Crafting the Mind of PROSOCS Agents

Andrea Bracciali[1], Neophytos Demetriou[3], Ulle Endriss[2],
Antonis Kakas[3], Wenjin Lu[4], Kostas Stathis[4,1]

[1]Dip. di Informatica, Università di Pisa braccia@di.unipi.it

[2]Dept. of Computing, Imperial College London ue@doc.ic.ac.uk

[3]Dept. of Computer Science, Cyprus University {nkd,antonis}@cs.ucy.ac.cy

[4]School of Informatics, City University London {lue,kostas}@soi.city.ac.uk

Abstract

PROSOCS agents are software agents that are built according to the KGP model of agency. KGP is used as a model for the *mind* of the agent, so that the agent can act autonomously using a collection of logic theories, providing the mind's reasoning functionalities. The behaviour of the agent is controlled by a *cycle theory* that specifies the agent's preferred patterns of operation. The implementation of the mind's generic functionality in PROSOCS is worked out in such a way so it can be instantiated by the platform for different agents across applications. In this context, the development of a concrete example illustrates how an agent developer might program the generic functionality of the mind for a simple application.

1 Introduction

PROSOCS (**Pro**gramming **So**cieties of **C**omputee**s**) [Stathis et al., 2004] is a prototype platform [1] that allows a developer to build software agents in global computing environments [GC, 2000]. PROSOCS seeks to offer for free the reasoning and communication capabilities an agent needs to operate in an open and distributed environment. The agent developer, as a result, is only required to specify the set of logic programming theories that describe the background knowledge necessary for the agent to operate within a specific application environment.

PROSOCS is characterised by a number of novel features including the combination of peer-to-peer computing and computational logic [Lu et al., 2003], a *society infrastructure* (presented in a companion paper of this volume [Alberti et al., 2004]), and an implementation reference model integrating the various components of the platform [Stathis et al., 2004]. The main focus and contribution of this work, however, is to provide an in-depth report on the development of one component, called in PROSOCS the *mind* [Stathis et al., 2004]. This component is built according to the KGP model of agency [Kakas et al., 2004b] so, in a sense, the work reported here summarises our attempt to implement the generic functionality of KGP in PROSOCS including how one might program components of an agent for a concrete application.

The development of PROSOCS has been motivated by the observation that techniques for developing interactions in open computing environments result either in low-level implementations with no obvious logical characterisation, which

[1]The platform is developed in SOCS (*Societies of Computees*), an EU research project investigating computational and logical models for the individual and collective behaviour of computational entities called *computees*, i.e. software agents developed in computational logic.

are, therefore, not verifiable, or in abstract specifications possibly employing expressive logics with modalities, but which – as argued in [Rao, 1996] – have shed very little light on actual implementations of agent-based systems. To bridge the gap between the logical specification and the implementation, agents in PROSOCS are constructed following the logical model of KGP [Kakas et al., 2004b], which has formal computational characteristics [Bracciali et al., 2004]. In this context, another contribution of this work is to show how the gap between implementation and specification is bridged using very concrete computational logic tools and techniques, to be discussed in the sequel.

The rest of this paper is structured as follows. Section 2 explains the KGP model of agency and discusses its main characteristics. Section 3 outlines how the logical KGP model is implemented to provide the generic functionality of an agent's mind. Then, in the context of a concrete scenario, Section 4 illustrates how to program agents and how to run them within PROSOCS. Section 5 discusses related work and, finally, Section 6 concludes by summarising the work and discussing our plans for the future.

2 The Architecture of PROSOCS Agents

PROSOCS agents implement the KGP (**K**nowledge, **G**oals and **P**lan) model of agency [Kakas et al., 2004b]. KGP is intended to represent logically the *internal* or *mental* state of an agent. Such a mental state in KGP is operated by a set of *reasoning capabilities* that allow the agent to perform *planning, temporal reasoning, identification of preconditions of actions, reactivity* and *goal decision*, together with a *sensing capability* for the agent to perceive the environment in which it is situated. The capabilities are used by a set of *transition rules* describing how the mental state changes as a result of the agent being situated in an environment. Agent behaviour is controlled by combining sequences of transitions determined by reasoning with a *cycle theory*.

2.1 KGP in a nutshell

The agent architecture of KGP is shown in Fig. 1 where the state of an agent is a triple $\langle KB, Goals, Plan\rangle$. Moreover, a set of Temporal Constraints TCS is associated with the time variables of the knowledge bases, goals, and plans.

- **KB** describes what the agent knows of itself and the environment and consists of separate modules supporting the different reasoning capabilities: KB_{plan} for planning, KB_{GD} for goal decision, KB_{pre} for the identification of preconditions, KB_{TR} for temporal reasoning, KB_{react} for reactivity, and KB_0 holds the (dynamic) knowledge of the agent about its external world. KB_0 is the only part of the knowledge of the agent that changes over time, typically due to information conveyed by the sensing capability.

- **Goals** is a set of properties that the agent has decided that it wants to achieve by a certain time possibly constrained via some *temporal constraints* (see TCS below). *Goals* are split into two types: *mental goals* that can be planned for by the agent and *sensing goals* that cannot be planned for but only sensed to find out from the environment whether they hold or not.

- **Plan** is a set of "concrete" actions, partially time–ordered, of the agent by means of which it plans (intends) to satisfy its goals, and that the agent can execute in the right circumstances. Each action in *Plan* relies upon

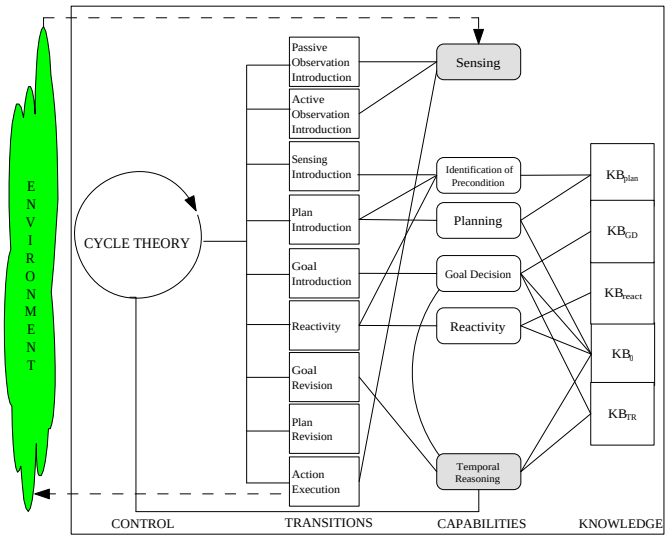

Figure 1: The KGP Model.

preconditions for its successful execution. These preconditions might be checked before the actions are executed.

- **TCS (Temporal Constraints Store)** is a set of constraints over integers and variables using the set of predicates $\{<, \leq, >, \leq, =, \neq\}$. The constraints put restrictions on the variables of goals in *Goals* and actions in *Plan*.

The state of an agent evolves by the cycle applying transition rules, which employ capabilities and a constraint solver for temporal constraints. The transitions are:

- Goal Introduction (GI), changing the top-level *Goals*, and using Goal Decision.
- Plan Introduction (PI), changing *Goals* and *Plan*, and using Planning and Identification of Preconditions.
- Reactivity (RE), changing *Goals* and *Plan*, and using the Reactivity capability.
- Sensing Introduction (SI), changing *Plan* by introducing new sensing actions for checking the preconditions of actions already in *Plan*, and using Sensing.
- Passive Observation Introduction (POI), changing KB_0 of KB by introducing unsolicited information coming from the environment, and using Sensing.
- Active Observation Introduction (AOI), changing KB_0 of KB, by introducing the outcome of (actively sought) sensing actions, and using Sensing.
- Action Execution (AE), executing all types of actions, and thus changing KB_0 of KB.
- Goal Revision (GR), revising *Goals*, and using Temporal Reasoning and Constraint Satisfaction.

