communicative acts would make also possible for the Society Infrastructure to control and monitor computees movements among physical nodes.

For these reasons, let us briefly sketch in more detail the requirements that this approach poses on SOCS computees/societies. As presented in deliverables D4 [57] and D5 [66], SOCS computees are able to send messages to a (set of) receiver(s) by communicative actions expressed as utterances in the *tell* predicate. To support migration we should provide a predicate *move*, that should allow migrating computees to explicitly utter `tell(Sender, Receiver, request(move(B)))` to perform the migration to the destination node B. In this case, the receiver could be a system agent as in [40]. In addition, each agent should be equipped with a suitable agent descriptor [40] containing the agent most relevant attributes for mobility (such as references to its code and data, its requirements for execution, etc.). In order to allow interoperability, the agent descriptor should also enclose one (or more) agent profile(s) [40], describing the software platform and the operating system supported by the migrating computee.

Use of explicit communication does not define neither how mobile computees operate nor how they are implemented. Of course, the platform should provide the needed mechanisms to transfer SOCS computees from one node to another. In our case, since the implementation approach is based on JXTA, we could take advantage of some existing work [21], where Java mobile agents are built on top of the JXTA P2P platform.

The previous considerations apply to both physical and logical mobility. The latter poses interesting issues related to the *social* level, that is one of the main focus of the SOCS project, as discussed in the next section.

7.2.4 Can we take advantage of SOCS to support mobility?

As we pointed out earlier on, one open issue related to MA is security. In SOCS we do not consider explicitly security issues. However, we think that our approach, expressing the *correct* behaviour of computees in terms of social integrity constraints and relying on a run-time compliance verifier, could be smoothly applicable also to the MA security problem. In particular, entering into a society would be interpreted as entering into a particular execution environment. Rules for expressing the right behaviour of migrating computees in accessing resources would be specified by social integrity constraints. Moreover, a computee could be made compliant with the society by learning the social integrity constraints while accessing the society, realising both an opening operation that is allowed to enter and interopertate with the society, and a closing operation that the society imposes its policies on it.

This scenario can be interpreted in a broader sense, if we think of the Society Infrastructure not at the logical, but at the physical level: a society represents a physical execution location. In this case, the social compliance verifier could act like an automatic detector of intrusions or of incorrect behavior of mobile computees. This research line could represent a contribution of the SOCS project to the general area of security and mobility.

A related approach can be considered the one of the MRG [70] project. This deals with the problem of security in MA and, in particular, with the problem of preventing a hosting node from executing an unsafe mobile agent piece of code. Differently from our approach, the solution in the MRG project aims to detect the safety of the agent code a-priori before agent execution at the new hosting node. The solution aims to develop an infrastructure needed to endow mobile code with independently verifiable certificates describing agent behaviour. These certificates are "proof-carrying", since they could independently be verified at the destination site.

One of the implications of proof-carrying certificates at the destination site, however, is that access is required to the internal structure of the migrating object. Notice that our approach is, instead, a "social" approach and therefore does not rely in general on the knowledge of the internal state of the computee. However, in principle, the logical foundation of the SOCS computational model and the hypothesis of moving computees as "knowledge bases", could be exploited to support a similar form of code-checking. Even if computees do not need to access the internal structure of each others in order to collaborate, societies could use proof-carrying techniques in order to inspect computee's internal state, certify and accept them as "safe".

7.3 Related work

A plethora of software platforms for building software agents is available, for example see [62, 34, 76, 28, 77, 4] for more details. This renders the task of reviewing all the related work available in this area gigantic [34]. As a result, to make our task more manageable, in this section we identify only a subset of existing platforms that we believe to be most relevant to PROSOCS. For example, we exclude platforms whose main focus is mobility. Another way to measure relevance is by selecting platforms that are based on the models that we have already compared with our work in deliverables D4 [57], D5 [66], and D8 [58], excluding when necessary proposals that to the best of our knowledge have not been updated recently (e.g. [19]). We expose the similarities and differences between PROSOCS and what we believe to be the most relevant proposals, resulting in an evaluation that is based on the relative advantages and disadvantages of PROSOCS and these chosen platforms.

In many respects, the comparison that we present here is by no means complete, in the sense that the status of PROSOCS is at an early prototyping stage that does not support an Agent Development Environment or any methodology, except for a very basic one. As a result we shall focus the comparison on the general approach taken and the implications of the reference models and implementation technologies used by PROSOCS and the chosen platforms, rather than issues such as development methodology and tools provided in the software engineering sense (e.g. knowledge editors; for a discussion along these lines of a subset of the platforms discussed here the interested reader is referred to [76]). In addition, we will not compare the logical or computational models that we use in PROSOCS with the corresponding models used in the selected platforms as this comparison is already available in [57, 66, 58].

This section is organised as follows. We start the discussion on the differences between our approach and platforms that use the concept of a directory facilitator component, such as JADE[16], FIPAOS[74], and ZEUS [72]. We then discuss well-known platforms that, like PROSOCS, offer reusable implementation of reasoning and communication mechanisms, treating agents as if they were first class objects. These include proposals such as IMPACT[12, 52], *SIM_Speak* [64], and 3APL [49, 1]. We also discuss platforms already used in industry, such as for instance AgentBuilder [3] and JACK [53]. We continue the discussion by looking separately at some logic-based approaches that are proposed as programming languages for agents, such as IndiGolog [46] and Go! [23]. We finally close with a discussion on related work for the implementation of the Society Infrastructure module.

