SOCS

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

IST-2001-32530

Deliverable D4: A logic-based approach to model computees

Project number:	IST-2001-32530
Project acronym:	SOCS
Document type:	D (deliverable)
Document distribution:	I (internal to SOCS and PO)
CEC Document number:	IST32530/UCY/006/D/I/b2
File name:	4006-b2[D4].pdf
Editor:	Antonis Kakas, Paolo Mancarella, Francesca Toni
Contributing partners:	ALL
Contributing workpackages:	WP1
Estimated person months:	55
Date of completion:	27 June 2003
Date of delivery to the EC:	30 June 2003
Number of pages:	119

ABSTRACT

A computee is an autonomous computational entity that operates in an open and dynamic environment. This report proposes a formal framework for computees, called *KGP* (*Knowledge, Goals, Plan*), that synthesises their reasoning and sensing capabilities into a model for their operation. Within this model the computee is able to reason about goals and plans and to execute its plans depending on the state of its (perception of) the external environment. The computee is also able to reason about its own behaviour and make intelligent choices about it. The model is based upon computational logic, both to model the reasoning capabilities and the reasoning about behaviour. The report also addresses the links of this model with the society model proposed in the companion deliverable D5.

Deliverable D4:
A logic-based approach to model computees

Antonis Kakas[*],
Paolo Mancarella[+],
Fariba Sadri[•],
Kostas Stathis[!],
Francesca Toni[•]

Departments of Computer Science,
∗ University of Cyprus, Cyprus
+ University of Pisa, Italy
• Imperial College London, UK
! City University, London, UK

ABSTRACT
A computee is an autonomous computational entity that operates in an open and dynamic environment. This report proposes a formal framework for computees, called *KGP* (*Knowledge, Goals, Plan*), that synthesises their reasoning and sensing capabilities into a model for their operation. Within this model the computee is able to reason about goals and plans and to execute its plans depending on the state of its (perception of) the external environment. The computee is also able to reason about its own behaviour and make intelligent choices about it. The model is based upon computational logic, both to model the reasoning capabilities and the reasoning about behaviour. The report also addresses the links of this model with the society model proposed in the companion deliverable D5.

Contents

1 Introduction **6**

2 General approach **10**

3 Background **15**
3.1 Logic programming . . . 15
3.2 Abductive logic programming . . . 16
3.3 Logic programming with priorities . . . 18
3.4 Constraint predicates in computational logic . . . 20
3.5 ALP and LPwNF with Constraints . . . 21

4 Preliminaries of the KGP model **22**

5 State of a computee **25**

6 Capabilities **28**
6.1 Planning . . . 28
6.1.1 KB_{plan}: Abductive Event Calculus . . . 28
6.1.2 Specification of $\models_{plan}$ . . . 31
6.1.3 Example of $\models_{plan}$ . . . 32
6.1.4 Possible variants of KB_{plan} and $\models_{plan}$ . . . 32
6.2 Identification of preconditions . . . 33
6.2.1 Specification of $\models_{pre}$ . . . 33
6.2.2 Example of $\models_{pre}$ . . . 33
6.3 Temporal reasoning . . . 33
6.3.1 KB_{TR}: Extended abductive event calculus . . . 34
6.3.2 Specification of $\models_{TR}$ . . . 35
6.3.3 Example of $\models_{TR}$ . . . 36
6.3.4 Extension of KB_{TR} for failing actions . . . 36
6.4 Reactivity . . . 38
6.4.1 KB_{react}: reactive constraints . . . 38
6.4.2 Specification of $\models_{react}$ . . . 39
6.4.3 Example of $\models_{react}$ . . . 40
6.5 Goal decision . . . 40
6.5.1 The knowledge base KB_{GD} . . . 40
6.5.2 Specification of $\models_{GD}$ . . . 42
6.5.3 Examples of $\models_{GD}$ . . . 42
6.5.4 Goal decision and personality . . . 45
6.5.5 Goal Decision and Reactivity . . . 46
6.6 Sensing . . . 46

7 Transitions **46**
7.1 Goal Introduction (GI) . . . 47
7.2 Plan Introduction (PI) . . . 48
7.3 Reactivity (RE) . . . 49
7.4 Sensing Introduction (SI) . . . 49

7.5 Passive Observation Introduction - (POI) . . . 50
7.6 Active Observation Introduction - (AOI) . . . 50
7.7 Action Execution (AE) . . . 51
7.8 Goal Revision (GR) . . . 52
7.9 Plan Revision (PR) . . . 52

8 Selection functions **53**
8.1 Core selection functions . . . 54
8.1.1 Action selection . . . 54
8.1.2 Goal selection . . . 55
8.1.3 Fluent selection . . . 56
8.1.4 Precondition selection . . . 57
8.2 Heuristic selection functions . . . 57
8.2.1 Heuristic action selection . . . 57
8.2.2 Heuristic goal selection . . . 58
8.2.3 Heuristic fluent selection . . . 58
8.2.4 Heuristic precondition selection . . . 59
8.3 Selection functions and revision transitions . . . 59
8.4 Resource-boundness . . . 59

9 Cycles of behaviour of computees **60**
9.1 Fixed cycles . . . 61
9.2 Cycle theories . . . 63
9.2.1 Operational Trace . . . 64
9.2.2 The basic component: $\mathcal{T}_{basic}$ . . . 65
9.2.3 The interrupt component: $\mathcal{T}_{interrupt}$ . . . 67
9.2.4 The initial component: $\mathcal{T}_{initial}$ . . . 67
9.2.5 The behaviour component: $\mathcal{T}_{behaviour}$ . . . 67
9.2.6 Properties of $\mathcal{T}_{cycle}$ . . . 68
9.3 Cycle Patterns and Profiles of Behaviour . . . 69
9.3.1 Fixed cycles via cycle theories . . . 69
9.3.2 Patterns and Profiles of Behaviour . . . 69
9.4 Hierarchies and multi behaviour criteria . . . 72

10 Computees in Societies **74**
10.1 Communication . . . 75
10.1.1 Language for communication . . . 75
10.1.2 Communication actions . . . 76
10.1.3 Generating communication actions as part of a plan . . . 76
10.1.4 Policies for communication . . . 76
10.1.5 Deciding who best to communicate with in order to achieve objectives . . . 77
10.2 Conforming to society's protocols . . . 78
10.3 Computees entering and leaving societies . . . 79
10.4 Responding to the society's expectations . . . 80

11 Possible extensions **81**
11.1 Plan introduction transition with intelligent selection of plans 81
11.2 Knowledge base revision transition 81
11.3 Conditional Goals 82
11.4 Concurrent execution and interruption of transitions 82
11.5 Utilities and Costs 83

12 Related work **83**
12.1 The BDI model 84
12.1.1 Classical BDI: Architectures, Logics, and Implementations 84
12.1.2 Classical BDI and *KGP*: A comparison 86
12.2 AGENT0 88
12.3 AgentSpeak 89
12.3.1 (Concurrent, Object-Oriented) AgentSpeak 89
12.3.2 AgentSpeak(L) 89
12.3.3 AgentSpeak(XL) 91
12.4 3APL 91
12.4.1 Agent Programs 91
12.4.2 Operational Semantics and Control 93
12.4.3 Agent Communication 94
12.5 DESIRE 95
12.6 Computational logic-based approaches 98
12.6.1 IMPACT 98
12.6.2 $\mathcal{MINERVA}$ 100
12.6.3 GOLOG 102
12.6.4 Vivid Agents 103

13 Evaluation **104**

14 Conclusion **109**

1 Introduction

The SOCS project aims at developing a model for distributed systems consisting of intelligent autonomous entities interacting with each other within the Global Computing (GC) environment. The systems are *distributed* and the entities are *autonomous* as activity, as envisaged by the GC vision, is not centrally controlled. The GC environment has the characteristic that it is *open* and thus can vary over time. Our model therefore needs to allow the dynamic evolution of its distributed systems (societies), with the entities composing them varying over time, both in number and nature, so that

- the interaction amongst entities cannot be hardwired in a fixed topology within these systems;
- the entities composing these systems cannot be guaranteed to be homogeneous, and might be highly *heterogeneous* instead.

Here, our interpretation of *openness* borrows from work by [Dav01] and by [AKGP01] (derived from [Hew91]). Indeed, [Dav01] proposes that in an open system (or artificial society) it is possible for any entity to enter the system simpy by starting an *interaction* with a member of it. This definition of openness is focused on membership, and therefore it necessarily needs to take into account how an entity can enter/leave the system. If entering/exiting a system can be done without any restriction, we can classify the system as open. On the other side, [AKGP01] gives a definition of open system (or artificial society) based on externally observable features of entities within the system. This openness definition implies that members could be heterogeneous, and possibly non-cooperative. Heterogeneity of entities might affect their internal architectures, the platforms on which they are realised as well as their behaviours (under the same circumstances).

Again as a consequence of the openess of GC systems, entities composing these systems need to

- *adapt* to changes, whilst making decisions;
- operate (e.g. reason and make decisions) despite having only *partial information* about the environment and the other entities in it.

The need to adapt arises both for entities joining systems and for entities remaining in systems when other entities join/leave them. The partiality of information might arise from the entities having newly joined one such system and having only a partial view over the system. This partiality might also arise from the autonomy of the entities, and their unwillingness to disclose information about themselves and the system itself.

Within SOCS we interpret the GC vision as follows.

Entities in these systems need to be "*intelligent*", by employing advanced forms of reasoning, in order to cope with the challenges posed by the GC vision. Autonomy required from these systems is addressed via intelligent decision making. Furthermore, in order to adapt, they need to operate and reason with changes taking place over time, having incomplete information. They also need to dynamically maintain a consistent view of their external world as this changes. In order to interact freely, they can use *high-level communication*, as understood in multi-agent systems.

We assume that entities are heterogeneous as far as behaviour, but aim at using *Computational Logic* (CL), as understood in [Kow79, Llo87, Kow90, KS02], as an abstraction for the

internal configuration of the entities, for their internal reasoning and for interactions with each other. We call the entities *computees*, standing for agents in CL. In particular, computees base their functional capabilities on CL reasoning techniques. Their internal operation is also specified and controlled using CL. We call the systems of such entities *societies*, as they are characterised by "social rules" for computees to interact and operate in the presence of each other.

In this context of societies of computees, adaptability amounts to adapting the "goals", "knowledge" and "plan" of the computees (namely their internal state), as the societies in which they are located evolve and as computees move from one society to another.

A number of techniques have been developed within CL for addressing tasks such as temporal reasoning in a changing environment, hypothetical reasoning for dealing with incomplete knowledge, hypothetical reasoning for planning, hypothetical reasoning to achieve communication, argumentation for decision-making and inductive logic programming for learning. However, in order to cope with the GC challenges, CL techniques in isolation are inadequate, as none serves all dimensions in the operation of computees.

In this deliverable, we present a model for individual computees integrating a number of existing CL techniques, in order to achieve the enhanced performance which is required by the GC vision. The model we propose has the following characteristics. The state of a computee consists of its *goals*, its *plan*, and its *knowledge base*. The terms "goals" and "knowledge base" are drawn from conventional CL literature, where they are used to refer, respectively, to the query to prove (or objective to achieve) and to the set of beliefs or data from which to prove (achieve) the goal. The term "plan" is drawn from conventional Artificial Intelligence literature on planning. We call the model KGP (Knowledge, Goals, Plan), based on the definition of such internal state. However, the characteristics of the model go beyond these choices for the representation of the internal state, as outlined below and explained in detail within this document.

All of the components of the state of a computee change over time, as a consequence of the evolution of the computee, its interaction with its environment and the other computees in it. A computee is equipped with reasoning *capabilities*, including planning and reactivity, which have a high-level semantic description within CL. Each of the capabilities is defined with respect to a specific knowledge base, which is part of the overall knowledge base of the state of the computee. Together with the reasoning capabilities, the computee has sensing capabilities that allow it to obtain (current) information from its environment.

A computee then synthesises these capabilities to produce a versatile behaviour that combines the achievement of its own individual goals and cooperation in the society (or societies) that it belongs to. The synthesis is obtained by employing the capabilities within *transitions*, which specify how the state of the computee changes as a result of the application of various capabilities. The transitions are then integrated together under *cycle theories*, that specify declaratively combinations of transitions depending on a desired profile of behaviour for the computee and the particular circumstances under which the computee is operating. The cycle theory of a computee therefore replaces more conventional one-size-fits-all theory of control as provided by conventional agent cycles in the multi-agent literature. The use of cycle theory as a form of declarative control is to the best of our knowledge novel in the agent literature. We share the same aims as those of [DdBD$^+$02], which however provides only a limited form of declarative control with a lower degree of flexibility than that we provide via cycle theories. This will be discussed further in the document.

More specifically the KGP model supports a general mode of operation of the computee where it:

- maintains a view of its environment,
- decides what its goals should be,
- decides which goals to address next,
- decides and plans how to achieve these goals,
- executes incrementally its plans,
- reacts to information received from the environment or communication received from other computees by modifying its goals,
- re-evaluates its previous decisions and knowledge based on new information that it receives from the environment and
- enriches its knowledge, by accommodating newly observed information,

all controlled by a cycle theory, expressing the desired profile of overall behaviour for the computee equipped with it. The notion of cycle theory and its use to determine the behaviour of computees could in principle be imported into any agent system, to replace conventional fixed cycles.

The main focus of this report is on setting up the formal model of a computee and identifying all its main components. The emphasis is on the development of such a model that would be able to support the functionalities required and that would be computationally realisable within current and/or extensions of computational logic frameworks (see WP3, WP4). The model of computees should be amenable to an abstract design and a well defined abstract architecture. This model also aims at paving the way to a formal analysis of the properties of the computee in particular with respect to its evolution (see WP5).

In summary, the main innovative features of our KGP model are the following:

- the KGP model allows for a very versatile knowledge base for a computee: within the same formalism of CL, the KGP knowledge base allows for multiple modules to deal with planning, reactivity, recording information, representing individual and social policies and decision making under these, representing and reasoning with communication protocols and policies, representing and reasoning with reactive, condition-action rules, temporal reasoning and temporal projection and prediction;
- the KGP model deals with different kinds of actions by the computee uniformly, including communicative actions, sensing actions, physical actions;
- the KGP model relies upon state transitions which are independent of any cycle of operation, and can be described, understood, verified, and have properties proven in a formal way;
- the KGP model has no fixed cycle, but rather relies upon a representation and reasoning framework to describe many different profiles of behaviour by means of cycle theories. Within any given cycle theory, the operation of a computee is dynamically sensitive to the particular circumstances at the time of operation;

- as well as the rest of the model, the cycle theory is also CL-based, and therefore verifiable with respect to formally specified properties. In particular, it allows to link features of the environment and behaviour of the computees, and thus prove properties conditionally on different types of environments;
- the KGP model readily facilitates dynamic interleaving of goal decision, (partial) planning, reacting, communicating and acting, together with plan and goal revision;
- the KGP model supports heterogeneity in many different respects due to its highly modular description of the components of the knowledge base of a computee, of the capabilities, of the transitions and of the cycle theory;
- all components, including the cycle theories, have declarative definitions independent of any operational concerns, but each come equipped with concrete computational counterparts and well–defined proof procedures.

With these features the KGP model is able to define computees as autonomous entities with diverse and versatile functionalities and operational behaviour meeting fully the following high-level GC objectives, as interpreted by our project:

- computees exhibit autonomous and adaptable behaviour in the open and changing GC environment;
- the KGP model supports different behaviours of computees, and allows the presence of behaviourally heterogeneous computees within societies;
- the KGP model allows for the interaction and operation of computees within societies.

The rest of this report is organised as follows.

In section 2 we describe at a high, abstract and intuitive level the technical aspects of the approach we take with the KGP model, and how different components within the model glue together.

In section 3 we give the necessary background for the various components of the framework. This includes background on CL, abductive logic programming, constraint logic programming and preference reasoning within logic programming with rule priorities.

In section 4 we give some preliminary definitions, useful for defining the state of a computee, which is given in section 5.

In section 6 we define the various reasoning and sensing capabilities of a computee within the KGP model.

In section 7 we define the possible state transitions of a computee within the KGP model.

In section 8 we define selection functions, which are used to select inputs to be given to transitions when applied within cycle theories.

In section 9 we define the concept of cycle theory and provide examples of a number of such theories corresponding to desired profiles of operation.

In section 10 we describe the features of the KGP model facilitating the functioning of a computee in a society.

In section 11 we describe some possible, useful extensions of the core KGP model.

In section 12 we compare our model with models proposed in the agent literature.

In section 13, we evaluate the progress of our work with respect to the objectives of WP1, according to the criteria set in deliverable D3. In this section, we also describe how other

criteria set in D3 were achieved by other work done by the Consortium, which is not part of D4. We also identify criteria for WP1 that have not been fulfilled by the Consortium so far.

Section 14 concludes.

2 General approach

The formal model of a computee described in this report is based on:

- an *internal (or mental) state* of the computee,
- a set of *reasoning capabilities* of the computee, for performing *planning, temporal reasoning, identification of preconditions of actions, reactivity* and *goal decision*,
- a *sensing capability* of the computee,
- a set of formal *transition rules* for the state of the computee defined in terms of the above capabilities,
- a set of *selection functions*, to provide appropriate inputs to the transitions,
- a set of execution *cycles* for the combination of the transitions and the selection functions, as provided by the *cycle theory* of the computee.

A snapshot of the model is given in figure 1. In this section, we overview all the components of the model and their integration within the model, as well as their role with respect to the general GC objectives as interpreted by SOCS. The overview in this section is non-technical. The technical details will be given in the remainder of this document.

We represent the **internal (or mental) state** of a computee as a triple $\langle KB, Goals, Plan\rangle$, whose components are as follows.

- **KB** is the knowledge base of the computee, and describes what the computee knows (or believes) of itself and the environment. KB consists of separate modules, supporting different reasoning capabilities. These modules are KB_{plan}, for Planning and for the Identification of Preconditions of actions, KB_{TR}, for Temporal Reasoning, KB_{GD}, for Goal Decision, KB_{react}, for Reactivity, and KB_0, holding the (dynamic) knowledge of the computee about the external world. By means of KB_0 the computee is able to record and reason about the current state of the external environment in which it is situated and also about the past and future states of this environment. We will assume that this environment includes communication messages received from other computees. KB_0 changes via changes in the environment and is typically updated by the computee when it observes the environment through its sensing capability. In the sequel, for convenience we will often assume that KB_0 is contained in all other modules in KB.

  Syntactically, KB_{plan} and KB_{TR} are abductive logic programs with constraint predicates (and $KB_{plan} \subset KB_{TR}$), KB_{GD} is a logic program with priorities over rules, KB_{react} is a set of integrity constraints in abductive logic programming, KB_0 is a set of facts (definite clauses) in logic programming.

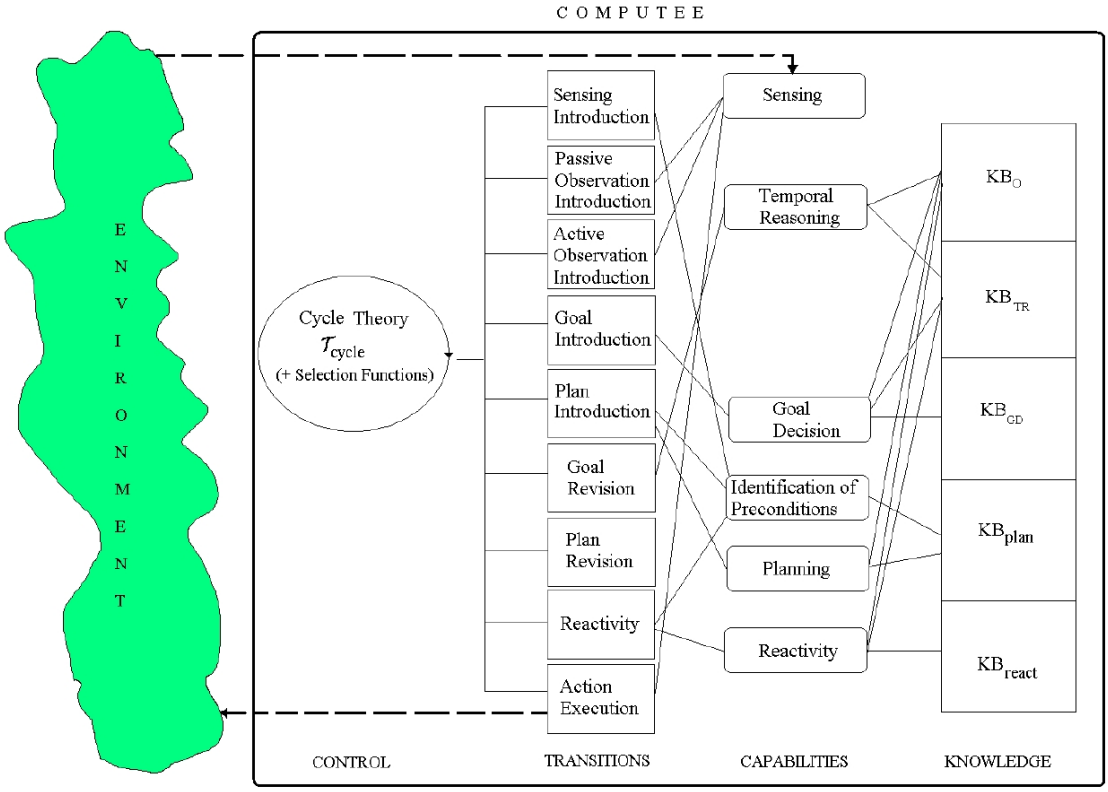

Figure 1: The Computee Reference Model

- **Goals** is a set of properties that the computee has decided that it wants to achieve (desires). The computee intends to achieve all such properties. Each such property is equipped with a time parameter, at which it is expected to hold, possibly constrained via temporal constraints. *Goals* are split into two types:

  mental goals, that can be observed (not) to hold via the *Sensing* capability as well as reduced to subgoals and actions by using KB via the *Planning* reasoning capability.

  sensing goals, that can only be dealt with by setting up sensing actions to find out from the environment whether they hold or not, via the *Sensing* capability.

  Goals are implicitly kept in a hierarchical structure, following a goal-subgoal relation. This structure is in the form of a tree. This structure is given simply by specifying, for each goal, its parent in the structure. The root of the tree is represented by $\perp$. All children of the root, referred to as the *top-level goals*, are either assigned to the computee by its designer at "birth", or they are determined by the *Goal Decision* reasoning capability.

The tree structure is generated via the *Planning* reasoning capability and augmented via the *Reactivity* reasoning capability.

- **Plan** is a set of "concrete" actions of the computee by means of which it plans (intends) to satisfy its goals, and that the computee can execute in the right circumstances.

  Each action is equipped with a time parameter, at which it is expected to be executed, possibly constrained via temporal constraints. The temporal constraints on actions give a partial order on the actions in *Plan*. Thus, *Plan* is at all effects a (partially ordered) sequence of actions.

  Each action in *Plan* is also equipped with the preconditions for its successful execution. These preconditions might be checked before the actions are executed. Preconditions are associated to actions via the reasoning capability for the *Identification of Preconditions*.

  Actions in *Plan* are split into the following types:

  physical actions, that the computee can execute to effect a change in the world,

  communicative actions , e.g. used by the computee to attempt to change the knowledge of other computees or to gain information from them, and

  sensing actions, with which the computee is able to get information from its external environment.

  Plan can contain any combination of types of actions.

  Actions in *Plan* are implicitly kept in the tree structure for *Goals*, again by specifying, for each action, its parent goal in the structure. However, note that actions are necessarily leaves of this tree, as they cannot have children. Actions are added to the tree structure via the *Planning* reasoning capability and via the *Reactivity* reasoning capability.

Overall, the **reasoning capabilities** employed within the KGP model are the following.

- **Planning**, to generate partial plans for sets of goals; each such plan consists of a set of sub-goals and a set of actions for each given input goal. The sub-goals and the actions in a partial plan for a goal are equipped with (possibly empty) temporal constraints on the times of actions and goals. A partial plan will only be generated by this capability if it is "globally consistent" with KB and the existing *Goals* and *Plan*.

  This capability contributes to rendering computees *autonomous* (capable to decide autonomously how to achieve their goals). The generation of partial (rather than total) plans paves the way to the computee being *adaptable* to changes in the environment without being wasteful of earlier planning efforts.

- **Reactivity**, to appropriately react to perceived changes in the environment, by adding goals to *Goals* and actions to *Plan*, e.g. as required to fulfil given "condition-action" rules in KB. The new goals and actions are equipped with (possibly empty) temporal constraints on the times of actions and goals, and with their parent in the new *Goals*. A set of goals and actions will only be generated by this capability if it is "globally consistent" with KB and the existing *Goals* and *Plan*.

  This capability contributes to render computees *adaptable* to changes in the environment.

- **Goal decision**, to continuolsy revise the top-most level goals of the computee, taking into account possible changes in the circumstances of the computee and its environment that affect the computee's preferences as to what it should aim at achieving. This capability differs form the earlier Reactivity in that it only modifies the top-level goals of a computee and it does not add actions to *Plan*. Moreover, differently from Reactivity, goal decision does not depend upon the *Goals* and *Plan* of the computee.

  This capability contributes to render computees *autonomous* (by deciding to modify its set of goals, and even dropping goals assigned to it by its designer, if any) as well as *adaptable* to changes in the environment.

- **Identification of Preconditions**, to identify the preconditions for the successful execution of given actions. The preconditions are properties that "cautious" computees might want to sense within the environment to establish whether the actions can be executed successfully. Thus, the presence of preconditions in actions, as identified by this capability, paves the way to modelling *heterogeneous* computees.

  The preconditions can also be reasoned upon to establish that some actions will never be successfully executable, as their preconditions will never be satisfied, and can be dropped from *Plan*. Thus, the presence of preconditions identified by this capability paves the way to the *adaptability* of computees.

- **Temporal Reasoning**, to reason about and from observations sensed within the environment and make predictions about properties holding in the world on the basis of these observations. This capability is used to define most of the other capabilities, and thus has a special role within the model; overall, it plays a fundamental role in rendering the computee *adaptable* to changes in its environment by dealing appropriately with *partial information* which evolves over time.

In this document we give concrete specifications of these reasoning capabilities, that we want to use for the development of the computational counterpart of the model as well as for its prototype implementation. However, note that the KGP model is in principle independent of such concrete choices and different such choices could have been made.

Note that other reasoning capabilities might be useful, for example to reason about preferred plans. Some of these additional capabilities are discussed as possible extensions of the KGP model later on in this document. Finally, note that computees do not necessarily need to have all capabilities listed here, and instead they could have any subset of these.

In addition to the reasoning capabilities above, the computee is equipped with a **sensing capability** which links the computee to its environment, by allowing it to observe properties holding in the environment and actions being executed by other computees. The KGP model assume the existence of such capability but does not model it. The concrete realisation of such a capability will be important for the implementation prototype to be delivered by WP4.

The state of a computee evolves by applying **transition rules**, which employ capabilities, as follows.

- **Goal Introduction**, which changes the top-level *Goals* of a state, and uses the *Goal Decision* capability.

- **Plan Introduction**, which changes *Goals* and *Plan* of a state, and uses the *Planning* and *Introduction of Preconditions* capabilities.

- **Reactivity**, which changes *Goals* and *Plan* of a state, and uses the *Reactivity* and *Introduction of Preconditions* capabilities.
- **Sensing Introduction**, which changes the *Plan* of a state by introducing new sensing actions for sensing the preconditions of actions already in *Plan*, and uses the *Sensing* capability.
- **Passive Observation Introduction**, which changes KB_0 of KB by introducing unsolicited information coming from the environment (a bit like an interrupt), and uses the *Sensing* capability.
- **Active Observation Introduction**, which also changes KB_0 of KB, but by introducing the outcome of sensing actions for properties of interest to the computee, and thus actively seeked, and uses the *Sensing* capability.
- **Action Execution**, for executing all types of actions, thus changing KB_0 of KB.
- **Goal Revision**, which revises *Goals*, e.g. by dropping goals that have already been achieved, by using the *Temporal Reasoning* capability.
- **Plan Revision**, for revising *Plan*, e.g. by dropping actions that have already been executed successfully.

We believe that this set of transitions is a suitable set of state transitions to accommodate the features required from computees to face the challenges of the GC vision. In order to be *autonomous*, computees need to be able to plan for ways to achieve their goals (Plan Introduction) and for introducing new goals (and possibly dropping existing goals) as they evolve (Goal Introduction). In order to be aware of changes in the environment, and thus being *adaptable*, computees need to be able to observe the environment, both proactively (Active Observation Introduction), after having decided what to observe (Sensing Introduction), and because of the environment (and computees in it) demanding attention (Passive Observation Introduction). In order to be *adaptable*, they need to be able to react to observed changes in the environment (Reactivity) and they need to be able to revise their goals and plans as their situation changes (Goal and Plan Revision). Communicative behaviour is accommodated within the framework by means of several of its features, in particular by means of Action Execution, as some of the actions in *Plan* are communicative, and Reactivity, to respond appropriately to incoming communication.

We also envisage, but do not concentrate upon, a **Knowledge Revision** transition to be developed as a possible extension of the model, to modify the knowledge of the computee, besides KB_0. This transition could make use of learning techniques as well in order to enlarge the knowledge base of the computee.

Finally, note that computees might be equipped with just a subset of the set of transitions above.

We adopt the distinction amongst capabilities and transitions for the following (software engineering-oriented) reasons. We encapsulate within a reasoning capability any reasoning process for which we could consider different specifications than the ones we provide within this document. We encapsulate within the sensing capability the physical environment of a computee, as this is not under the computee control and cannot be defined within the model. We encapsulate within transitions any process that affects the state of a computee. Moreover, the use of capabilities aims at increasing readability of the transitions.

The behaviour of a computee is then given by the application of transitions in sequences, thus producing progressive changes over the state of the computee. In the KGP model, these sequences are not determined by fixed cycles of behaviour, as in conventional agent architectures, but rather by reasoning with **cycle theories**. These are logic programs with priorities over rules, defining preference policies over the order of application of transitions. These policies are sensitive to changes in the environment and in the internal state of a computee, and provide a mean of declarative and intelligent control for computees. By adopting different cycle theories we aim at obtaining (behaviourally) *heterogeneous* computees.

The provision of declarative control for computees in the form of cycle theories is a highly novel feature of computees with respect to intelligent agents systems presented in the literature, as discussed later on in the paper when providing a comparison with related work. Note that the notion of cycle theory and its use to determine the behaviour of computees could in principle be imported into any agent system, to replace conventional fixed cycles.

Computees are agents whose diverse components are all expressed uniformly within computational logic: both the state of computees, their reasoning capabilities and their control (cycle) theories are formulated and realised within computational logic. We envisage that this will ease the task of formulating and verifying properties of computees in later stages of the project.

3 Background

In this section we give essential background on the techniques we rely upon in the remainder of the document to define the data structures underlying the model of computees, the capabilities which form the building blocks of the model, and behaviour cycle of computees. In particular, we give background on logic programming (section 3.1) (which is the form of computational logic we adopt for developing the model of individual computees) and some of its extensions, namely abductive logic programming (section 3.2), a logic programming framework extended to perform hypothetical reasoning (also called abduction), and logic programming extended to deal with preferences over rules (section 3.3), with particular emphasis on $LPwNF$, a concrete form of such extended logic programming. In section 3.4, we overview how to handle constraint predicates (such as $\leq, =$, etc) within logic programming. Finally, in section 3.5, we indicate how to incorporate constraint predicates within abductive logic programming and $LPwNF$.

