SOCS

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

IST-2001-32530

The SOCSDemo User Manual

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/011/IN/I/a1
File name:	3011-a1[socs-demo].pdf
Editor:	K. Stathis
Contributing partners:	ALL
Contributing workpackages:	WP4
Estimated person months:	1
Date of completion:	29 January 2004
Date of delivery to the EC:	31 January 2004
Number of pages:	28

ABSTRACT
We explain how to download, install, and run the examples of the SOCSDemo. We also outline the kind of knowledge representation and steps required for a user of the SOCSDemo to specify in the PROSOCS platform new examples, with the aim of executing them in the platform. Our longer term goal is to identify the issues that need to be addressed in order build a complete user manual for PROSOCS by the end of the SOCS project.

The SOCSDemo User Manual

Marco Alberti [6] and Andrea Bracciali[2] and Federico Chesani[5] andNeophytos Demetriou [4] and Ulle Endriss [1] and Antonis Kakas and Wenjin Lu [3] and Kostas Stathis[3] and Paolo Torroni[5]

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

[2] Dipartimento di Informatica, Università degli Studi di Pisa
Email: braccia@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: {fchesani,ptorroni}@deis.unibo.it

[6] Dipartimento di Ingegneria, Università degli Studi di Ferrara
Email: {malberti}@deis.unibo.it

ABSTRACT
We explain how to download, install, and run the examples of the SOCSDemo. We also outline the kind of knowledge representation and steps required for a user of the SOCSDemo to specify in the PROSOCS platform new examples, with the aim of executing them in the platform. Our longer term goal is to identify the issues that need to be addressed in order build a complete user manual for PROSOCS by the end of the SOCS project.

Contents

1 Introduction

The SOCSDemo is the prototype demonstrator of societies of computees as implemented using the PROSOCS platform [2]. In this context, the purpose of this document is twofold:

- to explain how to download, install and run the already available examples of SOCSDemo; and
- to provide an overview of the knowledge representation that a user will have to understand in order to write new examples using the PROSOCS platform.

We must say at the outset that by writing this document is not intended to provide an explanation of how a user can program PROSOCS. Instead, our goal is more of wanting to give an idea what it takes to build examples of the kind demonstrated by the prototype. In particular, our concern is more on the knowledge reprsentation used and not on providing a coherent example explaining all the required details.

We do, however, want to support a user, once the SOCSDemo has been correctly installed, to run components of the PROSOCS platform that will execute the examples and view the behavior of computees/societies through the provided Graphical User Interfaces (GUIs).

The document is structured as follows. Section 2 explains how to download and install the SOCSDemo package. Section 3 illustrates how to use PROSOCS to start a computee component. In particular, we explain the usage of the GUI provided by a computee and we show how to run examples with it, including the knowledge represenation required to write new instances of computees. Then in section 4 we show how to start the PROSOCS society. More specifically, we explain the usage of the GUI, and we show how to run examples with it, including writing new instances of societies. Finally, in section 5, we explain how to animate both computees and societies in one single example; in doing so we use example 7 from the Examples document [1].

2 Installation

In this section, we show how to download and install the SOCSDemo package. Before that we will specify what the prerequisites are.

2.1 Prerequisites

In this section we list the prerequisites required to install SOCSDemo.

Architecture and OS Prerequisites This version of the SOCSDemo runs only on Microsoft Windows XP (both Home and Professional Edition) and MS Windows 2000 operating systems, on Intel hardware architectures(or Intel compatibles).

Software prerequisites A Java Software Development Kit must be installed prior to running the SOCSDemo. Java SDK version 1.4.2 or higher is recommended, otherwise the SOCS-Demo may not execute properly. The Java "`bin`" directory must be accessible through the path variable. It is possible to check whether it is accessible by opening the command prompt and typing "`java -version`". You should be able to see the version of the java installed. The output should be similar to the following:

```
Microsoft Windows XP [Versione 5.1.2600] (C) Copyright 1985-2001
Microsoft Corp.

C:\>java -version

java version "1.4.2_02"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_02-b03)
Java HotSpot(TM) Client VM (build 1.4.2_02-b03, mixed mode)

C:\>
```

If not, please refer to the Java manual of the installed version, and to the OS manual on how to add a directory to the PATH variable.

2.2 Installation Procedure

The SOCSDemo prototype is distributed as a zipped archive file. It can be downloaded from the url:
`http://lia.deis.unibo.it/research/projects/SOCS/partners/demonstrator/`
which is part of the private area of the SOCS home page. For accessing this url a reviewer can use the keyword `reviewer` both for username and password.

The installation procedure consists of extracting the content of the archive in a desired location; no further configuration steps or installation procedures are required. It is possible to use any archive compressor/decompressor able to process the zip format.

The files inside the archive are organized in a directory tree, which must be kept as it is. Most of the extraction tools keep the original tree structure by default.

It is possible to extract the prototype anywhere in the file system. At the end of the installation process, a new directory named "socs" has been created. Inside it three subdirectories are present:

`socs\demo` contains the demo examples and the script to execute them. It is structured as a set of subdirectories, each directory for a specific example.

`socs\docs` contains the documentation about this prototype, including also this manual.

`socs\platform` contains the binaries of the prototype, and it is organized in three more subdirectories: one related to computee binaries, one related to society, and a third one dedicated to all the common libraries used by both computees and societies.

3 Computees

3.1 Starting a Computee

A computee component is by default configured with standard parameters, which allow to start the computee without any particular configuration. There can be cases in which users need to modify the defaults parameters. The configuration values are stored in a text file, named `computee.config` and situated in the directory `socs\platform\computee`. It is possible to edit this file. For each parameter the file contains also a description of the meaning and the allowed values it can assume.

3.2 Starting a computee

If the installation process has been terminated successfully, inside the directory `socs\platform\computee` there is a script named `run.bat`. This script can be used for launching a generic computee. Here we describe how to execute a generic society using this script.

A computee component needs some base information before it is launched. More precisely, it needs to know which KB to use when it is started.

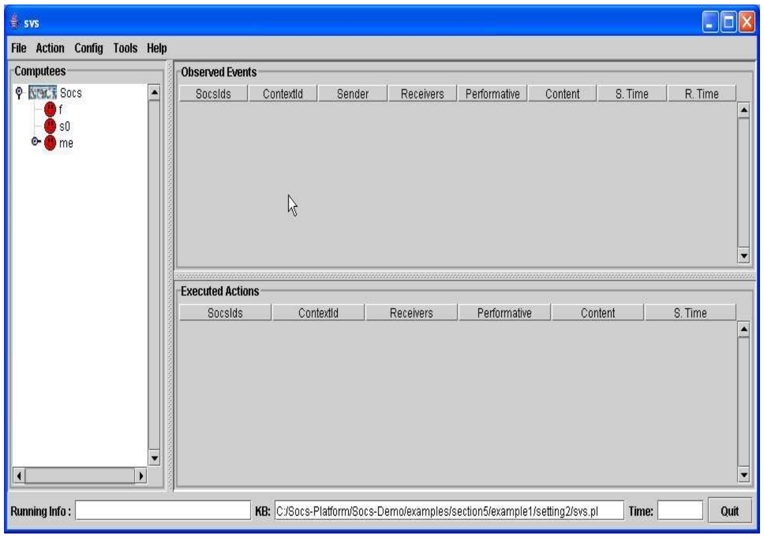
a) **By setting default values.** It is possible to specify the information in the `computee.config` file. Executing the script `run.bat` without any parameter will load the configuration values stored in the file.

b) **By specifying the values by the command line.** It is possible to specify the values on the command line, as parameters after the script name. For a detailed example, please type in at the command prompt the command `run --help` and read the instructions.

c) **Through the computee GUI.** It is possible to specify this parameter through the computee's Graphical User Interface. In this case we have to run the computee from the command prompt with the command `run --nokb`: the GUI of the computee will appear. We suggest that new users follow this procedure as opposed to (*a*) and (*b*) (because it is simpler).