7.3.1 Platforms based on a Directory Facilitator Approach

In PROSOCS we use P2P computing as the underlying model for the implementation of the reference model we presented in section 3.1. To deliver this implementation we use at the

lower level JXTA [56], a set of open protocols that allow any connected device on the network ranging from PCs and servers to cell phones and wireless PDAs to communicate and collaborate in an open but distributed manner. In PROSOCS we have seen how this lower level is linked with the logical part of the computee, via a mind-body computee architecture that links the reasoning capabilities of the mind with the physical actions carried out by the computee body that situates the computee in a networked environment. The motivation behind this approach has been to keep the link of PROSOCS and GC as close as possible, through the use of P2P protocols that projects in GC other than SOCS are currently developing.

The P2P approach used in PROSOCS contrasts with the FIPA approach, followed by platforms such as JADE[16], FIPAOS[74], ZEUS[72], and 3APL [1] whose FIPA reference model relies on specific types of agents such as the Directory Facilitator (DF), Agent Management System (AMS) and Agent Communication Channel (ACC) to support agent management in a distributed network. The DF provides "yellow pages" services to other agents. The AMS and ACC support inter-agent communication. The ACC supports interoperability both within and across different platforms. The Internal Platform Message Transport (IPMT) provides a message routing service for agents on a particular platform which must be reliable, orderly and adhere to the requirements specified in the FIPA standards.

An approach similar to the DF approach is used by the IMPACT platform [52], which relies on a *yellow pages server* for keeping information about agents. However, in this approach when an agent is deployed the registration is done automatically by the roost the agent is deployed on (a roost is the runtime environment wrapper that houses agents in that platform), and thus an agent does not have to register explicitly by executing a communicative act.

One implication of our approach using JXTA compared to those that use a DF model is that when we start a computee in PROSOCS we do not have to make the computee register with a DF via an explicit communicative action (i.e. similarly to IMPACT). In PROSOCS the body of a computee uses JXTA protocols, through the platform's JXTA API, to make itself dynamically present in the environment (to be more precise we use JXTA's advertisement features combined with the notion of peer groups), as is the case with ordinary physical environments, where being present suffices. At any point in time, a computee can use the JXTA API provided by PROSOCS to obtain all the peers that are available in the environment. In other words, by using JXTA we push the presence of an agent in the implementation of the platform's medium, so that the developer will not need to think about the use of yellow pages. This is arguably closer – at least conceptually – to the social organisation of a MAS application.

7.3.2 Platforms that treat Agents as First Class Objects

In PROSOCS a computee is treated as a first-class object in the sense that the developer can start a computee and inherit a set of tools that support the development of a reasoning component and the interaction with the environment for free. The computee has a specific mind-body architecture inspired by the architecture originally developed by Bell [14] and recently applied to build web-based agents in [51]. The advantage of looking at the mind and the body of the computee functioning as co-routines is very useful in that a computee can execute an action, often simultaneously with the activities of reasoning, planning and observation.

Our work with PROSOCS goes one step further from those described in [14, 51], in that, we are also providing a very specific computational theory for building the mind of an agent, based on proof–procedures, capabilities, transitions, and cycle theories. This seems to be missing from platforms such as the ones described in [16, 74, 72] that, when creating an agent they all

provide what we call a body without any reusable tools that will allow the developer to build easily a mind. The latter will have to be supplied by the developer separately, without a lot of help from the platform.

IMPACT. The IMPACT platform treats agents as first-class objects too[12, 89], facilitating the creation, deployment, interaction, and collaborative aspects of applications in a heterogeneous, distributed environment. We have seen already that IMPACT relies on servers (yellow pages, thesaurus, registration, type and interface) that facilitate agent inter-operability in an application independent manner. It also provides an Agent Development Environment for creating, testing, and deploying agents. In this context the agent's components are:

- *Application Program Interface* (API): provides a set of functions which may be used to manipulate the data structures managed by the agent in question.
- *Service Description*: specifies the set of services offered by the agent.
- *Message Manager*: manages the incoming and outgoing messages of the agent.
- *Actions, Action Policies, and Constraints*: describe the set of actions that the agent can physically perform, an associated action policy that states the conditions under which the agent may, may not or must do some actions.
- *Meta–knowledge*: holds beliefs about the environment and other agents, used to produce action policies.
- *Temporal Reasoning*: supports an agent to schedule actions that take place in the future, which could be interpreted as the commitments of the agent.
- *Reasoning with Uncertainty*: allows the agent to take into account that a state can be uncertain. For example, based on its sensors, an agent may have uncertain beliefs about the properties of the environment's state, as well as uncertainty about how the environment is likely to change.
- *Security*: supports the designer of the agent to enforce security policies according to the application requirements of the agent.

Although the mind-body architecture of PROSOCS provides a more intuitive separation between the part of the agent that is required to interact with the environment and the part that supports the reasoning, from the specification point of view, IMPACT agents are a superset of PROSOCS agents. As with PROSOCS they are supported by execution mechanisms whose to their computational features as well as their complexity are well understood. However, IMPACT provides generic types of agents that the programmer can reuse across applications which PROSOCS does not provide. On the other hand, IMPACT does not provide a Social infrastructure like PROSOCS, as this system was originally developed with different aims, namely, integrating heterogeneous sources of information, where the notion of society plays a less prominent role.