The reader familiar with these notions can skip this section, except for the conventions here introduced, as they will play an important role in the remainder of the document.

3.1 Logic programming

A *logic program* [Kow79, Llo87, Apt90] is a set of rules of the form

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

with A atom, $L_1, \ldots, L_n$ (positive or negative) literals, and $n \geq 0$. Each negative literal L_i is of the form $not\, B_i$, where B_i is an atom. The negation symbol *not* indicates *negation as failure* [Cla78, Llo87, Apt90]. All variables in A, L_i are implicitly universally quantified, with scope the entire rule. A is called the *head* and $L_1 \wedge \ldots \wedge L_n$ is called the *body* or the *conditions* of a rule of the form above. If $A = p(t)$, for some vector of terms t, the rule is said to *define* p.

A *definite logic program* is a set of *definite clauses*, namely rules without any occurrence of negation as failure, of the form

$$A \leftarrow A_1, \ldots, A_n$$

with A, $A_1, \ldots, A_n$ atoms, and $n \geq 0$.

A *fact* is a definite clause of the form above but with $n = 0$.

Given a definite logic program, the meaning of the program is given by its *least Herbrand model*. This is unique, for any given definite logic program, and thus can be chosen as the undisputed semantics of the program. Given an ordinary logic program, with negation as failure occurring in the body of rules in the program, in general there are many different semantics that can be adopted for that program. These include the *completion semantics* [Cla78] and its three-valued variant [K.87] *stable models* [GL88], *partial stable models* [SZ90] and *preferred extensions* [Dun91], *stationary expansions* [Prz91] and *complete scenaria* [Dun91], *well-founded model* [GRS88], *stable theories* [KM91] and *acceptability semantics* [KMD94b] (and semantics equivalent to these). Most of these semantics coincide for special classes of logic programs, e.g. stratified logic programs [ABW78]. Most of these semantics can be expressed uniformly within an argumentation framework [Dun95, BDKT97, KT99].

In the sequel, unless otherwise specified, we will not commit to any specific semantics for the logic programs we will encounter. The definitions we present in this report are parametric with respect to the adopted semantics, and, unless otherwise specified, any of the semantics above will be suitable for our framework. We foresee that, in the second phase of the project, when specifying the computational counterpart for the model presented in this document, we will have to adopt some concrete semantics for the logic programs in the model, and possibly use different semantics for different components of the framework.

In the sequel we will adopt the following conventions:

Convention 3.1 Given a logic program P, $P \models_{LP} C$ stands for (the implicitly existentially quantified conjunction of literals) C is true in the (chosen) semantics of P.

Convention 3.2 Given a definite logic program P, $P \models_H C$ stands for (the implicitly existentially quantified conjunction of atoms) C is true in the least Herbrand model of P.

Many of the reformulations of semantics of logic programs in argumentation terms, e.g. [BDKT97, KT99], rely upon $\models_H$, in that every logic program with negation can be understood as a definite logic program by interpreting the negative literals in it as new atoms, as first presented in [EK89]. The negative literals can be seen as hypotheses (or abducibles) that can be added to the logic program reinterpreted as definite, if some argumentation-based criteria are satisfied. Additional material about argumentation for logic programming, and its extensions, will be discussed in section 3.3 below.

Some of these semantics mentioned above are *sceptical* (e.g. the well-founded model semantics) whereas others may be used in a sceptical or in a *credulous* manner (e.g. the stable model semantics). Credulous versions of semantics allow to represent non-determinism.

3.2 Abductive logic programming

Abductive logic programming [KKT93, KKT98, KD02] is a powerful knowledge representation framework, that can be used to realize many diverse applications such as diagnosis, planning, natural language understanding, default reasoning and image processing as well as, more recently, active databases and intelligent agents [KS96b, KST98, KS99, ST99, ST00, STT02a, KM02].

An *abductive logic program* $\langle P, A, I \rangle$ consists of

- a logic program, P, seen as an "incomplete" theory,
- a set, A, of *abducible predicates*, whose (variable-free) *abducible atoms* are used to expand the theory,
- a set, I, of first-order sentences, referred to as the *integrity constraints*, that must be "satisfied" by any sets of abducibles expanding the theory.

The concept of integrity constraint first arose in the field of databases and to a lesser extent in the field of AI knowledge representation. The basic idea is that only certain knowledge base states are considered acceptable, and an integrity constraint is meant to enforce these legal states.

Given a goal (conjunction of literals) G, a set of abducible atoms $\mathcal{D}$, and a variable substitution θ for the variables in G, a pair $(\mathcal{D}, \theta)$ is an *abductive answer* for G, with respect to an abductive logic program $\langle P, A, I \rangle$, iff

1. $P \cup \mathcal{D}$ "entails" $G\theta$, and
2. $P \cup \mathcal{D}$ "satisfies" I.

Fulfilment of other properties is sometimes required, e.g. minimality, with respect to set inclusion, of the set of abducibles in the answer [KKT98].

Various notions of entailment and satisfaction can be adopted. As far as entailment is concerned, every semantics for the logic program P can provide one such entailment. Namely, part 1 of the definition of abductive answer above can be replaced by

1. $P \cup \mathcal{D} \models_{LP} G\theta$.

As far as satisfaction is concerned, there are several ways to define what it means for a knowledge base KB ($P \cup \mathcal{D}$ in our case) to satisfy an integrity constraint ϕ (in our framework $\phi \in I$). The *consistency view* requires that:

$$KB \text{ satisfies } \phi \text{ iff } KB \cup \phi \text{ is consistent.}$$

Alternatively the *theoremhood view* requires that:

$$KB \text{ satisfies } \phi \text{ iff } KB \models \phi.$$

Another view regards the integrity constraints as *epistemic* or *meta-level* statements about the content of the knowledge base. They specify what must be true about the knowledge base rather than what is true about the world modelled by the knowledge base. In conventional work on abductive logic programming, starting from the work of [EK89, KM90a, KM90b], the view adopted is stronger than consistency, weaker than theoremhood, and arguably similar to the epistemic or meta-level view. The view can be expressed by the following reformulation of part 2 of the definition of abductive answer above:

2. $P \cup \mathcal{D} \models_{LP} I$.

In this document we will consider integrity constraints of the following forms (L_i positive or negative literal, A atom, $false$ a special atom in the language of the abductive logic program):

$$L_1, \ldots, L_n \Rightarrow false \quad (n > 0)$$

conventionally called *denials*,

$$L_1 \vee \ldots \vee L_n \quad (n > 0)$$

conventionally called *disjunctive integrity constraints*, and

$$L_1, \ldots, L_n \Rightarrow A \quad (n > 0)$$

that we call *implicative integrity constraints*. In such integrity constraints, $L_1, \ldots, L_n$ is referred to as the *body* and A is referred to as the *head* of the constraint. All variables in the integrity constraints are implicitly universally quantified from the outside (as for rules), except for variables occurring only in the head which are implicitly existentially quantified with scope the head.

Often, in this document, we will allow for implicative integrity constraints to be expressed in "looser" forms. Notably, we will allow integrity constraints of the form

$$L_1, \ldots, L_n \Rightarrow K_1 \wedge \ldots \wedge K_m \quad (n > 0, m > 0)$$

with K_j atoms. Note that any abductive logic program $\langle P, A, I \rangle$ with an implicative integrity constraint

$$L_1, \ldots, L_n \Rightarrow K_1 \wedge \ldots \wedge K_m$$

of the looser form is equivalent to an abductive logic program $\langle P', A, I' \rangle$ with implicative integrity constraints of the stricter form, where $P' = P \cup \{p(X) \leftarrow K_1 \wedge \ldots \wedge K_m\}$ and $I' = I \setminus \{L_1, \ldots, L_n \Rightarrow K_1 \wedge \ldots \wedge K_m\} \cup \{L_1, \ldots, L_n \Rightarrow p(X)\}$, where p is a new predicate, not already occurring in $\langle P, A, I \rangle$, and X is the vector of all variables occurring in $L_1, \ldots, L_n$.

The notion of abductive answer given earlier relies upon "grounding" all variables in the goal G and extending the theory P by means of *ground* abducible atoms. In this document, we will instead rely upon non-ground abductive answers, whose variables may be constrained by means of conjunctions of literals in an underlying language of (arithmetical) constraints as in constraint logic programming, in the spirit of [KM95, KTW98, AKM00, KvD01] (see section 3.4).

3.3 Logic programming with priorities

$LPwNF$ [KMD94a, DK95, KM03b] is a logic programming framework for preference reasoning with an argumentation-based semantics. Within this framework we have a powerful form of decision making under a given preference policy. This policy can be sensitive to different information in a dynamic environment allowing the decision making to adapt to the particular circumstances at the time of the decision.

The preference reasoning within $LPwNF$ is based on a model of argumentation where local priority information, given at the level of the rules of a theory (or policy), is lifted to give a global preference over sets of rules that compose arguments and counter arguments for a certain decision. A theory or policy within $LPwNF$ is viewed as a pool of sentences or rules from which we need to select a suitable subset, i.e. an argument, in order to support a conclusion.

In $LPwNF$ knowledge is represented in a classical background logic $(\mathcal{L}, \models_H)$ by means of rules of the form

$$L \leftarrow L_1, \ldots, L_n, \quad (n \geq 0)$$

where $L, L_1, \ldots, L_n$ are positive or negative (classical) literals. A negative literal is a literal of the form $\neg A$, where A is an atom. Note that the symbol $\neg$ stands for classical negation, and is therefore distinguished from the symbol *not* used to represent negation as failure above. Indeed, no negation as failure occurs in an $LPwNF$ theory [1]. Note also that, differently from rules in conventional logic programs as given above, here the head L of rules may be negative as in the framework of Extended Logic Programming.

The background derivability $\models_H$ relation of the framework is the monotonic Horn logic given by the single inference rule of modus ponens, treating negative literals as ordinary atoms. In general, we can separate out an *auxiliary part* of a given theory from which the other rules can draw background information in order to satisfy some of its conditions. The reasoning of the auxiliary part of a theory is independent of the main argumentation-based preference reasoning of the framework and hence any appropriate logic can be used.

Some of the sentences in a $LPwNF$ theory express priorities over the rules of the theory. These have the same form as the rules above except that their head, L, refers to a *higher-priority relation,* h_p, and so such a rule has the general form

$$L = h\_p(rule1, rule2) \leftarrow L_1, \ldots, L_n, \quad (n \geq 0)$$

where $rule1$ and $rule2$ are the names of two rules in theory.

A rule of this form then means that under the conditions $L_1, \ldots, L_n$, $rule1$ has priority over $rule2$. The role of this priority relation is therefore to encode locally the relative strength of (argument) rules in the theory. The priority relation given by h_p is required to be irreflexive. The rules $rule1$ and $rule2$ can in fact be themselves rules expressing priority between other rules and hence the framework allows higher-order priorities. For simplicity of presentation we will assume that the conditions of any rule in the theory do not refer to the predicate h_p thus avoiding self-reference problems.

The preference reasoning of $LPwNF$ uses the priority relation between rules to find out conclusions that are preferred over their conflicting conclusions. Indeed, an $LPwNF$ theory might give rise, under its background derivability $\models_H$, to *conflicts*, namely between a literal L and its negation $\neg L$. More generally, we can define other forms of conflict within a given theory through an auxiliary predicate and rules of the form

$$incompatible(L_1, L_2) \leftarrow B,$$

stating that literals L_1 and L_2 are conflicting under some (auxiliary) conditions B (see sections 6.5 and 9.2 for examples of this). Also for any ground atom $h\_p(rule1, rule2)$, its conflicting literal is defined to be $h\_p(rule2, rule1)$ and vice-versa.

Conflicts together with the priority relation of the theory give rise to a notion of *attack* between sets of sentences in the theory $\mathcal{T}$. Δ attacks Δ', where Δ, Δ' are sets of sentences in the $\mathcal{T}$, means that these two sets have conflicting conclusions (under $\models_H$) and that the rules of Δ that derive this are rendered by Δ to have at least the same priority as that rendered

[1] This is where the framework gets its name: Logic Programming without Negation as Failure but the historical reasons for this name are not important in this document.

by Δ' for its own rules that derive the conflicting conclusion. For the case where the priority relation is static, i.e the rules for h_p have no conditions and so express globally that one rule is always stronger than another, this definition is given as follows (see [KM03b] for the formal details when h_p is not static).

Let $\mathcal{T}$ be an $LPwNF$ theory and $\Delta, \Delta' \subseteq \mathcal{T}$. Then Δ' *attacks* Δ (or Δ' is a *counter argument* of Δ) iff there exists L, $\Delta_1 \subseteq \Delta'$ and $\Delta_2 \subseteq \Delta$ such that:

(i) $\Delta_1 \vdash_{min} L$ and $\Delta_2 \vdash_{min} \overline{L}$

(ii) $(\exists r' \in \Delta_1, r \in \Delta_2 \text{ s.t. } h\_p(r, r')) \Rightarrow (\exists r' \in \Delta_1, r \in \Delta_2 \text{ s.t. } h\_p(r', r))$,

where $\overline{L}$ is any literal that conflicts L (e.g. $\overline{L} = \neg L$ or $incompatible(L, \overline{L})$ holds).

Here $\Delta \vdash_{min} L$ means that $\Delta \models_H L$ and that L cannot be derived from any proper subset of Δ. The second condition in this definition states that an argument Δ' for L attacks an argument Δ for the contrary conclusion only if the set of rules that it uses to prove L are at least of the same strength (under the priority relation h_p) as the set of rules in Δ used to prove the contrary. Note that the attacking relation is not symmetric.

Given this notion of attack that lifts the priority relation from individual rules to sets of rules, we define the *admissible* subsets or arguments of a given theory as follows.

Let $\mathcal{T}$ be a theory and $\Delta \subseteq \mathcal{T}$. Then Δ is an *admissible* argument iff Δ is consistent (i.e. conflict free) and for any $\Delta' \subseteq \mathcal{T}$ if Δ' attacks Δ then Δ attacks Δ'.

Then, preference reasoning is based on the maximal admissible arguments of a given theory. Usually, two entailment relations are defined:

Convention 3.3 Given an $LPwNF$ theory $\mathcal{T}$,

- $\mathcal{T} \models_{pr}^{cred} G$ means that there is at least one maximal admissible subset of $\mathcal{T}$ where G holds;
- $\mathcal{T} \models_{pr}^{scep} G$ means $\mathcal{T} \models_{pr}^{cred} G$ and, for any $\overline{G}$ such that $incompatible(G, \overline{G})$ holds, $\mathcal{T} \not\models_{pr}^{cred} \neg G$.

In the sequel, $\models_{pr}$ will indicate $\models_{pr}^{scep}$.

Additional details on $\models_{pr}$ can be found in [KMD94a, DK95, KM03b].

3.4 Constraint predicates in computational logic

Constraint Logic Programming (CLP) [JM94] extends logic programming with *constraints predicates* which are not processed as ordinary logic programming predicates, defined by rules, but are checked for satisfiability and simplified by means of a built-in, "black-box" constraint solver. These are typically used to constrain, together with unification which is also treated via an equality constraint predicate, the values that a conclusion of a rule can take. For example, constraints can be used to compute the value of time variables, in the rules of a program, under a suitable temporal constraint theory.

In CLP, constraints are build as first-order formulae in the usual way from primitive constraints of the form $c(t_1, \ldots, t_n)$ where c is a constraint predicate symbol and $t_1, \ldots, t_n$ are terms constructed over a give domain, $\mathrm{D}(\Re)$, of values. Then the rules of a constraint logic program P take the same form as rules in conventional logic programming given by

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

where C is a set of constraints.

A *valuation* θ of a set of variables is a mapping from these variables to the domain D($\Re$) and the natural extension which maps terms to D($\Re$). A valuation θ, on the set of all variables appearing in a set of constraints C, is called an $\Re$-solution of C iff C_θ, obtained by applying θ to C, is satisfied i.e. C_θ evaluates to true under the given interpretation of the constraint predicates and terms. The set C is called $\Re$-*solvable or simply satisfiable* iff it has at least one $\Re$-solution.

One way to given the meaning of a constraint logic program P is to consider the grounding of the program over its Herbrand base and all possible valuations, over D($\Re$), of its constraint variables. In each such rule if the ground constraints C are evaluated to true then the rule is kept with the condition of C dropped otherwise the whole rule is dropped. The meaning of P is then given by the meaning of the resulting logic program as described in section 3.1.

3.5 ALP and LPwNF with Constraints

The frameworks of abductive logic programming (ALP) and logic programming with priorities (as in $LPwNF$) described above can be usefully extended with constraints in the same way that LP extends logic programming. In $LPwNF$ the extension is exactly analogous to that of CLP simply allowing constraints in the body of the rules and hence constraining the conclusions that these rules can have.

In ALP together with this extension of constraining the conclusions of the rules we can use constraints to constrain abducible assumptions. In fact the link with constraints allows us to extend an ALP framework to include non-ground abducible hypotheses, an extension that increases significantly the versatility of applying abductive reasoning to various problems and that plays an important role within the model we propose in this document.

One such framework of integration of ALP and CLP is that of $ACLP$ [AKM00, KMM00] and the A-system [KvD01] that has followed it. In this framework abductive theories are defined as usual but now over the combined language of a given underlying framework of D($\Re$) and a user given language for the problem domain that is represented by the abductive logic program. The essential extension is that now the set of abducible atoms in the predicates of A is extended to consist of the following formulae:

1. $a(d)$, where a is an abducible predicate and d is a vector of constants in D$\Re$ (such abducibles are called ground abducibles)

2. $\exists X(a_1(X), .., a_n(X), C(X))$, where $n \geq 1$, a_i is an abducible predicate $\forall i. 1 \leq i \leq n$ and $C(X)$ is a (possibly empty) set of constraints.

The subset $\mathcal{D}$ of abducibles of an abductive answer can now include also non-ground abducibles, e.g. $\mathcal{D} = \exists X(a(X), C(X))$. The notion of an *abductive answer* is build on the previous definition for (ground) abduction generalising this by considering the different possible groundings of non-ground elements of $\mathcal{D}$ allowed by the constraints C that they involve. Given a theory and a goal G then a set $\mathcal{D}$ of abducibles is an abductive answer for G iff:

1. there exists a valuation (or grounding) g such that, for each abducible formula ϕ in $\mathcal{D}$, the constraints in ϕ are satisfied, and

2. for any grounding $\mathcal{D}_g$ of $\mathcal{D}$, $\mathcal{D}_g$ is a ground abductive explanation of G.

Hence $\mathcal{D}$ is an abductive solution for a goal G iff there exists (in $\Re$) at least one consistent grounding of $\mathcal{D}$ and for any such grounding, $\mathcal{D}_g$, this constitutes a ground abductive solution of G. Note that essentially this grounding θ gives also the associated variable substitution θ for the variables in the goals G, since in CLP the implicit unification of logic programming is replaced by an explicit equality theory that is always part of the constraint theory of the CLP framework.

Operationally, an abductive answer can be computed, in CLP fashion, by interleaving the generation (and consistency checking with respect to the integrity constraints) of abducible hypotheses together with the generation of an associated constraint store C of constraints on these hypotheses and its own check for satisfiability. This constraint store can grow during the computation provided that it remains satisfiable. A "black box" constraint solver, that is transparent to the abductive reasoning itself, is used to decide on the satisfiability of this store when necessary and to reduce the constraints accordingly. The satisfiability or not of C in turn affects back the computation of abductive hypotheses and their check of satisfying the integrity constraints.

An alternative approach to incorporating constraint predicates into ALP and (potentially) $LPwNF$ is that of [KTW98]. Here, constraint predicates are dealt with (semantically as well as procedurally) just like any abducible predicate, possibly constrained by integrity constraints that serve as a glass-box for their transparent treatment. This is achieved by ordinary abductive reasoning. In this framework, abductive answers do not rely upon grounding, and are instead of the form 2. above for ACLP.

Finally, we note that in both ALP and $LPwNF$ it is useful to extend the form of rules and integrity constraints to allow the heads of certain (top-level) rules or of certain implicative integrity constraints to contain constraints that limit the values of existentially quantified variables over the head. This extension will be described below in the document in the sections where it will be used.

4 Preliminaries of the KGP model

In the sequel we assume (possibly infinite) vocabularies of *time constants* (e.g., the set of all natural numbers), *time variables* (that we will indicate with $t, t', s, \ldots$), *fluents* (that we will indicate with $g, g', \ldots$), *action operators* (that we will indicate with $a, a', \ldots$), and *names of computees* that we will indicate with c, c', ... Given a fluent g, g and $\neg g$ are referred to as *fluent literals*. [2]

We assume that the set of fluents is partitioned in two disjoint sets: *mental fluents* and *sensing fluents*. Intuitively, mental fluents represents properties such that the computee itself is able to plan 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 computee 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 (given) resources are accepted and that some (given) connection is active.

[2] Note that $\neg$ represents classical negation. Negation as failure occurs in the model only within (some parts of) KB. All negations in *Goals* and *Plan* are classical negations.

We also assume that the set of action operators is partitioned in three disjoint sets: *sensing action operators*, *physical action operators* and *communication action operators*. In the sequel, we will refer to physical and communication action operators jointly as *non-sensing action operators*. Intuitively, sensing actions represent actions that the computee 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 computee may need to check in the environment. On the other hand, *physical* actions are actions that the computee 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 computees. 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 computee c_1 is requesting computee c_2 the resource r_1 within a dialogue d, at time t (see Section 10.1.1).

Temporal constraints

In the sequel we will use temporal constraints associated with goals and actions of a computee. Temporal constraints are formulae defined by the following syntax:

TC	::=	AtomicTC | TC $\wedge$ TC | TC $\vee$ TC | $\neg$ TC
AtomicTC	::=	Variable Relop Term
Relop	::=	$= \mid \neq \mid < \mid > \mid \leq \mid \geq$
Term	::=	Time_Constant | Time_Variable | Term Op Term
Op	::=	$+ \mid - \mid * \mid \div$

As mentioned above, time constants are simply natural numbers. Time variables are distinguished variables which can be instantiated to time points. Notice that no explicit quantification is introduced in temporal constraints: indeed, in a temporal constraint occurring in goals and actions (see below) all variables are implicitly existentially quantified.

We use the following terminology and notations.

Goals

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

- $l[t]$ is the *fluent literal* of the goal possibly referring to the time t; we will refer to $l[t]$ as a *timed fluent literal*;
- G' is the *parent* of G;
- Tc is a (possibly empty) temporal constraint which typically refers to the time t.

Top-level goals are goals which have no parent. We will denote them by triples of the form $G = \langle l[t], \bot, Tc\rangle$.

As an example, we may have a top-level goal G of the form

$$\langle problem\_fixed(p2, t), \bot, 5 \leq t \leq 10\rangle$$

and a subgoal G' of G of the form

$$\langle get\_resource(r1, t'), G, 5 \leq t' \leq t\rangle$$

meaning that to fix problem $p2$ within a certain time interval, the computee needs to have (or acquire) a resource $r1$.

Given a set of goals $Goals$ and $G \in Goals$ we write

- $parent(G) = G'$ if $G = \langle \_, G', \_ \rangle$ (G' can be $\perp$ if G is a top-level goal) [3]
- $children(G, Goals) = \{G' \in Goals \mid G' = \langle \_, G, \_ \rangle\}$
- $descendants(G, Goals) = children(G, Goals) \cup$
  $\{G' \in Goals | \exists G'' \in descendants(G, Goals).G' \in descendants(G'', Goals)\}$
- $ancestor(G, Goals) = \{G' \in Goals | G \in descendants(G', Goals)\}$.

We will also refer to two goals with the same parent as *siblings*.

In the sequel *mental goals* are goals whose fluent is mental, and *sensing goals* are goals whose fluent is sensing.

Actions

An action A is a 4-tuple of the form $\langle a[t], G, C, Tc \rangle$ where

- $a[t]$ is the *operator* of the action, referring to the execution time t; we will refer to $a[t]$ as a *timed (action) operator*;
- G is the goal towards which the action contributes (i.e., the action belongs to a *plan* for the goal G); G is a post-condition for A (but there may be others such post-conditions, as given within the knowledge base of the computee);
- 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;
- Tc is a (possibly empty) temporal constraints which typically refers to the time t of the action.

Note that we are assuming that actions are atomic and do not have a duration. The temporal constraints specify a time window over which the time of the action can be instantiated, at execution time. If the temporal constraints are empty then the action can be executed at any time.

As an example, we may have an action

$$\langle tell(c_1, c_2, request(r1), d, , t''), G', \{\}, 5 \leq t'' \leq t' \rangle$$

where G' is given as above.

Given and action A, we write

- $parent(A) = G$ if $A = \langle \_, G, \_, \_ \rangle$ (G can be $\perp$).

[3] We use "_" to denote a component which is not relevant in the context, inheriting the Prolog tradition of denoting by "_" an anonymous variable.

We will also refer to two actions with the same parent and to an action and a goal with the same parent as *siblings*. Note that, in practice, actions can be seen as special kinds of goals which are directly *executable*.

In the sequel *sensing actions* are actions whose action operator is a sensing action operator, *non-sensing actions* are actions whose action operator is a non-sensing action operator. We distinguish two types of sensing operators, denoted by $sense(l[t])$ and $sense\_precondition(l[t])$. The latter is a sensing action which a computee may explicitly introduce in its plan in order to check whether or not a precondition of a certain action holds at the current time.

In both a timed fluent literal $l[t]$ and a timed operator $a[t]$, the time t may be a time constant (in which case the associated temporal constraint will be empty) or a time variable. This variable is treated as an existentially quantified variable, the scope of which is the whole *state* of the computee (see Section 5). Whenever a goal (resp. action) is introduced within a state, the time variable associated with the goal (resp. action), is a distinguished, fresh variable. When a time variable is instantiated (e.g., at action execution time) the actual instantiation is recorded in the state of the computee. This allows us to keep different instances of the same action (resp. goal) distinguished.

Finally, note that the temporal constraints of a goal/action might be empty even though the time of the goal/action is a variable.

5 State of a computee

At any given time, the state of a computee is a triple

$$\langle KB, Goals, Plan \rangle$$

where $Goals$ is a set of goals and $Plan$ is a set of actions, as defined in section 4.
The $Goals$ and $Plan$ components of the state can be represented as a tree where:

- the root is $\perp$;
- the nodes of the tree other than the root are labelled by goals in $Goals$ or actions in $Plan$;
- the children of the root are the top-level goals in $Goals$;
- action can label only leaf nodes;
- for each non-leaf node labelled by a goal G, the children of the node are all the actions in $Plan$ and goals in $Goals$ whose parent is G.

As an example, the tree for the following $Goals$ and $Plan$ of computee c_1

$$\begin{aligned} Goals &= \{G, G_1, G_2\}, \text{ where} \\ \\ G &= \langle problem\_fixed(p2, t), \perp, 5 \leq t \leq 10 \rangle \\ G_1 &= \langle get\_resource(r1, t_1), G, 5 \leq t_1 \leq t \rangle \\ G_2 &= \langle get\_resource(r2, t_2), G, 5 \leq t_2 \leq t \rangle \\ \\ Plan &= \{\langle tell(c_1, c_2, request(r1), d, t'), G_1, \{\}, 5 \leq t' \leq t_1 \rangle\} \end{aligned}$$

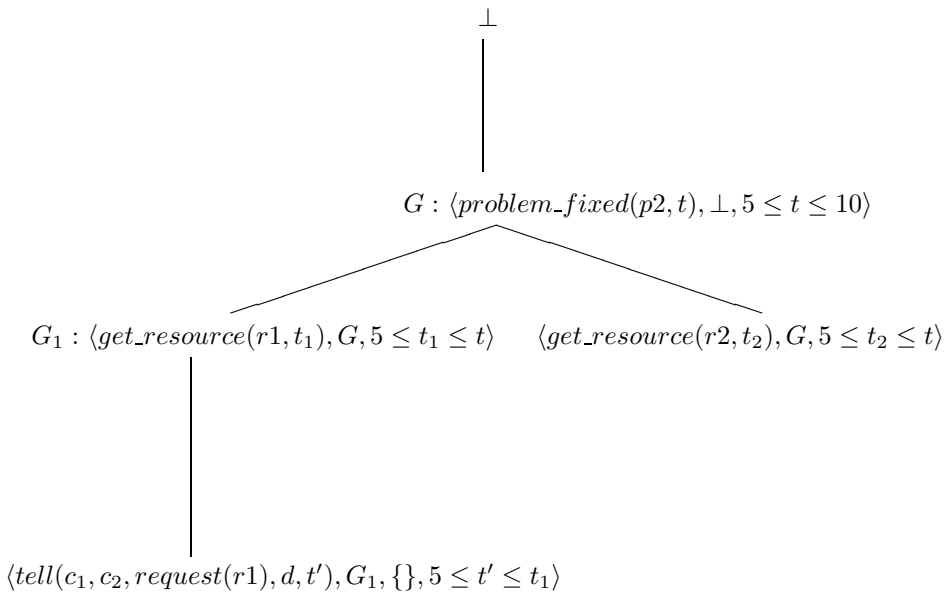

Figure 2: The tree view of the state of computee c_1

is the tree in Figure 2.

It is worth pointing out that all variables occurring in the tree (and in particular the time variables t, t_1, t_2 and t' in the example above) are implicitly existentially quantified with scope the whole tree.

For simplicity, we will assume that, given a state $\langle KB, Goals, Plan\rangle$, all occurrences of variables in $Goals$ and $Plan$ are time variables (implicitly existentially quantified globally within the $Goals$ and $Plan$). 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. We avoid dealing with the other variables to keep the model simple.

The KB component of the state is the union of various (not necessarily disjoint) knowledge bases. Among them we distinguish KB_0 which records the actions which have been executed 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. This fact means that an action a has been executed at time $t = \tau$ by the computee.
>
> $observed(l[t], \tau)$ where $l[t]$ is a timed fluent literal and τ is a time constant. This fact means that the property l has been observed to hold at time $t = \tau$.
>
> $observed(c, a[\tau'], \tau)$ where c is a computee's name, different from the name of the computee whose state we are defining, τ and τ' are time constants, and a is an action operator.

This fact means that the given computee has observed at time τ that computee c has executed the action a at time τ' (of course $\tau' \leq \tau$).

Note that assertions in KB_0 of the third kind are variable-free. These are intended to represent reception of communication from other computees. These type of assertions have no variable as they represent actions executed by other computees, whose (internal) time variables are of no interest to the computee in question.

Instead, assertions of the first two kinds refer explicitly to the time t. We could have represented these assertions simply as (variable-free) facts of the form $executed(a[\tau])$ and $observed(l[\tau])$, without referring explicitly to their time t. We have chosen the above representation with explicit variables in order to link the record in KB_0 of observed properties and execution of actions by the computee in order to link this record to the time of actions in $Plan$ and goals in $Goals$. Thus, the time of an action and of a goal serves implicitly as an identifier, uniquely identifying the action or goal (due to the assumption on the uniqueness of such variables, as indicated in section 4). Note that, as a consequence of the above, the variables in KB_0 are not properly speaking variables as such. Rather, they can be equated to "named variables" as in [KMM00].

Note that at this stage of the development of the KGP model we assume that the computee trusts its environment absolutely and therefore considers that the information in KB_0 is irrevocable.