3.3 The computee GUI

Once the user has started the computee, in a few seconds, the computee application Graphical User Interface (GUI) will be displayed on the screen, as shown in Figure 1.



SocsIds	ContextId	Sender	Receivers	Performative	Content	S. Time	R. Time

SocsIds	ContextId	Receivers	Performative	Content	S. Time

Running Info :

KB: C:/Socs-Platform/Socs-Demo/examples/section5/example1/setting2/svs.pl

Time:

Figure 1: Computee GUI

This user interface supports a user to:

- configure a computee by selecting the knowledge base (this will be typically be selected for you in the examples of the SOCSDemo);
- start/suspend/resume/stop the computee;
- communicate with other computees by sending/receving message manually;
- display and animate presence information of other online computees in the SOCS platform;
- display interactive information between the computee and other computees in the sociecty.

Configuration To configue a computee, a user at the moment can select the knowledge base where the initial state of the computee is specified.

1. To select a knowledge base, go to the menu "config → Select kb", a standard file chooser window will be displayed, from which a user can browse the files and select one as a knowledge base.

Control Computee After the configuration of the knowledge base, a computee can be controlled through the menus provided by the computee GUI.

1. To start a computee, go to the menu "file → start". By choosing this menu item the computee will be started.
2. While the computee is running, it can be suspended by selecting the manu "file → suspend". After selecting this menu item, the computee will be suspended.
3. In the suspend state, the computee can be resumed by selecting on the menu "file → resume". Once this item is chosen, the computee will resume from the state of the execution it was suspended.
4. A computee can be stopped to run at any state, by selecting the item "file → stop" from the menu. In the stop state, a user is allowed to re-configure the computee by selecting a different knowledge base.

Display Information The GUI also provides areas to display information about other online computees as well as interactions between the computee and other computees.

- Computees area: display and animated online computees in the society. By clicking on the menu "action → online computee", all the online computees in the society will be displayed in this area. Any change of the status (alive/suspended/stopped) of an online computee will automatically animated (smiley face/normal face/frowny face repsectively) of the computee.
- Observed events area: this displays the events (e.g. messages received by the computee) that happened in the society.
- Executed actions area: this presents every action executed (e.g. sent out messages) by the computee.

3.4 Writing the KBs of a Computee

3.4.1 Writing a KB_{GD}

In this section, we show how one can write KB_{GD} theories, which form the base for the goal decision (GD) capability, which in turn runs on top of the Gorgias system.

KB_{GD} Representation
Typically, the knowledge base KB_{GD} contains three parts: the *auxiliary part* with rules defining auxiliary predicates, the *lower-level part* with rules to generate goals and the *higher-level part* with rules that specify priorities between other rules of the theory. A subset of (fluent) predicates in the language of the knowledge base is separated out as the set of *goal fluents* of the computee.

Rules in the lower-level or goal generation part, of KB_{GD} are writen as Gorgias rules of the form:

```
kb_gd__rule(Label, gd(Goal), Arg_Body) :- Body.
```

where *Label* is a Prolog term naming this rule, *Goal* is a goal fluent literal of the form $(Literal, Time)$, and *Body* has the syntax of a Prolog rule body with conditions of the form $holds\_at(Literal, Time)$ or of the usual form of (definite) Prolog referring to auxiliary predicates. *Arg_Body* is usually the empty list as the rules of KB_{GD} cannot depend (see [3]) on a condition that is itself a goal or refers to a predicate on which we carry out preference/argumentation reasoning. Should we want to add such conditions then these are added here as a list. The overall name *kb_gd__rule* is necessary for every rule in KB_{GD} (for non-auxiliary predicates) in order to specify that the rule indeed belongs to the KB_{GD} module.

As an example, suppose that we have the following GD policy, where the only goal fluents are Leaving San Vincenzo (lsv) and low-battery alert (lba), given below in D4 syntax.

$$\begin{aligned}
r(lsv) : lsv(T) \leftarrow & \\
& holds\_at(finished\_work, T_{now}), \\
& T = T_{now} + 6. \\
r(lba) : lba(T) \leftarrow & \\
& holds\_at(low\_battery, T_{now}), \\
& T = T_{now} + 2.
\end{aligned}$$

These rules are coded in the KB_{GD} module as Gorgias rules:

```
kb_gd__rule(r(lsv,T), gd((lsv,T)), []) :-
    self__timestamp_current(Tnow),
    holds_at(finished_work, Tnow),
    T is Tnow + 6.

kb_gd__rule(r(lba,T), gd((lba,T)), []) :-
    self__timestamp_current(Tnow),
    holds_at(low_battery, Tnow),
    T is Tnow + 2.
```

where $self\_\_timestamp\_current/1$ is a system predicate that returns the current time at the time of application of GD. The user should not define this nor should it define rules for

$holds\_at(Literal, Time)$ which are evaluated by calling the temporal reasoning capability to check whether *Literal* holds at time T. In the current implementation equality and inequality constraints in the body of the rules are simply evaluated under the standard Prolog assumption that they will be fully ground when they are called.

The auxiliary part of KB_{GD} is just a definite [1] Prolog program that does not refer at all to any goal predicates of the theory. In addition, to accomodate conflicts of goals other than the ones due to the classical negation, KB_{GD} can include statements of incompatibility using the predicate $kb\_gd\_\_incompatible/2$. For example, the statement

```
    kb_gd__incompatible((lsv,T),(lba,T)).
```

states that the goals of lsv and lba are incompatible.

Finally, to express preferences over goal generation rules the higher-level part of KB_{GD} contains priority rules of the form:

```
    kb_gd__rule(Label, prefer(Label1,Label2), Arg_Body) : - Body.
```

where again *Label* is a Prolog term now naming this priority rule between two rules whose names are *Label*1 and *Label*2. *Body* and *Arg_Body* are as above for rules in the lower-level part. For example, the following code snippet

```
kb_gd__rule(gd_pref(X,Y), prefer(r(X,_), r(Y,_)), []) :-
    typeof(X,operational),
    typeof(Y,required).
```

codes the preference rule (in D4 sytle):

$$\begin{aligned} gd\_pref(X,Y) : r(X) > r(Y) \leftarrow\ & typeof(X, operational), \\ & typeof(Y, required). \end{aligned}$$

stating that operational goals are ranked higher than required goals. Hence for example if we have in the auxiliary part

```
    typeof(lsv, required).
    typeof(lba, operational).
```

then the goals lba are preferred over those of lsv.