3APL. Similar to the PROSOCS approach, when an agent is started in the 3APL platform [1] the system provides a set of tools that support the reasoning capabilities of the agent (for the mind), including a deliberation cycle. This platform also supports a more sophisticated interface for the agent than in PROSOCS, that allows the developer to edit the different belief, action and goal bases that are supported. In addition, the system supports communication amongst agents based on the FIPA DF approach (which - to the best of our knowledge - is not yet fully implemented). Moreover, the platform provides support for a sniffer agent that can present the communication that has been exchanged between agents running on the platform (this feature is a limited version of what is already available in [16]). However, the platform does not have a Society Infrastructure to test for conformance of protocols such as PROSOCS. We could not find any additional information about the reference model of a 3APL agent nor any implementation architecture of agents in 3APL, so that to provide a more detailed the comparison.

SIM_Speak. This platform [64] results from an implementation of the AgentSpeak(L) language [75] and its interpreter using the *SIM_AGENT* toolkit[82, 81]. In *SIM_Speak* an agent state comprises of a set of events E, a set of beliefs B, and a set of intentions I. Events are either external or internal. The external events correspond to inputs from the agent's environment. Internal events are generated during execution of a plan.

As in AgentSpeak(L), an agent executes in a cycle, which is similar to an instance of the PROSOCS cycle theory, as follows: select some event e from the pending set of events E using an event selection function SE; look for and choose an applicable plan p for e using a plan selector function SP; generate a new intention (e,p) if e is external, extend intention i with p if e is an achieve goal event generated by i; select an intention i from the current set of intentions I using an intention selector SI; execute the next goal or action of i.

Asynchronously the environment is adding events to the agent's event store, and it is asynchronously absorbing the agents actions. Each i in I is an execution thread and corresponds to the stack of plans currently being used to respond to some external event. However, the agent explicitly time shares between them using SI, viz., it has no internal concurrency. There is also no inter-agent communication model in the language - no special communication actions. Also, the three selection functions SE, SP, SI are black boxes, there is no means of programming them within the AgentSpeak(L) language.

7.3.3 Commercial-grade platforms

Apart from platforms that are freely available, there are also a number of platforms with commercial-grade interfaces and programming tools, such as AgentBuilder [3] and JACK[2], that support too the notion of agents as first class objects.

AgentBuilder. AgentBuilder[3] is based on the Agent0 [79] model and the extensions outlined for this model in the Placa [91] language. An agent is defined by the developer specifying behavioural rules, initial beliefs, commitments, intentions and agent capabilities. These are similar to specifying the different PROSOCS knowledge bases based on a basic agent class. However, as most agents require more capabilities than the basic agent class, the notion of Project Accessory Classes (PACs) is introduced. PACs are custom classes coded in Java and designed to perform specific tasks that augment the basic agent's behaviour.

To interpret the internal state of the agent, AgentBuilder provides a Run-Time Agent Engine. This is a high-performance mechanism that interprets the agent program and performs actions specified in agent action libraries, the user interface, or communication with other agents supported via a communication module. The Run-Time Agent Engine is similar to providing an implementation for a normal cycle theory together with a body and a common interface in PROSOCS.

JACK. JACK[2] is an environment for building, running and integrating commercial-grade multi-agent systems using a component-based approach. The system is based on the BDI model dMARS [32]. The developer of an agent in JACK can fully describe the functionality of an agent by extending an agent class (similar to what we call in PROSOCS an agent body) with a number of components using the JACK Agent Language. In general, the agent class allows the developer to add to an agent reasoning, communication, and interaction capabilities implementing the following conceptual components:

- BeliefSets and Views which the agent can use and refer to;
- Events (both internal and external) that the agent is prepared to handle;
- Plans that the agent can execute;
- Events the agent can post internally (to be handled by other plans);
- Events the agent can send externally to other agents.

Unlike PROSOCS, where we use a mixed approach of Computational Logic and Java, the JACK Agent Language is a super-set of Java - encompassing the full Java syntax while extending it with constructs to represent agent-oriented features. Each of the Java extensions are included in JACK along with their expected usage and semantic behaviour. A compiler pre-processes JACK Agent Language source files and converts them into pure Java. This Java source code can then be compiled into Java Virtual Machine code to run on the target system. The JACK Agent Kernel is the runtime engine for programs written in the JACK Agent Language. It provides a set of classes that give JACK Agent Language programs their agent-oriented functionality. Most of these classes run behind the scenes and implement the underlying infrastructure and functionality that agents require, while others are used explicitly in JACK Agent Language programs, inherited from and supplemented with callbacks as required to provide agents with their own unique functionality.

7.3.4 Agent Programming Languages

IndiGolog IndiGolog [46] is a high-level programming language for robots and intelligent agents that supports on-line planning and plan execution in dynamic and incompletely known environments. Programs may perform sensing actions that acquire information at runtime and react to exogenous actions. It could be used to write robot control programs that combine planning, sensing, and reactivity.

IndiGolog supports complex agents that are:

- able to do reasoning and planning;
- able to react to exogenous events;

- able to monitor plan execution and sense the environment;
- both reactive and proactive;
- written using very high-level language constructs.

IndiGolog is a member of Golog family of languages. Golog [61] is a language for expressing high-level programs for robots and autonomous agents. Golog supports program structures such as sequence, conditionals, loops, and non-deterministic choice of actions and arguments. It uses a Situation Calculus theory of action to perform the reasoning required in executing the program.