In the sequel we will assume that KB_0 is contained in all the remaining knowledge bases that form KB.

Given a state $S = \langle KB, Goals, Plan \rangle$, we will denote by $\Sigma(S)$ the valuation obtained as follows:

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

When the state S we are referring to is clear from the context, we will write simply Σ instead of $\Sigma(S)$. Intuitively, 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). KB_0 provides a "virtual" representation of Σ.

Below, $\Sigma(t)$, 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 temporal constraints associated with goals or actions in a state S will always take Σ into account. Let Tc be a (set of) temporal constraint with variables $t_1, \ldots, t_n, t_{n+1}, \ldots, t_m$, $m \geq n \geq 1$, such that:

- for each $i = 1, \ldots, n$, $\Sigma(t_i) = \tau_i$ for some time point τ_i;
- for each $i = n+1, \ldots, m$, there exists no τ_i such that $\Sigma(t_i) = \tau_i$.

Then, a *total* $\Sigma-$*valuation* σ for Tc is a valuation for the time variables in Tc which are not evaluated by Σ already, namely σ is a valuation $\{t_{n+1} = \tau_{n+1}, \ldots, t_m = \tau_m\}$, for some time points $\tau_{n+1}, \ldots, \tau_m$. Given a temporal constraint Tc and a total $\Sigma-$valuation σ for Tc, we will write $\sigma \models Tc$ if Tc is satisfied by the valuation $\sigma \cup \Sigma$. Moreover, we will write $F\sigma$ to denote the application of the valuation $\sigma \cup \Sigma$ to a formula F.

6 Capabilities

We will assume the reasoning capabilities given below. To refer to these capabilities, we use entailment symbols with subscripts. However, these are not always intended to formally denote entailment relationships but operators to achieve core forms of reasoning.

6.1 Planning

The Planning capability $\models_{plan}$ supports the planning by means of an abductive event calculus theory [KS86, Sha89]. In effect, $\models_{plan}$ is a partial planning operator, which selects one individual partial plan, if some exists, for every mental goal in the set of goals it is given as input.

Given a state $\langle KB, Goals, Plan\rangle$, and a non-empty set of mental goals $\{G_1, \ldots, G_n\} \subseteq Goals$,

$$KB, Plan, \{G_1, \ldots, G_n\} \models_{plan} \{\langle G_1, \mathcal{A}_1 s, \mathcal{G}_1 s\rangle, \ldots \langle G_n, \mathcal{A}_n s, \mathcal{G}_n s\rangle\}$$

standing for "the set of goals $\{G_1, \ldots, G_n\}$ can be reduced to the set $\mathcal{A}_1 s \cup \ldots \cup \mathcal{A}_n s$ of actions and $\mathcal{G}_1 s \cup \ldots \cup \mathcal{G}_n s$ of subgoals, given KB and $Plan$". In particular, each goal G_j can be reduced to the set $\mathcal{A}_j s$ of actions and $\mathcal{G}_j s$ of subgoals, given KB and $Plan$. Note that each of the $\mathcal{A}_j s$, $\mathcal{G}_j s$ may be empty, e.g., if G_j has already been planned for. Moreover, each of the $\mathcal{A}_j s$, $\mathcal{G}_j s$ may be $\bot$, if there exists no "viable" plan for G_j (in which case $\mathcal{A}_j s = \mathcal{G}_j s = \bot$). [4]

The Planning capability is defined with respect to a subset KB_{plan} of the knowledge base KB. KB_{plan} is based upon the ontology of the *event calculus* [KS86] in the sense that

1. it allows parallel actions,
2. it deals with local states (not global states as with the situation calculus snap-shot approach),
3. it follows the approach of the event calculus in the definition of postconditions/effects and preconditions.

Concretely, KB_{plan} is an *abductive* variant of the event calculus (e.g., see [Sha89]). Note that the event calculus theory we adopt can be obtained from translating theories in the $\mathcal{E}$-language [KM97b, KM97a] into the event calculus. In section 6.3 we will see how to extend KB_{plan} so that it can be used to perform temporal reasoning, again in the spirit of the $\mathcal{E}$-language.

6.1.1 KB_{plan}: Abductive Event Calculus

As the knowledge base KB_{plan} used to represent the knowledge required for partial planning we adopt (a variant of) the abductive event calculus, namely $KB_{plan} = \langle P_{plan}, A_{plan}, I_{plan}\rangle$. In the following specification of KB_{plan} we adopt the notational conventions of the abductive event calculus. In particular, an atom of the form $holds\_at(G, T)$ stands for *the fluent G holds at time T* and an atom of the form $happens(A, T)$ stands for *the action A takes place at time T*. In order to link these event calculus formulation to our time fluent and timed operators, we extend the basic theory, originating from the conventional event calculus, with suitable *bridge rules*.

[4] Note that we overload the symbol $\bot$, to indicate both the parent of top-level goals and the absence of a "viable" plan for a goal.

- P_{plan} consists of two parts: *domain-independent rules* and *domain-dependent rules*. In the sequel, we assume that 0 is the initial time.

  Domain independent rules

  $holds\_at(G, T_2) \leftarrow happens(A, T_1), T_1 < T_2, initiates(A, G), not\ clipped(T_1, G, T_2)$

  $holds\_at(\neg G, T_2) \leftarrow happens(A, T_1), T_1 < T_2, terminates(A, G), not\ declipped(T_1, G, T_2)$

  These rules ensure that positive fluents persist from the time they have been initiated by some event and negative fluents persist from the time their corresponding positive fluents have been terminated by some event.

  $holds\_at(G, T) \leftarrow holds\_initially(G), 0 < T, not\ clipped(0, G, T)$

  $holds\_at(\neg G, T) \leftarrow holds\_initially(\neg G), 0 < T, not\ declipped(0, G, T)$

  These rules ensure that fluents persist from the initial time, if they held at that time.

  $clipped(T_1, G, T_2) \leftarrow happens(A,\ T), terminates(A,\ G), T_1 < T < T_2$

  $declipped(T_1, G, T_2) \leftarrow happens(A,\ T), initiates(A,\ G), T_1 < T < T_2$

  These rules ensure that fluents do not persist if events occur terminating them (if they are positive) of initiating their complementary fluent (if they are negative).

  $holds\_at(G, T_2) \leftarrow observed(G[\_], T_1), T_1 < T_2, not\ clipped(T_1, G, T_2)$

  $holds\_at(\neg G, T_2) \leftarrow observed(\neg G[\_], T_1), T_1 < T_2, not\ declipped(T_1, G, T_2)$

  $happens(A, T) \leftarrow executed(A[\_], T)$

  $happens(A(C), T) \leftarrow observed(C, A[\_], T)$

  These rules serve as *bridge rules* for connecting the abductive event calculus theory to KB_0. Note that the third such rule implies that the effects of the action performed by the computee C will be initiated at the moment of the observation of this action by the computee whose knowledge base we are considering here. Alternatively, we could have written

  $happens(A(C), T') \leftarrow observed(C, A[T'], T)$

  $happens(A, T) \leftarrow assume\_happens(A, T)$

  This rule links *happens* to the abducible *assume_happens* (see below the definition of A_{plan}).

  Domain dependent rules

  P_{plan} also contains domain-dependent rules defining the predicates *holds_initially*, *initiates*, *terminates*, e.g.

  $holds\_initially(at(c, (1, 1)))$

  This rule represents that computee c is initially (i.e., at time 0) at location (1,1) (assuming that the environment in which c operates is a two-dimensional grid).

$initiates(go(X, L_1, L_2), T, at(X, L_2)) \leftarrow holds\_at(mobile(X), T)$

$terminates(go(X, L_1, L_2), T, at(X, L_1) \leftarrow holds\_at(mobile(X), T), L_1 \neq L_2$

These rules represent that an action $go(X, L_1, L_2)$ executed at time T initiates the property that the computee X is at location L_2 and terminates the property that the computee X is at location L_1, provided that the computee X is mobile and, for the rule for *terminates*, that L_1 and L_2 are different. In practice, the conditions in the body of these rules serve as *preconditions for the effects* of the action to hold, after the execution of the action. In addition to this kind of preconditions, KB_{plan} might specify *preconditions for actions to be executable*, e.g.

$precondition(go(X, L_1, L_2), at(X, L_1))$

This rule represents a precondition for the event/action *go* to be executable, in that a plan containing an action $go(X, L_1, L_2)$ needs to contain also actions guaranteeing that the precondition $at(X, L_1)$, at the same time as the action. The I_{plan} below specifies how such preconditions are enforced by the Planning capability, as we will see in the next section 6.1.2.

P_{plan} may also contain additional rules, e.g., defining domain-dependent predicates which may be static.

- A_{plan} consists of the predicate $assume\_happens$.
- I_{plan} contains the following domain-independent integrity constraints:

  $holds\_at(F, T), holds\_at(\neg F, T) \Rightarrow false$

  These integrity constraints enforce that a fluent and its negation can never hold at the same time, and that at every time either a fluent or its negation must hold.

  $happens(A, T) \wedge precondition(A, P) \Rightarrow holds\_at(P, T)$

  This integrity constraint enforces that (all) the preconditions for the executability of actions hold at the time of the execution of the action.

  I_{plan} may also contain additional integrity constraints, e.g., defining domain-dependent *ramification statements*, e.g.,

  $holds\_at(at(O, X), T), holds\_at(contains(O, I), T) \Rightarrow holds\_at(at(I, X), T)$

  This integrity constraint represents that if an object O is located at some location X, then whatever item I contained in O is also located at X.

Given a plan $Plan$, we denote by $\mathcal{EC}(Plan)$ its representation in the event calculus formulation, that is the conjunction

$$\bigwedge_{\langle a[t], \_, \_, Tc \rangle \in Plan} (happens(a, t) \wedge Tc)$$

Similarly, given a set of goals $Goals$, we denote by $\mathcal{EC}(Goals)$ its representation in the event calculus formulation, that is the conjunction

$$\bigwedge_{\langle l[t],\_,Tc\rangle \in Goals} (holds\_at(l,t) \wedge Tc)$$

6.1.2 Specification of $\models_{plan}$

Let $S = \langle KB, Goals, Plan\rangle$ be a state, and $\mathcal{G}s$ be the (non-empty) set of mental goals $\{\langle l_1[t_1], G'_1, T_1\rangle, \ldots, \langle l_n[t_n], G'_n, T_n\rangle\}$. Then:

$$KB, Plan, \mathcal{G}s \models_{plan} \quad \{\langle G_1, \mathcal{A}_1 s, \mathcal{G}_1 s\rangle, \ldots, \langle G_n, \mathcal{A}_n s, \mathcal{G}_n s\rangle\}$$

where, for each $j = 1, \ldots, n$

- either $\mathcal{A}_j s = \mathcal{G}_j s = \bot$, representing that there exists no "viable" partial plan for goal G_j
- or

  $\mathcal{A}_j s = \{(a^j_1[t^j_1], T^j_1), \ldots, (a^j_{m_j}[t^j_{m_j}], T^j_{m_j})\}$, $m_j \geq 0$, each $a^j_i[t^j_i]$ is a timed operator and T^j_i are temporal constraints and

  $\mathcal{G}_j s = \{(l^j_1[t^j_1], S^j_1), \ldots, (l^j_{k_j}[t^j_{k_j}], S^j_{k_j})\}$, $k_j \geq 0$, each $l^j_i[s^j_i]$ is a timed literal, and S^j_i are temporal constraints, representing a partial plan for goal G_j,

such that, if $\mathcal{T}$ is the set of all temporal constraints in $\mathcal{G}_s, \mathcal{A}_1 s, \ldots, \mathcal{G}_1 s, \ldots, \mathcal{A}_n s, \ldots, \mathcal{G}_n s$:

- there exists a total $\Sigma-$valuation σ for $\mathcal{T}$ such that $\sigma \models \mathcal{T}$.

Namely, all temporal constraints in the given set of goals and in the partial plans for them, if any, are satisfiable. [5]

$\mathcal{T}$ can be formally defined as

$$\bigcup_{j=1,\ldots,n} T_j \cup \bigcup_{j=1,\ldots,n, \mathcal{A}_j s \neq \mathcal{G}_j s \neq \bot} \bigcup_{i=1,\ldots m_j} T^j_i \cup \bigcup_{j=1,\ldots,n, \mathcal{A}_j s \neq \mathcal{G}_j s \neq \bot} \bigcup_{i=1,\ldots k_j} S^j_i.$$

For each $j = 1, \ldots, n$, the sets $\mathcal{A}_j s, \mathcal{G}_j s$ are defined as follows:

- either there exist sets $X_j = \{(a^j_1[t^j_1], T^j_1), \ldots, (a^j_{m_j}[t^j_{m_j}], T^j_{m_j})\}$, $m_j \geq 0$ and $Y_j = \{(l^j_1[t^j_1], S^j_1), \ldots, (l^j_{k_j}[t^j_{k_j}], S^j_{k_j})\}$, $k_j \geq 0$ such that

  (i) $P_{plan} \wedge [\bigwedge_{i=1,\ldots,m_j} assume\_happens(a^j_i, t^j_i) \wedge \bigwedge_{\ell=1,\ldots,k_j} holds(l^j_\ell, s^j_\ell)]\sigma \models holds\_at(l_j, t_j)\sigma$, and

  (ii) $P_{plan} \wedge [\bigwedge_{j=1,\ldots,n} \bigwedge_{i=1,\ldots,m_j} assume\_happens(a^j_i, t^j_i) \wedge \mathcal{EC}(Plan)]\sigma$
  $\wedge [\bigwedge_{j=1,\ldots,n} \bigwedge_{i=1,\ldots,k_j} holds\_at(l^j_i, t^j_i) \wedge \mathcal{EC}(Goals)]\sigma \models_{LP} I_{plan}$

  and $\mathcal{A}_j s = X_j$ and $\mathcal{G}_j s = Y_j$
- or there exist no sets X_j, Y_j as above, and $\mathcal{A}_j s = \mathcal{G}_j s = \bot$.

[5] See section 5 for the definition of $\models$.

Intuitively, for each given goal in the initially given set of goals $\mathcal{G}s$, the Planning capability yields a (partial) plan which contains a (possibly empty) set of actions and a (possibly empty) set of goals. Each action and goal is equipped with temporal constraints, which are overall satisfiable, jointly with the temporal constraints of all goals and other partial plans. The actions and goals in each partial plan are such that they allow to entail the goal (condition (i)) and together with the current plan in $Plan$ and with the current goals in $Goals$ satisfy as a whole the set of integrity constraints in I_{plan} (condition (ii)). If no partial plan exist satisfying condition (i) or some exist but it does not satisfy condition (ii) then the Planning capability returns $\bot$, which stands for "no viable plan".

6.1.3 Example of $\models_{plan}$

Suppose the Planning capability of a computee x is given in input, at time $\tau = 5$, the goal G:

$$\langle in\_touch\_with(y,t), \bot, t \leq 10 \rangle$$

to be in touch with another computee y by time 10. Suppose KB_{plan} contains the following domain-specific axioms (in addition to the domain-independent ones):

$$initiates(write(x,Y),T,in\_touch\_with(Y))$$
$$precondition(write(x,Y),have(connection))$$
$$precondition(write(x,Y),know\_address(Y))$$
$$holds\_initially(know\_address(y))$$

Then, $\models_{plan}$, might return the following:

$$\langle G, \{(write(x,y,t'), t' < t)\}, \{(have(connection,t'), t' < t)\} \rangle$$

Note that other outputs are also allowed by the formal specification of $\models_{plan}$, e.g.

$$\langle G, \{(write(x,y,t'), t' < t)\}, \{(know\_address(y,t'), t' < t), have(connection,t'), t' < t)\} \rangle.$$

6.1.4 Possible variants of KB_{plan} and $\models_{plan}$

There are a number of abductive logic programs onto which to map the original event calculus [KS86, Sha89]. However, they can be proven to be equivalent to each other [TK95]. Above, we have chosen one concrete such abductive logic program, but others would have been possible.

All of possible choices of abductive event calculus theories provide an intensional description of plans. In alternative, P_{plan} in KB_{plan} could be a library of plans, providing an extensional description of the plans, for given goals. Such library could be obtained by partially evaluating (any of) the above event calculus. For example, P_{plan} might consist of

$$\begin{aligned} holds\_at(in\_touch\_with(Y),T) \leftarrow \quad & happens(write(Y),T-1), \\ & happens(get\_address(Y),T-2), \\ & happens(get\_connection,T-3) \end{aligned}$$

6.2 Identification of preconditions

The capability $\models_{pre}$ supports the reasoning capability of identifying (observable) *preconditions for the executability* of actions in plans. This capability is used when an action has to be inserted in the plan of the computee.

$KB, a[t] \models_{pre} C$ stands for "C is a conjunction of (observable) preconditions of the action operator $a[t]$, given KB". In effect, $\models_{pre}$ allows for conditional planning.

This capability does not require a separate part of the knowledge base KB. It is instead defined in terms of the knowledge base for planning, KB_{plan}.

6.2.1 Specification of $\models_{pre}$

Given a state $\langle KB, Goals, Plan\rangle$ and a timed action operator $a[t]$,

$KB, a[t] \models_{pre} Cs$ iff

- either there exists c such that $P_{plan} \models_{LP} precondition(a, c)$ and $Cs = \bigwedge\{c[t] \mid P_{plan} \models_{LP} precondition(a, c)\}$
- or, otherwise, $Cs = true$. [6]

Note that, for a given action operator a, Cs is $true$ whenever no preconditions are explicitly indicated for a in P_{plan}. Note also that the time of the preconditions of an action is set to that of the action itself. Further, note that preconditions of actions, as stored within $Plan$, are not equipped with temporal constraints: temporal constraints for preconditions are indeed inherited from the action itself. Finally, note that this capability, as we have concretely defined it, involves hardly any reasoning. Indeed, the presence within the event calculus formulation of KB_{plan} of explicit $precondition$ statements greatly facilitates the specification of this capability.

6.2.2 Example of $\models_{pre}$

Let us consider the example in section 6.1.3, with

$precondition(write(x, Y), have(connection))$
$precondition(write(x, Y), know\_address(Y))$

and the timed operator $write(x, y, t)$. Then, $\models_{pre}$ will return

$have(connection, t) \wedge know\_address(y, t)$.

6.3 Temporal reasoning

The Temporal Reasoning capability $\models_{TR}$ allows to perform *temporal reasoning* by means of the event calculus given in section 6.1, extended to reason with incomplete information about fluents and for dealing with inconsistencies arising from observations in the environment.

[6] We assume that $true$ is a formula which is always entailed by KB.

A computee's knowledge for performing temporal reasoning is a subset of KB denoted by KB_{TR}, that together with the direct information that the computee obtains from the environment (and recorded in KB_0) allows to derive new knowledge about how properties of the world evolve in time, by means of $\models_{TR}$.

Given a state $\langle KB, Goals, Plan\rangle$ and a (ground) timed fluent literal $l[t]$ $KB \models_{TR} l[t]$ stands for "the timed fluent literal $l[t]$ holds in KB, namely it can be proven, within KB_{TR}, to follow from the set of actions known to have been executed and from the observations recorded in KB_0".

Note that, differently from all other capabilities, $\models_{TR}$ is a genuine entailment relationship. Note also that $\models_{TR}$ will be used to define all remaining reasoning capabilities within this section 6, as well as some of the transitions in section 7.

In a nutshell, the Temporal Reasoning capability $\models_{TR}$ is used to:

- Derive extra knowledge about the state of the world at a certain time (in the future and past) without the need to observe the world directly.
- Manage the change of the transient information obtained directly from the external world so that the computee has a consistent view of the history of the world.
- Help the computee to recognise the successful or failed execution of an action to produce its desired effects (and consequently recognise the success or failure of an attempt to satisfy a goal via a particular plan).

Overall, this capability plays a fundamental role in rendering the computee adaptable to changes in the environment of the computee.

We could adopt any framework of Reasoning about Actions and Change for the purpose of performing the required temporal reasoning. The particular framework that we adopt here, based on the event calculus and its E-Language extension as we have for planning, facilitates the interface between these two modules of KB_{TR} and KB_{plan} in KB.

6.3.1 KB_{TR}: Extended abductive event calculus

KB_{TR} is an abductive logic program $\langle P_{TR}, A_{TR}, I_{TR}\rangle$ where

- P_{TR} is P_{plan} together with the additional rules [7]
  $holds\_at(F,T) \leftarrow assume\_holds(F,T'), T' < T, not\, clipped(T',F,T)$
  $holds\_at(\neg F,T) \leftarrow assume\_holds(\neg F,T'), T' < T, not\, declipped(T',F,T)$
- A_{TR} consists of the predicate $assume\_holds$;
- I_{TR} is I_{plan} together with the additional integrity constraints
  $holds\_at(F,T) \vee holds\_at(\neg F,T)$
  $holds\_at(F,T), assume\_holds(\neg F,T) \Rightarrow false$
  $holds\_at(\neg F,T), assume\_holds(F,T) \Rightarrow false$

[7] If we assume that 0 is the initial time point, as we have done in section 6.1, then we can replace these two rules by
$holds\_at(F,T) \leftarrow assume\_holds(F,0), not\, clipped(0,F,T)$
$holds\_at(\neg F,T) \leftarrow assume\_holds(\neg F,0), not\, declipped(0,F,T)$.

Basically, reasoning with KB_{TR} as given above allows making *assumptions on fluents* holding or not, but does not allow making assumptions on events having happened (differently from reasoning with KB_{plan}). Assumptions on fluents allow filling gaps in the (incomplete) knowledge of the computee.

The abductive logic program given so far for KB_{TR} implicitly relies upon the assumption that KB_0 has a complete record of the relevant events that have occurred in the world (as no *assumptions on events* having happened can be made within such KB_{TR}). This assumption is not appropriate for all problem domains, and in particular not for domains in global computing environments. In such environments, it is possible for $KB_{TR} \cup KB_0$ to be inconsistent, and that the only way to recover from this is to assume a minimal set of occurrences of events not recorded explicitly in KB_0. In order to cope with such domains, we can extend the above formulation by allowing A_{TR} to contain abducibles of the form $assume\_happens(A, T)$, for every action operator A and time point T, as for the Planning capability.

- A_{TR} consists of thee predicates *assume_holds* and *assume_happens*.

The knowledge base can also become *classically inconsistent* as we add new information in KB_0 namely in the case where there exists at least one time point where the set of facts in KB_0 together with the integrity constraints in KB_{TR} cannot be satisfied together as classical formulae. For example, the computee wants to record that at the same time point the same fluent is true and false. A general way to deal with this would be to introduce a form of belief revision for classical theories along the lines for example of the AGM framework [AGM85]. This is beyond the scope of our work and we can adopt a simple (but weaker solution) solution of not accepting to record (via the Passive and Active Observation Transitions as we will see in section 7) a fluent observation when this makes $KB_{TR} \cup KB_0$ classically inconsistent. When we have a set of observations we similarly record only a maximal subset of this that keeps $KB_{TR} \cup KB_0$ classically consistent. Other middle ground solutions, where the observations that are deleted could be earlier ones already recorded in KB_0 according to the reliability of the source of the information etc, are also possible.

6.3.2 Specification of $\models_{TR}$

Given an abductive logic program KB_{TR} as discussed above and a (ground) timed fluent literal $l[t]$, $\models_{TR}$ can be defined in two alternative ways.

credulous $\models_{TR}$ ($\models_{TR}^{cred}$): $KB \models_{TR}^{cred} l[t]$ iff there exists a set Δ of (ground) atoms in the predicates in A_{TR} such that

1. $P_{TR} \cup \Delta \models_{LP} holds\_at(l, t)$,
2. $P_{TR} \cup \Delta \models_{LP} I_{TR}$, and
3. $\Delta \cap \{assume\_happens(a, t') | a$ is an action operator and t' is a time point $\}$ is minimal (with respect to set inclusion),
   namely, for all $\Delta' \subseteq A_{TR}$ satisfying 1 and 2, $\Delta' \cap \{assume\_happens(a, t') | a$ is an action operator and t' is a time point $\} \supset \Delta \cap \{assume\_happens(a, t') | a$ is an action operator and t' is a time point $\}$.

Intuitively, condition 1 says that the timed fluent under consideration is entailed by appropriately "completing" the logic program in KB_{TR} by means of a set of assumptions

Δ from A_{TR}, condition 2 says that the "completed" $P_{TR} \cup \Delta$ satisfies the integrity constraints in KB_{TR}, and condition 3 says that the subset of Δ consisting solely of abducibles on events having happened should be minimal with respect to set inclusion.

sceptical $\models_{TR}$ ($\models_{TR}^{scep}$): $KB \models_{TR}^{scep} l[t]$ iff

- $KB \models_{TR}^{cred} l[t]$, and
- $KB \not\models_{TR}^{cred} \overline{l}[t]$, where, if l is a fluent f, then $\overline{l} = \neg f$, and, if l is a fluent literal $\neg f$, then $\overline{l} = f$.

In the sequel we will assume that $\models_{TR}$ amounts to $\models_{TR}^{scep}$.

6.3.3 Example of $\models_{TR}$

Consider the following domain-specific rules and integrity constraints in KB_{TR}:

$initiates(turn\_on\_key, T, running \leftarrow holds\_at(battery, T)$
$terminates(turn\_off\_key, T, running)$
$holds\_at(broken, T) \Rightarrow holds\_at(\neg running, T)$
$holds\_at(\neg petrol, T) \Rightarrow holds\_at(\neg running, T)$

and the following KB_0:

$observed(battery(\_), 0)$
$executed(turn\_on\_key(\_), 2)$

In this example, *running* is initiated at time 2 and the computee would derive that it holds true from time 3 onwards as no known event occurrence after 2 terminates *running*. If the computee now observes that a *turn_off_key* action has been executed at time 5 and so its KB_0 is updated with the additional statement $executed(turn\_off\_key(\_), 5)$ then its Temporal Reasoning capability would derive sceptically that *running* is true at times $3, 4, 5$ and false from time 6 onwards.

6.3.4 Extension of KB_{TR} for failing actions

The version of KB_{TR} presented earlier is not able to recognise and deal with failing actions, so as to avoid obtaining inconsistencies.

Consider for example what would happen to the temporal reasoning of a computee if we add to KB_0 of the earlier example (in section 6.3.3), the fact $observed(broken(\_), 0)$. Then since this fact would persist everywhere (we have no information of an event occurrence that would terminate *broken*) after time 2 we have that on the one hand, the integrity constraint

$$holds\_at(broken, T) \Rightarrow holds\_at(\neg running, T)$$

would require that *running* is false, and on the other hand that *running* should be true because of the *turn_on_key* action. A similar problem would arise in the simpler situation when the computee observes at some time after 2 that *running* is false whereas the action should have made it true. Such situations could easily arise in a Global Computing environment where the computee is expected to operate.

We can handle this problem [8] by changing the form of the temporal reasoning knowledge base KB_{TR} without the need to change the Temporal Reasoning $\models_{TR}$ capability specification. We follow again the approach of the E-Language. This is based on turning the effect laws, represented via rules defining the predicates *initiates* and *terminates*, into default laws. Informally, we then have that an action occurrence produces its effect under some preconditions but only if we do not have information to the contrary, i.e., information that the effect is false after the occurrence. We can achieve this by replacing the rules for *initiates* of the form:

$$initiates(A, T, F) \leftarrow holds\_at(L_1, T), \ldots, holds\_at(L_k, T)$$

by the rules:

$$initiates(A, T, F) \leftarrow holds\_at(L_1, T), \ldots, holds\_at(L_k, T), not\ contrary(F, T)$$

$$contrary(F, T) \leftarrow observed(\neg F(\_), T_1), T_1 > T, not\ clipped(F, T, T_1)$$

Similarly, for the *terminates* rules. Note that *clipped* and *declipped* are defined via *terminates* and *initiates* respectively and will thus use their new definitions automatically.

This extended formulation of KB_{TR} is though only correct if the domain does not contain any integrity constraints of the form:

$$holds\_at(L_1, T), \ldots, holds\_at(L_k, T) \Rightarrow holds\_at(L, T)$$

that link fluents via ramifications. If we have such constraints then we need to replace in the above rule for $contrary(F, T)$ the condition $observed(\neg F(\_), T_1)$ by a more general $evidence(\neg F(\_), T_1)$, defined either through a direct observation as before or indirectly through some minimal proof of $\neg F$ drawn from the theory. For example, in the above example $evidence(\neg running, T)$ will be given by:

$$evidence(\neg running, T) \leftarrow observed(\neg running(\_), T)$$

together with the rules:

$$evidence(\neg running, T) \leftarrow holds\_at(broken, T)$$

$$evidence(\neg running, T) \leftarrow holds\_at(\neg petrol, T).$$

In fact these extra rules correspond to the prime implicants for $\neg F$ drawn from the integrity constraints when these are treated as statements of classical logic. The details for this can again be found in [KM03a].

To illustrate the utility of this extension let us return to the example above and suppose that $observed(broken(\_), 0)$ is added to KB_0. Then $\models_{TR}$ will derive that *running* holds false sceptically for any time with the occurrence of *turn_on_key* at 2 failing to produce its effect. This failure of the action is due to the fact that $contrary(running, 2)$ holds as we can derive from the rule $evidence(\neg running, T) \leftarrow holds\_at(broken, T)$ evidence for $\neg running$ at time 3. This then blocks the initiates rule for *running*. Further examples that illustrate failing actions and how such failing effects would interact with each other can be found in [KM03a].

[8] This problem is in fact directly linked to the qualification problem in reasoning about actions and change. For details see [KM03a].

6.4 Reactivity

The capability $\models_{react}$ supports the reasoning capability of reacting to stimuli from the external environment as well as to decisions taken while planning. $\langle KB, Goals, Plan\rangle \models_{react} \mathcal{G}s, \mathcal{A}s$ stands for "the set of goals $\mathcal{G}s$ and set of actions $\mathcal{A}s$ are introduced in order to react to some observation recorded in (the KB_0 part of the) given KB or to some goals in $Goals$ and actions in $Plan$".

6.4.1 KB_{react}: reactive constraints

KB_{react} is a set of *reactive constraints*, of the form

$$Triggers, Pre \Rightarrow h[T] \wedge Tc$$

where

- $Triggers$ is a non-empty conjunction of items of the form $observed(l[T'], T'')$ or $observed(c, a[T'], T'')$,
- Pre is a conjunction of any of the following:
  - $holds\_at(l, T')$, where $l[T']$ is a timed fluent literal,
  - $happens(a, T')$, where $a[T']$ is a timed action operator, and
  - temporal constraints;
- $h[T]$ is either a timed fluent literal or a timed action operator, and
- Tc are temporal constraints.

All variables in $Triggers$ and Pre are implicitly universally quantified from the outside. All variables in $h[T] \wedge Tc$ not occurring in $Triggers$ or Pre are implicitly existentially quantified on the righthand side of the implication.

Intuitively, a reactive constraint $Triggers, Pre \Rightarrow h[T] \wedge Tc$ is to be interpreted as follows: if (some instantiation of) all the observations in $Triggers$ hold in KB_0 and (some corresponding instantiation of) Pre holds (to be tested by means of $\models_{TR}$), then (the appropriate instantiation of) $h[T]$, with associated (the appropriate instantiation of) the temporal constraints Tc, should be added to $Goals$ or $Plan$ (depending on the nature of h). The $Triggers$ serve as a set of triggers for the reactive rule, and are somewhat responsible for that rule to be "fired". There must be at least one such trigger. We will see that reactive rules are "fired" after new observations are made in the environment and recorded in KB_0.