Rules for non-ground goals

The specification of KB_{GD} in D4 allows rules that derive non-ground goals. For example, we can have the following rules:

$$\begin{aligned} r(lsv) : lsv(T), T < T' \leftarrow & \\ & holds\_at(finished\_work, T_{now}), \\ & T' = T_{now} + 6. \\ r(lba) : lba(T), T < T' \leftarrow & \\ & holds\_at(low\_battery, T_{now}), \\ & T' = T_{now} + 2. \end{aligned}$$

[1]In fact, this program can contain Negation as Failure provided that it is stratified, i.e it has no non-determinism in it.

In each of these rules the variable T that appears in the head is existentially quantified and it is constrained by constraints of the form $T_{low} < T < T_{high}$ where T is such that it does not appear in the conditions of the rule. These rules will be coded as follows:

```
rule(r((lsv,T,[Tnow<T<T'])), gd((lsv,T,[Tnow<T<T']), []) :-
    self__timestamp_current(Tnow),
    holds_at(finished_work, Tnow),
    T' is Tnow + 6.

rule(r((lba,T,[Tnow<T<T'])), gd((lba,T,[Tnow<T<T'])), []) :-
    self__timestamp_current(Tnow),
    holds_at(low_battery, Tnow),
    T' is Tnow + 2.
```

Note that in this example we have passed all the temporal constraints of the heads of the rule in their names. This may not be necessary if the conditions of the priority rules that the user wants to write do not depend on the times and their constraints.

3.4.2 Writing a KB_{react} and KB_{plan}

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. Recall that C-IFF operates on completed logic programs (with constraints) together with integrity constraints. The completion of logic programs is a service performed by the system. Therefore, in order to program the planning and the reactivity capabilities of a new computee, only a logic program (to be completed by the system) and a set of integrity constraint need to be provided. We first briefly review the syntax of logic programs and integrity constraints, before moving on to explain what knowledge needs to be defined in order to set up the planning and the reactivity capabilities.

The syntax for facts and rules of a logic program is familiar 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. To indicate negative fluents (arguments inside subgoals, heads of rules, or facts), on the other hand, we use the functor `neg/1`. Here is an example for a rule with all these different features:

```
holds(neg(Goal),T2) :-
  holds_initially(neg(Goal)),
  0#<T2,
  not(declipped(0,Goal,T2)).
```

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. Additional restrictions are required to be able to avoid the explicit representation of quantifiers. Appropriate allowedness conditions are given in deliverable D8.

The planning knowledge base KB_{plan} of each computee consists of two parts, a domain dependent and a domain independent part. The latter is provided by the system; it is stored in the file `kbplandi.alp`. A similar file for the domain dependant part of a computee's planning knowledge, say `kbplan-example.alp`, needs to be provided for each new computee. Such a file would typically provide rules and facts defining the following predicates: `holds_initially/1`, `initiates/3`, `terminates/3`, `precondition/2`, and `executable/1` (in addition, it may also define auxiliary predicates or it may extend the definition of predicates such as `holds/2` defined in `kbplandi.alp`). The first of these predicates defines the set of fluents that are assumed to be true initially. The next two are used to describe the effect of actions on the persistence of certain fluents. The fourth is used to define preconditions that need to be fulfilled before an action can be executed. The final predicate defines the range of actions that the computee is able to execute itself (as opposed to actions that may only be observed). In what follows, we give a number of examples.

For instance, to define that computee Bob is initially known as a customer within our society of computees, we may write the following fact into `kbplan-example.alp`:

```
holds_initially(customer(bob)).
```

To specify that the action of buying an object initiates the fluent of having that object we may use the following fact:

```
initiates(buy(Object),T,have(Object)).
```

Here, `T` refers to the time point at which this action takes place (it is in fact not relevant in this particular example). Selling an object again would terminate the fluent of having it:

```
terminates(sell(Object),T,have(Object)).
```

Next let us look at an example for the definition of a precondition. The following fact expresses that it is a precondition for a buying action to be executed that the object to be bought is not yet sold out:

```
precondition(buy(Object),neg(sold_out(Object))).
```

Recall that the `neg/1` operator is used to refer to negated fluents. In some cases, KB_{plan} may refer to both actions that the computee in question can execute and other actions that may only be observed. The former range of actions needs to be identified using the predicate `executable/1`. the following rule, for example, expresses that the computee Francisco may perform any kind of communicative action as long as it does not address itself with that action:

```
executable(tell(francisco,C,Subject,D)) :- not(C=francisco).
```

In cases where all actions referred to in the knowledge base are executable, we may use the following simple fact:

```
executable(Anything).
```

As we can see, many sample knowledge bases will simply consist of a list of facts. For examples of more complex rules to define planning knowledge we refer to the domain independent planning knowledge base specified in the file `kbplandi.alp`. Finally, we recall that the domain dependent part of KB_{plan} may also contain integrity constraints, although this is typically not the case. The planning capability will load the planning knowledge defined in `kbplan-example.alp` (or a any other file specified to contain the domain dependent planning knowledge) and merge it with the domain independent knowledge of `kbplandi.alp`. The system will compute the completion of the resulting abductive logic program and use it as a basis for any queries to the planner.

The reactivity knowledge base KB_{react} is an extension of KB_{plan}. In particular, the domain independent parts of both knowledge bases are identical. In principle, KB_{react} extends KB_{plan} only by a number of domain dependent integrity constraints, called reactive rules. In practice, however, the domain dependent part of KB_{react} may also define a number of auxiliary predicates (by means of giving rules or facts) and refer to these auxiliary predicates within the reactive rules. Reactive rules are integrity constraints (following the syntax defined earlier) over predicates used in the knowledge base $KB0$ (i.e. `observed/2`, `observed/3`, and `executed/2`), the predicate `holds/2` defined in KB_{plan}, the abducible predicate `assume_happens/2`, temporal constraints, and possibly the aforementioned auxiliary predicates. For details of what types of rules are allowed we refer to deliverable D8.

Let us now consider an example for such a reactive rule. We first define an auxiliary predicate to express that `assume_happens(Action,T)` holds for some time `T` after a particular reference time (here called the `Limit`):

```
  assume_happens_after(Action,T,Limit) :-
    assume_happens(Action,T),
    Limit#<T.
```

We are now in a position to write an integrity constraint (for the computee Francisco) expressing the following (part of a) negotiation strategy: Whenever another computee `C` requests a resource `R` and you do not have that resource at the time `T` the request is observed then reply at some later time `T1` refusing to give away `R`. Here is the corresponding reactive rule:

```
[observed(C,tell(C,francisco,request(give(R)),D,T0),T),holds(neg(have(R)),T)]
implies [assume_happens_after(tell(francisco,C,refuse(give(R)),D),T1,T)].
```

All such reactive rules and definitions of auxiliary predicates are stored in a file, say `kbreact-example.alp`, which will be merged with the planning knowledge base on starting the system. The resulting knowledge base will then be queried using the C-IFF proof procedure whenever the reactivity capability is called.