An extension of Golog called ConGolog [45] was introduced later, adding support for concurrent processes with possibly different priorities, interrupts, and exogenous events. These new constructs were useful for writing controllers that react to environmental events while working on certain tasks. However, key features of real-world agent and robot applications are that the environment is dynamic and that the system has incomplete knowledge and must acquire information at run-time by performing sensing actions. Like earlier planning-based systems, Golog and ConGolog, assume an off-line search model. That is, the interpreter is taken to search all the way to a final state of a program before any action is really executed. This can be a serious problem if, for instance, the program involves a long running application, or if part of the program depends on information that can only be obtained by doing sensing at run-time. It is also impractical to spend large amounts of time searching for a complete plan when the environment is very dynamic.

Although IndiGolog addressed the limitations of Golog and ConGolog, the notion of exogenous actions in it is based on the assumption that there is a concurrent process executing these actions outside the control of the agent. In other words, one main difference between IndiGolog and the language of abductive logic programming that is available in PROSOCS, is that IndiGolog assumes that it should be the developer of an application that should interface the language to application domains of a distributed systems nature, while with PROSOCS the support for distribution comes for free. In addition, IndiGolog is not based on any specific agent architecture, while in PROSOCS a computee is built with a specific architecture encompassing a mind and a body.

Go!. The logic programming language Go![23] is descendant of the multi-threaded symbolic programming language April[38], with influences from IC-Prolog II[22] and L&O[65]. Go! has many features in common with Prolog, particularly multi-threaded Prologs, however, there are significant differences related to transparency of code and security. Features of Prolog that mitigate against transparency, such as the infamous cut (!) primitive, are absent from Go!. Instead, its main uses are supported by higher level programming constructs, such as single solution calls, iff rules, and the ability to define 'functional' relations as functions.

Go! is strongly typed to reduce the programmer's burden. The programmer can declare new types and thereby introduce new data constructors. Go! is also multi-threaded, a feature which the developers of this system consider essential for building sophisticated agents. Threads primarily communicate through asynchronous message passing. Threads, executing action rules, react to received messages using pattern matching and pattern based message reaction rules. A communications daemon enables threads in different Go! processes to communicate transparently over a network. Typically, each agent will comprise several threads, each of which can directly communicate with threads in other agents.

Threads in the same Go! process, hence in the same agent, can also communicate by manipulating shared cell or dynamic relation objects. Updates of these objects are atomic. Moreover, threads can be made to suspend until a term unifying with some given term is added to a shared dynamic relation by some other thread. This enables dynamic relations to be used to coordinate the activities of different threads within an agent. This is a powerful implementation abstraction for building multi-threaded agents.

Go! does not directly support any specific agent architecture or agent programming methodology, hence the system does not support agents as first class objects. In addition, facilities such as planning, temporal reasoning, and preference reasoning, the programmer has to build from scratch (although they could be reused using library modules, once they are developed).

7.3.5 Social Infrastructures

We would also like to discuss briefly two existing implementations of social frameworks.

The social approach to the definition of interaction protocols and semantics of Agent Communication Languages has been documented in several noteworthy contributions of the past years. Other frameworks are proposed in the literature, aimed 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. Among them, as we cited in D8, [13], present a formal framework for specifying systems where the behaviour of the members and their interactions cannot be predicted in advance, and for reasoning about and verifying the properties of such systems. The authors provide a tool (*Society Visualiser*) to demonstrate animations of protocol runs in such systems. The Society Visualiser's main purpose is to explicitly represent the institutional power of the members and the concept of valid action. Our work is not based on any deontic infrastructure. For this reason, our framework can be used for a broader spectrum of application domains, from intelligent agents to reactive systems.

ISLANDER [36] is a tool for the specification and verification of interaction in complex social infrastructures, such as electronic institutions. ISLANDER allows to analyse situations, called scenes, and visualise liveness or safeness properties in some specific settings. The kind of verification involved is static and is used to help designing institutions. Although our framework could also be used at design time, its main intended use is for on-the-fly verification of heterogeneous and open systems.

Part IV
CONCLUSION

8 Summary

The prototype demonstrator of SOCS is a set of application examples developed using the PROSOCS platform, a system supporting the programming of autonomous entities called computees, viz., software entities specified in computational logic and operating in global computing environments, and then societies via a Society Infrastructure. The PROSOCS platform combines advanced computational logic techniques to build the reasoning capabilities of a computee, with peer-to-peer computing techniques to allow an embodied computee to interact and communicate with an open and unpredictable environment.

A modular approach allows a user to program computees in terms of components, in particular a mind and a body. These function as co-routines and in turn consist of additional components represented as extended logic programs (e.g. the mind's cycle theory and knowledge) or concurrent objects and their methods (e.g. the body's control, the sensors, and the effectors). The advantage of looking at the mind and the body of the computee functioning as co-routines is that a computee can execute an action, often simultaneously with the activities of reasoning, planning and observation.

One of the strengths of the platform is that it provides a Social Infrastructure component that allows interactions amongst computees to be regulated, according to a set of rules that represent the ideal interactions of computees and their societies. In this context, a society is represented as a set of protocols and a knowledge base that are different for different applications.