Each such reactive constraint represents either a condition-action rule or a commitment rule (if h is an action operator) or a condition-goal rule (if h is a fluent). Each such constraint is a (conventional) condition-action rule if $Triggers$ and Pre share the same time parameter S, and the time T of h is $S + 1$.

Note that the left-hand side of a reactive constraint is written in the language of the event calculus, as it is meant to be evaluated by means of $\models_{TR}$. Instead, the right-hand side of the constraint is written in terms of fluent literals and action operators, as it will contribute to the $Goals$ and $Plan$ of the state. We could have written such constraints uniformly with left- and right-hand sides written within the same language, but this would have complicated the semantics of this capability, that we will see below.

For simplicity, below we will refer to a reactive rule simply as

$$Body \Rightarrow h[T] \wedge Tc$$

Finally, note that reactive constraints are special kinds of implicative integrity constraints in abductive logic programming.

6.4.2 Specification of $\models_{react}$

Given a state $\langle KB, Goals, Plan\rangle$,

$$\langle KB, Goals, Plan\rangle \models_{react} \mathcal{G}s, \mathcal{A}s$$

where:

- $\mathcal{G}s = \{(l_1[t], l_1'[t'], Tc_1), \ldots, (l_n[t], l_n'[t'], Tc_n)\}$, $n \geq 0$, where l_i, l_i' are fluent literals and Tc_i are temporal constraints,
- $\mathcal{A}s = \{(a_1[s], j_1[s'], Sc_1), \ldots, (a_m[s], j_m[s'], Sc_m)\}$, $m \geq 0$, where a_i is an action operator, j_i is a fluent literal and Sc_i are temporal constraints,

$\mathcal{G}s, \mathcal{A}s$ are defined as follows. Let δ, δ^* stand for either a fluent literal or an action operator.

(i) for each $(\delta^*[t], l'[t'], Tc) \in \mathcal{G}s$, there exists a (variant of a) reactive rule in KB_{react} of the form

$$Body \Rightarrow \delta[t] \wedge Tc$$

and a (possibly empty) subset $Goals' \subseteq Goals$ such that

- $\delta^*[t] = \delta[t]\theta$ and $KB_{TR} \cup \mathcal{EC}(Plan) \cup \mathcal{EC}(Goals) \models_{TR} Body\theta$, for some (total) valuation θ of the variables in $Body$,
- $KB_{TR} \cup \mathcal{EC}(Plan) \cup \mathcal{EC}(Goals) \setminus \mathcal{EC}(Goals') \not\models_{TR} Body\theta$ and
- either $Goals' \neq \{\}$ and $\langle l''[t'], l'[t'], \_\rangle \in Goals'$, or $Goals' = \{\}$ and $l'[t'] = \bot$.

(ii) $KB_{TR} \cup \mathcal{EC}(Plan) \cup \mathcal{EC}(Goals) \cup \mathcal{EC}(\mathcal{A}s) \cup \mathcal{EC}(\mathcal{G}s) \models_{LP} I_{TR}$.

Intuitively, the conclusion of reactive rules which have "fired" are returned (condition (i)), but only if they satisfy the integrity constraints in KB_{TR} "globally" (condition (ii)), namely together with all existing actions in $Plan$ and goals in $Goals$, and together with all actions and goals returned by this capability at the current time.

Note that the goals associated with the newly generated actions and goals will become their parents within the appropriate transition (see section 7.3). These goals are the parents of goals (l'') that are "responsible" for the newly generated actions and goals. So, the newly generated actions and goals, after the appropriate transition, will be siblings of these "responsible" goals. Note that there might be many such "responsible" goals, and for each one of them there will be a triple in $\mathcal{A}s$ and $\mathcal{G}s$, respectively. This means that we will get many copies of the same reaction in the state after applying the appropriate transition.

6.4.3 Example of $\models_{react}$

A possible reactive constraint in the knowledge base KB_{react} of a computee x is the following:

$$observed(Y, request(Y, x, R, T), T'), holds\_at(have(R), T') \Rightarrow \\ give(x, Y, R, T'') \wedge T' < T'' \leq T' + 10$$

This represents the commitment rule that, if some computee Y requests some resource R from x at some time T and x has the requested resources, then x commits to giving the resource to the requesting computee Y at some later time, but within 10 units of time.

Suppose KB_0 of the computee x consists of the facts

$$executed(acquire(pen, t), 2)$$
$$observed(y, request(y, x, pen, 3), 5)$$

and that KB_{TR} contains rules

$$initiates(acquire(pen), T, have(pen))$$
$$terminates(give(pen), T, have(pen))$$

Then, $\models_{react}$ will return the triple

$$\langle give(y, pen, t'), \bot, 5 < t' \leq 15 \rangle.$$

6.5 Goal decision

Part of the knowledge base KB of a computee, denoted by KB_{GD}, describes the policy of the computee under which it decides which (top-level) goals it is to set in its internal state (in *Goals*) at a certain time. The generated goals are the goals of current interest to the computee and for which it will go on to generate plans in order to achieve them. This Goal Decision capability, denoted by $\models_{GD}$, depends on KB_{GD} as well as on the state of the external environment as this is perceived by the computee in KB_0 and reasoned with using KB_{TR}.

The Goal Decision capability $\models_{GD}$ is defined in terms of the entailment $\models_{pr}$ for reasoning under preferences, as discussed in section 3.3. KB_{GD} is an $LPwNF$ theory that describes the conditions or rules under which a goal can be chosen together with other local information of priority amongst these rules that $\models_{pr}$ turns into a global preference.

6.5.1 The knowledge base KB_{GD}

In general, the knowledge base KB_{GD} contains 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. The lower-level part of KB_{GD} consists of rules in $LPwNF$ of the form

$$L \leftarrow L_1, \ldots, L_n \quad (n \geq 0)$$

where

- $L_1, \ldots, L_n$ are either time-dependent conditions of the form $holds\_at(l, t)$, where $l[t]$ is a timed fluent literal, or time-independent conditions, formulated in terms of auxiliary predicates defined within KB_{GD}, or temporal constraints.

- L is a timed fluent literal, together with (possibly empty) temporal constraints.

Thus, rules in the lower-level part of KB_{GD} are of the form

$$g[t], Tg \leftarrow L_1, \ldots, L_m, T_C \quad (m \geq 0)$$

where g is a fluent, Tg is a (possibly empty) set of temporal constraints and the time variable t is existentially quantified with scope the conclusion of the rule. As usual all variables in the body of the rules are implicitly universally quantified from the outside and a rule then represents all its ground instances under any total valuation of the time variables in the body that satisfies T_C. All variables in $g[t], Tg$ not occurring in the body of the rule are implicitly existentially quantified with scope the conclusion of the implication.

The conditions of the rules are meant to be evaluated in KB_{GD} together with KB_{TR} by combining the background derivability $\models_H$ of the $LPwNF$ framework with the Temporal Reasoning capability $\models_{TR}$. With abuse of notation we will denote in this subsection this derivability simply by $\models_{TR}$.

The higher-level part of KB_{GD} consists of rules in $LPwNF$ of the form

$$h\_p(rule1, rule2) \leftarrow L_1, \ldots, L_n, T_C$$

where $rule1, rule2$ are (parameterised) names of other rules in the knowledge base KB_{GD}. These rules in the higher-level part of KB_{GD} represent the (local) priorities amongst rules in the lower-level part or other priority rules in the higher-level part. The conditions of these rules can also be time depended and contain temporal constraints as for the case of the lower-level part rules.

In order to accommodate conflicts other than the ones due to the classical negation, namely between a fluent f and its negation $\neg f$, KB_{GD} includes also statements of incompatibility of the form

$$incompatible(l_1, l_2)$$

expressing that the fluent literals l_1 and l_2 are incompatible. In some cases this incompatibility could hold only under some conditions B (a conjunction of literals), expressed by the rule

$$incompatible(l_1, l_2) \leftarrow B$$

This notion of incompatibility extends to timed fleunt literals $l_1[t_1], l_2[t_2]$ with temporal constraints $Tg1, Tg2$, respectively, then we can extend the notion of incompatibility as follows. $l_1[t_1]$ and $l_2[t_2]$ are incompatible iff for every (total) valuation σ satisfying $Tg1, Tg2$, namely such that $\sigma \models Tg1$ and $\sigma \models Tg2$, the ground instances of the goals $l_1[t_1]\sigma$ and $l_2[t_2]\sigma$ are incompatible.

Finally, KB_{GD} will contain rules defining any auxiliary predicates occurring in the remaining part of KB_{GD}. We will refer to the set of these additional rules as KB_{GD}^{aux}.

6.5.2 Specification of $\models_{GD}$

Formally, the capability of $\models_{GD}$ is defined, using the admissibility semantics for $LPwNF$ as discussed in section 3.3, in the following way. Given a state $\langle KB, Goals, Plan\rangle$,

$$KB \models_{GD} \mathcal{G}s$$

where $\mathcal{G}s = \{\langle g_1[t_1], \bot, Tg_1\rangle, \ldots, \langle g_n[t_n], \bot, Tg_n\rangle\}$, $n \geq 0$, where g_i are fluent literals and Tg_i are temporal constraints on t_i, such that:

- $KB_{GD} \models_{pr} g_i[t_i], Tg_i$ for each $i = 1, \ldots, n$.

This means that a new (possibly empty) set of goals $\mathcal{G}s$ is generated that is currently preferred under the policy in KB_{GD} and the current information in KB_0 as used by the Temporal Reasoning capability to evaluate conditions in these policy rules. Note that some of these goals may already be present in the current state of the computee and their reappearance in $\mathcal{G}s$ simply means that they remain preferred goals to be achieved. The use of the Goal Decision capability as we will see below in the Goal Introduction transition (see section 7) takes care of such repeated goals and indeed of goals that need to be deleted from the current state as they are not preferred anymore.

More explicitly, $KB \models_{GD} \mathcal{G}s$ iff there exists a subset Δ of rules in KB_{GD} such that for each $\langle G, \bot, T_G\rangle \in \mathcal{G}s$

- Δ contains an instance $R\sigma$ of a rule R of the form $L, T_L \leftarrow B, T_C$ such that σ is a valuation of the temporal constraints T_C, $L\sigma = G$, $T_L\sigma = T_G$ and $KB \models_{TR} B\sigma$,[9]
- Δ is admissible in KB_{GD}
- no such Δ can be constructed for any goal that is incompatible with G, T_G.

6.5.3 Examples of $\models_{GD}$

Let us consider two examples of $\models_{GD}$ to illustrate the above definitions. For simplicity, in these examples conditions of the form $l(T)$, where l is a fluent literal, are meant to stand for their event-calculus counterpart $holds\_at(l, T)$, unless otherwise specified. For example, $weekend(T)$ stands for $holds\_at(weekend, T)$ and $deadline(Job, T)$ stands for $holds\_at(deadline(Job), T)$.

As a first example, suppose KB_{GD} expresses a policy for deciding which top-level goals amongst $gardening, holiday$ and $work$ a computee should set for itself. The lower-level or goal generation part of KB_{GD} contains the rules:

$r_1(T) : gardening(T) \leftarrow weekend(T)$
$r_2(Trip, T) : holiday(Trip, T) \leftarrow weekend(T), short(Trip)$
$r_3(Job, T) : work(Job, T) \leftarrow deadline(Job, T)$

that generates goals when certain conditions hold.

[9] We remind the reader that here in $KB \models_{TR} B\sigma$ the auxiliary part KB^{aux}_{GD} of KB_{GD} is used alongside $KB_{TR} \cup KB_0$ to evaluate the conditions $B\sigma$ of the rules.

Note that $short(Trip)$ can be seen as an auxiliary predicate. Note also that KB_{TR} may contain information about the future, e.g. that tomorrow is a weekend.

Let us also assume that the higher-level part of KB_{GD} contains the priority rules:

$R_1(T) : h\_p(r_1(T), r_2(\_, T)) \leftarrow gardening\_season(T)$
$R_2(T) : h\_p(r_2(Trip, T), r_1(T)) \leftarrow special\_offer(Trip, T)$
$R_3(T) : h\_p(r_3(Job, T), r_1(T)) \leftarrow bonus\_offer(Job, T)$
$R_4(T) : h\_p(r_3(\_, T), r_2(\_, T)) \leftarrow \neg bank\_holiday(T)$

and the higher order priority rule:

$C_1(T) : h\_p(R_2(T), R_1(T)) \leftarrow bank\_holiday(T).$

This theory expresses (via rule R_1) the policy that during a *gardening_season* the computee would prefer to do gardening over taking a holiday trip. Rule R_2 expresses the fact that the computee would prefer to take a holiday trip for which there is a special offer rather than do the gardening. Rules R_3 and R_4 express preferences a similar way. Finally, rule C_1 expresses the higher order preference that amongst the preference to do gardening when it is a gardening season and the preference to take a special offer holiday trip we would choose the second at a bank holiday.

In addition to the above rules, KB_{GD} contains rules stating that the three given goals are incompatible namely that no pair can be true simultaneously. For example, KB_{GD} has statements such as:

$$incompatible(gardening(T), holiday(Trip, T))$$

or if we want to render this incompatibility conditional on the $Trip$ been abroad then we have the statement:

$$incompatible(gardening(T), holiday(Trip, T)) \leftarrow abroad(Trip)$$

.

Consider now a scenario where a computee has to decide what to do at a weekend w amongst the goals $gardening(w), work(j, w), holiday(t, w)$, for all possible jobs j and trips t. Suppose first that the computee has no jobs with a deadline. Then no *work* goal can be generated by the lower-level part of KB_{GD} and hence $\models_{GD}$ will not produce such a goal under any circumstances. With this information alone both *gardening* and $holiday(t)$ for any short trip t are admissible and hence will each be a credulous conclusion of the policy theory KB_{GD}. To decide amongst these incompatible goals we need further information. So, if in addition this particular weekend w is during a *gardening_season* then only $gardening(w)$ will be admissible and hence a sceptical conclusions. Then $\models_{GD}$ will produced the goal $gardening(w)$. But if also there is a special offer for a particular trip t this weekend w, $holiday(t, w)$ will also be admissible and hence again we will have a dilemma and $\models_{GD}$ will not be able to generate any goal.

Further information that w is a bank holiday weekend will render $gardening(w)$ non admissible and then only $holiday(t, w)$ will be admissible. Hence under the circumstances of a bank holiday weekend $\models_{GD}$ generates the goal $holiday(t, w)$. Finally, suppose that the computee has

a particular job j with a deadline at the weekend w which is not a bank holiday. Then the goal $work(j, w)$ is admissible as is $gardening(w)$ but not any holiday trip goal. With the additional knowledge that there is a bonus for this job the only admissible goal is $work(j, w)$ and will be generated by the goal decision capability.

We have seen that in some cases it is possible for the computee to be in a dilemma as more than one goal which are incompatible with each other are admissible from its goal decision policy. In order to decide we either need more information, as in shown in the example above, or we can use additional policies that refer to other preference criteria. For example, it maybe possible (if the goals are appropriate) for the computee to use its general personality policy (see below) to help it decide amongst two such goals. In the example above in order to decide between $work(j, w)$ and $gardening(w)$ the computee could use its personality policy as follows. Suppose $work(j, w)$ is labelled addressing its "self-achievement" need and $gardening(w)$ its "social-affiliation" need. Then if its personality policy is selfish, i.e. "achievement" needs have higher priority over "social" needs, then under this policy only $work(j, w)$ is admissible and would be chosen by $\models_{GD}$ as the only goal to pursue. On the other hand, a computee with an altruistic personality would favour $gardening(w)$ over $work(j, w)$. More details on personality policies will be given below (see also [KM03b]).

The second example of KB_{GD} describes a more specific policy for deciding whether to *give* or *keep* objects as can arise for example within the context of a resource allocation problem (see [STT02b, STT02a] for an exposition of this problem within abductive logic programming). The lower-level part of KB_{GD} contains the rules:

$r_1(Res, Ag, T_1) : give(Res, Ag, T_1), T_1 \geq T_0 \leftarrow requested(Res, Ag, T_0), have(Res, T_0)$
$r_2(Res, T_0) : keep(Res, T_0) \leftarrow need(Res, Goal, T_0), have(Res, T_0)$[10]

and the higher-level or priority part of KB_{GD} contains the rules:

$R_1(Res, Ag, T_0) : h\_p(r_1(Res, Ag, \_), r_2(Res, T_0)) \leftarrow higher\_rank(Ag), alternative(Res, Res', T_0)$
$R_2(Res, Ag) : h\_p(r_2(Res, \_), r_1(Res, Ag, \_)) \leftarrow competitor(Ag)$
$C_1(Res, T_0) : h\_p(R_2(Res, \_), R_1(Res, \_, T_0)) \leftarrow urgent\_need(Res, T_0)$

This expresses a policy that depends on the relative roles of computees (e.g. $higher\_rank, competitor$) and specific context (e.g. $urgent\_need$) that the computee may find itself. The auxiliary predicates *alternative* and *urgent_need* are defined as follows:

$alternative(Res, Res', T_0) \leftarrow have(Res', T_0), same\_type(Res', Res)$
$urgent\_need(Res, T_0) \leftarrow need(Res, Goal, T_0), urgent(Goal, T_0)$

together with a set of facts for the auxiliary predicate *same_type* and an appropriate definition for $urgent(Goal, T_0)$.

We also have the following incompatibility statement between goals of giving and keeping:

$$incompatible(give(Res, \_, T), keep(Res, T))$$

[10] More realistically, this rule will give a goal to keep a resource for an interval of time starting from T_0 till the estimated time of use of the resource by the computee.

Consider now the case where a computee needs some resource res_1 at the same time t_0 that this resource is requested from another computee ag_1. Then if ag_1 is neither of higher rank nor a competitor then both goals $G_1 = keep(res_1, t_0)$ and $G_2 = \exists T_1 : T_1 \geq t_0, give(res_1, ag_1, T_1)$ are admissible and would both be generated by $\models_{GD}$ as they are compatible with each other (G_1 can be done at t_0 and G_2 at some later time).

If the computee has an alternative resource and the request comes from a higher ranking computee then only G_2 is admissible (G_1 is attacked by the ground goal $G_3 = give(res_1, ag_1, t_0)$) and would be the only goal generated by $\models_{GD}$. In the case where the request comes from a higher ranking but also competitor then again both G_1 and G_2 will be admissible and thus both will be generated by $\models_{GD}$. In the special case where the need for the resource is urgent the theory will derive that G_1 is a sceptical conclusion and hence will be generated by $\models_{GD}$, i.e. the computee will prefer to keep the resource at time t_0.

6.5.4 Goal decision and personality

As a more general example of KB_{GD} we can consider the case where this contains a personality theory (see [KM03b]) for the computee. In such a theory goals are generated and selected according to a general theory of needs for the computee. The *Goals* of a computee are separated into classes according to a set of high-level needs that they address. For example, an anthropomorphic computee will, according to theories of cognitive psychology, have its needs separated into five major categories: *Physiological, Safety, Affiliation or Social, Achievement or Ego and Self-actualisation or Learning.*

Then the lower-level or generation part contains, for each i that labels one of these categories of needs, rules of the form:

$$G_i[T'] \leftarrow R_i[T]$$

where: (1) $R_i[T]$ is a set of conditions under which this ith category of needs is not satisfied at time T and (2) $G_i[T']$ is any goal that when achieved at T' addresses the ith need. For example for $i = 2$, i.e. for the need of Safety, we can have a set of such rules in the general form of: $protect\_from(Hazard, T) \leftarrow danger(Hazard, T)$.

The higher-level or preference part of such a personality contains priority statements implementing a basic hierarchy of needs under "normal conditions" together with exceptions under "special circumstances". Such statements of basic hierarchy can have the form:

$$R^1_{ij} : h\_p(r_1(G_i), r_2(G_j)) \leftarrow N_i,$$
$$R^2_{ij} : h\_p(r_1(G_i), r_2(G_j)) \leftarrow R_i, \neg N_j$$

where $r_1(G_i)$ and $r_2(G_j)$ are any two rules for goals G_i and G_j pertaining to the ith and jth need respectively, and N_i denotes conditions under which the ith need is critical. These rules under the appropriate conditions give higher priority to rules (and hence to the respective goals that they produce) generating goals for the ith need over those rules that generate goals for the jth need. Hence a selfish computee will have in its personality theory the rules R^1_{43} and R^2_{43} (since the 4th motivation is that of "self-achievement" and the 3rd motivation is that of "social-affiliation") while an altruistic one will have the rules R^1_{34} and R^2_{34}.

This preference theory can be extended to cater for exceptions from the basic preference given by the above rules under some given special circumstances. For the details of this see [KM03b].

6.5.5 Goal Decision and Reactivity

The capabilities of Goal Decision and Reactivity are related in the sense that they both generate new goals. In fact, the rules in KB_{react} can be shared by the KB_{GD} in its lower-level part. But the goals generated by the two capabilities serve two different purposes. In the case of Goal Decision these are new goals that do not depend on the current set of *Goals* and *Plan* in the state of the computee. These goals can be selected from an a-priori set of candidate top-level goals that is identified at the design phase of the computee. In contrast, in the case of Reactivity, the new goals that it generates depend on the existing *Plan* and *Goals* in the current state of the computee. Its purpose is to adapt this *Plan* and *Goals* to take into account new information that it might have acquired. In addition, Reactivity may also introduce actions, as well as goals.

6.6 Sensing

In addition to the reasoning capabilities we have defined earlier, the computee is equipped with a sensing capability that allow it to obtain (up-to-date) information from its environment. This information can include reception of communication from other computees, observation of (communication) actions performed by other computees, and satisfaction of some observable property of the environment. We represent the sensing capability of a computee as an operator $\models^{\tau}_{Env}$. In particular, we use the following conventions:

- $\models^{\tau}_{Env} l$: a fluent literal is sensed at time τ.
- $\models^{\tau}_{Env} c : a$: it is observed, at time τ, that a computee c is performing at that time an action a.

We assume that this capability is time-stamped, to indicate the time at which it is applied within transitions. Also, we assume that $\models^{\tau}_{Env}$ is not total, in that it can be that for some timed fluent $f[t]$, neither $\models^{\tau}_{Env} f[t]$ nor $\models^{\tau}_{Env} \neg f[t]$ holds, for some τ. The implementation of this capability (within WP4) will have to guarantee that these properties are satisfied.

7 Transitions

The state of a computee evolves by applying transition rules. In this section we give a catalogue of such transitions, defined in terms of the capabilities given in section 6. We believe that the given catalogue is a suitable set of transitions to accommodate the features required from computees to face the challenges of the GC vision, as discussed in section 2. Note that computees might be equipped with just a subset of the set of transitions above, and that other transitions might be useful, as will be discussed in section 11.

Transitions will be denoted as

$$(\mathbf{L}) \qquad \frac{\langle KB, Goals, Plan\rangle \quad X}{\langle KB', Goals', Plan'\rangle}\,\tau$$

where $\mathbf{L}$ is the label of the transition, meaning that the transition takes place at time τ, it is given as input X, and changes the state from $\langle KB, Goals, Plan\rangle$ to $\langle KB', Goals', Plan'\rangle$. Depending on the transition, there might be no input and/or some components of the state

might not change. When the input exists, it represents a (non-empty) set of selected items (e.g. goals and/or actions) which will be provided via selection functions, given in section 8.

The time τ only plays a role in some of the transitions, namely *Passive Observation Introduction*, *Active Observation Introduction*, *Action Execution* and the revision transitions, as we will see later.

7.1 Goal Introduction (GI)

Goal Introduction revises the top-level goals of the computee (and accordingly the rest of the goal set) given the information from the goal decision capability $\models_{GD}$ which decides the goals the computee should focus on depending on its current circumstances. This transition revises also the current plan of the computee, in order to keep only actions which are relevant to the goals in the new set of goals.

$$(\mathbf{GI}) \qquad \frac{\langle KB, Goals, Plan\rangle}{\langle KB, Goals', Plan'\rangle}\,\tau$$

where

let $Gs = \{G \in Goals \mid parent(G) = \bot\}$. Then,

(i) If either there exists no Gs' such that $KB \models_{GD} Gs'$ or $Gs' = \{\}$, then

- $Goals' = \{\}$
- $Plan' = \{\}$.

(ii) If $KB \models_{GD} Gs'$, then

- $Goals' = Gs'' \cup des(Gs'', Goals)$ where:
  - (1) $Gs'' = \{G \in Gs' | G = \langle g[t], \bot, TC\rangle$ and there does not exist a (total) valuation σ such that $\sigma \models TC$ and $KB \models_{TR} g[t]\sigma\}$
  - (2) $des(Gs'', Goals) = \{G \in Goals | \exists G'' \in Gs''$ such that $G \in descendants(G'', Goals)\}$
- $Plan' = \{A \in Plan | A = \langle \_, G, \_, \_\rangle$ and $G \in Goals'\}$

In case (i) the computee has no goal to focus on, and this is reflected on the resulting state, where both the goals and the plan are empty. In case (ii), the set of goals in the new state is composed by the set of top-level goals returned by the goal decision capability, after having filtered out those goals which have been achieved already. Moreover, from the current set $Goals$, only (sub)goals relevant to the newly generated and not filtered away top-level goals (represented by the set $des(Gs'', Goals)$) are kept within the $Goals'$. Finally, the plan component of the state is updated in order to keep only actions related only to goals which were kept in the new state. Namely, actions whose parent is not in $Goals'$ are dropped from $Plan$. Note that the resulting $Plan'$ may be empty as a consequence.

Notice that this transition performs some kind of revision too. Indeed, the $\models_{GD}$ capability might remove as well as introduce some top-level goals. The goals that are eliminated are those that are not preferred any longer. These may be replaced with more preferred goals, if any.

7.2 Plan Introduction (PI)

Plan Introduction revises the state of the computee by updating the current set of goals and the current plan in order to take into account new plans for the goals selected for planning. These new plans are provided by the Planning capability $\models_{plan}$.

$$(\mathbf{PI}) \qquad \frac{\langle KB, Goals, Plan\rangle \qquad SGs}{\langle KB, Goals', Plan'\rangle}\,\tau$$

where SGs is a non-empty set of goals selected for planning (see section 8), and

$Goals' = Goals \cup \bigcup_{G\in SGs} Subg(G)$

$Plan' = Plan \cup \bigcup_{G\in SGs} Pplan(G)$

where, for each $G \in SGs$, the sets $Subg(G)$ and $Pplan(G)$ are obtained as follows.

(i) **Mental goals**: let $\{G_1, \ldots, G_n\} \subseteq SGs$, $n \geq 0$, be the set of all mental goals in SGs. If $n > 0$, let

$$KB, Plan, \{G_1, \ldots, G_n\} \models_{plan} \quad \{\langle G_1, \mathcal{A}_1 s, \mathcal{G}_1 s\rangle \ldots \langle G_n, \mathcal{A}_n s, \mathcal{G}_n s\rangle\}$$

Then for each $i = 1, \ldots, n$,

(i.1) either $\mathcal{G}_i s = \mathcal{A}_i s = \bot$ and $Subg(G_i) = Pplan(G_i) = \{\}$,

(i.2) or $\mathcal{G}_i s \neq \mathcal{A}_i s \neq \bot$ and
$Subg(G_i) = \{\langle l[t], G_i, S\rangle | (l[t], S) \in \mathcal{G}_i s\}$, and
$Pplan(G_i) = \{\langle a[t], G_i, C, T\rangle | (a[t], T) \in \mathcal{A}_i s \wedge KB, a[t] \models_{pre} C\}$.

(ii) **Sensing goals**: for each sensing goal $G = \langle l[t], G', T\rangle \in SGs$,

- $Subg(G) = \{\}$, and
- $Pplan(G) = \langle sense(l[t]), G', C, T\rangle$,
  where $KB, sense(l[t]) \models_{pre} C$.

Basically, plan introduction adds to the current state a plan for each of the goals selected for planning. For each such goal G, a (partial) plan has two components: a set of (sub)goals represented by $Subg(G)$, and a set of actions represented by $Pplan(G)$. The new state is updated by adding each set $Subg(G)$ to the current set of goals and each set $Pplan(G)$ to the current plan.

In the case of *sensing* goals (case ii) no subgoal is added and only the corresponding sensing action needs to be added to the current plan. Notice that the capability $\models_{pre}$ is used in order to determine the preconditions of this sensing action.

In the case of a *mental* goal there are two possibilities: either the Planning capability cannot determine any plan for the goal (case i.1), in which case the goal does not contribute to the updating of the state (both $Subg(G)$ and $Pplan(G)$ are empty); or the Planning capability actually determines a plan for G (case i.2). In this case, the Planning capability returns a set of

timed literals, each associated with a (possibly empty) temporal constraint, and a set of timed operators, each associated with a (possibly empty) temporal constraint. For each timed literal, the corresponding goal structure (in which the parent goal G is recorded along with the timed literal and the temporal constraint) is built and added to the current set of goals. Analogously, for each timed operator the corresponding action structure is built and added to the current plan. Notice that the capability $\models_{pre}$ is needed to determine the preconditions of the actions being built.

Notice that if all the goals in SGs are mental goals and have already been planned for, then the state will not change, i.e. $Goals' = Goals$ and $Plan' = Plan$. However, as we will see in section 9, this transition will only be triggered when the set of goals SGs does not have this feature. Hence, it cannot be the case, in (i), that both $\mathcal{A}_i s = \{\}$ and $\mathcal{G}_i s = \{\}$ for some mental goal G_i which can still be planned for.

Our definition of $\models_{plan}$ does not currently allow to take into account preferences amongst all possible plans for a goal. If the preferences of a computee over plans were allowed to change over time then, in the plan revision and goal revision transitions (see sections 7.9 and 7.8, respectively), actions in old, less preferred plans would need to be tidied up, by deleting them, in a more complex manner.

7.3 Reactivity (RE)

Reactivity revises the current state according to the results provided by the $\models_{react}$ capability, which determines goals and actions deriving from the reactive constraints.

$$(\mathbf{RE}) \qquad \frac{\langle KB, Goals, Plan\rangle}{\langle KB, Goals', Plan'\rangle}\,\tau$$

where, if $\langle KB, Goals, Plan\rangle \models_{react} \mathcal{G}s, \mathcal{A}s$, then

$$Goals' = Goals \cup \{\langle l[t], l'[t'], Tc\rangle | (l[t], l'[t'], Tc) \in \mathcal{G}s\}$$

$$Plan' = Plan \cup \{\langle a[t], l'[t'], C, Tc\rangle | (a[t], l'[t'], Tc) \in \mathcal{A}s \wedge KB, a[t] \models_{pre} C\}.$$

The $\models_{react}$ capability returns a set of goals and a set of actions deriving from the firing of reactive constraints. The goals are added to the current set of goals, and the actions are added to the current plan, each equipped with the corresponding preconditions determined by the $\models_{pre}$ capability. The parent of each newly added action and goal is determined by the reactive capability itself.

Notice that, if the $\models_{react}$ capability returns an empty set of goals and an empty set of actions, the state is unchanged (i.e. $Goals' = Goals$ and $Plan' = Plan$).