- Plan Revision (PR), revising *Plan* and using Constraint Satisfaction.

Given an agent's knowledge, capabilities, transitions and cycle theory, the KGP model assumes (possibly infinite) vocabularies of *time constants* (e.g., the set of all natural numbers), *time variables* (indicated with $t, t', s, \ldots$), *fluents* (indicated with $f, f', \ldots$), *action operators* (indicated with $a, a', \ldots$), and *names of agents* (indicated with c, c', $\ldots$). Given a fluent f, f and $\neg f$ are referred to as *fluent literals*. We use $l, l', \ldots$ to denote fluent literals. Moreover, given a fluent literal l, by $\overline{l}$ we denote its complement, namely $\neg f$ if l is f, f if l is $\neg f$.

KGP also assumes that the set of fluents is partitioned in two disjoint sets: *mental fluents* and *sensing fluents*. Mental fluents represent properties such that the agent itself is able to plan for so that they can be satisfied, but can also be observed. On the other hand, sensing fluents represent properties which are not under the control of the agent and can only be observed by sensing the external environment. For example, $problem\_fixed$ and $get\_resource$ may represent mental fluents, namely the properties that (given) problems be fixed and (given) resources be obtained, whereas $request\_accepted$ and $connection\_on$ may represent sensing fluents, namely the properties that a request for (given) resources is accepted and that some (given) connection is active.

The set of action operators is further divided in three disjoint sets: *sensing*, *physical*, and *communication action operators*. Sensing actions represent actions that the agent performs in order to establish whether some fluents hold in the environment. These fluents may be *sensing* fluents, but they can also represent effects of actions that the agent may need to check in the environment. On the other hand, *physical* actions are actions that the agent performs in order to achieve some specific effect, which typically causes some changes in the environment. Finally, *communication* actions are actions which involve communications with other agents. For example, $sense(connection\_on, t)$ is a sensing action, aiming at checking whether or not the sensing fluent $connection\_on$ holds; $do(clear\_table, t)$ may be a physical action operator, and $tell(c_1, c_2, request(r1), d, t)$ may be a communication action expressing that agent c_1 is requesting from agent c_2 the resource r_1 within a dialogue with identifier d, at time t.

2.2 A closer look at the state of a KGP agent

A goal G is a pair of the form $\langle l[t], G' \rangle$ where:

- $l[t]$ is the *fluent literal* of the goal, referring to the time t;
- G' is the *parent* of G.

Top-level goals are goals of the form $G = \langle l[t], \bot \rangle$. As an example, we may have a top-level goal G of the form: $\langle problem\_fixed(p2, t), \bot \rangle$ and a subgoal G' of G of the form $\langle get\_resource(r1, t'), G \rangle$, with $TCS = \{5 \leq t \leq 10, 5 \leq t' \leq t\}$, meaning that to fix problem $p2$ within a certain time interval, the agent needs to have (or acquire) a resource $r1$ within an appropriate other time interval. *Mental (sensing) goals* are goals whose fluent is mental (sensing, resp.).

An action A is a triple of the form $\langle a[t], G, C \rangle$ where:

- $a[t]$ is the *operator* of the action, referring to the execution time t;
- G the goal towards which the action contributes (i.e., the action belongs to a plan for the goal G). G may be a post-condition for A (but there may be other such post-conditions);
- C are the *preconditions* which should hold in order for the action to take place successfully; syntactically, C is a conjunction of (timed) fluent literals.

As an example, we may have an action $\langle tell(c_1, c_2, request(r1), d, , t''), G', \{\}\rangle$ where G' is given above and $5 \leq t'' \leq t'$ also belongs to TCS. *(Non-)Sensing actions* are actions whose operator is a (non-)sensing one.

In both a timed fluent literal $l[t]$ and a timed operator $a[t]$, the time t is a time variable. This variable is treated as an existentially quantified variable, the scope of which is the whole *state* of the agent. Whenever a goal (resp. action) is introduced within a state, the time variable associated with the goal (resp. action), is to be understood as a distinguished, fresh variable. When a time variable is instantiated (e.g., at action execution time) the actual instantiation is recorded in (KB_0 in) the state of the agent. This allows us to keep different instances of the same action (resp. goal) distinguished.

For simplicity, we assume that, given a state $\langle KB, Goals, Plan, TCS\rangle$, all occurrences of variables in $Goals$ and $Plan$ are time variables. In other words, our goals and actions are ground except for the time parameter. Variables other than time variables in goals and actions can be dealt with similarly. We concentrate on time variables as time plays a fundamental role in our model, and we avoid dealing with the other variables to keep the model simple.

Amongst the various modules in KB, we distinguish KB_0, which records the actions which have been executed (by the agent or by others) and their time of execution as well as the properties (i.e. fluents and their negation) which have been observed and the time of the observation. Formally, KB_0 contains assertions of the form:

- $executed(a[t], \tau)$ where $a[t]$ is a timed operator and τ is a time constant, meaning that action a has been executed at time $t = \tau$ by the agent.
- $observed(l[t], \tau)$ where $l[t]$ is a timed fluent literal and τ is a time constant, meaning that the property l has been observed to hold at time $t = \tau$.
- $observed(c, a[\tau'], \tau)$ where c is an agent's name, different from the name of the agent whose state we are defining, τ and τ' are time constants, and a is an action operator. This means that the given agent has observed at time τ that agent c has executed the action a at time τ' ($\tau' \leq \tau$).

Assertions in KB_0 of the third kind are variable-free. These are intended, e.g., to represent reception of communication from other agents. Instead, assertions of the first two kinds refer explicitly to a time t. This representation with explicit variables allows us to link the record in KB_0 of observed properties and execution of actions by the agent with the time of actions in $Plan$ and goals in $Goals$. As a consequence, the time variables in KB_0 are not strictly speaking variables but they can be equated to "named variables".

Since KB_0 is used in all the remaining modules in KB, and these are represented in a logic programming style, we are not allowed to have assertions with existentially quantified variables. Hence, the various knowledge bases will include a *variant* of KB_0, namely $KB_0\Sigma(S)$, where $\Sigma(S)$ is defined below. We will refer to $KB_0\Sigma(S)$ simply as KB_0.

Given a state $S = \langle KB, Goals, Plan, TCS\rangle$, we denote by $\Sigma(S)$ (or simply Σ, when S is clear from the context) the valuation:

$$\Sigma(S) = \{t = \tau \,|\, executed(a[t], \tau) \in KB_0\} \cup \{t = \tau \,|\, observed(l[t], \tau) \in KB_0\}$$

Σ extracts from KB_0 the instantiation of the (existentially quantified) time variables in $Plan$ and $Goals$, derived from having executed (some of the) actions in $Plan$ and having observed that (some of the) fluents in $Goals$ hold (or do not hold). Also, $\Sigma(t)$, for some time variable t, will return the value of t in Σ, if there

exists one, namely, if $t = \tau \in \Sigma$, then $\Sigma(t) = \tau$. The valuation of a temporal constraint Tc in a state S will always take Σ into account. Namely, any ground valuation for the temporal variables in Tc must agree with Σ on the temporal variables assigned to in Σ.