We have exemplified the use of the PROSOCS platform and we have developed a series of examples (documented in [7]) showing how to use the resulting generic functionality to build a simple application for the provision of location-independent smart services. This application is intended to demonstrate the feasibility of functionalities and computational viability of the logical models developed in the SOCS project (WP1, WP2, and WP3). It is also intended as the starting point for the final year of the project, where larger scale experiments are being planned (WP6).

We have also evaluated the implementation and placed it in the context of what we think are the most relevant existing works in the literature. Both the evaluation of our work and the review of related work give rise to the next section, where we discuss our immediate plans for future work.

9 Future work

We will continue with the experimentation based on the current examples [7] document. In parallel, additional experimentation will be based on the second example document that we have developed in [31], which is also attached as part of this deliverable. In this way we hope to continue with the testing and the debugging of the prototype, to prepare it for the final year experimentation plans (WP6).

Our immediate plan for future work for improving computees is as follows.

- Focus on the scalability of the C-IFF and Gorgias proof-procedures;
- Extend the temporal reasoning capability in order to make it more efficient and relax the simplifying assumptions made in D8. This would lead to a TR theory able to deal with non–ground theories, containing narrations with events occurring at existentially quantified time points within intervals, and able to recover from a given narrative with inconsistent observations.
- Investigate how to represent and implement variables other than temporal variables and work alongside with the developments of the computational model of D8 in this direction.
- Incorporate an environment computee that can be reused across applications to support physical actions and interactions of computees with the environment.
- Investigate the practical problems of incorporating the notion of global time in the system, so that the system can reason with deadlines.

In our future work about the social infrastructure, we are planning the following tasks.

- Allow for a tighter interaction between societies and computees. We show some motivating example in an attached document of this deliverable [31].
- Have the society actively communicate expectations to computees so that they can be used as instances for helping the computees in the goal decision procedure, and for managing trust and reputation in open societies.
- Try to make the proof of the society more efficient and scalable as the number of social events increases. In the next year we will do an experimental evaluation of the performance of the implemented SCIFF, which will help us making choices in this respect.
- Augment SOCSDemo with a set of protocols from the literature, to facilitate integration with legacy systems and heterogeneity. We already started doing this, taking some protocols from FIPA/AUML.

Acknowledgments

Thanks to Nicolas Maudet for implementing a first version of the transitions in PROSOCS, while employed at City University for this project.

References

[1] 3APL Platform User Guide. http://www.cd.uu.nl/3apl/download.html, visited Dec. 2003.

[2] Agent Oriented Software (AOS), Carlton, Victoria. 2001. Version 3.0. http://www.jackagents.com.

[3] The AgentBuilder Web-site. http://www.agentbuilder.com/, visited Dec 2003.

[4] The AgentBuilder Web-site: Agent development tools. http://www.agentbuilder.com/AgentTools/index.html, visited Dec 2003.

[5] AGILE: Architectures for Mobility IST–2001–32747, 2003.

[6] M. Alberti, A. Bracciali, F. Chesani, N. Demetriou, U. Endriss, A. Kakas, W. Lu, , K. Stathis, and P. Torroni. Socsdemo user manual. Discussion Note IST3250/CITY/011/DN/I/a2, SOCS Consortium, Dec. 2003.

[7] M. Alberti, A. Bracciali, F. Chesani, N. Demetriou, U. Endriss, F. Sadri, A. Kakas, E. Lamma, W. Lu, N. Maudet, P. Mello, M. Milano, K. Stathis, G. Terreni, F. Toni, and P. Torroni. Examples for the functioning of computees and their societies. Discussion Note IST3250/ICSTM//DN/I/a2, SOCS Consortium, Dec. 2003.

[8] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Compliance verification of agent interaction: a logic-based tool. 2004. Submitted.

[9] M. Alberti, A. Ciampolini, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Logic Based Semantics for an Agent Communication Language. In B. Dunin-Keplicz and R. Verbrugge, editors, *Proceedings of the International Workshop on Formal Approaches to Multi-Agent Systems (FAMAS)*, pages 21–36, Warsaw, Poland, Apr. 12 2003.

[10] 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 *Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004). Special Track on Agents, Interactions, Mobility, and Systems (AIMS)*, Nicosa, Cyprus, Mar. 14–17 2004. ACM Press. to appear.

[11] M. Alberti, S. Melchiori, and P. Torroni. A feasibility study for the implementation of a prototype demonstrator of societies of computees using jade and prolog. Internal document. attached to deliverable [18], SOCS Consortium, 2003.

[12] K. A. Arisha, F. Ozcan, R. Ross, V. S. Subrahmanian, T. Eiter, and S. Kraus. IMPACT: a Platform for Collaborating Agents. *IEEE Intelligent Systems*, 14(2):64–72, March/April 1999.

[13] 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.

[14] J. Bell. A Planning Theory of Practical Rationality. In *Proceedings of AAAI'95 Fall Symposium on Rational Agency*, pages 1–4. AAAI Press, 1995.

[15] P. Bellavista, A. Corradi, and C. Stefanelli. A mobile agent infrastructure for the mobility support. In *SAC (2)*, pages 539–546, 2000.

[16] F. Bellifemine, A. Poggi, and G. Rimassa. JADE: a FIPA2000 compliant agent development environment. In J. P. Mller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 216–217. ACM Press, May 2001.

[17] A. Bracciali, A. C. Kakas, E. Lamma, P. Mello, K. Stathis, F. Toni, and P. Torroni. D11: Evaluation and self assessment. Technical report, SOCS Consortium, 2003. Deliverable D11.