7.4 Sensing Introduction (SI)

The *Sensing Introduction* transition allows a computee to explicitly add to its current plan a set of sensing actions in order to check whether or not some preconditions of other actions in $Plan$ are satisfied.

$$(\mathbf{SI}) \qquad \frac{\langle KB, Goals, Plan\rangle \qquad SPs}{\langle KB, Goals, Plan'\rangle}\,\tau$$

where SPs is a non-empty set of preconditions of actions, associated with the parent of these actions, selected for sensing (see section 8), and

$$Plan' = Plan \cup \{\langle sense\_precondition(c[t'], G, D, Tc\rangle \mid (c[t], G) \in SPs\}$$

where, for each $(c[t], G)$,

(i) $Tc = (t' < t)$, and

(ii) $KB, sense\_precondition(c[t']) \models_{pre} D_i$.

Intuitively, SPs is a set of preconditions of actions in $Plan$, together with the parents of those action. Those parents are used as parents of the sensing actions added to $Plan$ by this transition. Condition (i) imposes that the actual sensing action for a certain precondition c should be executed before the action of which c is a precondition is executed. On the other hand, condition (ii) amounts at determining (if any) the preconditions for the sensing action. Depending on KB_{plan}, it might be that $KB, sense\_precondition(c[t]) \models_{pre} \emptyset$, for some or all c. Indeed, this will be the case if the sensing actions are not explicitly represented in KB_{plan}.

7.5 Passive Observation Introduction - (POI)

The *Passive Observation Introduction* transition updates the current knowledge base of the computee by adding new observed facts which derive from changes in the environment.

$$(\textbf{POI}) \qquad \frac{\langle KB, Goals, Plan\rangle}{\langle KB', Goals, Plan\rangle}\,\tau$$

where, if $\models^{\tau}_{Env} l_1 \wedge \ldots \wedge l_n, c_1 : a_1[\tau_1] \wedge \ldots \wedge c_k : a_k[\tau_k]$, $n, k \geq 0$, either $n > 0$ or $k > 0$, each l_i being a fluent $g_i[\_]$ or the negation of a fluent $\neg g_i[\_]$, each c_j being the name of a computee and each $a_j[\tau_j]$ being a timed action operator,

$$\begin{aligned} KB'_0 = KB_0 \cup \{&observed(l_1, \tau), \ldots, observed(l_n, \tau)\} \\ \cup \{&observed(c_1, a_1[\tau_1], \tau), \ldots, observed(c_k, a_k[\tau_k], \tau)\} \end{aligned}$$

For each fluent $l[\_]$ which the computee passively observes to hold in the environment at the current time, a corresponding observation is added to the knowledge base. For each action $a[\_]$ which the computee passively observes other computees have performed, a corresponding observation is added to the knowledge base. Notice that the observation records both the time at which the action was performed and the (current) time of observation.

A passive observation may be seen as the reaction of the computee to some (unexpected or unpredictable) event (e.g. an interrupt) which happens in the environment. The transition allows the computee to passively absorb or incorporate the observable effects of these events (interrupts) into its knowledge base.

7.6 Active Observation Introduction - (AOI)

Similarly to (POI), the *Active Observation Introduction* transition allows the computee to update its current knowledge base by possibly adding new observations from the environment. Differently from (POI), however, the computee is deliberately looking for some properties to hold in the environment.

$$(\mathbf{AOI}) \qquad \frac{\langle KB, Goals, Plan\rangle \qquad SFs}{\langle KB', Goals, Plan\rangle}\,\tau$$

where $SFs = \{l_1[t_1], \ldots, l_n[t_n]\}$, $n > 0$, is a set of fluents selected for being actively sensed (see section 8) and

$$KB'_0 = KB_0 \cup \bigcup_{i=1,\ldots,n} \mathcal{S}_i$$

where, for each $i = 1, \ldots, n$:

- $\mathcal{S}_i = \{observed(l_i[t_i], \tau)\}$ if $\models^{\tau}_{Env} l_i[\tau]$
- $\mathcal{S}_i = \{observed(\neg l_i[t_i], \tau)\}$ if $\models^{\tau}_{Env} \neg l_i[\tau]$
- $\mathcal{S}_i = \{\}$ if neither $\models^{\tau}_{Env} l_i[\tau]$ nor $\models^{\tau}_{Env} \neg l_i[\tau]$

For each fluent the computee is checking in the environment, there may be three possibilities: either the fluent holds in the environment at the current time (first bullet), or the negation of the fluent holds (second bullet), or the environment has no evidence for the fluent nor for its negation (last bullet). In the latter case, no update is made to the knowledge base as far as the fluent is concerned.

7.7 Action Execution (AE)

Action Execution is the transition which caters for actually performing actions in plans. Its effect amounts at recording in KB_0 the fact that certain actions have been executed and, in the case of sensing actions, the effect of sensing.

$$(\mathbf{AE}) \qquad \frac{\langle KB, Goals, Plan\rangle \qquad SAs}{\langle KB', Goals, Plan\rangle}\,\tau$$

where SAs is a non-empty set of actions selected for execution (see section 8), and for each $A \in \mathcal{A}s$, $A = \langle a[t], G, C, Tc\rangle$ such that $\{t = \tau\} \models Tc$

(i) If A is not a sensing action then

$$KB'_0 = KB_0 \cup \{executed(a[t], \tau)\}.$$

(ii) if A is a sensing action, namely if $A = p(l[t])$ where $p = sense$ or $p = sense\_precondition$, then:

$KB'_0 = KB_0 \cup \{executed(p(l[t]), \tau)\} \cup \mathcal{S}$

where:

- $\mathcal{S} = \{observed(l[t], \tau)\}$ if $\models^{\tau}_{Env} l[\tau]$
- $\mathcal{S} = \{observed(\neg l[t], \tau)\}$ if $\models^{\tau}_{Env} \neg l[\tau]$
- $\mathcal{S} = \{\}$ if neither $\models^{\tau}_{Env} l[\tau]$ nor $\models^{\tau}_{Env} \neg l[\tau]$

First of all, notice that, given the set of actions selected for execution, only actions whose temporal constraints are satisfied by the current time are actually executed. The execution of the action is recorded in KB_0. In the case of non-sensing actions (i.e. physical or communication actions) nothing else needs to be updated in the state. In the case of a sensing action, the actual effect of sensing is added, if any, to the current knowledge base, similarly to what is done in the case of active observation introduction.

Also, notice that we are not addressing, in the definition of (AE) above, the problem of actually *performing* the selected actions so that other computees may be able to observe them. This issue will have to be addressed by any concrete realisation of the model.

Finally, note that actions performed by (AE) are not deleted from the *Plan*. Indeed, their execution might not have been successful and thus it might need to bee repeated at a later stage. We will see below how the *Plan Revision* transition takes care of the removal from *Plan* of any action that has been executed successfully. The success of an action is measured in terms of the achievement of the goal that the action has been introduced for, namely its parent.

7.8 Goal Revision (GR)

The *Goal Revision* transition caters for revising the state by keeping only those goals which are still worth achieving.

$$(\mathbf{GR}) \qquad \frac{\langle KB, Goals, Plan\rangle}{\langle KB, Goals', Plan\rangle}\,\tau$$

where $Goals'$ is such that for each $G = \langle l[t], G', Tc\rangle \in Goals'$

(i) either $G' = \bot$ or $G' \in Goals$, and

(ii) there is no total valuation σ such that $\sigma \models Tc \wedge KB \models_{TR} l[t]\sigma$, and

(iii) if G is a mental goal, there exist $\mathcal{A}s \neq \bot, \mathcal{G}s \neq \bot$ such that

$$KB, Plan, \{G\} \models_{plan} \{\langle G, \mathcal{A}s, \mathcal{G}s\rangle\}, \text{ and}$$

(iv) there exists a total valuation σ such that $\sigma(t) = \tau'$, $\tau' > \tau$ and $\sigma \models Tc$.

First of all, all goals kept in the new state are either top-level goals or descendants of goals themselves kept in the new state (case i). Furthermore, a goal is kept if it has not been achieved yet (case ii). Moreover, a mental goal is kept in the new state if there is still a "viable" plan for it, i.e. it can still be planned for (case iii). Finally, only goals whose time has not run out, i.e. their temporal constraints are still satisfiable, are kept in the new state (case iv).

It is possible to introduce a further *Goal Revision capability* which takes into account some heuristics for goal revision, and to incorporate such heuristics (via the new capability) within the (GR) transition given earlier (see later in section 11).

7.9 Plan Revision (PR)

Similarly to Goal Revision, the *Plan Revision* transition caters for revising the state by keeping only those actions in the plan which are still relevant and which can still be executed.

$$(\mathbf{PR}) \qquad \frac{\langle KB, Goals, Plan\rangle}{\langle KB, Goals, Plan'\rangle}\,\tau$$

where $Plan'$ is such that for each $\langle a[t], G, C, Tc\rangle \in Plan'$:

(i) $G \in Goals$ or $G = \bot$, and

(ii) there exists a total valuation σ for the (implicitly existentially quantified) variables in Tc and t such that $\sigma \models Tc \wedge t > \tau$, and

(iii) if C is a non empty conjunction of preconditions, then there exists a total valuation σ such that $\sigma \models Tc \wedge t \geq \tau$, and $KB \models_{TR} C\sigma$

(iv) if $a[t] = sense\_precondition(C)$ then there exists an action

$$\langle a'[t'], G', C, Tc'\rangle \in Plan'.$$

First of all, an action from the original $Plan$ is kept in the current state (case i) only if it belongs to a plan for one of the current goals (i.e. the goal which generated that action is still in the current set of goals). Furthermore, the time of the action has not run out yet (case ii), i.e. the temporal constraints associated with it can still be satisfied in the future, and the preconditions of the action, if any, are still satisfiable (case iii). Finally, an action which has been introduced to sense the preconditions of another action, should be kept if the latter is still an action in $Plan'$ (i.e. it still belongs to the current plan.)

As for (GR), we may incorporate within this transition some heuristics for plan revision, via a further *Plan Revision capability* (see section 11 for more details).

8 Selection functions

Some of the transitions given in the earlier section take, in addition to a state, some other input, as indicated. We will see, in the next section 9, that such additional inputs are selected by means of *selection functions*, defined here. These selection functions will play an important role in the cycle theories of computees that control its operational behaviour.

The selection functions are *goal selection*, *action selection*, *fluent selection* and *precondition selection*, and they are all defined as mappings of the form

$$m : States \times Time\,Constants \rightarrow Sets$$

from the set of all possible states of the computee and the set of all possible time-points to the set of all possible sets of items of interest. Depending on the concrete selection function, these items are

- actions in $Plan$,
- goals in $Goals$,
- fluents occurring anywhere in $Goals$,
- preconditions of actions in $Plan$.

We define two classes of selection functions. The first class, called *core selection functions*, are fixed in the model [11], while the second class, of *heuristic selection functions*, are variable and can be chosen differently for each computee. As we will see below, the core selection functions only select goals (for goal selection) and actions (for action selection), preconditions (for precondition selection) and fluents (for fluent selection) that still have a chance of "success" (that have not become invalid in some way), for example, they have not yet timed out [12]. On the other hand, the heuristic selection functions are used to complement the core selection functions in order to capture different behaviours for different computees. For example, if we want a computee which is timely, then its heuristic selection functions will select goals/actions/fluents according to their urgency.

Hence the core selection functions are used to indicate items that are viable selections, whereas the heuristic selection functions are used to decide which of all candidate items (as chosen by the core selection functions) are preferred and thus effectively selected. Items chosen by the heuristic selection functions are items that are preferable, in being selected, over those that are not. The actual selection of items preferred according to the heuristic selection functions will depend on the semantics of the cycle theories we will see in section 9.

8.1 Core selection functions

8.1.1 Action selection

Intuitively the action selection function only selects actions that are not yet timed out, are not redundant and are still "useful" to perform.

This is a function c_{AS} from the set of all possible $States$ and the set of all possible $TimeConstants$ to the set of all actions in $Plan$.

Informally, the set of conditions for core action selection is as follows. Given a state $S = \langle KB, Goals, Plan\rangle$ and a time-point τ, the set of all actions selected by c_{AS} is the set of all actions A in $Plan$ such that at time τ:

1. A is not timed out,
2. no ancestor (except for $\bot$) of A in $Goals$ is satisfied in the state S,
3. no ancestor or sibling of A in $Goals$ and $Plan$ is timed out,
4. no precondition of A is known to be false in the state S,
5. there exists no goal G' with no children in $Goals$ that is a sibling of A or that has an ancestor in common with A (except for $\bot$) such that there exists no plan for G'.

Formally, given a state $S = \langle KB, Goals, Plan\rangle$ and a time-point τ, the set of all actions selected by c_{AS} is the set of all actions

$$A = \langle a[t], G, C, Tc\rangle \in Plan$$

such that:

[11] We note that in principle these could be changed, without the need for adjusting any other component of the model. However, we believe that the concrete choices for the core selection functions that we give below, form a basis for any choice of core selection functions that would extend them.

[12] Informally, an action/goal has timed out if there is no possible valuation of its temporal constraint at or after the current time.

1. there exists a total valuation σ for the variables in Tc and t such that $\sigma \models Tc \wedge t > \tau$,
2. there exists no $G' = \langle l[t'], G^*, Tc' \rangle \in Goals$, $G^* \neq \bot$, such that
   - $G^* = G$ or $G^* \in ancestors(G, Goals)$
   - there exists a total valuation σ for the variables in Tc' and t' such that $\sigma \models Tc' \wedge t' \leq \tau$ and $KB \models_{TR} l[t']\sigma$,
3. there exists no $A' = \langle a'[t'], G^*, C', Tc' \rangle \in Plan$, and there exists no $G' = \langle l[t'], G^*, Tc' \rangle \in Goals$ such that
   - $G^* = G$ or $G^* \in ancestors(G, Goals)$, Tc',
   - there exists no total valuation σ for the variables in Tc' and t' such that $\sigma \models Tc' \wedge t' > \tau$,
4. let $C = l_1[t_1] \wedge \ldots \wedge l_n[t_n]$; if $n > 0$, then for every $i = 1, \ldots, n$ there exists a total valuation σ for the variables in Tc, t and t_i such that $\sigma \models Tc$, $KB \models_{TR} l_i[t_i]\sigma$
5. there exists no $G' = \langle l[t'], G^*, Tc' \rangle \in Goals$ such that
   - there exists no goal $G'' \in Goals$ whose parent is G',
   - $G^* = G$ or $G^* \in ancestors(G, Goals)$,
   - $KB, \{G'\} \models_{plan} \{(G', \bot, \bot)\}$. [13]

8.1.2 Goal selection

Intuitively, the goal selection function only selects goals that are not already satisfied or goals that have not yet timed out.

Goal selection is a function c_{GS} from the set of all possible $States$ and the set of all possible $TimeConstants$ to the set of all goals in $Goals$.

Informally, the core set of conditions for goal selection is as follows. Given a state $S = \langle KB, Goals, Plan \rangle$ and a time-point τ, the set of all goals selected by c_{GS} is the set of all goals G^+ in $Goals$ such that, at time τ:

1. G^+ is not timed out,
2. no ancestor (except for $\bot$) or sibling of G^+ in $Goals$ and $Plan$ is timed out,
3. no ancestor of G^+ in $Goals$ is satisfied in the state S,
4. there is a (partial) plan for G^+,
5. G^+ has no children and there exists no goal G' with no children in $Goals$ that is a sibling of G^+ or that has a common ancestor (except for $\bot$) with G^+ such that there exists no plan for G'.

[13] Recall that $KB, \{G'\} \models_{plan} \{(G', \bot, \bot)\}$ means that there is no "viable" plan for G'.

Formally, given a state $S = \langle KB, Goals, Plan\rangle$ and a time-point τ, the set of all goals selected by c_{GS} is the set of all goals

$$G^+ = \langle l[t], G, Tc\rangle \in Goals$$

such that:

1. there exists a total valuation σ for the variables in Tc and t such that $\sigma \models Tc \wedge t > \tau$,
2. there exists no $A' = \langle a'[t'], G, C', Tc'\rangle \in Plan$, and there exists no $G' = \langle l[t'], G^*, Tc'\rangle \in Goals$ such that
   - $G^* = G$ or $G^* \in ancestors(G, Goals)$,
   - there exists no total valuation σ for the variables in Tc' and t' such that $\sigma \models Tc' \wedge t' > \tau$,
3. there exists no $G' = \langle l'[t'], G^*, Tc'\rangle \in Goals$, $G^* \neq \bot$, such that
   - $G^* = G$ or $G^* \in ancestors(G, Goals)$,
   - there exists a total valuation σ for the variables in Tc' and t' such that $\sigma \models Tc' \wedge t' \leq \tau$ and $KB \models_{TR} l'[t']\sigma$,
4. there exists $As \neq \bot, Gs \neq \bot$ such that $KB, \{G^+\} \models_{plan} \{(G^+, As, Gs)\}$,
5. there exists no $\langle \_, G^+, \_\rangle \in Goals$, $\langle \_, G^+, \_, \_\rangle \in Plan$ and there exists no $G' = \langle l'[t'], G^*, Tc'\rangle \in Goals$ such that
   - there exists no goal $G'' \in Goals$ whose parent is G',
   - $G^* = G$ or $G^* \in ancestors(G, Goals)$ for some t' and Tc',
   - $KB, \{G'\} \models_{plan} \{(G', \bot, \bot)\}$.

8.1.3 Fluent selection

Fluent selection is a function c_{FS} from the set of all possible *States* and the set of all possible *TimeConstants* to the set of all fluents. Informally, a core set of conditions for fluent selection is as follows. Given a state $S = \langle KB, Goals, Plan\rangle$ and a time-point τ, the set of all (timed) fluents selected by c_{FS} is the set of all (timed) fluents F such that:

- F or its negation is one of the effects of some action that has recently been executed.

Note that F selected by c_{FS} may not occur in *Goals* but could be some other (observable) effect of the executed action not necessarily the same as the goal that the action aims to achieve.

Formally, the set of all (timed) fluents selected by c_{FS} is the set of all (timed) fluents F such that:

- either $KB_{plan} \models_{LP} initiates(A, F, T)$ or $KB_{plan} \models_{LP} terminates(A, F, T)$ and
  $KB_0 \models_{LP} executed(A, T)$ and
  $\tau - \epsilon < T < \tau$,
  where ϵ is a sufficiently small number.

8.1.4 Precondition selection

Intuitively precondition selection selects all those preconditions not yet known to hold or not to hold of those actions in $Plan$ that would be selected by the action selection function.

Precondition selection is a function c_{PS} from the set of all possible $States$ and the set of all possible $TimeConstants$ to the set of all preconditions of actions in $Plan$, each associated with a goal in $Goals$.

Informally, a core set of conditions for precondition selection is as follows. Given a state $S = \langle KB, Goals, Plan \rangle$ and a time-point τ, the set of all preconditions (of actions) selected by c_{PS} is the set of all pairs (C, G) of preconditions C and goals G such that:

1. there exists an action A in $Plan$ such that C is a precondition of A and G is the parent of A,
2. C is not known to be true in the state S,
3. $A \in c_{AS}(S, \tau)$.

Formally, given a state $S = \langle KB, Goals, Plan \rangle$ and a time-point τ, the set of all preconditions of actions selected by c_{PS} is the set of all pairs (C, G) of preconditions C and goals G such that:

1. there exists $A = \langle a[t], G, Cs, Tc \rangle \in Plan$ such that C is a conjunct in Cs,
2. there exists no total valuation σ for the variables in Tc and t such that $\sigma \models Tc$ and $KB \models_{TR} C\sigma$,
3. $A \in c_{AS}(S, \tau)$.

8.2 Heuristic selection functions

In this section we will provide a catalogue of possible requirements of the heuristic selection functions. These requirements are added to the requirements posed by the core selection functions. Namely, the set of items selected by the heuristic functions are (possibly proper) subsets of the sets of items selected by the core selection functions.

The catalogue we provide in this section is by no means exhaustive, and it aims at illustrating the range of possibilities available in constructing computees. In principle, any combination of these characteristics is allowed.

As mentioned above, the heuristic selection functions will play a fundamental role in defining the cycle theories (see section 9), thus determining the range of operational behaviours that computees can exhibit by adopting different cycle theories. In the third phase of the project, both within WP5 and WP6, we will study the correlation between choices of heuristic functions and behaviours/properties of computees.

8.2.1 Heuristic action selection

Heuristic action selection is a function h_{AS} from the set of all possible $States$ and the set of all possible $TimeConstants$ to the set of all actions in $Plan$. Informally, we will assume that given a state $S = \langle KB, Goals, Plan \rangle$ and a time-point τ, the set of all actions selected by h_{AS} is the set of all actions A such that A is selected by c_{AS} and in addition any (subset) of the following criteria holds for A:

- A is an urgent action in $Plan$; in the sequel we will refer to a heuristic action selection function which selects the most urgent actions as h^{u}_{AS};
- all the preconditions of A are known to hold in S, providing a measure for predicting the successful execution of the action; in the sequel we will refer to a heuristic action selection function which selects actions whose preconditions are known to hold as h^{pre}_{AS};
- the computee has a "high" level of confidence that executing A will lead to achieving its intended effects; a measure of this confidence could be provided by counting the number of attempted but failed executions of the action; in the sequel we will refer to a heuristic action selection function which does not select actions tried once and failed as h^{fail}_{AS};
- A belongs to the same plan as the actions previously executed, if any; here we can interpret two actions as belonging to the same plan if they have a common ancestor (except for $\perp$); in the sequel we will refer to a heuristic action selection function which selects actions belonging to the same plan as actions previously executed as h^{sp}_{AS};
- there exists no other earlier action in $Plan$ as yet unexecuted that is linked (i.e. have a common ancestor apart from $\perp$) to A.

8.2.2 Heuristic goal selection

Heuristic goal selection is a function h_{GS} from the set of all possible $States$ and the set of all possible $TimeConstants$ to the set of all goals in $Goal$. Informally, we will assume that given a state $S = \langle KB, Goals, Plan \rangle$ and a time-point τ, the set of all goals selected by h_{GS} is the set of all goals G such that G is selected by c_{GS} and in addition any (subset) of the following criteria holds for G:

- G is the most urgent goal in $Goals$; in the sequel we will refer to a heuristic goal selection function which selects the most urgent goals as h^{u}_{GS};
- G belongs to the same plan as the actions previously executed, if any; in the sequel we will refer to a heuristic goal selection function which selects goals belonging to the same plan as actions previously executed as h^{sp}_{GS};
- there exists no other earlier goal in $Goals$ that is linked (i.e. have a common ancestor apart from the root) to G and has no children.

8.2.3 Heuristic fluent selection

Heuristic fluent selection is a function h_{FS} from the set of all possible $States$ and the set of all possible $TimeConstants$ to the set of fluents. Informally, we will assume that given a state $S = \langle KB, Goals, Plan \rangle$ and a time-point τ, the set of all fluents selected by h_{FS} is the set of all fluents F such that F is selected by c_{FS} and in addition criterion holds for F:

- F is the effect of an action that has already been tried a given number of times (more than once) unsuccessfully; the computee had tried the action again, and wants to check its effects soon after retrying.

8.2.4 Heuristic precondition selection

Heuristic precondition selection is a function h_{PS} from the set of all possible $States$ and the set of all possible $TimeConstants$ to the set of all pairs consisting of a precondition of an action in the $Plan$ and a goal in the $Goals$ of the state. Informally, we will assume that given a state $S = \langle KB, Goals, Plan \rangle$ and a time-point τ, the set of all pairs selected by h_{PS} is the set of all pairs (C, G) such that (C, G) is selected by c_{PS} and in addition the following criterion holds:

- there exists $A = \langle a[t], G, Cs, \_ \rangle \in Plan$ such that A would be selected by h_{AS} and C is a conjunct in Cs.

8.3 Selection functions and revision transitions

Note that there exists a strong link between the core selection functions and the revision transitions (PR) and (GR). Indeed, none of the items selected by the selection functions will ever be deleted by the revision transitions (PR) and (GR), if these are applied before the selections.

Note also that the revision transitions (PR) and (GR) could be extended to incorporate heuristics for goal and plan revision. These heuristics would correspond to the heuristic selection functions.

8.4 Resource-boundness

Selection functions typically select a set of items, i.e. a set of goals, actions etc, rather than one such item at a time. These sets are then intended to be used as input to the appropriate transitions. In many cases though, the computee may not have the resources (e.g. time) to execute a transition for all the selected items. For example, computees may not have enough resources to execute more than a certain number of actions at a time, or plan for a certain number of goals at a time.

Resource-boundness considerations thus may force the computee to restrict its selections to sets of items for which it has sufficient resources.

The issue of resource-boundness depends strongly on the problem application domain but nevertheless we can accommodate this within the KGP model in a general way as an additional selection criterion. One way to do this is to use a set of *projection functions* on the output of the selection functions which separate the items selected into subsets each one of which is within the resource-bounded capabilities of the computee. These projection functions are formalised as mappings of the form:

$$r : Sets \rightarrow 2^{Sets}$$

from the set of all possible sets of items to its power-set where given $A \in Sets$

- if $A' \in r(A)$ then $A' \subseteq A$,
- if A_i, A_j $(i \neq j) \in r(A)$ then $A_i \cap A_j = \emptyset$,
- $\bigcup_{i:A_i \in r(A)} A_i = A$.

The detailed definition of these functions will depend on the problem domain and the resource cost of the various items.

These projection functions will be composed with the selection functions to split the set of selected items into subsets that fulfil the resource-boundedness criteria.

Finally, we note that an alternative way to handle the issue of resource-boundness is to include this within the selection functions rather than apply it a-posteriori on the items selected by the selection functions. This would be more appropriate for example in the case where even some singleton items could not be "executed" due to the limited resources that the computee has. In this case, we could use the resource-boundness criterion within the core selection function to avoid selecting such items. Furthermore, resource-boundness can also be used inside the heuristic selection functions as a heuristic selection criterion where for example the computee selects according to resources needed by the selected items.

9 Cycles of behaviour of computees

So far our model of computees has defined in isolation its internal state and the possible individual state transitions that a computee may have, in terms of the reasoning and sensing capabilities of the computee. In order to complete this model we need to specify how a computee operates via the execution of its transitions. The operation of a computee will be understood in terms of sequences of transitions. Such sequences can be obtained from *fixed cycles* of operation of computees as in most of the literature on agents. Alternatively, such sequences can be obtained via fixed cycles together with the possibility of selecting amongst such fixed cycles according to some criteria e.g. the type of external environment in which the computee will operate (see recent work of [DdBD$^+$02]) Yet another possibility is to specify the required operation via more versatile *cycle theories*. Whereas fixed cycles can be seen as providing a conventional procedural control, cycle theories act as declarative control theories specifying requirements on the operation of computees. These cycle theories will form the basis for specifying the operation of computees.

Fixed cycles can be defined simply as sequences of transitions, to be repeatedly applied. A cycle theory will be defined as a logic program with priorities over rules with its argumentation based semantics (see section 3.3).

The role of the cycle theory is to control the sequence of the internal transitions that the computee does in its life. It regulates these "narratives of transitions" according to certain requirements that the designer of the computee would like to impose on the operation of the computee where any sequence of transition rules can be allowed in the "life" of a computee by a cycle theory. Thus, whereas a fixed cycle can be seen as a restrictive and rather inflexible catalogue of allowed sequences of state transitions (possibly under pre-defined specific conditions), the cycle theory is there to identify *preferred patterns* of sequences of transitions and in this way regulate in a flexible way the operational behaviour of the computee.

The aim of the cycle theory is therefore twofold. On the one hand it aims at *controlling* and at helping to *characterise* the operational behaviour of a computee, by giving a form of *intelligent control* that is rooted in the knowledge of the computee and is responsive to changes in its external environment. On the other hand the use of different selection functions (see section 8) and in particular the heuristic selection functions and other such conditions, using criteria such as urgency, utility and significance, aims at obtaining different kinds of computee. Examples of types of computees are timely computees, focused computees, impatient computees, cautious computees, etc. The formal definition of such "profiles of behaviour" for computees and the formal verification that certain choices of selection functions and cycle theories give these

profiles will be studied in the third phase of the project, in workpackage WP5. In section 9.3, we give concrete examples of some such profiles that can be modelled within our approach.

We will assume that the operation of a computee follows a simple top level loop of the kind "Receive Information - Respond", allowing the computee to constantly aim at achieving its own goals while being alert to the environment and its changes. The computee receives information from its environment via the transitions of Passive Observation Introduction (POI) and Active Observation Introduction (AOI). POI is the only transition that the computee cannot control itself. A cycle theory will interpret POI as a form of *interrupt* in the operation of the computee. The new information, obtained via POI, can change the decision for the next transition to be executed, as specified and thus regulated by the cycle theory itself. A fixed cycle, instead, cannot be interrupted, and it will consider information as obtained via POI only when POI's turn will come within the cycle.

Both for fixed cycles and cycle theories, we will assume that the operation of a computee will start from some *initial state*. This can be seen as the state of the computee at its "birth". The state then evolves via the transitions, as commended by the fixed cycle or cycle theory. For example, the initial state of the computee could have an empty set of goals and an empty set of plans, or some designer-given goals and an empty set of plans. In the sequel, we will indicate the given initial state as S_0.

In this section, we will denote by I the set $I = \{GI, PI, RE, SI, POI, AOI, AE, GR, PR\}$, namely I is the set of all possible labels (indexes) of transitions, as given in section 7.

Finally, in this section we will assume the existence of a *clock* (external to the computee) whose task is to mark the passing of time. Each clock tick could be modelled as a social event, as in deliverable D5. The clock is responsible for deciding the time at which the transitions are applied.

9.1 Fixed cycles

In this section, any transition of the computee (as defined in section 7) of the form

$$(\mathbf{L}) \qquad \frac{S \quad X}{S'}\,\tau$$

where S, S' are states of the computee, and X may be empty, will be denoted by an atom

$$T_L(S, X, S', \tau)$$

Whenever the time τ of the transition is not relevant to the discussion, this will be written simply as

$$T_L(S, X, S')$$

Also, we will mostly incorporate within transitions the selection, via appropriate selection functions, of the conditions X prior to the application of a transition, and write

$$T_L(S, X, S', \tau)$$

instead of

$$f(S, \tau) = X,\ T_L(S, X, S', \tau)$$

where f is the appropriate selection function. [14] Indeed, for fixed cycles, the role of selection functions is exclusively to select the inputs for the appropriate transition when the turn of the transition comes within the fixed cycle. (Instead, as we will see in the next section, the role of selection functions for cycle theories is to help deciding which transition should be applied next.) Thus, for fixed cycles, selection functions and (appropriate) transitions are strongly coupled. In particular,

$T_{PI}(S, X, S', \tau)$ will stand for $f_{GS}(S, \tau) = X$, $T_{PI}(S, X, S', \tau)$
$T_{SI}(S, X, S', \tau)$ will stand for $f_{PS}(S, \tau) = X$, $T_{SI}(S, X, S', \tau)$
$T_{AOI}(S, X, S', \tau)$ will stand for $f_{FS}(S, \tau) = X$, $T_{AOI}(S, X, S', \tau)$
$T_{AE}(S, X, S', \tau)$ will stand for $f_{AS}(S, \tau) = X$, $T_{AE}(S, X, S', \tau)$

namely, PI must be applied immediately after its inputs have been selected by f_{GS}, SI must be applied immediately after its inputs have been selected by f_{PS}, and so on.