2.3 Reasoning features of KGP Agents

In the KGP model we commit to a number of computational logic mechanisms to support the agent's reasoning capabilities. We rely on abduction [Kakas et al., 1998] to make the mind of the agent capable of reacting, planning, and reasoning temporally. We also use logic programming with priorities [Kakas et al., 1994] so that the mind of the agent can perform preferential reasoning for deciding goals and selecting transitions. Changes in the environment are interpreted in the mental state of the agent using a specialised version of the abductive event calculus [Shanahan, 1989].

2.3.1 Abductive reasoning

A KGP agent is capable of reasoning abductively using a new proof procedure called CIFF [Endriss et al., 2004c]. This integrates abductive and constraint reasoning in a logic programming-like environment. It extends the IFF procedure [Fung and Kowalski, 1997] for Abductive Logic Programming (ALP) [Kakas et al., 1998], which derives its name from the fact that it operates on *completed* logic programs that can be represented as sets of "iff-definitions", i.e. as formulas of the form:

$$p(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \cdots \vee D_n \tag{1}$$

where each of the disjuncts D_i is a conjunction of literals.

Motivated by PROSOCS, CIFF extends IFF in two ways: (*i*) by dealing with constraint predicates (hence the C in the acronym); and (*ii*) by relaxing restrictions on allowed patterns on quantification in the input (this is achieved by replacing overly restrictive static definitions with a so-called dynamic allowedness rule [Endriss et al., 2004c]). The knowledge bases used for three of the reasoning capabilities in the KGP model (i.e. planning, reactivity and temporal reasoning), expressed in the abductive event calculus, can be mapped to a suitably expressive ALP framework. Temporal constraints, in particular, can be modelled using constraint predicates over integers.

In our framework, an abductive logic program is a pair $\langle Th, IC \rangle$, consisting of a *theory* Th and a finite set IC of *integrity constraints*. Th is a logic program in iff-form that provides definitions for certain predicates (but not special predicates such as $=$ or $\bot$). Any predicate that is neither defined nor special is called *abducible* and instances of these abducible predicates may be assumed to be either *true* or *false*. Integrity constraints are formulas that restrict the range of possible interpretations of these predicates and, in our framework, they are all of the form:

$$L_1 \wedge \cdots \wedge L_m \rightarrow A_1 \vee \cdots \vee A_n \tag{2}$$

where each of the L_i is a literal and each of the A_i is an atom.[2]

Finally, a *query* Q is simply a conjunction of literals. A query may be regarded as an *observation* against the background of the world knowledge encoded in a given abductive logic program. An *answer* to such a query would then provide

[2]Integrity constraints are not to be confused with constraint predicates (as in constraint logic programming).

an *explanation* for this observation: it would specify which instances of the abducible predicates have to be assumed to hold for the observation to hold as well. In addition, such an explanation should also validate the integrity constraints. CIFF can compute such an abductive answer for inputs expressed in a first-order language that includes constraint predicates. Details of the proof-procedure, such as the rewrite rules or the relevant *soundness* results, are outside the scope of the present paper and may be found in [Endriss et al., 2004c].

2.3.2 The Abductive Event Calculus

KGP agents reason temporally using a specialised version of the *event calculus* [Kowalski and Sergot, 1986], based on abduction [Shanahan, 1997], and appropriately extended to fit the KGP mode of operation. The specialised event calculus used is shown in Fig. 2, showing how a situated KGP agent concludes that a fluent F holds at a time T.

$holds\_at(F, T_2) \leftarrow$	$happens(O, T_1), initiates(O, T_1, F),$
	$T_1 < T_2, \neg\ clipped(T_1, F, T_2).$
$holds\_at(\neg F, T_2) \leftarrow$	$happens(O, T_1), terminates(O, T_1, F),$
	$T_1 < T_2, \neg\ declipped(T_1, F, T_2).$
$holds\_at(F, T) \leftarrow$	$holds\_initially(F), 0 < T, \neg\ clipped(0, F, T).$
$holds\_at(\neg F, T) \leftarrow$	$holds\_initially(\neg F), 0 < T, \neg\ declipped(0, F, T).$
$holds\_at(F, T_2) \leftarrow$	$observed(F, T_1), T_1 \leq T_2, \neg\ clipped(T_1, F, T_2).$
$holds\_at(\neg F, T_2) \leftarrow$	$observed(\neg F, T_1), T_1 \leq T_2, \neg\ declipped(T_1, F, T_2).$
$holds\_at(F, T) \leftarrow$	$assume\_holds\_at(F, T).$
$clipped(T_1, F, T_2) \leftarrow$	$happens(O,\ T), terminates(O, T, F), T_1 \leq T < T_2.$
$declipped(T_1, F, T_2) \leftarrow$	$happens(O,\ T), initiates(O, T, F), T_1 \leq T < T_2.$
$clipped(T_1, F, T_2) \leftarrow$	$observed(\neg F, T), T_1 \leq T < T_2.$
$declipped(T_1, F, T_2) \leftarrow$	$observed(F, T), T_1 \leq T < T_2.$
$happens(O, T) \leftarrow$	$executed(O, T).$
$happens(O, T) \leftarrow$	$observed(O, T', T).$
$happens(O, T) \leftarrow$	$assume\_happens(O, T).$
Integrity Constraints:	$holds\_at(F, T), holds\_at(\neg F, T) \rightarrow false$
	$assume\_happens(A, T),$
	$precondition(A, P) \rightarrow assume\_holds\_at(P, T).$

Figure 2: The Abductive Event Calculus for KGP.

From Fig. 2 we can see that fluents holding at the initial state are represented by what *holds_initially*. Fluents are *clipped* and *declipped* by events happening at different times. Events that happen will *initiate* and *terminate* fluents holding at or between times. Fluents will also hold if they have been *observed* or result from actions that have been *executed* by the agent. An operation can be made to happen and a fluent can be made to hold simply by assuming them. In KGP we also use domain-independent integrity-constraints, to state that a fluent and its negation cannot hold at the same time and, when assuming (planning) that some action will happen, we need to enforce that each of its preconditions hold.

2.3.3 Preference Reasoning

A KGP agent is able to reason about preferences using GORGIAS [Demetriou and Kakas, 2003, Kakas and Moraitis, 2003], a new proof-procedure for preference reasoning in an environment of logic programming with priorities. GORGIAS extends a previous proof procedure [Dimopoulos and Kakas, 1995] for

the framework of Logic Programming without Negation as Failure ($LPwNF$) allowing priorities between rules of the theory to be non-static, i.e. depending on the particular context (external circumstances) at the time the reasoning takes place.

In general, an extended $LPwNF$ program is a pair $\langle P, H \rangle$ where P, called the lower-level part, consists of labelled propositional rules of the form:

$$Label : L \leftarrow L_1 \wedge ... \wedge L_n \tag{3}$$

where $L, L_1, ..., L_n$ are atoms a or explicitly negative literals $\neg a$, and H, called the higher-level part, consists of propositional rules of the form:

$$Label : L \succ L' \leftarrow L_1 \wedge ... \wedge L_n \tag{4}$$

where $L_1, ..., L_n$ are atoms or explicitly negative literals, L, L' are labels of rules in P or H, while the symbol $\succ$ can be read as *has higher priority*. With these rules we have a background entailment, $\models_{LP}$, given by the single inference rule of modus ponens. The program also contains rules defining the special predicate:

$$incompatible(p(X), q(X)) \tag{5}$$

expressing the fact that any instance of the literal $p(X)$ is incompatible with the corresponding instance of the literal $q(X)$. This notion of incompatibility includes $incompatible(p, \neg p)$, for all atoms p, and $incompatible(r \succ s, s \succ r)$, for all labels of rules r and s. A $LPwNF$ program may also contain the definitions of auxiliary predicates used in the conditions of rules in P and H and rules.