[18] A. Bracciali, T. Kakas, E. Lamma, F. Sadri, K. Stathis, and F. Toni. Wp1-wp4: progress report. Technical report, SOCS Consortium, 2003. Deliverable D7.

[19] F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6(1):67–94, 1997.

[20] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.

[21] R. Y. Chen and B. Yeager. Java mobile agents on project jxta peer-to-peer platform. In *36th Hawaii International Conference on System Sciences (HICSS'03)*. IEEE Computer Society, 2003.

[22] D. Chu. and K. L. Clark. IC-Prolog II: A Multi-threaded Prolog System. In *Proceedings of the ICLP-93 Post-conference Workshop on Concurrent, Distributed and Parallel Implementations of Logic Programming*, Budapest, 1993.

[23] K. Clark and F. McCabe. Go! for multi-threaded deliberative agents. to appear in Annals of Mathematics and AI, also available http://www.doc.ic.ac.uk/ klc/gowp.html, 2003.

[24] K. Clark and P. Robinson. Agents as multi-threaded logical objects. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, pages 33–65. Springer-Verlag, LNAI 2407, 2002.

[25] K. Clark, N. Skarmeas, and F. McCabe. Agents as clonable objects with knowledge base state. In V. Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'96)*, Kyoto, Japan, 1996. The MIT Press: Cambridge, MA, USA.

[26] M. Cohen and K. Stathis. Strategic Change stemming from E-Commerce: Implications of Multi-Agent Systems in the Supply Chain. *Strategic Change*, 10:139–149, 2001.

[27] A. Corradi, R. Montanari, E. Lupu, and C. Stefanelli. Policy controlled mobility.

[28] P. Dart, E. Kazmierczak, M. Martelli, V. Mascardi, L. Sterling, V. Subrahmanian, and F. Zini. Combining Logical Agents with Rapid Prototyping for Engineering Distributed Applications. In *Proc. of 9th International Conference of Software Technology and Engineering (STEP'99)*, Pittsburgh, PA, September 1999. IEEE.

[29] 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, Dec. 2001. 2nd International Workshop (ESAW'01), Prague, Czech Republic, 7 July 2001, Revised Papers.

[30] O. deBruijn and K. Stathis. Socio–Cognitive Grids: The Net as a Universal Human Resource. In *Proceedings of Tales of the Disappearing Computer*, pages 211–218. CTI Press, 2003.

[31] N. Demetriou, A. Kakas, and P. Torroni. Further examples of the functioning of computees. Technical report, SOCS Consortium, 2003. Examples Document.

[32] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. Singh, A. Rao, and M. Wooldridge, editors, *Intelligent Agents V, Agent Theories, Architectures, and Languages, 5th International Workshop, ATAL '98, Paris, France, Proceedings*, number 1365 in Lecture Notes in Artificial Intelligence, pages 155–176. Springer-Verlag, 1998.

[33] M. d'Orazi Flavoni. A qu-prolog implementation of the IFF proof procedure and example application use for communicating agents. Technical report, Department of Computing, Imperial College, London, UK, 2002. Advanced MSc project.

[34] T. Eiter and V. Mascardi. A comparison of environments for developing software agents. *AI Communications*, 15:169–197, 2002.

[35] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Resource Allocation by Negotiation. Technical Report IST32530/ICSTM/019/IN/I/a2, 2002.

[36] 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.

[37] B. T. et al. The Project JXTA Virtual Network. http://www.jxta.org/docs/JXTAprotocols.pdf, May 2001.

[38] M. F.G. and C. K.L. April: Agent process interaction language. In W. M. J. and J. N. R., editors, *Intelligent Agents*, pages 324–340. LNCS, Vol. 890, Springer Verlang, 1995.

[39] FIPA: Foundation for Intelligent Physical Agents.

[40] FIPA agent management support for mobility specification, Aug. 2001. Published on August 10th, 2001, available for download from the FIPA website, http://www.fipa.org.

[41] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, Oct. 1998.

[42] A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361, May 1998.

[43] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, Nov. 1997.

[44] Global Computing: Co-operation of Autonomous and Mobile Entities in Dynamic Environments.

[45] G. D. Giacomo, Y. Lesperance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[46] G. D. Giacomo, H. J. Levesque, and S. Sardia. Incremental execution of guarded theories. *ACM Transactions on Computational Logic*, 2(4):495–525, October 2001.

[47] D. Gilbert and C. Palamidessi. Concurrent constraint programming with process mobility. *Lecture Notes in Computer Science*, 1861:463–??, 2000.

[48] Gorgias system manual. http://www.cs.ucy.ac.cy/ nkd/gorgias/ (visited 20/12/2003).

[49] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[50] C. Holzbaur. Specification of constraint based inference mechanism through extended unification. Dissertation, Dept. of Medical Cybernetics & AI, University of Vienna, 1990.

[51] Z. Huang, A. Eliens, , and P. de Bra. An Architecture for Web Agents. In *Proceedings of EUROMEDIA'01*. SCS, 2001.

[52] Impact Software Library User Documentation. http://www.cs.umd.edu/projects/impact/Docs/, visited Dec 2003.

[53] Agent Oriented Software Group Web-site. http://www.agent-software.com.au/shared/home/, visited Dec. 2003.

[54] J. Jaffar and M. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[55] The JAVA Programming Language. http://www.sun.com/software/java/.

[56] Project JXTA. http://www.jxta.org/, visited Dec 2003.