3.4.3 Writing a KB_{TR}

The computational model for the Temporal Reasoning (TR) capability (see Deliverable D8 for its definition) builds upon the computational model of (Constraint) Abductive Logic Programmming (namely, the CIFF proof procedure in this implementation). The main task of TR is to deduce whether a property (a fluent) holds at some time point, given a domain knowledge, and a (possibly partial) *narration* regarding the facts observed. This is done by answering to queries like

```
?   holds_at(Fl, T).
```

where `Fl` is a ground fluent literal, and `T` is a time point that can be ground, in the basic case, or an existentially quantified variable, possibly together with a set of temporal constraints on the existentially quantified variable of the query.

The knowledge base KB_{TR} consists of four parts:

1. a *domain independent* theory, based on Abductive Event Calculus [5], which explains how events cause fluents and how fluents persist in time. It is contained in the file `KBtrdi.pl`, consulted in the initialisation phase of each computee. 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, that is contained in a specific file for each computee, `KBtrddNAMESCENARIO.pl` say. It contains the definition of the predicates like `initiates/3`, `terminates/3`, and `holds_initially` (as explained in D8).

   For example,

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

   "putting the switch on causes light if the bulb is not broken". Note the different treatment of preconditions for action effects with respect to how general preconditions are represented, for instance, in KB_{plan}.

3. a *narration*, contained in KB_0, that is passed to TR as a parameter by all the capabilities/transitions which use it. It represents facts that are known to have occurred (`observed/2`, `observed/3` and `executed/2`), or to hold initially (expressed as `observed(fl,0)`). For example,

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

   states that the computee did an action of switch on at time 10.

4. the *consistency* integrity constraint

   ```
   holds_at(F, T), holds_at(neg(F), T) -> false.
   ```

which prevents a fluent and its negation to hold at the same time, and currently is the only constraint currently supported. Other (user defined) constraints could be used with a simple extension of the implementation.

Example 1 *Given KB_{TR} containing the following domain dependent theory and KB_0 the following narration*

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

executed(switch_on, 10).
observed(not(light), 20).
```

the query

```
? holds_at(broken_bulb,30)
```

can be proved to hold both credulously and skeptically. Informally, something can be credulously proved if there exists a possible explanation for it whereas for skeptically we also require that an explanation for its negation does not exist, (see D4,D8 for details).

Example 2 *In several components of the computee, e.g. the Goal Revision (GR) transition, may use TR in order to check whether a fluent, e.g. a (sub)goal or an action precondition, holds within a time interval. This is done via queries containing an existentially quantified time variables, possible subject to a set of constraints on these variables and other existentially quantified variables in the state of the computee. Referring to the previous example above, the query*

```
? holds_at(broken_bulb,T1), 10 < T2, T2 < T1.
```

can be proved to hold both credulously and skeptically (the time variable are existentially quantified) , while the query

```
? holds_at(neg(broken_bulb),T1), 15 < T1, T1 < 18.
```

can not be proved either skeptically, or credulously. Finally, the query

```
? holds_at(neg(broken_bulb),T1), T1 < 5.
```

credulously holds (by assuming `assume_holds(neg(broken_bulb), 0)`*).*

In several places within the computee model, e.g. the cycle theory, it is sufficient in most cases to have ground skepticall temporal queries. Such queries are invoked by the call

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

where `KB0` is the current narration as a list of facts, and `Goal` is a ground goal of the form `[holds_at(fl, t)]`, where `fl` and `t` are a ground fluent and a ground time point.

Note that as it is currently used, this does not return an answer, but only succeeds when the goal holds against the current narration.

Example 3 *In the above example the query $holds\_at(broken\_bulb, 30)$ will be called by:*

```
ground_temporal_reasoning_skeptical(
                [executed(switch_on, 10),observed(not(light), 20)],
                holds_at(neg(broken_bulb), 30))
```

In this case the query simply succeeds, since we are not interested in any computed answer. On the other hand, if the fluent was not entailed, the call would have failed.

Instead, the skeptical reasoning of TR for existentially quantified variables, together with a set of temporal constraints, can be invoked by the call

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

where `KB0` is as before, and `EqvGoal`, `EqvTCs` and `EqvSigma` are a goal, a set of temporal constraints, and a substitution for existentially quantified variables in the state of the computee, respectively. The implementation could be extended to compute a time interval where the existentially quantified query holds.

3.4.4 Writing a Cycle Theory

A cycle theory is an *LPwNF* theory and consists of three components, namely a basic, an interrupt, and a behaviour component. They are written in Gorgias in a way similar to the knowledge base KB_{GD} of Goal Decision. We note that this section should be seen more as a systems manual rather than as a user's manual as cycle theories form the control mechanism of computees.

A user typically will be confined to choosing for her/his application the basic high-level characteristics of behaviour of the computee that are appropriate. This is done by giving the computee an appropriate *behaviour* part of its cycle theory. How a user writes this part will be explained in some detail in the subsection below. Before doing this we will explain how the other parts of a cycle theory are written out which although more or less fixed can still be changed by an (advanced) user in order to customize further the computees to meet her/his requirements.

Basic and Interrupt Components of Cycle Theories
The *basic* part of any cycle theory consists of rules of the form:

$$r_{i|k}(S', X) : T_k(S', X, S'') \leftarrow T_i(S, S'), C_{i|k}(S', X).$$

where T_i and T_k are any two transitions different from the Passive Observation, i.e. $i, k \neq PO$. Such a rule specifies which transition T_k might follow a transition T_i The conditions $C_{i|k}$ in such a cycle-step rule are called enabling conditions as they determine when a cycle-step from T_i to T_k is allowed or enabled. In particular, they determine the input X, if any is required, of the ensuing transition T_k. Such input will be determined by calls to the appropriate selection functions, when required.

For example, the following cycle-step rule

$$r_{PI|AE}(S', As) : T_{AE}(S', As) \leftarrow T_{PI}(S, S'), C_{PI|AE}(S', As).$$

that expresses the possibility that a PI transition can be followed by an AE transition is represented in Gorgias as follows

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

The first argument of the conclusion is a label naming the rule and 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 (here 'AE') and its parameters chosen by this rule (here 'As). 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. Rules are wrapped in *ct__rule* to indicate the module they belong to.

As explained above the enabling conditions, i.e. the predicate *ct__ec*/2, determine the input parameters of the ensuing transition, and thus in the given example they determine the actions to be executed. Furthermore, without loss of generality, we have simplified the form of the basic rules in the implementation in such a way that the condition for the previous transition is included within the enabling conditions. In the example above the Plan Introduction (PI) transition is not "visible" in the conditions of the rule as it is part of *ct__ec*/2.

The enabling conditions are auxiliary predicates of the *LPwNF* theory specifying the cycle theory and are written in Prolog. For the example of the AE transition above this is given by:

```
ct__ec('AE',As) :-
        fun__action_selection(As),
        ct__ec_aux_p('AE',As).
```

The first predicate, namely *fun__action_selection*/2, is a core selection function (denoted as c_{AS} in D4) that determines from the current state actions that may be executed. This predicate is coded as follows:

```
fun__action_selection(((Op,AT),Goal,Preconditions,TC)) :-
        self__action((Op,AT),Goal,Preconditions,TC),
        self__temporal_constraints_validate(AT,TC),
        self__goal_ancestors_eq(Goal,Ancestors),
        self__goal_check_ancestors_nopass(Ancestors).
```

where (a) the first condition picks an action from the state of the computee via the predicate *self__action*/1, then (b) the second condition checks that the temporal constraints are still satisfied, and (c) the last two conditions check that none of the action's ancestors (i.e. a goal or subgoal) has been satisfied already.

We note that this implementation does not cover the complete specification of this selection function as given in [3]. An advanced user can extend this definition in appropriate ways.

The other predicate, namely *ct__ec_aux_p*/2 checks that action execution (AE) is an allowed type of transition based on the previous one. It calls the predicate *ct__ec_user_p*/2 through which the user specifies the basic allowed transition steps. For example, if we have:

```
ct__ec_user_p('AE','PI').
ct__ec_user_p('AE','AE').
ct__ec_user_p('AE','AO').
ct__ec_user_p('AE','PR').
```

then this specifies that the only type of transitions that are allowed to follow an action execution transition are PI, AE, AO and PR. If we want to allow any type of transition to follow any other type (possibly the same) then we simply need to define *ct__ec_user_p*/2 by the single clause:

```
ct__ec_user_p(Tr1,Tr2).
```

The *interrupt* component of the cycle theory is analogous, in syntax, to the basic component. However, each interrupt rule specifies what might follow a PO transition, which acts as an interrupt. Concretely, this is accomplished as follows:

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

Specifying the Pattern of Behaviour
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 then are used to determine, amongst all the enabled cycle-steps, which ones are preferred under the current circumstances.

In D4 notation, the behaviour rules are of the form

$$R^i_{k|l} : r_{i|k}(S', X_k) > r_{i|l}(S', X_l) \leftarrow BC^i_{k|l}(S', X_k, X_l).$$

where $BC^i_{k|l}$ are called *behaviour* conditions. These are heuristic conditions (e.g. heuristic selection functions) under which the cycle step to transition $T_k(X_k)$ is preferred over the cycle step to transition $T_l(X_l)$. Through the behaviour part of the cycle theory we can encode different patterns of operation.

In the sequel we provide coded examples of behaviour rules that specify a certain pattern of operation that can be taken as an underlying basis on which we can build different and additional patterns of behaviour — we can call this 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.

Hence for example, we have the behaviour rule of

$$R^{GI}_{PI|*}(Gs) : r_{GI|PI}(S, Gs) > r_{GI|*}(S, X).$$

to capture that after goal introduction (GI) we plan for the goal introduced, and the behaviour rule of

$$R^{AE}_{GR|*} : r_{AE|GR}(S) > r_{AE|*}(S, \_) \leftarrow \mathit{empty\_plan}(S).$$

to capture the fact 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([prefer(normal,step('GR',_), step(_,_)),
    prefer(step('GR',_), step(_,_)), []):-
        self__last_transition('AE'),
        empty_plan.
```

where such rules are wrapped by *patt_rule*, the first argument of the head of the rule is a user chosen name of the rule and the second argument, namely the head of the priority rule that is coded up must have the form $prefer(step(Tr1, I1), step(Tr2, I2))$.

The condition, *empty_plan*/0, used in this example is called a behaviour condition 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

$$R^{PO}_{GI|*} : r_{PO|GI}(S, Obs) > r_{PO|*}(S, Obs) \leftarrow comm\_msg(Obs).$$

and thus the corresponding Gorgias rule is given by:

```
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 which through its goal decision capability will decide the response to *Obs*. In addition, in the normal pattern of behaviour the rule

$$R^{PI}_{AE|*}(As) : r_{PI|AE}(S, As) > r_{PI|*}(S, X).$$

gives preference to action execution (AE) after a plan introduction (PI) transition while the rule

$$R^{*}_{AE|AE}(As) : r_{*|AE}(S, As) > r_{*|AE}(S, \_) \leftarrow comm\_action\_selection(As).$$

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

$$C^{*}_{AE|AE} : R^{*}_{AE|AE}(A_1) > R^{*}_{AE|AE}(A_2) \leftarrow more\_urgent(A_1, A_2)$$

ensures that the more urgent communication action is preferred. These rules are coded, respectively, as follows:

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

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

```
        comm_action_selection(As).