The preference reasoning of $LPwNF$ are realized through *argumentation* [Bondarenko et al., 1997, Dung, 1995, Prakken and Sartor, 1997]. In this, a query Q holds if and only if there exists a subset A of the rules of the given program $\langle P, H \rangle$, in which Q holds under the background logic $\models_{LP}$ and this set A is an *admissible* argument for Q. Informally, an argument A is admissible [Kakas et al., 1994, Dung, 1995] iff all its counter-arguments, namely arguments for conclusions that are incompatible with any of the conclusions of A, are not stronger (under the priority of rules) than A. Hence this makes any query Q supported by an admissible argument a preferred conclusion under the given program.

GORGIAS is based on a general proof theory [Kakas and Toni, 1999] for computing argumentation. This proof theory is given in terms of derivations of trees, where nodes are arguments and each node is labelled as "attack" or "defense". A defense node is followed by a set of children attack nodes, one for each of its possible counter-arguments to it. An attack node is followed by a defense node containing a counter-argument against its parent. *Successful derivations* terminate with a tree whose root is an admissible argument supporting the given query. GORGIAS specialises this proof-theory by incorporating a specific way of computing arguments and counter-arguments and specific techniques to deal with the dynamic nature of the attacking relation between arguments. Any node, N, of the tree results from first choosing a "culprit" conclusion c of the parent node of N and then reducing (by resolution) some conflicting incompatible literal, $\overline{c}$, of c so that N minimally entails $\overline{c}$ under the background logic $\models_{LP}$. The root node of the initial tree is computed by reducing the given query formula Q into a minimal set that concludes Q via $\models_{LP}$.

3 The generic functionality of an agent's mind

To situate the mind of a PROSOCS agent in a specific environment, the mind component is wrapped by another component that in PROSOCS we call the *body*. Through the body a PROSOCS agent is capable of accessing a networked environment consisting of other PROSOCS agents. The detailed interaction of the body and the mind, or the body and the environment, is however beyond the scope of this work; we refer the reader to [Stathis et al., 2004] for more details. Here we prefer to discuss instead how the mind is implemented using logic-based tools and PROLOG. We start from the representation of the mental state, we outline how the agent interprets this state, we show how the capabilities (such as temporal reasoning) build upon the agent's interpretations, we continue with the implementation of the transitions that handle state changes, and we close with the cycle theory that controls the agent's behaviour.

3.1 The KGP State in PROSOCS

An important issue that we need to resolve when implementing KGP is the representation of the existentially quantified variables (and the associated notion of substitution) used in a KGP state, i.e. those occurring in goals and actions, to keep them distinguished from normal PROLOG variables. In the current implementation, we represent existentially quantified variables by terms of the form `eqv(Id)`, where `Id` is an identifier of the form `t1, t2, ...`, uniquely generated by the implementation. Variables in actions and goals may be instantiated when actions are executed (to the execution time) and when fluents are observed (at the time of observation). The instantiation cannot be performed explicitly on the state since time variables are also used as identifiers of actions and goals. So we choose to keep this instantiation implicit, by recording it in what we referred to in the previous section as the valuation Σ. We represent the valuation Σ as an assertion containing a list of equalities of the form:

```
[eqv(t1) = 18, eqv(t2) = 21, ..., eqv(t12) = 39].
```

We also define a predicate `get_Sigma/1` to access the list at any time, as well as predicates that append to that list as required. Similarly, we implement all the temporal variables of the *temporal constraint store* TCS (to be shared by all transition and capabilities) as a list of inequality constraints of the form:

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

The `#` symbol indicates a constraint of TCS. As before, we define a predicate `get_TCS/1` to access the list at any time, together with predicates that update the list when new goals and plans are introduced. Constraint satisfaction with the TCS is handled by the proof-procedure CIFF, which is discussed in Section 2.3.1.

Goals and actions in a KGP state are hierarchically organised and represented as a tree with goals and actions as nodes. Goals are represented as:

```
goal(Child, Parent, TCs).
```

`Child` and `Parent` are both of the form `(Fluent, Time)`, indicating that the agent is trying to achieve the sub-goal `Child` specified as `Fluent` at a `Time`; a time point or a temporal variable is constrained by the temporal constraints `TCs`. For example, the goal:

```
goal((have(ticket),eqv(t4)), root, [eqv(t4) #=< 10]).
```

is a top-level goal (i.e. a goal with `root` as a `Parent`) for an agent to have a ticket before or at time 10 (note the existentially quantified variable). Similarly:

```
action(Action, Goal, Preconditions, TCs).
```

represents an action `Action` that contributes to the parent goal `Goal`, where `Action` is a pair `(Operator, Time)`, whose `Operator` must be executed at a `Time` (constrained as before by a list of temporal constraints `TCs`), provided that the list of `Preconditions` is satisfied in the current state of the agent. The example below:

```
action((call_taxi(Taxi), eqv(t17)),
       (get_to_station, eqv(t5)),
       [taxi_is_available(Taxi)],
       [eqv(t17) #=< 12]).
```

describes an action `call_taxi(Taxi)`, with parent goal `get_to_station`, preconditions `taxi_is_available(Taxi)`, and a constraint on the time of the action to be less than or equal to 12.

The knowledge base of an agent is represented as a set of assertions representing facts and rules. The (dynamic) knowledge of the agent about the external world in KB_0 consists of initial facts and happened events of the form:

```
observed(Property, Time).
```

for the `Fluent` observed to hold at `Time`, or:

```
observed(Agent, Action, Time).
```

for an `Agent` that is observed to execute an `Action` at a specific `Time`. For instance a typical utterance in agent dialogues, stating that "agent `c1` has recorded at time 7 that the agent itself informed agent `c2`, within dialogue `d`, at time 5 that there are `no_member_seats`" is represented as:

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

As discussed before, variable instantiations are recorded in the valuation Σ dynamically when the observation is asserted in the knowledge base.

3.2 Interpreting the state using Proof-Procedures

CIFF. To build PROSOCS we have implemented in PROLOG the CIFF proof-procedure [3], whose implementation is described in [Endriss et al., 2004b]. Iff-definitions of a CIFF program, as described in equation 1 of section 2.3.1, are written in the form:

```
p(X1,...,Xk) iff [D1,...,Dn].
```

Similarly, integrity constraints, as described in equation 2 of section 2.3.1, are written in the form:

```
[L1,L2,...,Lm] implies [A1,...,An].
```

To interpret rules of the above kind, we call the predicate `ciff/4`:

```
ciff(Theory, ICs, Query, Answer).
```

[3] The CIFF system is available at `http://www.doc.ic.ac.uk/~ue/ciff/`.

The first argument is a list of iff-definitions, the second is a list of integrity constraints, and the third is the list of goals 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 constraints. The procedure relies on a constraint solver based on the built-in finite domain solver of SICStus PROLOG [Carlsson et al., 1997]. In principle, CIFF could be integrated with different constraint solvers; in practice we require a constraint system that is suitable to reason about temporal constraints as they are used in the KGP model.