[57] A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. A logic-based approach to model computees. Technical report, SOCS Consortium, 2003. Deliverable D4.

[58] A. C. Kakas, E. Lamma, P. Mancarella, P. Mello, K. Stathis, and F. Toni. Computational model for computees and societies of computees. Technical report, SOCS Consortium, 2003. Deliverable D8.

[59] D. Kotz and R. Gray. Agent tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4), 1997.

[60] E. Lamma, P. Mello, P. Mancarella, A. Kakas, K. Stathis, and F. Toni. Self-assessment: parameters and criteria. Technical report, SOCS Consortium, 2003. Deliverable D3. Distribution restricted to the GC programme.

[61] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[62] B. Logan. Classifying agent systems. In J. Baxter and B. Logan, editors, *Software Tools for Developing Agents: Papers from the 1998 Workshop*, pages 11–21. AAAI Press, July 1998. Technical Report WS–98–10.

[63] W. Lu, N. Maudet, and K. Stathis. Building socio-cognitive grids by combining peer-to-peer computing with computational logic. In O. de Bruijn and K. Stathis, editors, *Proceedings of 1st International Workshop on Socio-Cognitive Grids*, pages 18–22, 2003.

[64] R. Machado and R. H. Bordini. Running AgentSpeak(l) agents on *sim_agent*. In M. d'Inverno and M. Luck, editors, *Working Notes of the Fourth UK Workshop on Multi-Agent Systems (UKMAS 2001)*, 13-14 December 2001.

[65] F. G. McCabe. *Logic and Objects*. International Series in Computer Science. Prentice Hall, 1992.

[66] P. Mello, P. Torroni, M. Gavanelli, M. Alberti, A. Ciampolini, M. Milano, A. Roli, E. Lamma, F. Riguzzi, and N. Maudet. A logic-based approach to model interaction amongst computees. Technical report, SOCS Consortium, 2003. Deliverable D5.

[67] P. Melo, M. Milano, A. Roli, R. Montanari, M. Gavanelli, E. Lamma, and F. Riguzzi. Combinatorial Auctions. Technical Report IST32530/UNIBO/0XX/IN/I/a2, 2002.

[68] MIKADO: Mobile Calculus based on Domains (IST–2001–32222), 2003.

[69] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Part I and II.*, 100(1):1–77, 1992.

[70] MRG: Mobile Resources Guarantees IST–2001–33149, 2003.

[71] MYTHS: Models and Types for Security in Mobile Distributed Systems IST–2001–32617. Web page, 2001.

[72] H. S. Nwana, D. T. Ndumu, L. C. Lee, and J. C. Collis. ZEUS: a toolkit and approach for building distributed multi-agent systems. In O. Etzioni, J. P. Müller, and J. M. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 360–361, Seattle, WA, USA, 1999. ACM Press.

[73] Object management group: Mobile agent system interoperability facility specification.

[74] S. Poslad, P. Buckle, and R. Hadingham. The FIPA-OS agent platform: Open Source for Open Sstandards. In *Proceedings of PAAM'00*, pages 355–368, 2000.

[75] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In R. van Hoe, editor, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer-Verlag, 1996.

[76] P.-M. Ricordel and Y. Demazeau. From analysis to deployment: A multi-agent platform survey. In A. Omicini, R. Tolksdorf, and F. Zambonelli, editors, *Engineering Societies in the Agents World*, volume 1972 of *LNAI*, pages 93–105. Springer-Verlag, Dec. 2000. 1st International Workshop (ESAW'00), Berlin (Germany), 21 Aug. 2000, Revised Papers.

[77] A. Serenko and B. Detlor. Agent Toolkits: A General Overview of the Market. Technical Report Working paper 455, McMaster University, Hamilton, Ontario, July, 2002.

[78] M. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 1055–1060, 1989.

[79] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[80] SICStus prolog user manual, release 3.8.4, May 2000.

[81] SIM_AGENT Web-Site. http://www.cs.bham.ac.uk/ axs/cog_affect/sim_agent.html, visited Dec 2003.

[82] A. Sloman and B. Logan. Building cognitively rich agents using the SIM_Agent toolkit. *Communications of the ACM*, 42(3):71–73, 75, 1999.

[83] Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. http://lia.deis.unibo.it/Research/SOCS/.

[84] J. Stamos and D. Grifford. Implementing remote evaluation. *IEEE Transactions on Software Engineering*, 16(7), 1990.

[85] K. Stathis. Location-aware SOCS: The Leaving San Vincenzo scenario. Technical Report IST32530/CITY/002/IN/PP/a1, 2002.

[86] K. Stathis, C. Child, W. Lu, and G. K. Lekeas. Agents and Environments. Technical Report Technical Report IST32530/CITY/005/DN/I/a1, 2002.

[87] K. Stathis, O. deBruijn, and S. Macedo. Living memory: agent-based information managment for connected local communities. *Interacting with Computers*, 14(6):665–690, 2002.

[88] K. Stathis, W. Lu, N. Demetriou, A. Kakas, U. Endriss, and A. Bracciali. PROSOCS: A platform for programming software agents in computational logic. In J. Müller and P. Petta, editors, *Fourth International Symposium from Agent Theory to Agent Implementations*, to appear, Vienna 2004.

[89] V. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross. *Heterogenous Active Agents*. MIT-Press, 2000.