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).
```

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 Society

4.1 Configuring the Society

The Society component is by default configured with standard parameters, which allow to start the society demonstrator without any particular configuration.

There can be cases in which users need to modify the defaults parameters. The configuration values are stored in a text file, named `society.config` and situated in the directory `socs\platform\society`. It is possible to edit this file. For each parameter the file contains also a description of the meaning and the allowed values it can assume.

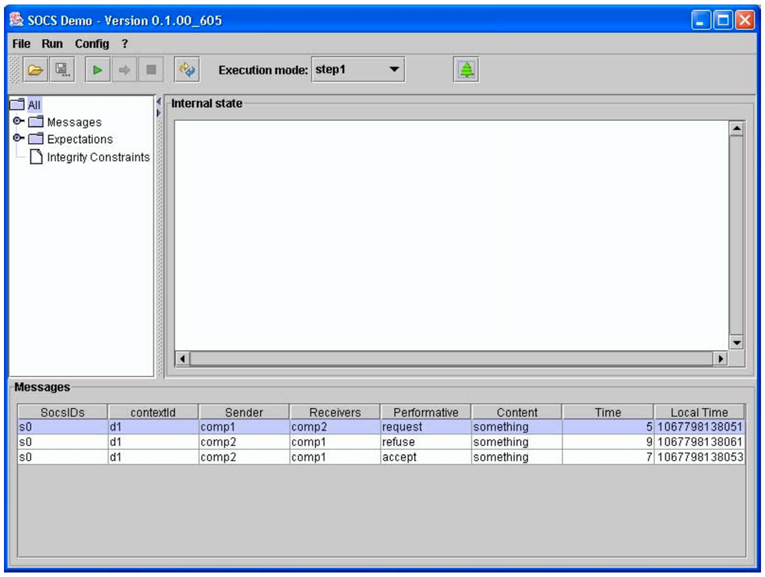
4.2 Starting a generic society

If the installation process has been terminated successfully, inside the directory `socs\platform\society` there is a script named `run.bat`. This script can be used for launching a generic society application. Here we describe how to execute a generic society using this script.

A society application needs some base information before it is launched. More precisely, it needs to know:

- which proof procedure to use (the society application is able to use different proof procedures with different characteristics);
- which events source to use;
- the source files (in case the file system is the selected event source);
- the protocol files in which the protocols are defined;
- one file containing the Social Organization Knowledge Base (SOKB);

There are three ways to provide these information:



SocsIDs	contextId	Sender	Receivers	Performative	Content	Time	Local Time
s0	d1	comp1	comp2	request	something	5	1067798138051
s0	d1	comp2	comp1	refuse	something	9	1067798138061
s0	d1	comp2	comp1	accept	something	7	1067798138053

Figure 2: Main *society GUI*

a) **By setting default values.** It is possible to specify the information in the `society.config` file. Executing the script `run.bat` without any parameter will load the configuration values stored in the file.

b) **By specifying the values by the command line.** It is possible to specify the values on the command line, as parameters after the script name. For a detailed example, please type in at the command prompt the command `run --help` and read the instructions.

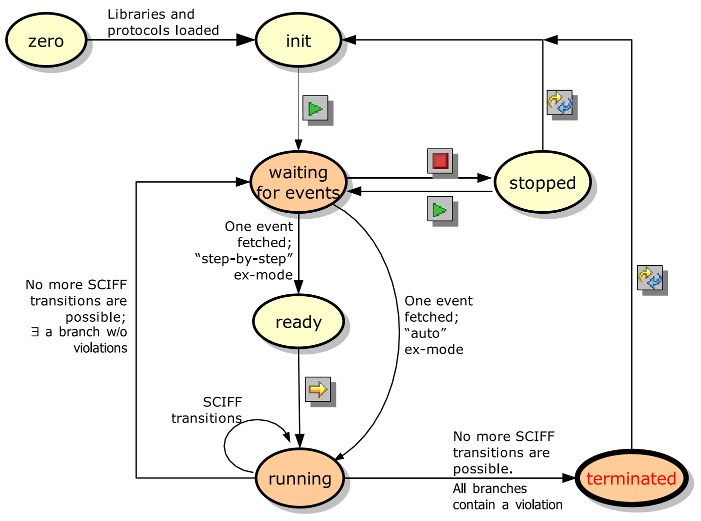
c) **Through a configuration GUI.** It is possible to specify these parameters through a Graphical User Interface. The procedure is two-fold: (1) execute the launching script with the parameter `--graphic`: a configuration window will appear; (2) select the parameters as desired, and then click to the button "start" to launch the society application. We suggest that new users follow this procedure as opposed to (*a*) and (*b*) (because it is simpler).

4.3 Graphical User Interface

The Graphical User Interface (GUI) allows the user to follow the interaction between the computees, and it shows if the interaction is compliant to the social integrity constraints. In fact, as it is described in Deliverables D8 [4] and D8, a society is characterized by a set of protocols, defined in terms of Social Integrity Constraints.

The GUI of the society component is composed of three main areas:

Messages. A table to show the messages exchanged between computees is situated at the bottom of the window, and it shows all the messages exchanged until a certain moment



zero
Libraries and protocols loaded
init
waiting for events
stopped
One event fetched; "step-by-step" ex-mode
ready
No more SCIFF transitions are possible; ∃ a branch w/o violations
One event fetched; "auto" ex-mode
SCIFF transitions
running
No more SCIFF transitions are possible. All branches contain a violation
terminated

Figure 3: State diagram of the society demonstrator. The proof states are in orange; the control states are in yellow.

between the computees. A light violet line shows the next message that will be elaborated by the proof.

Left pane. A list of the computees members of the society at a certain moment is located on the left, and each computee is listed with its name. The name "All" is reserved by default, and it is used to select the data regarding all the computees. It is possible to click on a computee to see the relevant information regarding it (e.g., the messages directed to or sent by it).

Central pane. The content of the central pane changes according to which computee is selected among those listed on the left. It can shows the set of messages exchanged by a particular computee (divided in sent and received) or it can shows the set of expectation regarding the computee (again divided in pending, fulfilled and violated).

A button bar, located on top of the three main areas presented above, contains some controls and buttons. They can be used in order to manage the elaboration of the events, by selecting different proofs or protocols, as well as to load new event files or to save through a file the events actually received.

The controls are not always enabled: the GUI respects the proof-state diagram presented in [2]. Some buttons can be enabled or disabled depending on the current state of the proof.

We give below a brief description of each control and of the effects it has on the application.

Play button It starts the proof procedure. This button must be pressed in the beginning in order to make the proof start elaborating the events; after the user presses the "stop" button, it is necessary to press again the play button to continue the elaboration.

Stop button It stops the elaboration of the events. The proof moves form the "waiting-for-events" state to the "stopped" state.

Step button This button is enabled only in "step-by-step" execution mode. It is used to let the proof elaborate the next available event.

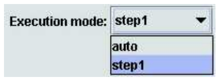
Close History button It closes the history, and provides to notify it to the proof. When the history is closed, no new messages can be elaborated by the proof. The proof will provide to apply the close history transition. If the society is re-initialized, the history is re-set to an "open" state.

Init button It re-initializes all the internal state of the society. All the results obtained until here are lost, and the application return to the "init" state. The events will be processed from the beginning. It is enabled only when the application is in the "stopped" state.

Load button It loads a new set of events from a file. The new events are appended to the list of available events. It is enabled only if the selected messages source is the file system.

Save button It saves all the events received until now in a file. The saved events can be used later for logging purposes or to repeat the elaboration.

Show Tree button It shows/hides the tree viewer of the information generated after the elaboration of the events.



Exec mode selection box It selects the desired execution mode. Valid modes are "auto" and "step1". In the "auto" execution, mode all the events are elaborated, and the elaboration stops only when there are no more events to process. In the "step1" execution mode instead for each event is required an acknowledge to elaborate it (step button). The "step1" mode is somewhat similar to the "step" execution mode which is available in most debuggers. Only, instead of instruction there are events. It is possible to change the execution mode only when the application is not elaborating, hence when the application is in the "stopped" state.

4.4 Running Demo Example 4 from the package

In this paragraph we illustrate how to run a built-in example of functioning of the society prototype. In Section 4.2 we introduced how to execute a generic society. The configuration of parameters, necessary to the proper functioning of the demo example, has been written into a script file (`.dot`).

The steps needed to execute Example 4 are as follows:

1. open the directory `socs\demo\4`
2. execute the script `demo4_soc.bat`

The script launches the society application and loads protocols and events for this example. To execute the others examples of the society the procedure is the same, except that the user should select the directory corresponding to the desired example (for example `socs\demo\5` for the example 5).

The purpose and expected behaviour of this example are discussed in the Examples document.

4.5 Creating your own examples

In order to specify a society of computees, the user needs to edit two text files:

1. A Prolog file containing the Social Organization Knowledge Base of the society;
2. A text file containing the set of Social Integrity Constraints that the computees will be required to comply with.

The specification of the society is independent of the source of events to be verified for compliance (Medium, User Prompt or History File); in Sect. 4.5.2, we describe the expected format of a History File.

4.5.1 Specification

Social Organization Knowledge Base. The Social Organization Knowledge Base should be defined in a single Prolog file (which will be consulted by means of the Graphical User Interface, see Sect. 4.3).

The file should be declared to be a SICStus module of name `sokb`, and to export the predicates which will be used by the proof procedure (e.g., those called in the body of Social Integrity Constraints). In addition, the module should import the `proof` module. See the SICStus Prolog manual [6] for information on how to deal with modules.

Abducibles $\mathbf{E}$, $\neg\mathbf{E}$, $\mathbf{EN}$, and $\neg\mathbf{EN}$ should be written as `e/2`, `note/2`, `en/2`, and `noten/2`, respectively.

Social Integrity Constraints. Social Integrity Constraints are specified in a text file, and are loaded into the proof by means of the GUI (see Sect. 4.3).
The formal syntax of SICs is described in [4]; in Table 1, we show the symbols in the SICs file that correspond to those in [4].

Type	Symbols in [4]	Symbols in the SICs file
Logical connectives	$\rightarrow$, $\wedge$, $\vee$	`--->`, `/\`, `\/`
Events & Expectations	$\mathbf{H}$, $\neg\mathbf{H}$, $\mathbf{E}$, $\neg\mathbf{E}$, $\mathbf{EN}$, $\neg\mathbf{EN}$	`H, !H, E, !E, EN, !EN`
CLP Constraints (unification)	$=$, $\neq$	`=, !=`
CLP Constraints (CLP-FD)	$=$, $\neq$, $<$, $\leq$, $>$, $\geq$	`==, <>, <, <=, >, >=`

Table 1: Correspondence between symbols found in [4] and symbols in the SICs file.

For instance, the SIC

$$\begin{aligned} & \mathbf{H}(\mathit{tell}(A, B, \mathit{request}(P), D), T_1) \\ \rightarrow\ & \mathbf{E}(\mathit{tell}(B, A, \mathit{accept}(P), D), T_2) \ \wedge\ T_2 \leq T1 + 100 \\ \vee\ & \mathbf{E}(\mathit{tell}(B, A, \mathit{refuse}(P), D), T_2) \ \wedge\ T_2 \leq T1 + 100 \end{aligned}$$

should be written as