GORGIAS. We have also implemented the GORGIAS proof-procedure in PROLOG [4]. The lower-level part of preference rules of a GORGIAS program, as described in equation 3 of section 2.3.3, are written in the form:

```
rule(Theory, Label, L, [L1,...,Ln]):- Conditions.
```

The translation requires that a *Theory* argument is added in rules, a term that distinguishes between different theories, such those required for goal decision and the cycle theory. Rules for the higher-level part of a GORGIAS program, as described in equation 4 of section 2.3.3, are described similarly as:

```
rule(Theory, Label, prefer(L, L'), [L1,...,Ln]):- Conditions.
```

Also, we write:

```
incompatible(Theory, p(X), q(X)).
```

to express facts, as described in equation 5 of section 2.3.3. Then to call the GORGIAS interpreter on preference rules such as those above, we need to call the predicate `gorgias_solve/2` as:

```
gorgias_solve(Theory, Query).
```

The first argument is a term representing an identifier that denotes a preference theory, and the second is a list of goals in the query. GORGIAS implements a general proof-theory [Kakas and Toni, 1999] for computing argumentation. This proof theory is given in terms of derivations of trees, where nodes are arguments and each node is labelled as "attack" or "defense".

3.3 Calling Proof-Procedures in Capabilities

Goal Decision. The goal decision capability, when called, allows the agent to decide the preferred goals at a given current time. Conditions about goals, e.g. rules of the form $holds\_at(P, T)$ for the fluent P being true at time T, are evaluated using the temporal reasoning capability. The derivability relation of goal decision is implemented by the following top-level rule:

```
goal_decision(OrderedGoals):-
        now(Tnow),
        findall(Goal, gorgias_solve(kb_gd(Tnow), gd(Goal)), Goals),
        list_to_ordered_set(Goals, OrderedGoals).
```

which, via the `gorgias_solve/2` predicate, links to the GORGIAS proof-procedure to select the list of goals to be considered next. Goals are returned by finding first the list of possible goals, and then filtering out incompatible goals in the returned answer set. Non-ground goals, i.e. goals with a non-fixed but possibly constrained time, may be returned.

[4] GORGIAS can be obtained from `http://www2.cs.ucy.ac.cy/~nkd/gorgias/`.

Planning, Reactivity & Temporal Reasoning. The Planning, Reactivity, and Temporal Reasoning capabilities are implemented by means of CIFF, which is used to reason about knowledge on fluents and actions that either initiate or terminate these fluents. Our approach to planning has been described in [Endriss et al., 2004a] and has been currently extended in [Mancarella et al., 2004]. The reactivity capability is closely related to planning; the reactivity knowledge base is an extension of the planning knowledge base. The main difference is that there is not a specific selected goal passed to reactivity, as in the case of planning, but rather a list of assumptions (goals and actions in the current state) and the list of temporal constraints with respect to which it is checked whether some reaction is needed.

Temporal Reasoning uses CIFF to check whether a fluent `F` holds at a given time point `T`, i.e. `[holds_at(F, T) | CS]` is entailed by the temporal reasoning knowledge base $KBtr$, with `T` an existentially quantified temporal variable, possibly subject to a constraint set `CS`. A *credulous* entailment, accounting for the possibility for a fluent to hold, and a *skeptical* one, for the certainty, are defined:

```
query_credulously_TR(KBtr, [holds_at(F,T) | CS], Answer):-
         extract_significant_points(KBtr, SPT),
         instantiate_theory(SPT, KBtr, GKBtr, GICs),
         ciff(GKBtr, GICs, [holds_at(F,T) | CS], Answer).

query_skeptically_TR(KBtr, [holds_at(F,T) | CS], Answer):-
         query_credulously_TR(KBtr, [holds_at(F,T) | CS], Answer),
         \+ query_credulously_TR(KBtr, [holds_at(neg(F),T) | CS], _).
```

The credulous predicate eventually calls CIFF over a grounded theory and uses CIFF's constraint solver. The skeptical predicate is defined in terms of the credulous one to prove the fluent and to check that the negation of the fluent cannot be proved. For the details of the temporal reasoning framework used in PROSOCS the interested reader is referred to [Bracciali and Kakas, 2004].

3.4 Implementing the Transitions

Transitions are implemented in PROSOCS following the formal computational model of KGP as described in [Bracciali et al., 2004]. To give an implementation example, we present here the definition of the goal revision (GR) transition, which is called to enable the agent to revise its goals. This is done by building the new goals for the tree, further updating the state with these new goals at the current time `Tnow` supplied as input. The following predicate implements this transition:

```
goal_revision(Tnow) :-
        sub_trees(SubTrees),
        select_children(SubTrees, TopLevelGoals),
        revised_goals(Tnow, TopLevelGoals, [], NewGoals),
        update_goals(NewGoals).
```

`sub_trees/1` above accesses a list with two members, one indexing the *reactive* and the other the *non–reactive* sub-trees of the goal tree. From these the `TopLevel` goals are selected as the children of the sub-trees using `select_children/2`. The `TopLevel` goals are then revised in order to build up the `NewGoals`. Revision is achieved using the predicate:

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

The recursive predicate `revised_goals/4` builds the new goals from the current state incrementally, that is, by adding only goals whose parent is in the list of the `NewGoals`. Checking that a list of goals persist involves checking that a single goal persists, and if it does, the goal is added in the list of (new) `Goals`; otherwise the goal is ignored in the new state. This process continues until checks are carried out for every top–level goal. To check that a goal persists we make sure that the goal has not been achieved and its time has not run out:

```
persists_goal(Tnow, G):-
        \+ goal_achieved(Tnow, G),
        goal_not_timed_out(Tnow, G).
```

In this context, the definition of an achieved goal is represented as:

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

The above definition shows how the capability of temporal reasoning, invoked through its API, assists the checking of the temporal constraints over the goal. We define similarly when a goal has not timed out.

3.5 Engineering the Cycle Theory

In the current implementation of PROSOCS, the mind of an agent is called by the agent's *body-control* (as described in [Stathis et al., 2004]) to control the behaviour of the agent in atomic cycle steps, i.e. applications of the above illustrated transitions, possibly provided with an opportune input. The program is therefore named `cycle_step/3` and it is defined as:

```
cycle_step(TransitionStep, Tnow, Tnext):-
        select_step(TransitionStep, Tnow),
        apply_step(TransitionStep, Tnow, Tnext).
```

The details of the `TransitionStep`, namely, the name of the next `Transition` and its `Input` are selected using the cycle theory represented in the mind of the agent, which relies on preference reasoning. This allows the behaviour control of an agent to be flexibly programmable, according to the environment where the agent operates.

```
select_step(step(Transition, Input), Tnow) :-
        now(Tnow),
        gorgias_solve(ct(Tnow, normal), step(Transition, Input)).
```

The execution of the query `step(Transition, Input)` depends on the given cycle theory `ct` of the agent and its state at the current time `Tnow`. Execution of the query chooses the first transition that can be proved admissible by GORGIAS from the theory `ct(Tnow, normal)`. The parameter `normal` indicates that the cycle relies on a pre-specified sequencing of transitions that in PROSOCS we call a pattern of behaviour. For example, to specify a rule in the `normal` pattern of behaviour in order to respond to communications received by another agent via the passive observation introduction (POI) transition (see section 2.1), we need to write:

```
rule(ct(_, normal), prefer(step('GI',_), step(_,_)), []) :-
      last_transition('POI', Obs),
      comm_msg(Obs).
```

The rule states that the goal introduction transition GI is preferred to any other transition after a POI transition. Hence, the agent will then decide on the response to the observation `Obs` through its goal decision capability. Such a pattern can be reused across applications, provided the agent developer chooses to select it as appropriate for the application at hand.