[90] P. Tarau. Jinni: Intelligent mobile agent programming at the intersection of java and prolog. In *Proceedings PAAM'99, London, UK*, 1999.

[91] S. R. Thomas. The PLACA agent programming language. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents*, Berlin, 1995. Springer-Verlag.

[92] F. Toni and K. Stathis. Access-as-you-need: a computational logic framework for flexible resource access in artificial societies. In *Proceedings of the Third International Workshop on Engineering Societies in the Agents World (ESAW'02)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2002.

[93] B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J. C. Hugly, and E. Pouyoul. Project JXTA-C:Enabling a Web of Things. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*, pages 282–287. IEEE Press, January 2003.

[94] Extensible Mark-up Language - (XML). http://www.w3.org/XML/, visited Dec 2003.

10 Appendices

10.1 Protocol definition language

```
#   Ics is a list of integrity constraints. It can be an empty list.
Ics                 ::=     ( IC )*
IC                  ::=     BodyIC ImplSymbol HeadIC FullStopSymbol

#   Syntax of BodyIC
BodyIC              ::=     [ NotSymbol ]( HEvent | Expectation )
                            ( AndSymbol [ NotSymbol ] BodyLiteral )*
BodyLiteral         ::=     HEvent | Expectation | Literal | Constraint
HEvent              ::=     HappenedSymbol LBracket Event "," Time RBracket
Event               ::=     CompoundTerm
Time                ::=     Variable | Num
Literal             ::=     CompoundTerm
CompoundTerm        ::=     TermId  [ LBracket Term ( "," Term )* RBracket ]
Term                ::=     Num | Variable | CompoundTerm | List |
                            (LBracket Term ( "," Term)* RBracket)
TermId              ::=     ConstantLiteral
List                ::=     LSquare [ Term  ( "," Term )* ] RSquare

#   Syntax of HeadIC
HeadIC              ::=     HeadDisjunct ( OrSymbol HeadDisjunct )* |
                            BottomSymbol
HeadDisjunct        ::=     [ NotSymbol ] Expectation ( AndSymbol [ NotSymbol ]
                            (Expectation | Constraint | CompoundTerm) )*

#   Syntax of Expectations
Expectation         ::=     PosExpectation | NegExpectation
PosExpectation      ::=     PosExpSymbol LBracket Event "," Time RBracket
NegExpectation      ::=     NegExpSymbol LBracket Event "," Time RBracket

#   Syntax of CLP Constraints
Constraint          ::=     CLPConstraint | UnifConstraint
CLPConstraint       ::=     Expr Relop Expr
Expr                ::=     AtomicExpr [ Op AtomicExpr ]
AtomicExpr          ::=     Variable | Num
Relop               ::=     EqualSymbol | NotEqualSymbol | LessThanSymbol |
                            GreaterThanSymbol | LessEqualSymbol |
                            GreaterEqualSymbol
Op                  ::=     PlusSymbol | MinusSymbol | ProductSymbol |
                            DivSymbol
UnifConstraint      ::=     Term UnifRelop Term
UnifRelop           ::=     UnifSymbol | NotUnifSymbol
```

```
#   Identifiers
Variable            ::=     ('A'-'Z'|'_') ( Character )*
ConstantLiteral     ::=     ( 'a'-'z' ) ( Character )*
Character           ::=     ( 'a'-'z' | 'A'-'Z' | '0'-'9' | '_' )
Num                 ::=     "0" | ( ('1'-'9') ('0'-'9')* )

#    Symbols
AndSymbol           ::=     "/\"
BottomSymbol        ::=     "false"
DivSymbol           ::=     "/"
EqualSymbol         ::=     "=="
FullStopSymbol      ::=     "."
GreaterEqualSymbol  ::=     ">="
GreaterThanSymbol   ::=     ">"
HappenedSymbol      ::=     "H"
ImplSymbol          ::=     "--->"
LBracket            ::=     "("
LessEqualSymbol     ::=     "<="
LessThanSymbol      ::=     "<"
LSquare             ::=     "["
MinusSymbol         ::=     "-"
NegExpSymbol        ::=     "NE"
NotEqualSymbol      ::=     "<>"
NotSymbol           ::=     "!"
NotUnifSymbol       ::=     "!="
OrSymbol            ::=     "\/"
PlusSymbol          ::=     "+"
PosExpSymbol        ::=     "E"
ProductSymbol       ::=     "*"
RBracket            ::=     ")"
RSquare             ::=     "]"
UnifSymbol          ::=     "="
```

10.2 A brief introduction to Constraint Handling Rules

Constraint Handling Rules [41] (*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 social proof-procedure.

Simplification CHRs. Simplification rules are of the form

$$H_1, \ldots, H_i \iff G_1, \ldots, G_j | B_1, \ldots, B_k \tag{2}$$

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 constraints, 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

$$H_1, \ldots, H_i \Longrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \tag{3}$$

where the symbols have the same meaning and constraints of those in the simplification rules (2).

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

$$H_1, \ldots, H_l \backslash H_{l+1}, \ldots, H_i \Longleftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \tag{4}$$

where $l > 0$ and the other symbols have the same meaning and constraints of those of simplification *CHR*s (2).

Declaratively, the rule of Eq. (4) is equivalent to

$$H_1, \ldots, H_l, H_{l+1}, \ldots, H_i \Longleftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k, H_1, \ldots, H_l \tag{5}$$

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 $B_1, \ldots, B_k$.