```
H(tell(A,B,request(P),D),T1)
---> E(tell(B,A,accept(P),D),T2) /\ T2 <= T1+100
\/ E(tell(B,A,refuse(P),D),T2) /\ T2 <= T1+100.
```

Blank spaces, tabs and newlines are ignored by the parser.

4.5.2 History

The history file is a sequence of entries of the format:

$$\mathit{Act}([\mathit{SocIds}], \mathit{DialogId}, \mathit{Performer}, \mathit{Addressee}, \mathit{Performative}, [\mathit{Arguments}], \mathit{Time}).$$

where *Act* is the type of action, *SocIds* is the list of identifiers of the societies in which the act is performed[2], *DialogId* is the interaction identifier, *Performer* is the computee that performs the action, *Addressee* is the computee that the action is addressed to, *Performative* and *Arguments* represent the content of the action, and *Time* is the time at which the action is performed.

Time is expected to be an integer number, and all the other fields are expected to be Prolog constants. For instance, the history entry

```
tell([s0],auction1,bidder,auctioneer,bid,[scooter,10],150).
```

could represent the event of `bidder` `telling` `auctioneer` a `bid` of `10` money units for a `scooter`, in the context of interaction `auction0` at time `150` (the society identifier `s0` is ignored in the current version of the demonstrator).

In order to allow for a correct processing of events by the proof procedure, entries should be listed in increasing order with respect to the *Time* field.

4.5.3 Running your own example

Once a SOKB and one (or more) file containing the Social Integrity Constraints have been specified, running the Society is just a matter of specifying these information properly. As already discussed in section 4.2, this can be done in three different ways. We strongly suggest to use the configuration GUI. In order to do that a user must:

[2]This field is meant to support the future extension of multiple societies, and is ignored in the current version of the prototype.

1. open a command prompt
2. move to the directory `socs\platform\society`
3. type the command `run --graphic`

A configuration window appears, and the user must select the proper files in which SOKB and ICS have been saved. If “file” is selected as the event source, the user must also select a history file from which to fetch the events. After clicking the button “start”, the society application GUI should appear.

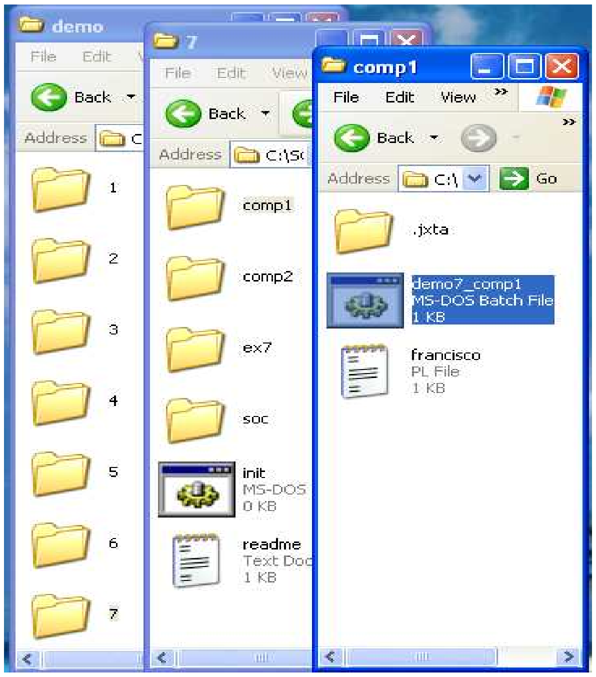
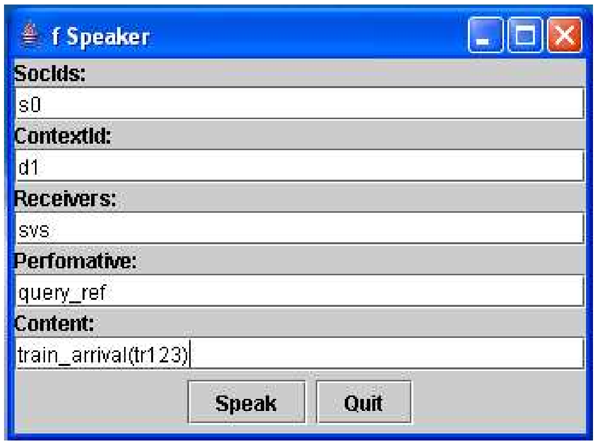


Figure 4: Starting a computee under MS Windows

5 Running Example 7 of SOCSDemo

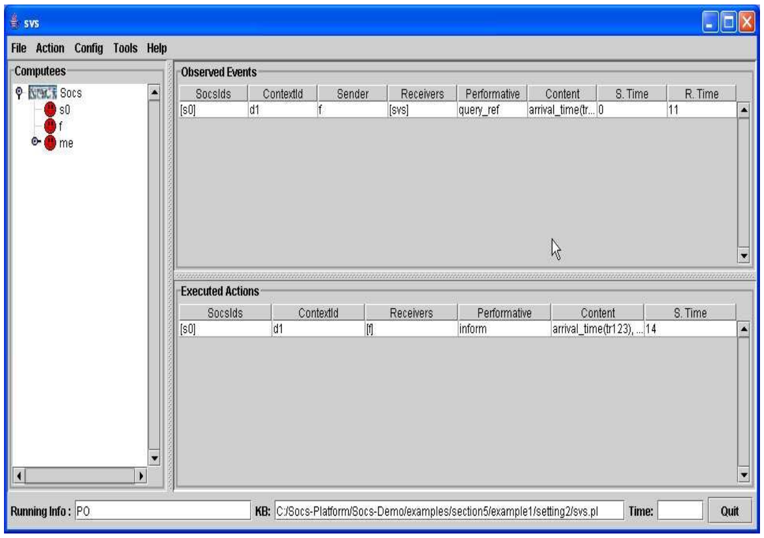
In general, once a user has selected an example to run from the SOCSDemo it will have to run the computees involved and the society. Once the user has opened the directory of the example, the user has to read first a `readme.txt` file in order to get information about the required order of the actions that need to be performed. The user has then to select a directory where a computee is defined, open it, and in it double-click on the file that contains an executable of a computee. An example of this situation for example 7 of the examples document is depicted in Fig. 4, where the user has having selected example 7 of the demo, is starting the first computee residing in the directory `comp1`, from the file `demo7-comp1.bat`. This will start the computee of Francisco, abbreviated as `f`. Similarly, the user will have to do the same for the other computee



SocIds: s0
ContextId: d1
Receivers: svs
Perfomative: query_ref
Content: train_arrival(tr123)

Figure 5: Sending a message manually.