Once a transition is selected it is applied on the state of the agent as follows:

```
apply_step(NextStep, Tnow, Tnext) :-
      commit(NextStep),
      record(NextStep, Tnow),
      increment(Tnow, Tnext).
```

The application of a selected step involves calling `commit/1` to execute the code implementing the transition. For example, to call the goal revision transition GR, discussed in the previous section, we write:

```
commit(step('GR', [Tnow])) :- goal_revision(Tnow).
```

Once a transition is committed to, any state changes will have been implicitly performed as an effect of the call, and then the system will invoke `record/2` to add the current step in the history of the transitions applied so far. Finally, the application of a transition is completed by calling `increment/2`, to increase by one the current time-stamp, which is recorded to indicate that the state has changed. For a more detailed account of the cycle theory framework used in PROSOCS, the reader is referred to [Kakas et al., 2004a].

4 Programming an agent's mind

To illustrate how to program the mind of a PROSOCS agent we draw upon a scenario that was originally reported in [Stathis, 2002] and has been recently related to practical applications for ambient intelligence in [Stathis and Toni, 2004]. The scenario describes a businessman who makes his trip easier by carrying with him a personal communicator, a device that is a hybrid between a mobile phone and a PDA. We assume that the application running on this personal communicator provides the environment for an agent that augments the direct manipulation interface of the device with proactive information management within the device and flexible connectivity to smart services, assumed to be available in objects available in the global environment the businessman travels within.

There are various kinds of interactions reported in the scenario, here we present the ones that allow us to show (a) how to program an agent's goal decision capability, and (b) how to program an agent's reactivity and planning capability. These

two simple situations will let us show how we can specify the various domain-specific parts of the capabilities, thus also exemplifying how to write a program for a PROSOCS agent that operates in a dynamic environment.

4.1 The Goal Decision capability

To program the goal decision knowledge base KB_{GD} involves writing two main parts: 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 agent.

Rules in the lower-level (goal generation part) are written as GORGIAS rules of the form:

```
rule(kb_gd(Tnow), Label, gd(Goal), []) :- Conditions.
```

where *Label* names the rule, $Goal = (Literal, Time)$ is a goal fluent literal and *Conditions* has the syntax of a PROLOG rule body. As an example, the goal of rising a low battery alarm (`lba`) within two time instants from when the low battery is detected, can be expressed by the GORGIAS rule:

```
rule(kb_gd(Tnow), r(lba,T), gd((lba,T)), []) :-
      ground(Tnow),
      temporal_reasoning((low_battery, Tnow)),
      T is Tnow + 2.
```

where `temporal_reasoning((Literal,Time))` is evaluated by calling the Temporal Reasoning capability (the skeptical version) to check whether *Literal* holds at time T. Conflicts of goals can be expressed by means of the predicate $incompatible/3$, like, to state that the goals lsv and lba are incompatible:

```
incompatible(kb_gd(_), (lsv,T), (lba,T)).
```

Preferences over goal generation rules are expressed in the higher-level part of KB_{GD} by priority rules of the form:

```
rule(kb_gd(Tnow), Label, prefer(Label1,Label2)) : - Body.
```

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

Sometimes, rules may derive non-ground goals. For example, the previous rule could also be written as:

```
rule(kb_gd(Tnow),r((lba,T,[Tnow<T<T1])),gd((lba,T,[Tnow<T<T1])),[]):-
        ground(Tnow),
        temporal_reasoning((low_battery, Tnow)),
        T1 is Tnow + 2.
```

where the T that appears in the head is existentially quantified and it is constrained by constraints of the form $T_{low} < T < T_{high}$, and all the temporal constraints of the heads of the rule have been passed in their names.

4.2 Reactivity and Planning capabilities

To exemplify reactivity and planning, one of the interactions exemplified in the scenario indicate that the agent of a businessman called *Francisco* (this agent is identified here as `f`) is connected in an ad-hoc network to ask another agent that provides train information for a train station called *San Vincenzo Station* (this other agent is identified here as `svs`). We want to show next how to program the train station agent to interact with any "passenger" agent that requires train information. The CIFF rule below, specified in KB_{react}, represents one of the rules the train station agent needs to reply to requests in the context of the `query-ref` protocol of FIPA:

```
[observed(Ag, tell(Ag, svs, query_ref(Q), D, T0), T1),
 holds(have_info(Q, I), T1)
] implies
[assume_happens_after( tell(svs, Ag, inform(Q,I), D, T2), T1)].
```

The above rule simply states that if an agent `Ag` asks a query `Q` in a dialogue `D` the agent `svs` and `svs` has the information `I` for `Q`, then `svs` must inform the content of `I` after it receives the request from `Ag`. `assume_happens_after(A,T)` can be thought of as a macro that can be used to express that action `A` should be executed at any time after `T`.

We also need to specify information in the KB_{plan}, for example:

```
holds_initially( have_info(arrival_time(tr123), 10:32) ).
precondition( tell(svs, _, inform(Q,I), _, _), have_info(Q, I) ).
executable( tell(svs, Ag, _, _, _) ) :- not (Ag = svs).
```

The above abductive event calculus program simply states that the agent holds initially information about the arrival time of a train. It also provides the preconditions of actions, in this case, this states that before giving a piece of information the agent must have this information at hand. Other types of checks are required, for example, that when a communicative action is executed by the agent, it is executed for other agents to receive it, but not the agent that sends it.

4.3 Example Run

Using the `normal` pattern of behaviour for the cycle theory, we have implemented a number of interactions (taken from the previously represented scenario) in PROSOCS and we have run examples using the facilities provided by the prototyping platform. Fig. 3 is taken from one of these experiments, showing the state of the `svs` agent as it is animated by the platform's interface.

On the top-left corner of the agent's interface the platform presents information about the other agents `svs` sees in the environment, including itself that is denoted by the keyword `me`. On the top-right area of the agent's interface the platform shows the incoming communication by other agents, in this case, a `query_ref` request. Answers to incoming communication are then presented in the bottom-right area of the agent's interface, in this case, an inform message to the agent that asked the query. Finally, the bottom-left area gives information about the steps that have been executed by the cycle theory, in this case, the agent has been started, has tried to introduce a goal through goal introduction (GI), but as there were no goals selected, it has been waiting for input (i.e. the reactivity transition (RE) has nothing to react to). Only when a message is received via passive observation introduction (POI), the agent executes a reply action using the action execution (AE) transition, and after that carries on waiting for input, as before.

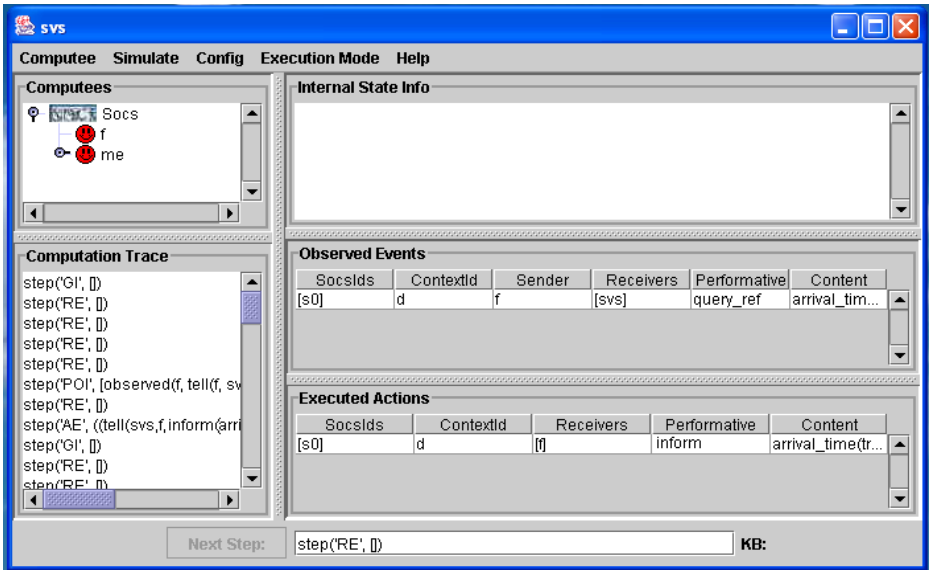

Figure 3: An Example Run.