In addition, in this section we will mostly drop the conditions X, and represent a transition simply as

$$T_L(S, S', \tau)$$

Then, a *fixed cycle* is a fixed sequence of transitions of the form

$$T_1, \ldots T_n$$

where $i \in I$, $i = 1, \ldots, n$, $n \geq 2$. This gives an *operational trace* of the computee of the form

$T_1(S_0, S_1, \tau_1), T_2(S_1, S_2, \tau_2), \ldots, T_n(S_{n-1}, S_n, \tau_n)$,
$T_1(S_n, S_{n+1}, \tau_{n+1}), \ldots, T_n(S_{2n-1}, S_{2n}, \tau_{2n})$,
$\ldots$

where S_0 is the initial state of the computee, and each τ_i is given by the clock of the system at the time that T_i is applied. Note that we assume that $\tau_i < \tau_j$, for $i < j$.

The classical "observe-think-act" cycle [KS99] (for a rather limited computee) can be represented in our approach as the cycle:

$$T_{POI}, T_{RE}, T_{PI}, T_{AE}.$$

A more sophisticated version of the "observe-think-act" cycle, incorporating goal decision and sensing actions, may be as follows:

$$T_{POI}, T_{RE}, T_{GI}, T_{PI}, T_{AOI}, T_{AE}, T_{PR}, T_{GR}.$$

A purely reactive computee (which has very limited knowledge) can execute the following cycle:

$$T_{POI}, T_{RE}, T_{AE}.$$

Note that POI is interpreted here as a transition which is under the control of the computee, but is passive in the sense that, via such transition, the computee does not look for anything

[14] In this subsection, we use a neutral symbol f for selection functions. f could be either a core selection function c or a heuristic selection function h, as the distinction amongst the two types of selection functions does not play a role for fixed cycles.

special. Rather, it opens its reception channel and waits for some input. Below, in section 9.2, we will see a different interpretation of POI as an interrupt.

Note that, although fixed cycles such as the above are quite restrictive, they may be "sensible" in some circumstances. For example, the cycle for a purely reactive computee may be fine in an environment which is highly dynamic, whereas the more sophisticated version of the "observe-think-act" cycle may be appropriate in an environment with few and infrequent changes. A computee may then be equipped with a catalogue of fixed cycles, and a number of conditions on the environment to decide when to apply which of the given cycles. This would provide for a limited form of intelligent control, paving the way towards the more sophisticated and fully declarative control given in the next section. This would be in the spirit of [DdBD$^+$02].

9.2 Cycle theories

Our model does not propose any fixed cycle or cycles for a computee. Any sequence of the transition rules can be allowed in the "life" of a computee. As mentioned above the cycle theory is there to allow the computee to reason about preferred patterns of sequence of transitions and to give a mechanism of how such patterns may be selected by the computee. It can be viewed as a *preference policy* that will determine at each step the preferred next internal transition(s) to be executed. In effect, this policy ranks the various alternative transitions that can follow the current [15] transition in the operation of the computee so that the most preferred can be chosen.

A *cycle theory* is a logic program $\mathcal{T}_{cycle}$ with priorities over rules in the logic programming framework $LPwNF$ described in section 3.3. Such a logic program is a *meta*-program in that it reasons on the whole state of the computee. It is, however, of the same format as the object level knowledge componentKB_{GD} of KB, that is expressed within the same framework.

Concretely, $\mathcal{T}_{cycle}$ consists of three components:

- A *basic* part $\mathcal{T}_{basic}$ that determines the basic steps of operation by specifying the allowed unitary cycle-steps from one transition to another.
- An *interrupt* part $\mathcal{T}_{interrupt}$ that specifies the cycle-steps that can follow a POI, i.e. an interrupt with new information. These are viewed as re-initialisation steps for the cycle operation.
- A *behaviour* part $\mathcal{T}_{behaviour}$ that determines via preference rules on the alternatives given in the basic and interrupt parts the special characteristics of the operation (and thus behaviour) of the computee.

Below, in sections 9.2.2, 9.2.3 and 9.2.5, we give formal definitions and examples for the above $\mathcal{T}_{basic}, \mathcal{T}_{interrupt}, \mathcal{T}_{behaviour}$, respectively. In section 9.2.4 we define the theory $\mathcal{T}_{initial}$, in a format analogous to that for $\mathcal{T}_{basic}$ and $\mathcal{T}_{interrupt}$ for deciding which transition the computee should start with. In the sequel, we will denote by $\mathcal{T}_{cycle} = \mathcal{T}_{initial} \cup \mathcal{T}_{basic} \cup \mathcal{T}_{interrupt} \cup \mathcal{T}_{behaviour}$.

The cycle-steps in $\mathcal{T}_{basic} \cup \mathcal{T}_{interrupt}$ are rules of the form

[15] We will assume that the choice for the next transition depends only on the current transition and not on the longer history of the previous transitions. Note that this does not mean that information from the past operation of the computee is not used in deciding the next transition. Such information will be used as this is recorded in the state of the computee. Our assumption rather means that the only explicit reference to the (type of) transitions that the computee has carried out till now that is needed in order to make the choice for the next transition is that to the current transition.

$$T'(S', X', \tau) \leftarrow T(S, X, S', \tau', \tau), C(S', \tau, X')$$

sanctioning that, if at time τ, which is the time at which the current transition T has finished (having started at time τ'), the conditions C evaluated in the resulting state S' are satisfied, then transition T' should follow transition T and applied with inputs the state S' and the set of items X', if required. Note that evaluating the conditions C allows us to compute X' from S'. Below, except for section 9.2.1, we will write cycle-step rules in short as

$$T'(S', X') \leftarrow T(S, X, S'), C(S', \tau, X')$$

concentrating on the arguments of interest.

The rules in $\mathcal{T}_{initial}$ are of the form

$$T(S_0, X) \leftarrow C(S_0, \tau, X)$$

sanctioning that, if the conditions C are satisfied in the initial state S_0 at time τ, then the initial transition should be T, applied to state S_0 and input X, if required. Note that $C(S_0, \tau, X)$ may be empty, and $\mathcal{T}_{initial}$ might simply indicate a fixed initial transition T_1.

In the following section 9.2.1 we show how a cycle theory $\mathcal{T}_{cycle}$ induces the operational trace of the computees, defined in terms of sequences of transitions.

9.2.1 Operational Trace

As earlier, let us suppose that S_0 is the given initial state of the computee. In addition, until later in this section, let us suppose that the computee is given some *initial transition* T_1, that the computee will start to operate from. A natural choice for T_1 could be GI, namely the computee starts by deciding which goals to set for itself, if not already equipped with some goals by its designer.

Then, the *operational trace* given by $\mathcal{T}_{cycle}$ is a sequence of sequences of transitions, each of the form

$$T_1^j(S_0^j, X_1^j, S_1^j, \tau_1^j), \ldots, T_i^j(S_{i-1}^j, X_i^j, S_i^j, \tau_i^j), T_{i+1}^j(S_i^j, X_{i+1}^j, S_{i+1}^j, \tau_{i+1}^j), \ldots$$

(where each of the X^js may be empty) such that

- $j = 1, \ldots, n, \ldots$ gives the number of sequences for the computee, which is typically infinite;
- $S_0^1 = S_0$, namely the state from which the first sequence starts is the initial state;
- $T_1^1 = T_1$, namely the first transition in the first sequence is the given initial transition T_1;
- for each $j = 2, \ldots, n, \ldots$, $T_1^j = POI$, namely each sequence (except for the first one), starts with a POI transition;
- for each $j = 1, \ldots, n, \ldots$, if there exists a $j+1$th sequence then the jth sequence is finite, say
$$T_1^j(S_0^j, X_1^j, S_1^j, \tau_1^j), \ldots, T_{m_j}^j(S_{m_j-1}^j, X_{m_j}^j, S_{m_j}^j, \tau_{m_j}^j)$$
for some $m_j \geq 1$, and $S_0^{j+1} = S_{m_j}^j$, namely the initial state of the $j+1$th sequence is the final state of the jth sequence; the transition $T_{m_j}^j$ is referred to as *final* within the sequence;

- for each $j = 1, \ldots, n, \ldots$, if there exists a $j+1$th sequence then there exists a POI (interrupt) between times $\tau^j_{m_j-1}$ and $\tau^j_{m_j}$, namely a new sequence is only started because of the occurrence of a POI;
- τ^j_i is given by the clock of the system at the time that T^j_i is applied (with the property that $\tau^j_i < \tau^j_{i+i}$, and $\tau^j_{m_j} < \tau^{j+1}_1$, for each j, i), namely time increases;
- for each $j = 1, \ldots, n, \ldots$, $i < m_j$,

  $$T_{cycle} \wedge T^j_i(S^j_{i-1}, X^j_i, S^j_i, \tau^j_i, \tau^j_{i+1}) \models_{pr} T^j_{i+1}(S^j_i, X^j_{i+1}, \tau^j_{i+1})$$

  namely each (non-final) transition in a sequence is followed by the most preferred transition, as specified by rules with priorities in T_{cycle}.

Note that the definition of operational trace above does not impose that all POI are taken into account. Moreover, POI does not interrupt a transition that has already started being applied. Rather, the definition above imposes that POI is kept waiting until the transition that had already started has completed. In a more advanced execution model, this restriction could be relaxed, to allow for a POI to effectively interrupt the computation of the current transition, or for POI to be executed concurrently with the ongoing transition. In such a case we need to decide which of the partial information (if any) computed by the current transition should be kept in the state of the computee. Also, the given definition of operational trace prevents the concurrent execution of transitions, and a more advanced execution model could avoid this. We will discuss some of these issues further in section 11.

Also, note that, some transitions might leave the state of a computee unchanged. This might happen, for example, if the *Goals* and *Plan* of the computee are empty and GI (the only transition that makes sense in this state), does not introduce any new goals.

Further, note that, by means of the last condition above, because of the definition of $\models_{pr}$ and because of the assumption that all transitions are incompatible with each other (see below), we assume that at most one cycle-step is enabled at any time. This requirement imposes certain conditions on the form of the cycle theory, as we will see below in section 9.2.6.

Finally, note that assuming that the computee is given the initial transition T_1 may be restrictive. Indeed, for the computee to be truly intelligent, we want it to be able to decide which transition to start with, depending on its initial state and its environment. For example, it might be useful for the computee to start with a POI, if one occurs, or with GI, if goals and plan in the initial state are empty. In general, a computee may be equipped with a theory $\mathcal{T}_{initial}$ to decide the initial transition by reasoning. Then, $\mathcal{T}_{cycle} = \mathcal{T}_{initial} \cup \mathcal{T}_{basic} \cup \mathcal{T}_{interrupt} \cup \mathcal{T}_{behaviour}$. We will discuss $\mathcal{T}_{initial}$ further in section 9.2.4.

If $\mathcal{T}_{initial}$ is given, then the third bullet above becomes:

- $\mathcal{T}_{initial} \cup \mathcal{T}_{interrupt} \cup \mathcal{T}_{behaviour} \models_{pr} T^1_1(S_0, X^1_1)$.

9.2.2 The basic component: $\mathcal{T}_{basic}$

Each rule in any given $\mathcal{T}_{basic}$ is called a *cycle-step* rule and is of the form

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

where $i, k \in I, i, k \neq POI$. Any such rule specifies which transition T_k might follow a transition T_i. Note that cycle-step rules do not specify what might follow a POI transition (as $i \neq POI$).

This is done by the $\mathcal{T}_{interrupt}$ theory. Note also that cycle-step rules cannot indicate that a POI transition should follow any transition (as $k \neq POI$) since POI is the only transition not under the control of the computee.

The conditions $C_{i|k}$ in a cycle-step rule as the above are called *enabling conditions* as they determine when a cycle-step from the transition T_i to the transition 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. Hence such a rule is parameterised by X as well as the state S' resulting from the application of the currently finished transition T_i.

For example, the following cycle-step rule

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

expresses the possibility that an Action Execution transition (AE) can be followed by a Plan Introduction (PI) transition. The enabling conditions $C_{AE|PI}(S', \tau, Gs)$ determine the set of goals Gs that are to be planned for by the ensuing PI transition. Such goals are determined by a call to the core goal selection function c_{GS}, namely

$$C_{AE|PI}(S', \tau, Gs) \leftarrow Gs = c_{GS}(S', \tau), Gs \neq \{\}$$

We will see below that the heuristic goal selection function will be used within $\mathcal{T}_{behaviour}$. Also we remind the reader that these conditions may also contain the application of a projection function to take into account resource bounds, in which case we will have a collection of such rules, one for each subset of the selected goals Gs, as given by the appropriate projection function.

A full $\mathcal{T}_{basic}$ part of a cycle theory may contain the following cycle-step rules for deciding what might follow an AE transition:

$r_{AE|PI}(S', Gs) : T_{PI}(S', Gs) \leftarrow T_{AE}(S, As, S'), C_{AE|PI}(S', \tau, Gs)$
$r_{AE|AE}(S', As') : T_{AE}(S', As') \leftarrow T_{AE}(S, As, S'), C_{AE|AE}(S', \tau, As')$
$r_{AE|AOI}(S', Fs) : T_{AOI}(S', Fs) \leftarrow T_{AE}(S, As, S'), C_{AE|AOI}(S', \tau, Fs)$
$r_{AE|PR}(S') : T_{PR}(S') \leftarrow T_{AE}(S, S')$

Namely, AE could be followed by another AE, or by a PI, or by an AOI, or by a PR. Any other possibility, e.g. for GI to follow AE, is excluded within this particular $\mathcal{T}_{basic}$ theory.

The enabling conditions, $C_{AE|AE}(S', \tau, As')$, of the second cycle-step rule above determine the set of actions As' that are to be executed within the ensuing AE transition. Such actions are determined by a call to the core action selection function c_{AS}, namely

$$C_{AE|AE}(S', \tau, As') \leftarrow As' = c_{AS}(S', \tau), As' \neq \{\}$$

Similarly, the enabling conditions, $C_{AE|AOI}(S', \tau, Fs)$, of the third cycle-step rule above determine the set of fluents Fs that are to be sensed next within the ensuing AOI transition. Such fluents are determined by a call to the core fluent selection function c_{FS}, namely

$$C_{AE|AOI}(S', \tau, Fs) \leftarrow Fs = c_{FS}(S', \tau), Fs \neq \{\}$$

We will see that the heuristic action selection and fluent selection functions will be used within $\mathcal{T}_{behaviour}$.

A cycle-step rule in $\mathcal{T}_{basic}$ only determines what might follow a transition that is different from POI. The potential follow-ups of POI are determined by the $\mathcal{T}_{interrupt}$ part of the cycle theory, as given in the following section.

9.2.3 The interrupt component: $\mathcal{T}_{interrupt}$

The interrupt component of $\mathcal{T}_{cycle}$ is analogous, in syntax, to the basic component. However, each rule in $\mathcal{T}_{interrupt}$ specifies what might follow a POI transition, which acts as an interrupt. Concretely, each rule in $\mathcal{T}_{interrupt}$ is an *interrupt cycle-step rule* of the form

$$r_{POI|k}(S', X) : T_k(S', X) \leftarrow T_{POI}(S, S'), C_{POI|k}(S', \tau, X)$$

where $k \in I, k \neq POI$. In fact, it is reasonable to allow only the rules below:

$r_{POI|GI}(S') : T_{GI}(S') \leftarrow T_{POI}(S, S')$
$r_{POI|RE}(S') : T_{RE}(S') \leftarrow T_{POI}(S, S')$
$r_{POI|GR}(S') : T_{GR}(S') \leftarrow T_{POI}(S, S')$

These concrete interrupt cycle-steps have no enabling conditions, and thus in principle they allow for any of GI, RE and GR to follow POI. Part of the $\mathcal{T}_{behaviour}$ part of the cycle theory can contain priority rules amongst the rules of $\mathcal{T}_{interrupt}$ so that there exists a unique preferred transition to follow POI.

9.2.4 The initial component: $\mathcal{T}_{initial}$

The $\mathcal{T}_{initial}$ part of a cycle theory consists of rules of the form

$$r_{0|k}(S_0, X) : T_k(S_0, X) \leftarrow C_{0|k}(S_0, \tau, X)$$

such that $k \in I, k \neq POI$, and S_0 is an initial state of the computee.
Examples of such rules are the following.

$$r_{0|GI}(S_0) : T_{GI}(S_0) \leftarrow S_0 = \langle KB, Goals, Plan \rangle, Goals = \{\}$$

namely the computee should start with GI if it is equipped with no goals by its designer.

$$r_{0|PI}(S_0, Goals) : T_{PI}(S_0, Gs) \leftarrow Gs = c_{GS}(S_0, \tau), Gs \neq \{\}$$

namely the computee should start with PI if it has some goals, and PI is given as input (some of) those goals in the state S_0, as selected by the goal selection function, applied to S_0 at time τ.

The initial transition could also be POI linking then with $\mathcal{T}_{interrupt}$ as above.

9.2.5 The behaviour component: $\mathcal{T}_{behaviour}$

In the previous subsections we have defined the components $\mathcal{T}_{basic}$ and $\mathcal{T}_{interrupt}$ of $\mathcal{T}_{cycle}$. We are now going to define the fourth and last component $\mathcal{T}_{behaviour}$. Its main task is to specify local priorities over rules in $\mathcal{T}_{basic}$ and $\mathcal{T}_{interrupt}$ of the cycle theory so that this can decide, amongst all enabled cycle-steps, which one should be preferred. Also it has the task to enforce

that the interrupt component $\mathcal{T}_{interrupt}$ overrides the decisions on the next transition to apply as given by the basic component $\mathcal{T}_{basic}$. Note that, although we have implied so far that indeed the interrupt cycle-step rules always override the basic cycle-step rules, in a more general setting we can also allow in $\mathcal{T}_{behaviour}$ priorities across the basic and interrupt cycle-step rules so that in some cases the basic rules would be preferred. [16]

The general form of the rules in $\mathcal{T}_{behaviour}$ is

$$R^i_{k|l} : h\_p(r_{i|k}(S, X_k), r_{i|l}(S, X_l)) \leftarrow BC^i_{k|l}(S, X_k, X_l, \tau)$$

where $r_{i|k}$ and $r_{i|l}$ are (names of) rules in $\mathcal{T}_{basic} \cup \mathcal{T}_{interrupt}$. This rule says that at time τ after transition T_i if the conditions $BC^i_{k|l}$ hold then we prefer the next transition to be T_k over T_l.

The conditions BC are called *behaviour conditions* and determine when the preferences apply. These conditions depend on the state of the computee after T_i and on the parameters chosen in the two cycle-steps $r_{i|k}$ and $r_{i|l}$. In contrast with the enabling conditions of the cycle-step rules, behaviour conditions are *heuristic* conditions that we can choose appropriately in order to get different patterns of behaviour. These conditions are defined in terms of heuristic selection functions, where appropriate. We will see several examples of such rules in the next subsections below.

Note also that $\mathcal{T}_{cycle}$ also contains that

$$incompatible(T_k(S, X_k), T_l(S, X_l))$$

for any k, l $k \neq l$, stating that all transitions are incompatible with each other. This condition can be relaxed if we want to allow for concurrent execution of transitions, as we will discuss in section 11, but this is beyond the scope of this report.

9.2.6 Properties of $\mathcal{T}_{cycle}$

We have seen in section 9.2.1 that we require that for any given cycle theory, *at most* one transition is enabled at each time. In order to ensure that at most one next cycle-step is preferred, we can require that the behaviour conditions must be such that no two cycle-steps are (in any possible state) given higher priority than all other cycle-steps in the same *family*, where by a family of cycle-steps rules we intend the set of all cycle-step rules for the same current transition. This means that families of behaviour conditions are *exclusive*. We will see below when it is appropriate to relax this last condition and how this can be replaced in a suitably generalised from.

Similarly, we can impose conditions over $\mathcal{T}_{initial}$ so that at most one initial transition can be generated from it.

In addition, we could impose that *at least* one transition is enabled at each time. Together with the earlier requirement this would mean that *exactly* one transition is enabled at each time. Again, this imposes some requirements on the form of $\mathcal{T}_{cycle}$, so that at least one cycle-step is enabled at each time. One such requirements, for example, could be that the set of enabling conditions for cycle-step rules in $\mathcal{T}_{basic}$ and $\mathcal{T}_{interrupt}$ are *exhaustive*, namely one of such conditions is always satisfied. Also, the behaviour conditions must be always (in any

[16] Note that, in a more advanced execution model, we could even force that the current transition is interrupted when a POI needs to be executed. In such a case we need to decide which of the partial information (if any) computed by the current transition should be kept in the state of the computee before engaging into the POI transition. This is beyond the scope of this report.

possible state) given higher priority than all other cycle-steps in the same family. This again can be achieved by requiring that each family of behaviour conditions are exhaustive.

If we impose this additional requirement, then we impose that computees cannot be ever idle.

9.3 Cycle Patterns and Profiles of Behaviour

In this section we will show how different patterns of operation can arise from different cycle theories aiming to capture different profiles of operational behaviour by the computees. We will first show how fixed cycles are a special case of cycle theories and then give examples of $\mathcal{T}_{behaviour}$ to show the ease with which we can capture in a cycle theory a certain pattern of behaviour.

9.3.1 Fixed cycles via cycle theories

Cycle theories generalise fixed cycles in that the behaviour induced by a fixed cycle can be obtained via the behaviour induced by a cycle theory, for some special cycle theories. These are theories where

- all rules in $\mathcal{T}_{behaviour}$ have empty (or true) behaviour conditions
- for each pair $k, l \in I, k \neq l$, there is only one rule $R^i_{k|l}$ or $R^i_{l|k}$ in $\mathcal{T}_{behaviour}$, and
- there exists no sequence of rules $R^i_{k|l_1}$, $R^i_{l_1|l_2}$, ..., $R^i_{l_n|k}$ in $\mathcal{T}_{behaviour}$, such that $k, l_t \in I, k \neq l_t$, for $t, j = 1, \ldots, n$, $l_t \neq l_j$, for $t \neq j$, and $n > 1$.

We then have a *fixed total order* amongst the cycle-step rules in the same family, for each current transition. This gives a pattern of operation of the computee that depends only on the enabling conditions of the cycle-steps. Assuming that the top-most transition, with respect to this total order, is enabled at each step, we get a fixed cycle that underlies the operation of the computee.

9.3.2 Patterns and Profiles of Behaviour

Relaxing the above simplification that all the behaviour conditions are true, and letting the preference rules in $\mathcal{T}_{behaviour}$ be conditional on the current state of the computee opens up the possibility to produce a variety of patterns of operation. The overall operational behaviour of the computee as given by generic cycle theories is thus dynamic, depending on the particular circumstances under which the transitions are executed. Many different *patterns of profiles of behaviour* can be defined by choosing these conditions appropriately. Changing the behaviour conditions we can engineer the pattern or profile of operational behaviour of the computee.

Some examples of profiles of (operational) behaviour are the following.

Punctual or Timely This is a pattern where the computee attempts to satisfy its goals on time. It plans and executes its actions in order to achieve a timely completion of its goals. Hence transitions for the completion of actions and goals that are becoming relatively urgent are given preference over ones for other goals and actions and over other operations of the computee.

Focused or Committed This is a pattern where once a computee has chosen a plan to execute prefers to continue with this plan (refining and/or executing further) until the plan is finished or it has become invalid at which point the computee can consider other plans or other goals etc. Hence transitions that relate to an existing plan have preference over transitions that relate to other plans e.g transitions that introduce other plans.

Impatient This is a pattern where whenever a computee finds out that an existing action or plan is invalidated by the environment prefers to abandon it. Hence it prefers to execute other plans for other goals or for the same goal.

Efficient This is a pattern where a computee prefers to follow a sequence of transitions that allows it to achieve its goals in an optimal way with respect to some utility or cost criterion e.g. minimise the number of observations. Hence as in the case of the punctual computee (where the utility is time) the utility will determine preferences amongst alternative choices of transitions.

Cautious This is a pattern where the computee prefers not to attempt to execute an action when it does not know that this can be done, i.e. it does not know that its preconditions hold. It prefers to execute actions for which it knows that the preconditions hold. Hence it would also prefer to do a sensing introduction transition over an action execution transition. Similarly, in a cautious pattern the computee would prefer to check that the desired effects (in a plan) of an action hold after its execution.

Careful This is a pattern where when some failure occurs, e.g. some action execution has failed or has timed out, then the computee prefers to first re-examine its current goals and plans before continuing with their further reduction and execution. Hence the computee prefers to do revision transitions over the other transitions in order to first let the effect of the failure propagate in its current state.

Let us illustrate, by means of examples, how we could capture some of these patterns of operation. Consider again $\mathcal{T}_{basic}$ as given earlier in section 9.2.2, for deciding what might follow an AE transition:

$r_{AE|PI}(S', Gs) : T_{PI}(S', Gs) \leftarrow T_{AE}(S, As, S'), C_{GI|PI}(S', \tau, Gs)$
$r_{AE|AE}(S', As') : T_{AE}(S', As') \leftarrow T_{AE}(S, As, S'), C_{AE|AE}(S', \tau, As')$
$r_{AE|AOI}(S', Fs) : T_{AOI}(S', Fs) \leftarrow T_{AE}(S, As, S'), C_{AE|AOI}(S', \tau, Fs)$
$r_{AE|PR}(S') : T_{PR}(S') \leftarrow T_{AE}(S, S')$

The behaviour component $\mathcal{T}_{behaviour}$ for a *punctual* profile of operation would then contain the following rules [17]:

$R^{AE}_{AE|*} : h\_p(r_{AE|AE}(S, As), r_{AE|*}(S, X)) \leftarrow As = h^u_{AS}(S, \tau), As \neq \{\}$
$R^{AE}_{PI|*} : h\_p(r_{AE|PI}(S, Gs), r_{AE|*}(S, X)) \leftarrow Gs = h^u_{GS}(S, \tau), Gs \neq \{\}$
$R^{AE}_{AOI|*} : h\_p(r_{AE|AOI}(S, Fs), r_{AE|*}(S, X)) \leftarrow Fs = h^u_{FS}(S, \tau), Fs \neq \{\}$
$R^{AE}_{PR|*} : h\_p(r_{AE|PR}(S), r_{AE|*}(S, X)) \leftarrow nothing\_urgent\_or\_to\_be\_sensed(S, \tau)$

where the behaviour condition $nothing\_urgent\_or\_to\_be\_sensed(S, \tau)$ can be defined as

[17] Here and below we will use * to denote a variable that can take the value of any transition index in I.

follows:

$$nothing\_urgent\_or\_to\_be\_sensed(S,\tau) \leftarrow h^u_{AS}(S,\tau) = \{\}, h^u_{GS}(S,\tau) = \{\}, h^u_{FS}(S,\tau) = \{\}$$

Here, we use the heuristic selection functions for actions and goals, h^u_{AS} and h^u_{GS}, respectively, encapsulating urgency, as given in section 8. Intuitively, As are "the most urgent actions" in S and Gs are "the most urgent goals" in S, at time τ. h^u_{FS} is some heuristic function for the selection of urgent fluents to be sensed, as given in section 8. The PR transition will be selected only if there is nothing urgent to be dealt with within the state, and nothing to be sensed. Basically, the last rule says that we prefer a PR transition when "there is time to tidy up the state of the computee". Note that these behaviour conditions are exhaustive so that always one of these rules will apply. If we also assume that they are exclusive then only one will apply and this will determine the next transition as the most preferred one.

If we want to capture a *careful* behaviour where the computee revises its state when one of its goals or actions times out (being careful not to have in its state other goals or actions that are now impossible to achieve in time) we would have in $\mathcal{T}_{behaviour}$ the rule:

$$R^*_{PR|*} : h\_p(r_{*|PR}(S), r_{*|*}(S)) \leftarrow time\_out(S,\tau)$$

where the PR transition is preferred over all other transitions. The behaviour condition $time\_out(S,\tau)$ can be defined as follows:

$$time\_out(S,\tau) \leftarrow S = \langle KB, Goals, Plan\rangle, A = \langle \_, \_, \_, Tc\rangle \in Plan, \nexists\sigma[\sigma \models Tc]$$

Moreover, if we wanted to have a PR transition always followed by a GR transition, then we would add the rule (with empty behaviour conditions):

$$R^{PR}_{GR|*} : h\_p(r_{PR|GR}(S), r_{PR|*}(S))$$

A *focused* profile of behaviour could be captured by rules:

$$R^{AE}_{AE|*} : h\_p(r_{AE|AE}(S,As), r_{AE|*}(S,X)) \leftarrow As = h^{sp}_{AS}(S,\tau), As \neq \{\}$$
$$R^{AE}_{AE|*} : h\_p(r_{AE|PI}(S,Gs), r_{AE|*}(S,X)) \leftarrow Gs = h^{sp}_{GS}(S,\tau), Gs \neq \{\}$$

which state that we prefer to execute actions or reduce goals from the same plan (i.e with a common ancestor) as the actions that have just been executed. Here, the behaviour conditions are defined in terms of the heuristic selection functions h^{sp}_{AS} and h^{sp}_{GS}, as given in section 8. Intuitively, As and Gs, respectively, belong to the same plan as the actions executed within the current transition AE.

A *cautious* profile of behaviour would involve rules of the form:

$$R^*_{SI|AE} : h\_p(r_{*|SI}(S,Fs), r_{*|AE}(S,As)) \quad \leftarrow \quad pre(As,Fs)$$

stating that a sensing introduction transition for the preconditions of an action is preferred over the execution of that action. The predicate $pre(As,Fs)$ is true if Fs is the set of all preconditions of all actions in As.

Alternatively, we could have a rule to prefer to execute actions whose preconditions are known to be true:

$$R^*_{AE|AE} : h\_p(r_{*|AE}(S,As_1), r_{*|AE}(S,As_2)) \quad \leftarrow \quad h^{pre}_{AS}(S,\tau) = As_1, As_1 \neq \{\}$$

Basically, the execution of "executable" actions (As_1) is preferred over the execution of "non-executable" actions (As_2). Actions are "executable" if their preconditions are known to hold. Here, the behaviour conditions are defined in terms of the heuristic action selection function h^{pre}_{AS}, given in section 8, that selects all actions in a state whose preconditions are known to hold in that state..

An *impatient* pattern where actions that have been tried and failed are not tried again would involve rules of the form:

$$R^{*}_{*|AE} : h\_p(r_{*|*}(S), r_{*|AE}(S, As)) \leftarrow h^{fail}_{AS}(S, \tau) = As, As \neq \{\}$$

where AE is given less preference than any other transition. Intuitively, As are "failed" actions, returned by the heuristic action selection function h^{fail}_{AS}, defined in section 8. As a result of this priority rule it is possible that such failed actions would remain un-tried again (unless nothing else is enabled) until they are timed out and dropped by PR.

Finally, let us give an example where preference rules on the interrupt cycle-steps can also determine a characteristic of the behaviour of the computee. Referring to the interrupt theory given in section 9.2.3, if we have the rule:

$$R^{POI}_{RE|*} : h\_p(r_{POI|RE}(S), r_{POI|*}(S))$$

then a computee will be focused on its current plans as it prefers to use the new external input provided by POI to adapt its current plans and goals by RE rather than to introduce possible new goals through GI or to revise its goals through GR.