specified in directory `comp2`, abbreviated as `svs`, and the society specified in directory `soc`, abbreviated as `s0`.



Observed Events

SocsIds	ContextId	Sender	Receivers	Performative	Content	S. Time	R. Time
[s0]	d1	f	[svs]	query_ref	arrival_time(tr...	0	11

Executed Actions

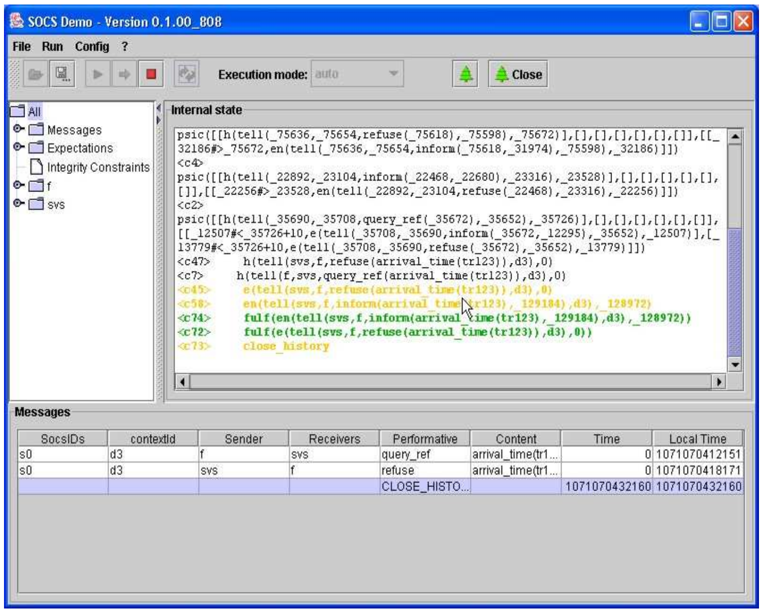
SocsIds	ContextId	Receivers	Performative	Content	S. Time
[s0]	d1	[f]	inform	arrival_time(tr123), ...	14

Running Info : PO
KB: C:/Socs-Platform/Socs-Demo/examples/section5/example1/setting2/svs.pl
Time:

Figure 6: Computee GUI

Once all components have been started properly, the user will have to select in the computee window of `f` the sending of message, which will need to be specified manually. Therefore, the user will have to select from the menu Action → Speak. Then, the window of Fig. 5 should appear on the screen. The user at this point should press the Speak button. This will have the effect of the computee `f` sending the message to computee `svs`, and immediately after this the user can observe that the interfaces of the two computees have recorded messages being exchanged, including the society recording these events.

At the end of the interaction the computee *svs* will have the state shown already in Figure 6, while the computee for f will have the actions executed from svs as observations and the observations of svs as actions. The society, on the other hand, at the end of the interactions will have the form shown in Figure 7.



```
psic([[h(tell(_75636,_75654,refuse(_75618),_75598),_75672)],[],[],[],[],[],[]],[[_
32186#>_75672,en(tell(_75636,_75654,inform(_75618,_31974),_75598),_32186)]])
<c4>
psic([[h(tell(_22892,_23104,inform(_22468,_22680),_23316),_23528)],[],[],[],[],[],
[]],[[_22256#>_23528,en(tell(_22892,_23104,refuse(_22468),_23316),_22256)]])
<c2>
psic([[h(tell(_35690,_35708,query_ref(_35672),_35652),_35726)],[],[],[],[],[],[]],
[[_12507#<_35726+10,e(tell(_35708,_35690,inform(_35672,_12295),_35652),_12507)],[_
13779#<_35726+10,e(tell(_35708,_35690,refuse(_35672),_35652),_13779)]])
<c47>     h(tell(svs,f,refuse(arrival_time(tr123)),d3),0)
<c7>     h(tell(f,svs,query_ref(arrival_time(tr123)),d3),0)
<c45>     e(tell(svs,f,refuse(arrival_time(tr123)),d3),0)
<c58>     en(tell(svs,f,inform(arrival_time(tr123),_129184),d3),_128972)
<c74>     fulf(en(tell(svs,f,inform(arrival_time(tr123),_129184),d3),_128972))
<c72>     fulf(e(tell(svs,f,refuse(arrival_time(tr123)),d3),0))
<c73>     close_history
```

SocsIDs	contextId	Sender	Receivers	Performative	Content	Time	Local Time
s0	d3	f	svs	query_ref	arrival_time(tr1...	0	1071070412151
s0	d3	svs	f	refuse	arrival_time(tr1...	0	1071070418171
				CLOSE_HISTO...		1071070432160	1071070432160

Figure 7: The Society at the end of the interaction between computees.

References

[1] Marco Alberti, Andrea Bracciali, Federico Chesani, Neophitos Demetriou, Ulle Endriss, Fariba Sadri, Antonis Kakas, Evelina Lamma, Wenjin Lu, Nicolas Maudet, Paola Mello, Michela Milano, Kostas Stathis, Giacomo Terreni, Francesca Toni, and Paolo Torroni. Examples for the functioning of computees and their societies. Discussion Note IST3250/ICSTM//DN/I/a2, SOCS Consortium, December 2003.

[2] Marco Alberti, Andrea Bracciali, Federico Chesani, Ulle Endriss, Marco Gavanelli, Wenjin Lu, Kostas Stathis, and Paolo Torroni. SOCS prototype. Technical report, SOCS Consortium, 2003. Deliverable D9.

[3] 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.

[4] 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.

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

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