5 Related Work

A plethora of software platforms for building software agents is already available [Logan, 1998, Eiter and Mascardi, 2002, Ricordel and Demazeau, 2000, Dart et al., 1999, Serenko and Detlor, 2002]. This renders the task of reviewing all the related work in this area gigantic [Eiter and Mascardi, 2002]. 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 agents. For example, we exclude platforms whose main focus is mobility. Also, the related work presented next is focused on comparing single agents and not the generic platforms in which these agents have been developed. For a comparison of PROSOCS with other platforms the reader is referred to [Stathis et al., 2004].

At a first glance, the KGP model used in PROSOCS and a number of agents systems built according to the classical BDI model [Bratman et al., 1988] appear to be similar. However, at a closer look, the models are quite different. One major difference is that KGP is not based on a modal-logic approach to represent an agent's beliefs but instead uses a non-monotonic computational logic that supports defeasible reasoning for the knowledge of agents. Also, unlike BDI, KGP is based on a cycle theory [Kakas et al., 2004a] that allows the specification of flexible patterns of operation and not a fixed cycle. Like BDI, however, KGP allows plans to be generated dynamically (as part of the reasoning process) and statically through the use of plan libraries. For lack of space, we refer the interested reader to [Kakas et al., 2004b], for a critical review of KGP with most logical models of agency available in the literature, including BDI.

Another important characteristic of PROSOCS agents is that they can be thought of as *first-class objects*, in the sense that the developer can start an agent and inherit a set of tools supporting the development of both a reasoning component and the interaction with the environment for free. Many FIPA compliant platforms, for example FIPAOS [Poslad et al., 2000], or ZEUS [Nwana et al., 1999], do not commit to a model of agency but support only the interaction of the agent with the environment. In such platforms the programmer has to develop from scratch

the reasoning capabilities of the agent. Other FIPA compliant approaches, such as JADE [Bellifemine et al., 2001] and 3APL [Hindriks et al., 1999, 3APL, 2003], make use of tools that support the reasoning capabilities of the agent (for the mind). Unlike our logic-based approach which is closer to the specification of an agent, JADE can be used with JADEX [Pokahr et al., 2003] to implement the goal-directed behaviour found in BDI agents using a JAVA API. On the other hand, 3APL uses a logic-based language which, like BDI, is linked to a modal-logic framework [Meyer et al., 2001].

An approach similar to PROSOCS is the recent development of the JASON [JASON, 2004] platform that implements AgentSpeak(L) [Rao, 1996] agents. Like JASON agents, PROSOCS agents attempt to narrow the gap between the specification and executable model of an agent. Also like JASON agents, agent interactions in PROSOCS can be verifiable because PROSOCS is based on a formal computational framework for individual agents and their social interactions. However, agents in PROSOCS and JASON differ in the ways AgentSpeak(L) and KGP differ: (a) in KGP there are explicit links amongst the goals and between the goals and plans, and the priorities amongst potential goals, which are not restricted to temporal orderings as in AgentSpeak(L). Also, the control of AgentSpeak(L) agents is based on a fixed cycle, and not on the flexible cycle theory of KGP.

Like agents in JASON, agents in the IMPACT platform (see for example [Arisha et al., 1999] and [Subrahmanian et al., 2000]) are treated too as first-class objects. Unlike PROSOCS, where agents are built from scratch by assuming a logic programming approach, an IMPACT agent may be built on top of an arbitrary piece of software, defined in any programming language. IMPACT uses deontic concepts to represent action and action policies which in KGP correspond to abductive integrity constraints, while the system relies on an interval–based logic for temporal reasoning rather than the Event Calculus approach of PROSOCS. Although IMPACT agents have additional features, for example reasoning about uncertainty and the representation of security policies, they do not have a flexible cycle theory and thus cannot support (behaviourally) heterogeneous agents, as we can in PROSOCS.

Agent programs developed in the high-level programming language INDIGOLOG [DeGiacomo et al., 2001] support on-line planning and plan execution in dynamic and incompletely known environments. Such agent programs can also perform sensing actions that acquire information at runtime and react to exogenous actions. INDIGOLOG is a member of the GOLOG family of languages [Levesque et al., 1997] that use a Situation Calculus theory of action to perform the reasoning required in executing the program. Unlike INDIGOLOG agents in PROSOCS we rely on abductive logic programming and logic programming with priorities combined with an Event Calculus approach to program an agent. Moreover, goals cannot be decided dynamically in INDIGOLOG, instead in PROSOCS they change according to the patterns specified in the goal decision theory.

CONGOLOG [DeGiacomo et al., 2000], an extension of the GOLOG family, has also been introduced to add support for concurrent processes with possibly different priorities, interrupts, and exogenous events to agent programs. However, like earlier planning-based systems, CONGOLOG assume an off-line search model. To overcome the limitation of off-line search, INDIGOLOG uses the notion of exogenous actions assuming that there is a concurrent process executing these actions outside the control of the agent. This assumption implies that it is up to the programmer of an agent to interface INDIGOLOG to the rest of the agents of an application domain, something that in PROSOCS the programmer gets for free.

Finally, agents developed in the logic-based language GO! [Clark and McCabe, 2003] has many features in common with multi-threaded

PROLOG systems and provides types, higher–level programming constructs, such as single solution calls, "iff" rules, and the ability to define 'functional' relations as functions. However, as with INDIGOLOG, GO! does not directly rely on any specific agent architecture or agent programming methodology, hence the system does not yet treat agents as first-class objects. As a result, facilities such as planning, temporal reasoning, and preference reasoning are not available, so the programmer has to build them from scratch; the system though provides library modules, so that components such a KGP module can be reused once developed generically, as in PROSOCS.

6 Conclusion

We have discussed how to develop PROSOCS agents, software agents that are built according to the KGP model of agency. KGP is a model for the mental state of the agent, so that the agent can act autonomously using a collection of logic theories, providing the mind's reasoning functionalities. The behaviour of the agent is controlled by a *cycle theory* that specifies the agent's preferred patterns of operation. In this context, we have shown how to implement the mind's generic functionality in such a way so that it can be instantiated by PROSOCS for different agents across applications. We have illustrated through the development of a concrete example how an agent developer might program (or instantiate) the generic functionality of the mind for a simple application.

For future work we plan to experiment with the platform on global computing applications and test the PROSOCS agents and social infrastructure with complex problems, along the lines already presented in [Stathis and Toni, 2004]. For this purpose, we have also started to extend PROSOCS to support interactions of agents with objects in the environment, other than the communicative interaction support available in the PROSOCS platform.