Note also that we can use a preference rule in $\mathcal{T}_{behaviour}$ to override the interrupt theory under certain circumstances. Suppose for example that we do not want to carry out any of the interrupt cycle-steps at the expense of delaying the execution of very urgent actions. Then we could have a *cross* preference rule:

$$R^{POI}_{AE|*} : h\_p(r_{POI|AE}(S, As'), r_{POI|*}(S)) \quad \leftarrow \quad h^{u}_{AS}(S, \tau) = As', As' \neq \{\}, \; very\_urgent(As', \tau)$$

with some appropriate definition of the predicate *very_urgent*. This rule achieves the preference to carry on with the execution of the very urgent actions despite the interrupt.

9.4 Hierarchies and multi behaviour criteria

The behaviour part $\mathcal{T}_{behaviour}$ of the cycle theory allows us to view the operation of the computee as a form of *heuristic search*, obtained by using the heuristic selection functions and other heuristic criteria to define the behaviour conditions. But how can we synthesise these different criteria to get an effective and intelligent search for the operation of a computee? Each heuristic corresponds to a criterion of evaluation and hence we need to have ways to perform, via the cycle theory, a multi-criteria decision in order to take into account simultaneously a variety of heuristics, e.g. resource limitations, urgency, utility and qualitative criteria such as executability or failure of actions.

We can study these questions in two steps. First we will consider further the issue of how one heuristic on its own ensures that a decision can be taken at each step of the operation. Then we will see how we can synthesise criteria so that ambiguities left at the level of one criterion could be resolved using another heuristic criterion.

Given a basic part of a cycle theory with the possibility of more than one cycle-step in a family to be enabled the behaviour part needs to contain preference rules that would decide which one of these enabled cycle steps will be preferred. Hence if we name these cycle-steps by $r_1, ..., r_n$ we would generally have in the behaviour part for each pair of these rules two rules[18]:

$R^k_{ij} : h\_p(r_i, r_j) \leftarrow BC_{ij}$
$R^k_{ji} : h\_p(r_j, r_i) \leftarrow BC_{ji}$

that show that r_i is preferred over r_j when the condition BC_{ij} holds and vice-versa when BC_{ji} holds. As we have seen, these conditions typically refer to a heuristic criterion that takes different values in the different rules. For example, in the punctual pattern of behaviour, where we have the heuristic criterion of "urgency", we can have the following two rules using this criterion:

$$R^{AE}_{AE|*} : h\_p(r_{AE|AE}(S, As), r_{AE|PI}(S, Gs)) \leftarrow As = h^u_{AS}(S, \tau), As \neq \{\}$$

$$R^{AE}_{PI|*} : h\_p(r_{AE|PI}(S, Gs), r_{AE|AE}(S, As)) \leftarrow Gs = h^u_{GS}(S, \tau), Gs \neq \{\}$$

If only one of such rules has its conditions satisfied then this effectively decides the cycle-step to be applied. But what happens if conditions BC_{ij} and BC_{ji} are not exclusive, i.e. it is possible that in some states more than one hold true? For example, what happens when in a state we have both actions and goals that are urgent and so the behaviour conditions of the above two rules are both true? Then the heuristic embodied in these conditions cannot decide on the next cycle-step to be applied. None of the possible next transitions given by these cycle steps would be a sceptical conclusion. These would be credulous conclusions i.e. possible next transitions, but as the transitions are incompatible with each other the theory could be in a dilemma [19].

In order to resolve this dilemma we can have additional preference rules in the behaviour part that would compare the overlapping preference rules and give a priority amongst them. These additional rules therefore express a *higher-order preference* on the lower-level preference rules. They have the same form as the preference rules examined so far except that now they apply on preference rules rather than cycle-step rules. Their form is:

$$C^i_{m|n} : h\_p(R^i_m(S, X_m), R^i_n(S, X_n)) \leftarrow EC^i_{m|n}(S, X_m, X_n, \tau)$$

where R^i_m and R^i_n are preference rules. The conditions EC can be seen as a *refinement* of the heuristic criterion involved [20], where more information from the criterion is used to evaluate the alternatives. In the above example, where we have a dilemma amongst urgent goals and actions, we could have:

$$C : h\_p(R^{AE}_{AE|PI}(S, Gs, As), R^{AE}_{AE|AE}(S, As, Gs)) \leftarrow more\_urgent\_goal(S, Gs, As, \tau)$$

[18] For simplicity, we will drop the parameters in the rules and the names of rules when these are not needed.

[19] We repeat here that if the particular transitions are not incompatible with each other then we can generalise the decision process of the cycle theory to allow all of them as next transitions to be executed concurrently.

[20] These conditions can be called *Exception conditions* as these higher order preference rules can change the "normal" default preference given by the lower-level preference rules of the form R^i_m.

$$C' : h\_p(R^{AE}_{AE|AE}(S, As, Gs), R^{AE}_{AE|PI})(S, Gs, As) \leftarrow more\_urgent\_action(S, As, Gs, \tau).$$

These say that if the goals are more urgent than the actions, then the preference to reduce these urgent goals further by a PI transition next is stronger than the preference to next execute the urgent actions and vice-versa. This can then resolve the dilemma of which is to be the next transition.

The behaviour (or exception) conditions at this higher level need to be exhaustive and exclusive otherwise it is possible to arrive again in a state of dilemma (the preference reasoning will not give a sceptical conclusion) where this has now been transferred one level higher. If this is the case we can use another level of preference rules that stipulate the relative priority of these higher-level rules and so on.

This then results in a *hierarchy* of preference rules where each level uses a more refined or detailed form of the heuristic. We require only that at some level the conditions of the preference rules are exhaustive and exclusive. Thus the overall decision process for the cycle step to be applied is carried out at stages following a hierarchy where each time we refine the information drawn from the heuristic.

We can generalise this extension with higher-order preference rules by allowing these rules to refer to other heuristic criteria, i.e instead of using further information from the same heuristic these rules could be conditional on a different criterion. For example, to decide amongst equally urgent actions and goals we can use some other utility criterion using the higher-order preference rules:

$$C : h\_p(R^{AE}_{AE|PI}(S, Gs, As), R^{AE}_{AE|AE}(S, As, Gs)) \leftarrow utility(S, Gs, U_1)$$
$$C' : h\_p(R^{AE}_{AE|AE}(S, As, Gs), R^{AE}_{AE|PI}(S, Gs, As) \leftarrow utility(S, As, U_2).$$

The hierarchy then becomes a hierarchy of different criteria and the decision using the preference reasoning $\models_{pr}$ becomes a multi-criteria decision problem. Starting from the basic part of the cycle theory and its enabling conditions we can have a hierarchy of criteria $Core\_Selection\_Criteria \sqsupseteq Criterion_1 \sqsupseteq Criterion_2 \sqsupseteq ...$, e.g. $Core\_Selection\_Criteria \sqsupseteq Urgency \sqsupseteq Utility$ as we have seen above. In this way we are effectively synthesising different patterns of behaviour corresponding to different heuristic criteria.

For total hierarchies where criteria are separated at different levels of the preference rules the multi-criteria decision required is simply following the hierarchy step by step. If this is not so and different criteria are mixed at the same level then we need to apply more complex forms of multi-criteria decision making.

10 Computees in Societies

Individual computees, as modelled in this document, will not be created to function in isolation, but instead they will be parts of artificial computee societies. In this section we discuss the features a computee should have in order to function within an artificial society. Such features will allow the computee to communicate with other computees and take into account any opportunities offered by the society without overlooking any requirements that the society may impose. In this section we describe four specific features that facilitate the functioning of computees within societies and that provide the necessary connections between the models

described in this document and deliverable D5 (where the reader can find out more details about the social modelling of computees). These features are:

1. Communication;
2. Conforming to society's (communication and other) protocols;
3. Entering and leaving societies;
4. Responding to society's expectations.

10.1 Communication

In this section, we describe in more detail how communication can be embedded within the KGP model developed in this document. The KGP model assumes that a computee has, within the set of actions it can perform, a set of communication actions. In general, in order for a computee to communicate with other computees it requires:

1. a language for communication,
2. a collection of communication actions,
3. a way of generating communication actions as part of plans to achieve goals,
4. policies for determining which communication acts to perform when, and policies for generating communication actions in response to utterances it receives or in reaction to events in the environment,
5. a way of deciding who best to communicate with in order to achieve its objectives.

In the following we address each point and show how the KGP model of D4 and our current and previous work cater for it.

10.1.1 Language for communication

In D5 we give a framework to equip a Computee Communication Language (CCL) with a semantics which is independent of the computee's internal state. The social semantics is provided to give a social meaning to the computees' communication actions. Here, instead, we explain the internal functioning of computees which allow for communication (as explained below).

Throughout this section, we rely upon a language for communication defined in [STT02a] for a resource exchange scenario, and grounded on a computational logic framework. This language is also adopted/described in some examples presented within D5.

The language can be summarised as follows. It contains a predicate of the form

$$tell(Utterer, Recipient, Content, Context, Time)$$

that denotes a communication act from the $Utterer$ to the $Recipient$ at time $Time$. $Context$ is a unique identifier for the context (dialogue in [STT02a]) and could relate to an instance of the protocol, if any (to which this dialogue conforms). In the sequel, we will often abuse the syntax and omit the identifier representing the $Context$ of an utterance, for simplicity of presentation. $Content$ is the content of the utterance. $Content$ can, for example, be: $request(Resource)$

$accept(request(Resource))$
$refuse(request(Resource))$

For an exhaustive list of performatives to be adopted for resource exchange, with detailed explanations, the reader should refer to [STT02a], while for additional information about the communication language and its social semantics, the reader should consult D5.

10.1.2 Communication actions

Within the KGP model, concrete utterances, namely instances of the generic 'tell' predicate, are interpreted as (communicative) actions. These can be used as actions in a plan, just as any other (physical or sensing) actions. Thus, from the viewpoint of the KGP model, communicative actions do not differ from any other actions.

Within the reasoning capabilities of planning and temporal reasoning, we can view the communication language as providing a collection of communication action operators. These can be used within the event calculus *happens* predicate (see below for some examples). Within the KB_0 of a computee, communicative actions performed by the computee itself (*output messages*) are recorded within the *executed* predicate (by means of AE), whereas communicative actions performed by other computees (*input messages*), and observed by the computee, are recorded within the *observed* predicate (via the Passive Observation Introduction).

10.1.3 Generating communication actions as part of a plan

The planning capability and subsequently the Plan Introduction transition (PI) depend on the theory KB_{plan}. This theory is an abductive event calculus theory that describes how actions initiate and terminate properties. It allows communication actions as well as physical ones. Consider the event calculus theory given in section 6.1. To this core we can add domain dependent axioms for the *initiates*, *terminates* and *precondition* predicates. For example the KB_{plan} of computee "a" will include the following program for the ownership of a resource:

$initiates(get(a, Owner, Rsrc), T, have(a, Rsrc)) \leftarrow holds\_at(have(Owner, Rsrc), T),$
$precondition(get(a, Owner, Rsrc), approves(a, Owner, Rsrc)),$
$initiates(tell(Owner, a, accept(request(Rsrc)), \_, T), T, approves(a, Owner, Rsrc))$

Similarly, we can define the dual *terminates* rules when a is giving away a resource after accepting a request from another computee.

Communicative actions can be introduced within the goals or plan of the computee also by the Goal Introduction and Reactivity transitions. More concretely, communication actions are introduced from the representation of communication policies/strategies within KB_{GD} and KB_{react}, respectively, as it will be discussed in the next section below.

10.1.4 Policies for communication

Policies for determining which communication acts to perform when and generating communication actions in response to utterances the computee receives or in reaction to events in the environment can be represented in abductive logic programming as integrity constraints [STT01, STT02b, STT02a] or in logic programming with priorities as policy rules.

The reactive capability and the Reaction transition depend on KB_{react}, which consists of a collection of integrity constraints. Amongst these we incorporate the communication policies

of the computees. These, in turn, will be used by the capability and the transition to generate (add to the plan) communication actions in response to other computees and the environment.

Here are examples for the KB_{react} of computee a[21]:

$tell(X, a, request(R), T), have(R, T) \Rightarrow tell(a, X, accept(request(R)), T + 5)$
$injured(X, T), first\_aid\_officer(Y, T) \Rightarrow tell(a, Y, request(bandage), T + 1)$

where $have$, $first\_aid\_officer$ and $injured$ are fluents.

The first constraint specifies that if a request is made to a for a resource R and a has that resource then 5 time points later a should respond by accepting the request. The second constraint specifies that if (a observes that) someone is injured then one time point later a should ask a first-aid officer for bandage.

An event happening in the environment or a communication action sent by another computee can be received and recorded by a computee through its Passive Observation Introduction transition. This will update the state of the computee, in particular the KB_0. Then in response to this update the application of the Reaction and/or Goal Introduction transitions will generate any appropriate reaction as specified by the policy captured by the integrity constraints similar to the above.

10.1.5 Deciding who best to communicate with in order to achieve objectives

In many cases a computee will need to decide which particular computee (or computees) it should communicate with. For example when requesting some information or a resource it will need to ensure that the computee to which it will make the request is appropriate (i.e. it is likely to have/be able to provide this type of information or resource). Moreover, amongst all such possible computees it may want to select one which it judges to be best suited for the particular request under the particular circumstances of the request.

In order to incorporate this type of decisions we can supply the computee with a specific policy for this. Such a policy will be part of its KB_{plan}, KB_{react} and KB_{GD} knowledge and can be represented either as integrity constraints or as rules with priorities. As an example consider the axioms given earlier for communicative actions of requests and accepting these. A computee will often need to select a suitable computee to which to make its requests. For this we can add in its knowledge the rule:

$$precondition(tell(a, Owner, request(Rsrc), T), suitable(Owner, Rsrc, T).$$

This now says that to ask for some resource, the computee needs to determine a *suitable* computee to ask. The definition of *suitable* to be incorporated in its knowledge can be in the form of simple rules that can allow one to "guess" intelligently which computees are helpful in which cases. For example:

$suitable(X, R, T) \leftarrow holds\_at(friend(X), T)$
$suitable(X, R, T) \leftarrow trades\_in(X, R, T)$

[21] In this example we simplify the notation and write the fluents without using *hold_at*

which say that X is a *suitable* computee to ask for resource R if X is a friend or X trades in R.

More generally, *suitable* can be decided according to a preference policy of the computee. For example,

$r_1(X,R) : suitable(X,R,T) \leftarrow trades\_in(X,R,T)$
$r_2(X,R) : \neg suitable(X,R,T) \leftarrow urgent(R,T), slow\_delivery(X,R)$
$R_1 : h\_p(r_1(X_1,R), r_1(X_2,R)) \leftarrow friend(X_1), \neg friend(X_2)$
$R_2 : h\_p(r_1(X_1,R), r_1(X_2,R)) \leftarrow \neg trust\_worthy(X_2), \neg trust\_worthy(X_1)$
$R_3 : h\_p(r_2(X,R), r_1(X,R))$
$C_1 : h\_p(R_2, R_1)$

This expresses the preference to select a friend and avoid computees which are not trustworthy even in the case when they are friends. Also according to whether the need for a resource is urgent or not the computee equipped with these rules will not select a computee which is slow in delivery when it needs the resource urgently.

10.2 Conforming to society's protocols

Societies may have their own protocols governing the communications of their members. Such protocols, typically, specify the range of acceptable responses that can be made to a communication act, possibly with time constraints imposed on such responses.

In D5 we have proposed the adoption of social integrity constraints to express communication protocols, and we have interpreted the society model in terms of abductive logic programming. Conformance to communication protocols can be checked on-the-fly using a proof procedure, able to consider (possibly in an incremental way) social events that have happened and detect fulfilment or violation with respect to the specified protocols. In accordance with the GC vision, this conformance is determined without any assumption on the computees' internal behaviour, but only requiring to monitor and check the utterances of the society members.

Within the project we have also explored two different approaches for individual computees to deal with such society protocols [EMST03a, EMST02, EMST03b], assuming that the internal behaviour of computees can be programmed appropriately. Recall that computees have their own (private) policies regarding communications with other computees, recorded within their knowledge base. On entering a society they can "observe" the protocols of the society and they may decide to ensure conformance to them or more generally to take integrated decisions from both private policies and public protocols.

One approach proposed in [EMST03a, EMST02, EMST03b] amounts to ensuring conformance by adding the society protocols, expressed as special kinds of integrity constraints (with disjunction in the head, as adopted in D5) to the KB_{react} of the computee. Thus the communication actions of the computee will be governed by the combination of its own private policies and the public protocols of the society (societies) it belongs to. This will ensure a form of weak conformance whereby the computee will never make any "illegal" utterances. The reactive capability and the Reaction transition attempt to ensure consistency of the computee's communication actions with respect to both the private policies and the public protocols now in its knowledge base.

The second approach proposed in [EMST03a, EMST02, EMST03b] does not ensure conformance but provides a technique for checking a priori whether or not the private policies of a

computee are conformant with the public protocols of a society before the computee makes any utterances.

We have also studied in [KM03b] how the decision of a computee can integrate its different private and public protocols. Policies are expressed in logic programming with priorities and are simply added together in the knowledge base of the computee. Then conflicts between the private policies and public protocols are treated, using the underlying preference reasoning $\models_{pr}$ of the computee, in the same way as conflicts within its own private policies are treated. This gives a uniform way to resolve conflicts between private policies and public protocols. We have also examined how this form of uniform integration of policies can be extended using methods from multi-criteria decision theory.

10.3 Computees entering and leaving societies

In D5 we have shown how social integrity constraints can be exploited to regulate expected or forbidden actions in terms of membership. Membership is, in turn, dynamically determined on the basis of relevant social events, e.g. joining, leaving, being expelled, etc, depending on the kind of society (open, semi-open and semi-closed, following the classification of [Dav01]).

In an open society there are no restrictions for computees to join/leave the society. In semi-open societies membership is a property that is initiated by the event of joining, if then admitted by the society, and terminated by the events of leaving or being expelled. Finally, in semi-closed societies, membership is a property that is initiated by the event of joining and being assigned a proxy computee within the society to represent the computee itself within the society.

In [TS02], the authors provide an event calculus based formalisation of membership of open, semi-open and semi-closed societies, following [Dav01]. More formally: [22]

$$
\begin{aligned}
holds\_at(member(C,SOC),T) \leftarrow \quad & happens(tell(C,SOC,join(C,SOC)),T'),\\
& T' < T, not\, clipped(T', member(C,SOC),T),\\
& open(SOC)\\
holds\_at(member(C,SOC),T) \leftarrow \quad & happens(tell(C,SOC,join(C,SOC)),T'),\\
& happens(tell(SOC,C,admit(C,SOC)),T''),\\
& T' < T'' < T, not\, clipped(T'', member(C,SOC),T),\\
& semi\_open(SOC)\\
holds\_at(member(C,SOC),T) \leftarrow \quad & happens(tell(C,SOC,join(C,SOC)),T'),\\
& happens(tell(SOC,C,represent(C,C',SOC)),T''),\\
& T' < T'' < T, not\, clipped(T'', member(C,SOC),T),\\
& semi\_closed(SOC)\\
clipped(T', member(C,SOC),T) \leftarrow \quad & happens(tell(C,SOC,leave(C,SOC)),T''),\\
& T' < T'' < T\\
clipped(T', member(C,SOC),T) \leftarrow \quad & happens(tell(SOC,C,expel(C,SOC)),T''),
\end{aligned}
$$

[22] The formulation given here is syntactically different from the one adopted in [TS02] but semantically equivalent to it. Here, we basically rewrite the formulation of [TS02] so that it is in line with the syntactical conventions introduced in section 6 and earlier on in this section.

$$T' < T'' < T$$

In [TS02], it is assumed that this knowledge is held by all computees, so that they can reason about themselves and other computees being part of societies, starting from the recording of actions and from the (subjective) knowledge of the nature of societies. In general, this knowledge may be held just by the society itself (or by any computee serving the role of authority in the society) so that it can "reason" about membership within itself from the observation of communication amongst computees and with the society, as shown in D5 . The society can also provide a "yellow pages" of members, updated by each new entry and departure, that computees can consult when needed.

Within the KGP model, knowledge such as outlined above is held within KB_{plan}. The recording of actions (*happens* events) is kept, as usual in this model, within KB_0.

Societies (or authority computees within them) can formulate their own policies regarding admission and expulsion of members. In [TS02], some examples of these policies are presented, formulated as integrity constraints in abductive logic programming. For example, if some computee attempts to join a semi-open society, then the society (possibly via a gatekeeper authority in it) will communicate to the computee that it has been admitted. More formally (and again using the conventions adopted in section 6):

$$happens(tell(C, SOC, join(C, SOC)), T), semi\_open(SOC) \Rightarrow \\ happens(tell(SOC, C, admit(C, SOC)), T') \wedge T < T'$$

This integrity constraint is assumed to be held by the society (or by any authority in it). Within the KGP model, knowledge such as this is held within KB_{react}, and represented as (we assume this knowledge is held by SOC):

$$tell(C, SOC, join(C, SOC), T), semi\_open(SOC) \Rightarrow \\ tell(SOC, C, admit(C, SOC), , T') \wedge T < T'$$

10.4 Responding to the society's expectations

The society model in SOCS allows for the society to generate expectations for individual computees inhabiting it. These expectations may, for example, be related to the society goals, if any, and might be communicated to computees. An appropriate response by the computee to such expectations could be to import these expectations as integrity constraints or reactive rules in its knowledge. Alternatively, it can generate goals for itself related to the expectations. Such responses to the expectations can then be catered for by the Reactive capability together with the Reactivity transition and the Goal Decision capability with the Goal Introduction transition of computees.

A computee can react to society's expectations in different ways. This is possible because the model of the society envisaged in D5 leaves the computees free to act as they wish even if this might be not compliant with protocols. The computee can weigh up the society's expectations and the (simple) goals that it generates against its own goals to make a new decision about its current goals.

Then a socially "obedient" computee would assign (in the higher level part of its KB_{GD}) to such generation of goals via expectations higher priority over all other types of rules. Hence these expectations will be adopted as goals for the computee when this executes its goal introduction transition. However, different computees could attach different levels of priority to goals

generated by society expectations in comparison to their own private goals thus allowing for different degrees of conformance to the societies expectations. Also this relative priority of goals from expectations and own goals can be dynamic depending on the particular circumstances in which the computee finds itself at the time of becoming aware of a particular expectation. The higher level part of the KB_{GD} can accommodate such a versatile spectrum of treatment of society expectations.

11 Possible extensions

In this section we discuss briefly some of the possible extensions of the KGP model. These are beyond the scope of this report, as they are not needed to meet the minimum success requirements set for WP1 within deliverable D3. Nonetheless, they would be useful extensions for a wider applicability of our approach.

11.1 Plan introduction transition with intelligent selection of plans

At the moment, the Plan Introduction (PI) transition is defined in such a way that any plan given by the Planning capability can be incorporated within the state of the computee, for the goals selected for planning. In general, it would be useful to be able to apply preferences over plans in order to select, say, plans that may be more likely to succeed than others or plans that have less cost in executing. We could use the same form of preferential reasoning, that we have adopted already for the goal decision capability and for the behaviour as given by cycle theories, in order to decide which plan to choose at any point according to some preference policy. We have already started studying this problem [DK03] by investigating the integration of abduction and argumentation where preference reasoning via argumentation on the abductive hypotheses is performed while generating an explanation or plan through abduction.

Apart from the Planning capability this extension would also affect the revision transitions, selection functions and the range of possibilities for cycle theories to take into account possible changes of preferences of plans over time.

11.2 Knowledge base revision transition

We have allowed for plans and goals in the state of a computee to be revised dynamically, within PR and GR transitions. For the knowledge base of the computee we have assumed that this can only change in its KB_0 component recording new information from the environment. Changes to KB_0 of course implicitly cause changes to the conclusions that can be derived within KB, as indicated via KB_{TR}. However, we have not allowed for the rest of the knowledge, $KB - KB_0$, to change directly. For this we could introduce a new transition for Knowledge Revision (KR), of the form

$$(\mathbf{KR}) \qquad \frac{\langle KB, Goals, Plan\rangle}{\langle KB', Goals, Plan\rangle}\,\tau$$

where KB' is obtained via a new reasoning capability $\models_{KBR}$, namely $KB \models_{KBR} KB'$, standing for "KB' is the result of revising KB", according to some knowledge revision policy.

With this transition we could address the problem of revising the knowledge base when the expansion of KB_0 with a new observation would imply that the knowledge base of the computee

becomes classically inconsistent (see section 6.3.1). We could then employ revision policies that take into account the reliability of the source of the observed information and the level of trust that the computee has in its sensing capabilities, in order to decide how to revise its KB_0 to avoid such an inconsistency.

Also the $\models_{KBR}$ capability may use evidence collected in KB_0 to update the general model of the world that is present in $KB \setminus KB_0$. In particular, this new capability and transition can be used for the computee to generalise and learn from its past experience. It can generate "compiled" knowledge that links direct observations to "internal" properties of the model of the world that the computee has. For this the $\models_{KBR}$ capability can use different methods of relational learning such as those of predictive Inductive Logic Programming (ILP) for learning rules for useful concepts and descriptive ILP for learning integrity constraints.

The modular definition of the transitions facilitates the addition of such a KR transition. Such an addition would affect only the spectrum of cycle theories that one could define but does not require a re-examination of the structure of cycle theories. In fact, the present form of cycle theories could include this transition giving its relative priority over the others and thus the only task that remains is for the transition itself to be defined, by defining the underlying capability.

11.3 Conditional Goals

In some cases when the computee operates in a highly unknown or unpredictable environment it is useful for it to be able to derive conclusions conditional on certain properties of the environment. The Planning capability already operates in this way as plans can contain, together with actions, subgoals some of which are not to be planned further but rather act as assumptions that can be tested in the environment. Similarly, we can extend the goal decision capability so that the computee can derive conditional goals for itself, e.g. "go on a holiday trip if the weather is good" or "give a resource to another computee if it agrees to pay a certain price" etc. This extension can be supported within an integrated framework [KM03b, DK03] of logic programming with priorities and abduction. This is the same framework mentioned above for the intelligent planning extension but where now the argumentative reasoning with logic programs and priorities is the primary form of reasoning that uses abduction if and where this is needed.

11.4 Concurrent execution and interruption of transitions

In the definition of cycle theories and their induced operation we have assumed that transitions may not be executed concurrently. This may be restrictive. For example, it might be useful to be able to plan for one goal, via PI, while executing actions in a plan for another goal, via AE. In order to accommodate concurrent execution of transitions we need to relax the conditions that all transitions are incompatible thus obtaining a partial order over the transitions, and have more than one cycle step equally preferred at some times. We should then modify the induced operation by a cycle theory to allow for concurrency and study how the state is updated through the concurrent execution of transitions.

In our model we impose that until a transition is completed, no interrupt, as provided by a POI, can be taken into account. In a more advanced execution model, we could force that the current transition is interrupted when new information arrives at the computee and a POI needs to be executed. In such a case we need to decide which of the partial information (if any)

computed by the current transition should be kept in the state of the computee before engaging into the POI transition.

11.5 Utilities and Costs

In this document we have not studied in any detail how a computee can use knowledge of explicit and numerical utility of goals and actions. Also global utilities of states can be useful in taking local decisions on actions and goals. Different measures of utility can be used in several places in the model. In particular, they can be used as heuristics in the model. For example, in goal selection we could select any goal G such that G is the most "useful" goal to achieve, in that achieving that goal will bring the computees into a state of maximal utility. We could also accommodate costs of actions within the framework, and add new heuristics for action selection such as the selection of all A in $Plan$ such that A is of low cost below a threshold.

12 Related work

During the past few years we have witnessed an explosion of proposed models and architectures for individual agents. In this section, we identify a set of proposals that we believe are directly relevant to the KGP model. We expose the similarities and differences between KGP and those related proposals, resulting in a critical evaluation that is based on the relative advantages and disadvantages of KGP with respect to these related models.

We start with a number of existing proposals that are popular in modelling agents and multi-agents systems, most notably, the classical BDI model [RG97], the modelling features of the agent-programming languages: Agent0 [Sho93], AgentSpeak [Rao96] and its variants, 3APL [HdBvdHM99a], and the agent-modelling framework DESIRE [BDKTV97]. Then we compare KGP with existing computational logic-based approaches that use, as we do here, non-monotonic logic programming frameworks and techniques to model, specify and implement software agents. These include the work developed by the IMPACT project [AEK$^+$99], the logic-based system $\mathcal{MINERVA}$ [LAP01b], the agent specification language GOLOG [LRL$^+$97] and its variants, and Vivid Agents [SW00].

Throughout this section we advocate a number of advantages for KGP, which we explicitly list below for the sake of readability. These are as follows:

- The KGP model provides a simple but powerful specification framework that synthesises in a single framework: Abductive Logic Programming (ALP), Temporal Reasoning based on the Event Calculus, Constraint Logic Programming (CLP), and Preference-based Reasoning based on Argumentation.
- The KGP model supports the computational logic formulation of an agent's knowledge, goals, plans, and reasoning capabilities represented as non-monotonic logic programs together with a logical cycle-theory that is specified separately. All these specifications are modular and executable, by relying computationally on existing proof-procedures and their extensions (to be developed within WP3).
- The KGP model is flexible in that it supports heterogeneity of computees because of the modular definition of the capabilities and the transitions, as well as the modularity of the cycle theory.

- The KGP model does not rely upon any fixed sequence of operations, computees can take run-time decisions about what to do next, depending on their personalities and the information they receive from the environment.

- The flexibility of the KGP model and the way certain transitions are specified allow for computees to be autonomous and adapt their operation to the open and continuously changing environment. In addition, the model allows computees to be tolerant of partial information and recover from some inconsistencies that may arise from directly observing the environment.

- Computees specified in the KGP model have social ability through communication and interaction with the environment, and can contain social knowledge and capability as specified and supported by the model.

We proceed to discuss how these advantages compare with the features of existing agent models.

12.1 The BDI model

Perhaps one of the most influential approaches for modelling complex systems in the form of agents, is based on the *intentional stance* proposed by Dennet in [Den87]. This approach refers to treating a complex system as if it had intentions, irrespective of whether it does or not. The advantage of treating a system as a rational agent is that one is able to predict the system's behaviour. The idea here is that first one ascribes beliefs to the system, as those the agent ought to have given its abilities, history and context. Then one attributes desires to the system as those the agent ought to have given its survival needs and means of fulfilling them. One can then predict the system's behaviour as that of a rational agent would undertake to further its goals given its beliefs.

Dennet argues for three main reasons in taking an intentional stance. First it fits well with our understanding of the processes of natural selection and evolution in complex environments. Second, it has been shown to be an accurate method of predicting behaviour. Third, it is consistent with our folk psychology of behaviour.

Within the intentional stance approach, the BDI (*Beliefs, Desires and Intentions*) model has been proposed by Bratman et al [BIP88] to represent resource-bounded practical reasoning for programmable agents. The main philosophy of the approach requires that the *Beliefs* hold the partial knowledge that agents have about their environment, the *Desires* contain the goals the agents are aiming at, and the *Intentions* include the plans agents set in order to achieve their goals.

12.1.1 Classical BDI: Architectures, Logics, and Implementations

One important contribution of the original paper by Bratman et al [BIP88] is an architecture for practical reasoning. In this architecture the agent's intentions are viewed as being structured into larger plans. The architecture distinguishes between plans that the agent has actually adopted (intentions that are structured into plans as a result of deliberation), and plans-as-recipes, or operators, that are stored in an additional structure called the *plan library*.