Acknowledgments

We would like to thank the anonymous referees for their comments to a previous version of this article. We would also like to thank our colleagues Paolo Mancarella and Nicolas Maudet for their help in an earlier implementation of some PROSOCS transitions and Francesca Toni for helpful discussions on the representation of the KGP state. This work was partially funded by the IST programme of the European Commission, Future and Emerging Technologies under the IST-2001-32530 SOCS project, within the Global Computing proactive initiative. The last author would also like to acknowledge the support of: (a) the Italian MIUR programme "Rientro dei Cervelli" and (b) Nuffield Foundation UK, grant no NAL/00320/G.

References

[GC, 2000] (2000). Global Computing, Future and Emerging Technologies. http://www.cordis.lu/ist/fetgc.htm.

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

[Alberti et al., 2004] Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., and Torroni, P. (2004). Compliance verification of agent interaction: a logic-based software tool. *Applied Artificial Intelligence*. This volume.

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

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

[Bondarenko et al., 1997] Bondarenko, A., Dung, P. M., Kowalski, R. A., and Toni, F. (1997). An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence*, 93:63–101.

[Bracciali et al., 2004] Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Mancarella, P., Sadri, F., Stathis, K., Terreni, G., and Toni, F. (2004). The KGP Model for Global Computing: Computational Model and Prototype implementation. In Quaglia, P. and Priami, C., editors, *Post-Proceedings of the Global Computing 2004 Workshop*, LNCS 3267, pages 342 – 369, Rovereto, Italy. Springer.

[Bracciali and Kakas, 2004] Bracciali, A. and Kakas, A. (2004). Frame consistency: Computing with frame explanations. In *Proceedings of NMR 2004*, pages 79–87.

[Bratman et al., 1988] Bratman, M.E., Israel, D.J., and Pollack, M.E. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4.

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

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

[Dart et al., 1999] Dart, P., Kazmierczak, E., Martelli, M., Mascardi, V., Sterling, L., Subrahmanian, V.S., and Zini, F. (1999). 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. IEEE.

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

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

[Demetriou and Kakas, 2003] Demetriou, N. and Kakas, A. C. (2003). Argumentation with abduction. In *Proceedings of the fourth Panhellenic Symposium on Logic*.

[Dimopoulos and Kakas, 1995] Dimopoulos, Y. and Kakas, A. C. (1995). Logic programming without negation as failure. In *Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon*, pages 369–384.

[Dung, 1995] Dung, P. M. (1995). On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77:321–357.

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

[Endriss et al., 2004a] Endriss, U., Mancarella, P., Sadri, F., Terreni, G., and Toni, F. (2004a). Abductive Logic Programming with CIFF: Implementation and Applications. In *Proceedings of the Convegno Italiano di Logica Computazionale (CILC-2004)*. University of Parma.

[Endriss et al., 2004b] Endriss, U., Mancarella, P., Sadri, F., Terreni, G., and Toni, F. (2004b). Abductive logic programming with CIFF: System Description. In Alferes, J. and Leite, J., editors, *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA-2004)*, LNAI 2329, pages 680–684. Springer.

[Endriss et al., 2004c] Endriss, U., Mancarella, P., Sadri, F., Terreni, G., and Toni, F. (2004c). The CIFF Proof Procedure for Abductive Logic Programming with Constraints. In Alferes, J. and Leite, J., editors, *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA-2004)*, LNAI 2329, pages 31–43. Springer.

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

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

[JASON, 2004] JASON (2004). A Java-based AgentSpeak interpreter used with Saci for multi-agent distribution over the Net. http://jason.sourceforge.net/.

[Kakas et al., 2004a] Kakas, A., Mancarella, P., Sadri, F., Stathis, K., and Toni, F. (2004a). Declarative Agent Control. In *Proceedings of 5th CLIMA Workshop*, Lisbon, Portugal. Springer.

[Kakas et al., 2004b] Kakas, A., Mancarella, P., Sadri, F., Stathis, K., and Toni, F. (2004b). The KGP Model of Agency. In *Proceedings of 16th European Conference of Artificial Intelligence (ECAI-04)*, pages 33–37, Valencia, Spain.

[Kakas et al., 1998] Kakas, A. C., Kowalski, R. A., and Toni, F. (1998). The role of abduction in logic programming. In Gabbay, D. M., Hogger, C. J., and Robinson, J. A., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press.

[Kakas et al., 1994] Kakas, A. C., Mancarella, P., and Dung, P. M. (1994). The acceptability semantics for logic programs. In *Proceedings of the Eleventh International Conference on Logic Programming,* Santa Marherita Ligure, Italy, pages 504–519.

[Kakas and Moraitis, 2003] Kakas, A. C. and Moraitis, P. (2003). Argumentation based decision making for autonomous agents. In Rosenschein, J. S., Sandholm, T., Wooldridge, M., and Yokoo, M., editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2003)*, pages 883–890, Melbourne, Victoria. ACM Press.

[Kakas and Toni, 1999] Kakas, A. C. and Toni, F. (1999). Computing argumentation in logic programming. *Journal of Logic and Computation*, 9:515–562.

[Kowalski and Sergot, 1986] Kowalski, R. A. and Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, 4(1):67–95.

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

[Logan, 1998] Logan, B. (1998). Classifying agent systems. In Baxter, J. and Logan, B., editors, *Software Tools for Developing Agents: Papers from the 1998 Workshop*, pages 11–21. AAAI Press.

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

[Mancarella et al., 2004] Mancarella, P., Sadri, F., Terreni, G., and Toni, F. (2004). Planning partially for situated agents. In *Proceedings of 5th CLIMA Workshop*, Lisbon, Portugal. Springer.

[Meyer et al., 2001] Meyer, J-J., de Boer, F., van Eijk, R., Hindriks, K., and van der Hoek, W. (2001). On Programming KARO Agents. *Journal of KGPL*, 9(2):245–256.

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

[Pokahr et al., 2003] Pokahr, A., Braubach, L., and Lamersdorf, W. (2003). JADEX: Implementing a BDI-Infrastructure for JADE Agents. *EXP - In Search of Innovation (Special Issue on JADE)*, 3(3):76–85.

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

[Prakken and Sartor, 1997] Prakken, H. and Sartor, G. (1997). Argument-based extended logic programming with defeasible priorities. *J. of Applied Non-Classical Logics*, 7(1).

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

[Ricordel and Demazeau, 2000] Ricordel, P.M. and Demazeau, Y. (2000). From Analysis to Deployment: A Multi-Agent Platform Survey. In Omicini, A., Tolksdorf, R., and Zambonelli, F., editors, *Engineering Societies in the Agents World*, LNAI 1972, pages 93–105. Springer. 1st International Workshop (ESAW'00), Berlin, Germany, Revised Papers.

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

[Shanahan, 1997] Shanahan, M. (1997). Event calculus planning revisited. In *Proceedings 4th European Conference on Planning*. LNCS 1348, pages 390–402.

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

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

[Stathis et al., 2004] Stathis, K., Kakas, A. C., Lu, W., Demetriou, N., Endriss, U., and Bracciali, A. (2004). PROSOCS: a platform for programming software agents in computational logic. In Müller, J. and Petta, P., editors, *Proceedings of the Fourth International Symposium "From Agent Theory to Agent Implementation" (AT2AI-4 – EMCSR'2004 Session M)*, pages 523–528, Vienna, Austria.

[Stathis and Toni, 2004] Stathis, K. and Toni, F. (2004). Ambient intelligence using KGP agents. In Markopoulos, P., Eggen, B., Aarts, E., and Crowley, J. L., editors, *Proceedings of 2nd European Symposium on Ambient Intelligence*. LNCS 3295 pages 351–362. Springer.

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