What makes the architecture suitable for practical reasoning are four main processes:

- *Means-End-Reasoner* – generates a set of options (which can be thought of as sub-plans or eventually actions) from the current beliefs, intentions, and plans in the plan library;
- *Opportunity Analyser* – generates a set of options based on the current beliefs and the current desires;
- *Filtering Process* – provides the options that survive given the options generated by the Means-End-Reasoner and the Opportunity Analyser, and the current beliefs, and intentions;
- *Deliberation Process* – generates a new set of intentions from the current set of surviving options generated by the Filtering Process, the current beliefs, and desires.

Together with an additional reasoning process that specifies how the agent's beliefs change due to the agent's perception of the environment, the above four processes constitute a system, by which an agent forms, fills in, revises and executes (actions from) plans.

It is important to note that the architecture requires that the plans are partial due to the bounded resources and knowledge that the agent has. Plans can be partial in two different ways. They may be temporarily partial, accounting for some periods of time and not for others. They may also be structurally partial, accounting for situations where they need to decide upon the ends, leaving open for later deliberation questions about the means to those ends.

Following [BIP88], Cohen and Levesque [PL90] have formalised some philosophical aspects of Bratman's theory [Bra87]. In their formalism, intentions are defined in terms of temporal sequences of an agent's beliefs and goals. In related work, Rao and Georgeff have developed a modal logic framework for agent theory based on the three primitive modalities of beliefs, desires, and intentions [RG91, RG97]. Their formalism is based on a branching model of time in which belief-, desire-, and intention-accessible worlds are themselves branching time structures.

To establish the link between BDI theory and practice Rao and Georgeff have also presented an abstract architecture [RG92, RG95] that focuses on practical/computational concerns (unlike that of Bratman et al), where amongst other things, it illustrates how a BDI system can be designed to have data structures that correspond to beliefs, desires, and intentions, together with update and query operations on these structures. The rationale for such a choice is useful, Rao and Georgeff argue, when an agent has to communicate with humans and other agents, and can be expected to simplify the building, maintenance, and verification of application systems.

However, the architecture does not rely on the use of modal-logic theorem provers, as one might have expected. The reason for this is that by using such computational tools the time taken to reason, and thus the time taken to act, is potentially unbounded, thereby destroying reactivity that is essential in the agent's survival. Instead, the update operations on the beliefs, desires, and intentions structures are controlled by an interpreter as shown below:

```
BDI-interpreter
initialise-state();
repeat
    options:= option-generator(event-queue);
    selected-options := deliberate(options);
    update-intentions(selected-options);
    execute();
    get-new-external-events();
```

drop-successful-attitudes();
drop-impossible-attitudes();
end repeat

At the beginning of every cycle, the option-generator() reads an event-queue structure and returns a list of options. The deliberate() selects a subset of selected-options to be adopted and adds these to the intentions structure. If there is an intention to perform an atomic action at this point in time, the agent then executes it by calling execute(). Any external events that have occurred during the interpreter cycle are then added in the event-queue by calling get-new-external events(). Internal events are added as the occur. Next, the agent modifies the intention and desire structures by calling drop-successful-attitudes() and drop-impossible-attitudes() to deal with successful as well as unrealisable (or impossible) intentions and desires.

The ideas behind this new, computing-centric, abstract architecture is to bridge the gap, between BDI theory - presented in terms of the BDI architecture and the modal logics on the one hand, and a number of existing BDI implementations - most notably the work on the systems PRS [GL86, GI89a, GI89b, IGR92] and dMARS [GL87] on the other.

12.1.2 Classical BDI and KGP: A comparison

Wooldridge and Jennings in [WJ95] inform us that: "... precisely which combination of information attitudes (such as knowledge or belief) and pro-attitudes (such as intentions, desires and obligations) is important to characterise an agent, is an issue of some debate". In fact, this issue is still ongoing and there is no universal agreement as to what such pro-attitudes should be. For the purposes of SOCS, and inspired by BDI, we chose the structure of Knowledge Bases, Goals and Plans.

We could safely say that, at a first glance, the KGP model seems to be falling within the general philosophy of the classical BDI model, but with more emphasis on a computational logic characterisation. Moreover, using the categories discussed in [WJ95], we can say that the relation between classical BDI and KGP can be described as follows. The information attitude of *belief* in BDI is similar to that of *knowledge* in KGP. The pro-attitude of *desires* in BDI is similar to the more specific computational notion of having explicit *goals* in KGP and, in particular, it can be thought of as a superset of the KGP goals. The pro-attitude of *intentions* in BDI is similar to the set of goals in KGP that have been chosen so far to be planned for, together with any (partial) plan generated for them.

By examining closer the BDI and KGP models, however, we find a number of important differences, most of which result from the notions of information attitudes used, the granularity of the pro-attitudes represented, and the different computational logic tools used to represent and reason with these attitudes and pro-attitudes.

One major difference between BDI and KGP is that KGP is not based on a modal-logic approach to represent an agent's beliefs but instead it is based on a non-monotonic computational logic that supports defeasible reasoning for the knowledge of agents. The KGP model takes a simpler (certainly a less expressive) specification language where belief is ascribed as a mental attitude from an external viewpoint.

Still the KGP model does not loose the generality of the BDI model in the way plans are constructed and used. Both in the BDI and in the KGP models, plans can be drawn from a plan library or generated as part of a deliberation process (in the case of KGP model we allow plans to be generated as a result of reasoning). However, even if both BDI and KGP address

resource boundedness by constructing partial plans, KGP also uses a meta-logical component focusing on urgency (see the action and goal selection discussed earlier in section 9) to facilitate timely action execution and planning.

Also, the KGP model uses goal–selection policies that are similar to the generate-options() functions in BDI. However, the KGP policies are specified as preference theories that can be used to describe the personality of the agent. These can be added in a modular way when the agent is created, or even in real time, thus providing more possibilities for engineering flexible behaviour, when this is required. In principle, we plug in different modules to reflect different personalities. Moreover, our goal–selection policies are interpreted in an argumentation–based framework, which attributes to our approach a more detailed and formal model than that proposed by the classical BDI.

Another important difference between the BDI and the KGP models is that in KGP we provide a more flexible cycle theory to regulate the behaviour of a computee, in the sense that this theory can be customisable, rather than relying on an one-size-fits-all cycle as in classical BDI (and its recent interpretations discussed in sections 12.3, 12.4 and 12.5.

Another advantage of the KGP model over the classical BDI is that in KGP the correspondence between agent specification and executable implementations is closer than that provided by classical BDI. In our review of the BDI work we agree with Rao's more recent views on BDI [Rao96], viz., that the complexity of the code written for classic BDI implementations such as PRS and the simplifying assumptions made by them have meant that these implementations have lacked a strong theoretical underpinning.

Rao's recent views is that the specification logics for BDI have shed very little light on the practical problems and, as a result, the two streams of work on theory and practice seem to have been diverging. By looking back at the abstract architecture provided in [RG92], Rao also argues that "...due to its abstraction this work was unable to show a one-to-one correspondence between the model theory, proof theory, and the abstract interpreter". He then goes on to further conclude that "...the holy grail of BDI agent research is to show a one-to-one correspondence with a reasonably useful and expressive language".

We will see later, in section 12.3, how Rao, who is undoubtedly one of the major contributors in the development of modal BDI logics, turns to the use of logic programming techniques to re-interpret his earlier BDI work. Independently of Rao's change of perspective, the development of our KGP model has been motivated by similar observations. However, in our work we are driven mainly by the possibility of providing a computational logic model for the specification and implementation of software agents that relies solely on logic programming techniques and their extensions, hoping to demonstrate that the so much desired one-to-one correspondence between agent theory and practice is indeed possible.

Still, one important advantage of BDI over KGP is that agents in BDI can be introspective in that they can reason about their own beliefs and reasoning capabilities. In addition to introspection, a BDI agent can also model the beliefs and capabilities of other agents. Although KGP does not support reasoning about other agents, it does support reasoning about actions, with incomplete information and how to recover from inconsistencies that arise from direct observation. We could have extended KGP to support introspection and modelling of other agents using existing computational logic techniques based on meta-logic [BK82, Sat92, Jia94]. Indeed, in earlier work we have made a start in modelling such introspection [DST98, DST99]. We have chosen not to deal with issues of introspection in this document, mainly because we wanted to concentrate on other issues of more direct relevance to Global Computing.

As a final remark on BDI it is worth saying that the model has been criticised for not

taking into account social notions, in particular obligations. It was extended by Broersen et al [BaJHHvdT01] to accommodate such notions. A new architecture contains four components that output: beliefs, obligations, intentions, and desires only for certain inputs. Conflicts between these outputs are either resolved by the architecture's control loop (BDI-like) or by a separate selection component that outputs new intentions. Agent types are represented by different control loops. In a *realistic* agent, beliefs override obligations, intentions or desires, while in a *single-minded* agent intentions override desires and obligations. Other types of agents are also defined such as *open-minded* and *selfish* agents. An implementation of the extended model is available as a production system using propositional rules with only a partially-specified semantics.

KGP takes into account social notions such as protocols and society expectations, as discussed in section 10. In common with Broersen et al [BaJHHvdT01] KGP attempts to facilitate the heterogeneity of computees by its modular and general cycle theories modelling different computee personalities.

12.2 AGENT0

AGENT0 [Sho93] is as an agent-oriented programming language that extends the AI language Lisp. It is probably one of the first attempts to promote a social view of computation based on the interaction of different co–operating agents. The approach is grounded on a multi–modal logic with an explicit representation of time, with modalities such as *beliefs* and *commitments*, and communication primitives, such as REQUEST and INFORM.

Using the AGENT0 architecture an agent has four component data structures: a set of *capabilities* – specifying what an agent can or is able to do; a set of *beliefs* – stating what an agent believes at certain times and about certain times; a set of *commitments* – representing the actions that the agent ought to do at specific times; and a set of *commitment rules* – describing how new commitments can be introduced or old commitments can be dropped.

An agent cycle interprets commitment rules in AGENT0 roughly as follows: a new incoming message may update the beliefs and will be matched against the agent's message conditions of the commitment rules set. These conditions are then matched against the beliefs of the agent. If the commitment rule fires (i.e. both the message and the conditions of the commitment rule are satisfied), then the agent becomes committed to the action. The execution of an action then may update the commitments and the beliefs of the agent.

In AGENT0, the capabilities, the beliefs the commitments and the commitment rules correspond, respectively, to the list of available actions that the computee can perform in the environment, the knowledge of a computee at specific times and about specific times, the goals, and the knowledge bases underlying the Reactive and Goal Decision capabilities in KGP.

One main difference between KGP and AGENT0 is that in KGP (like BDI) we have plans and goals that are explicitly related through transitions, while in AGENT0 these relations are missing. Also, goals in KGP can have varying degrees of priority, which can be determined dynamically (not like commitments which are prioritised only according to time).

AGENT0 was only intended as a prototype, to illustrate the principles of agent–oriented programming. AGENT0 makes some limited provision for agents making requests for actions to other agents. This is an issue which we have not addressed in the KGP model. A further refinement of AGENT0 to deal with the limitation of such requests has been reported by Thomas in the Planning Communicating Agents (PLACA) language [Tho95]. Despite the improvements of PLACA over AGENT0, the language still inherits the gap that there is in AGENT0 between

the high–level concepts such as beliefs and commitments and the implementable architecture. In contrast, one of the main aims of KGP in using computational logic has been to bridge the gap between the formal model and the implementation.

Moreover, as we have seen in the introduction of this section, the KGP model has a more flexible control theory than AGENT0, which relies on a one-size-fits-all interpreter. In addition to that the KGP model is defined to facilitate a more concrete computational counterpart (to be developed within WP3). For example, the notion of how beliefs persist in AGENT0 is described in terms of informal guidelines, which need to be followed at the agentification (implementation) stage. Instead, in KGP, the way the knowledge persists and changes is described with concrete event calculus axioms that are precisely specified and can be be directly executed as logic programs.

Finally, the resulting AGENT0 language, although quite expressive, lacks sensing actions and at the same time makes a strong simplifying assumption, namely, that the internal beliefs of an agent are assumed to be consistent. Instead in KGP, the internal consistency of the knowledge base of the computee is not assumed but is ensured by the choice of the representation language of the computational logic used and the way observations are assimilated in the agent's knowledge base.

12.3 AgentSpeak

12.3.1 (Concurrent, Object-Oriented) AgentSpeak

The first version of AgentSpeak [WRR95] attempted to provide an agent-oriented programming language with BDI–like modelling capabilities such as PRS [IGR92] and appropriate language constructs, influenced by work in object-based programming languages. Agents in the system are organised into *agent families*, a class of agents whose instances offer specific types of *services* to other agents. Services are realized through the execution of an associated *plan*. Each agent is also associated with a *database*. Some of the services and a portion of the database could be public; i.e. available outside the agent. The remainder of the database and the services and all of the plans are private to the agent family. The language supports and extends concurrent object-oriented language features such as synchronous and asynchronous messages and has well-developed communication primitives for groups.

12.3.2 AgentSpeak(L)

Like with PRS, however, there was a large gap between AgentSpeak programs and the theory of the BDI model. To bridge this gap, Rao in [Rao96] proposes AgentSpeak(L), a programming language that can be viewed as an abstraction of the BDI implemented systems (such as PRS - described in [GI89a] and dMARS - in the way formalised in [dKLW98]) and allows agent programs to be written in a restricted first-order language with events and actions. In this context, Rao argues that the shift in perspective of taking a simple specification language as the execution model of an agent and then ascribing the mental attitudes of beliefs, desires and intentions, from an external viewpoint is likely to have a better chance of unifying theory and practice. It is worth saying here that KGP has been constructed very much in this spirit. In fact, the argument above, presented by Rao relates quite closely the KGP model and AgentSpeak(L), while at the same time it also enforces the link between of KGP with BDI.

The current state of an AgentSpeak(L) agent, which is a model of itself, its environment, and other agents, can be viewed as its current belief state. From this abstract description of an

agent's state one can conclude that an AgentSpeak(L) agent is like a BDI agent but different from a KGP one, in that it contains a model of itself and other agents. However, it is not clear from the description in [Rao96] how this claim is supported, as no examples are given in the modelling language to model, say, the self or other agents. An example is provided of how the environment can be represented, not in a "meta-level" sense but as a set of object-level predicates. If the modelling of the self and the other agents is meant to be done in this way (i.e. via ordinary object-level predicates), then an equivalent object-level representation can easily be expressed in KGP too.

States which the agent wants to bring about in AgentSpeak(L) can be viewed as desires, while the adoption of partially instantiated rules/programs to satisfy such desires can be viewed as intentions. Although the notion of goals are used to represent desires, the treatment of goals in AgentSpeak(L) is different from the treatment of goals KGP, where goals are data structures with temporal constraints and links to a goal hierarchy, forming part of the agent's state. On the other hand, intentions are similar to KGP plans in that in KGP plans are set of partially instantiated actions (the un–instantiated part may include the temporal constraints of the action) that change as a result of new incoming observations reflecting changes in the environment. However, KGP plans are not programs in the AgentSpeak(L) sense, as we will see shortly.

In AgentSpeak(L), an agent contains apart from a set of beliefs, a set of plans, and a set of intentions, also a set of events, a set of actions, and a set of selection functions. The selection of plans, their adoption as intentions, and the execution of these intentions are described by providing the operational semantics in terms of an interpreter that runs the agent programs specified in AgentSpeak(L).

The beliefs, desires and intentions are not defined as modal formulas, but instead as a set of base beliefs (or as Rao puts it: *facts in the logic programming sense*) and a set of plans. Plans are context–sensitive, event–invoked recipes that allow hierarchical decomposition of goals as well as decomposition of actions. The key characteristic to remember here is that plans are drawn from a plan library, that is, they are different from plans in KGP that are generated on–line. Similarly, although events (especially those generated externally from changes in the environment) in AgentSpeak(L) are like the KGP observations, a new observation in KGP may cause amongst other things a Goal Introduction or a Goal Revision, which combined with the dynamic planning facility allows for a more dynamic behaviour from that provided by AgentSpeak(L).

The proof theory of the AgentSpeak(L) language is provided in terms of a labelled transition system. The notion of *configuration* is a labelled description of the set of events, beliefs, intentions, and actions of an agent. Proof rules define how the agent transits from one configuration to the next. It is argued that these transitions have a direct relationship to the operational semantics of the language and hence help to establish the strong correspondence between the interpreter and the proof theory.

However, the interpreter of AgentSpeak(L), like the BDI cycle, is a one-size-fits-all interpreter, and not the flexible cycle provided in the KGP model. Moreover, up until recently, there was no implementation of the AgentSpeak(L) interpreter. AgentSpeak(L) also suffers from proof–rules being embedded in the cycle algorithm, that is, they are not defined separately and modularly, as in the KGP cycle theories.

Undoubtedly the AgentSpeak(L) work has opened up an alternative, restricted, first–order characterisation of BDI agents, which bridges the gap between theory and practice. However, another disadvantage of AgentSpeak(L), with respect to some other abstract agent–oriented

programming languages (e.g. 3APL as we will discuss it in section 12.4), is that it does not provide ways of dealing with plan failure. In fact, Rao has pointed out that AgentSpeak(L) events for goal deletion were supposedly intended for dealing with plan failures, but he did not include them in the semantics of the language. The *KGP* model deals with plan failure and goal failure with the Plan Revision and Goal Revision transitions, which update the agent state.

12.3.3 AgentSpeak(XL)

Bordini et al [BBJ+02, HM02] have implemented an extended AgentSpeak(L) interpreter which they call AgentSpeak(XL). This extended interpreter allows the programmer to handle plan failures as well as specify un–instantiated variables within negated belief atoms in the context (*KGP* preconditions) part of plans (AgentSpeak(L) required that only ground atoms are used in negated context conditions). However, this work emphasises that the formal semantics for these extensions is not as yet fully specified, but is work in progress. In the AgentSpeak(XL) paper, not even an informal account of these extensions have been defined, as the main focus of this more recent work has been to extend the original AgentSpeak(L) interpreter with a scheduler for the on-the-fly generation of plans.

Another important extension specified in AgentSpeak(XL) is that it improves AgentSpeak(L) by adding agent communication capabilities in the style of KQML. To accommodate these capabilities, changes to the original AgentSpeak(L) abstract interpreter have been defined in order to reflect the communications in which agents are engaged.

Finally, it is not clear in AgentSpeak and AgentSpeak(L) how resource boundedness can be supported, although an example of how to use AgentSpeak(XL) with deadlines is provided in [BBJ+02]. In *KGP* we have made some limited provision for resource boundedness within the selection functions, as discussed in section 8.

12.4 3APL

Another programming language for agent programming relevant to *KGP* is 3APL, presented in a number of articles by Hindriks et al [HdBvdHM98, HdBvdHM99a, HdL00, HdBvdHM99b]. Unlike *KGP* which is based purely on a declarative language, the 3APL language is a combination of imperative and logic programming. From the imperative programming viewpoint, 3APL inherits the full range of regular programming constructs, including recursive procedures and state-based computation. States of agents in 3APL, however, are belief (or knowledge) bases, which are different from the usual variable assignments of imperative programming. From the computational logic perspective, also taken by our *KGP* model, answers to queries in the beliefs of a 3APL agent are proofs in the logic programming sense.

12.4.1 Agent Programs

At run–time an agent program in 3APL is viewed as consisting of a set of Beliefs, a set of Basic Actions, a set of Goals, and a set of Practical Reasoning Rules. These concepts are discussed below, the discussion includes also their relation to concepts of the *KGP* model.

Beliefs – Although the beliefs in [HdBvdHM99a] are exemplified in a logic programming like (first-order) language with integrity constraints, any logical language, even a modal language could in principle be used. In *KGP*, however, we have already seen that the representation language is fixed to be a combination of ALP, CLP, and LP with Preferences.

Basic Actions – Another interesting feature of 3APL is that basic actions are viewed as simple goals in the language (because actions are seen as the capabilities that can achieve a particular state of affairs). This perspective is due to the fact that the language treats an agent more like a mental concept rather than something that requires an interface to an external environment (but see section 12.4.3 on communication). This is an important difference with the *KGP* model that models an agent as a mental entity too, but it also models explicitly the link of the agent with the environment through sensing and action execution.

Goals – Apart from basic actions, there are two other types of goals: *achievement goals* and *test goals*. Achievement goals act like procedures in imperative programming and have a procedural meaning for things that agent has to do. Test goals on the other hand, allow the agent to query its beliefs, and are evaluated relative to the current beliefs of the agent, similar to preconditions of actions in *KGP*. Together, the basic actions, achievement goals and test goals are the *basic goals* of the 3APL language. *Complex goals* are then composed from basic goals by using programming constructs for sequential composition and non-deterministic choice, constructs drawn from imperative programming.

It is important to say that the notion of goal is used in 3APL as a more general notion to that of intention, the reason being that intentions in the BDI context are often viewed as some kind of choice with an associated level of commitment [PL90]. The commitment made to a choice determines when an agent will reconsider or drop its intentions. An agent may adopt several commitment strategies towards its intentions. In 3APL, however, a goal reflects a choice that the agent has made, while there is no explicit level of commitment associated with each of the goals. The commitment or the revision strategies of an agent are more or less implicit in the practical reasoning rules of the agent (to be described below). Put another way, it is the practical reasoning rules that determine which goal can be considered as an intention.

Note the similarities and differences of 3APL and *KGP* goals. Like in 3APL, *KGP* goals also reflect choices made by the agent but the difference is that *KGP* supports a specific Goal Decision theory for choosing goals. In addition, *KGP* also provides explicit transitions for Goal Introduction and Goal Revision, as a way to deal with the similar notions of commitment and revision strategies in the style of BDI.

Practical Reasoning Rules – The main purpose of the practical reasoning rules is that they supply the agent with a facility to manipulate goals. In our framework we can interpret the practical reasoning rules as specifying plans to achieve goals Thus, practical reasoning rules can be used to build a plan library from which an agent can retrieve plans for achieving goals. In addition, they provide a facility for monitoring goals, similar to the concept of guards, as in concurrent programming languages, to allow the agent to commit to a plan. It is important to note that in [HdBvdHM99a] practical reasoning rules are classified under four classes:

- *Failure rules* – these are rules that revise the goals of an agent in order to avoid failure, or clean up the state of the agent after failure. Alternatively, the agent may substitute some alternative means to deal with a failure situation and try to achieve a goal in some other way. To avoid behaviour that leads to catastrophic consequences, this type of rules are assigned highest priority.

- *Reactive rules* – the application of these rules does not depend on the current goals of the agent but only on the current beliefs. This class of rules specify a plan of action in the body of the rule which is required to respond appropriately to a type of situation that is represented by the conditions (guard) of the rule. As agent responses using these rules are time critical, the priority assigned to these rules is the second highest.

- *Plan rules* – agents use these rules to find the appropriate means to achieve their achievement goals. Plan rules provide a plan in the body of the rule to achieve the achievement goal in the head of the rule. The lower priority assigned to this class of rule, third highest, reflects the assumption that it is important to avoid failure and be reactive, and that only after this has been accomplished the agent should worry about how to be proactive.

- *Optimisation rules* – these rules are used to identify situations where a suboptimal plan is being pursued by the agent to achieve a goal, and, if such situations are found, then the rules optimise the behaviour of the agent by choosing a more optimal way to achieve it achievement goals. Without these rules nothing can go wrong, but without them more cost may be induced by the agent's current plan than necessary. For this purpose, optimisation rules have the lowest priority.

Given the above structure of the practical reasoning rules, there is an obvious similarity between Failure rules and Goal Revision in KGP, Reactive Rules and Reactivity in KGP, Plan Rules and Plan Introduction in KGP. The difference is that the classification and prioritisation scheme above is not imposed by the 3APL language, but it is recommended by the designers of the language when writing a practical application. Instead, in KGP we have committed to a specific structure of transitions that are explicitly supported by the model, in that they have a formal specification, whose prioritisation is specified modularly in the cycle theories according to the agent personality. There are no Optimisation Rules in KGP, but we discuss such rules in section 11 as a possible extension to the existing model.

12.4.2 Operational Semantics and Control

The operational semantics of 3APL are specified in terms of transition rules, which – similarly to AgentSpeak(L) – are relations on so-called configurations, but here transitions are not labelled. Two distinct classes of transition rules are defined. The first type defines what it means to execute a single goal given the current belief base of an agent and the current computational state (which include the variable bindings). At this level, transition rules provide a formal specification for 3APL basic actions, achievement goals, test goals, complex goals, and practical reasoning rules. The second type of transitions is defined in terms of the first type and defines what it means to execute an agent in terms of executing multiple goals.

To deal explicitly with selection mechanisms for goals and actions, 3APL introduces a separate formal specification for the control structures of the agent language. A second transition system is introduced, called the *meta-transition* system, which includes features for referring to the object level language, as well as operators for programming control structures for the object (agent) level. The meta-transition system also supports a set of basic actions that allow the agent to select, apply rules, and execute goals. For this purpose four actions are introduced: (1) an action for selecting an applicable rule, (2) an action for the application of a number of rules, (3) an action for selecting an enabled goal, and (4) an action for the execution of a set of goals. Finally, constructs for expressing the preference order over goals and rules (such as the practical reasoning rules base) are also provided.

The control structure that is proposed by 3APL [DdBD$^+$02] is a specialisation of a one-size-fits-all update-act interpreter [Sho93, Rao96, KS96a] as follows:

Update-Act Cycle
1 Select a rule R to fire

2 Update the goal base by firing R
3 Select a goal G
4 Execute (part of) G
5 Goto 1

Steps (1) and (2) form part of the *application phase*. In step (1) a rule is selected using the classification of rules. In step (2) the goal base of the agent is modified by applying the selected rule. Steps (3) and (4) constitute the *execution phase*. At step (4), it is decided when part of the goal is to be executed. A goal is not completely executed, as there is no way of deciding in advance whether or not it is possible to execute a goal completely. In addition, if a goal is completely executed, there is no reason of having fail rules to check possible failure when executing a goal. So, at each cycle at most one rule is fired and at most one goal (is partly) executed. The full representation of the cycle in the 3APL meta-language is presented in [HdBvdHM99a]. In [DdBD$^+$02] an extension of the meta-language is discussed, whose aim is to make the cycle of 3APL programmable, however, all articles about 3APL control leave many issues as part of future work.

In *KGP* we share the 3APL aims, in particular, to make the cycle theory programmable and the selection mechanisms explicit [DdBD$^+$02]. For this purpose *KGP* proposes a core set of selection functions that can be extended via heuristic rules to model different behaviours and types of agents. *KGP* also supports declarative theories for the cycle in order to provide declarative control, while in 3APL the cycle relies more on imperative control constructs. In addition, we allow for flexible orderings of transitions in that in *KGP* we can reason with preferences about which transitions can be applied at a specific point in time. These preferences may change according to external events or changes in the knowledge of the agent. Instead in 3APL the ordering is fixed. Within this fixed cycle it is possible to program specific operational strategies on how the agent is to achieve its goals depending on the goals and other conditions in the environment. This is in contrast with *KGP* where such strategies are separated out as part of the preference policies of the computee in its knowledge base, and where in the cycle we can arrange for the general operation of the computee according to some overall profile of behaviour that we want the computee to have. Hopefully, our use of declarative control theories will also allow us to specify and prove formal properties.

12.4.3 Agent Communication

3APL also provides support for communication between agents in two ways. The first, described in [HdBvdHM99b], extends the set of basic actions with synchronous communication actions such as tell and ask to exchange information or offer and request for requests between agents. The arguments of the synchronous communication actions are unified and the resulting variable bindings are the result of the communication that form the contents of the messages being exchanged. In this approach, the synchronised actions are matched on the basis of the performatives they enact, i.e. the tell action is a speech act with performative 'tell', which should be matched with the complementary performative 'ask'. Deduction is used to derive information from a received message, while abduction is used to provide semantics for request and offer.

To avoid some of the problems that arise from synchronous communication as in Hindriks et al [HdBvdHM99b], the work of Dastani et al [DvdHD02] extend 3APL with asynchronous communication. In this approach 3APL agents send and receive messages with contents com-

pliant to the FIPA specification. In order to model asynchronous communication in the 3APL framework, the 3APL specification is extended with a buffer, called the *message–base*. The message-base of a 3APL agent contains messages that either are sent by the agent to other agents or are received from other agents. The message base makes possible for an agent to continue with its own goals after sending a message. It does not have to wait for the receiving agent to synchronise before it continues. In the same way, the receiving agent can receive messages at any time and does not have to form a goal to receive a message.

Like 3APL, the KGP model supports both synchronous and asynchronous communication as follows. Messages are special kinds of actions called *communicative actions*. Sending messages involves applying Action Execution on communicative actions. Receiving messages involves applying Active Observation - if the mode of communication is synchronous, or Passive Observation - if the mode of communication is asynchronous. Note, however, that KGP abstracts away from the model the issue of how messages are being sent and received, as this is a matter of the model's implementation. Instead, in 3APL, the work discussed in [DvdHD02] mixes in the modelling part implementation issues such as the notion of the message–base, which is essentially an implementation concept of a message buffer. In addition, in KGP we can specify private communication policies, which can be imported in the agent in a modular way.

12.5 DESIRE

DESIRE (DEsign and Specification of Interacting REasoning components) is a high–level modelling framework that explicitly models the knowledge, interaction, and coordination of complex tasks and reasoning capabilities in agent systems [BDKJT97, BDKTV97, BKJT95]. The framework views both individual agents and the overall system in terms of a compositional architecture – where all functionality is designed as a series of interacting, task-based, hierarchically structured components.

Tasks are characterised in terms of their inputs, their outputs and their relationship to other tasks. Interaction and co–ordination between components, between components and the external world, and between components and users is specified in terms of informational exchange, sequencing information and control dependencies. The components themselves can be of any complexity, from simple functions and procedures up to whole knowledge-based systems, and can perform any domain function (e.g. numerical calculations, information retrieval, optimisations, etc).

In [BDKTV97], DESIRE has been extended to define a generic BDI model to incorporate beliefs, desires and intentions (in which intentions with respect to goals are distinguished from intentions with respect to plans). The result is a more specific BDI agent where an agent's task control is capable of six tasks: the own process control deals with how the agent determines its own beliefs, desires and intentions, the agent specific tasks deals with the agent performing its own tasks, the world interaction management deals with managing interaction with the environment, the agent interaction management deals with communication with other agents, the maintenance of world information deals with modelling the world, and the maintenance of agent information deals with modelling other agents.

In KGP terms, we can think of the agent's task control as the cycle theory, whose various tasks map into (combinations of) transitions and/or capabilities. Briefly, we can say that the world interaction management maps to the Sensing capability and the Action Execution transition in KGP, the agent interaction management maps naturally to the combination of

Active Observation and Passive Observation transitions of KGP, the agent specific tasks maps to the Action Execution transition of KGP, and the maintenance of world information maps to the way we use the Temporal Reasoning in the KB_0 part of the knowledge base. A more detailed comparison between KGP and the rest of an agent's tasks in DESIRE follows next.

The agent's own process control task consist of three main components, as follows.

- The belief determination component – performs reasoning on relevant beliefs and includes beliefs that change as a result of observations. The equivalent of belief determination in KGP is the notion of proof (in the logic programming sense), combined with the temporal reasoning using an extended version of the Event Calculus, in order to deal with beliefs that change as a result of observations.

- The desire determination component – determines the desires of the agent (knowledge on how desires are generated is left unspecified in the model). In KGP we handle goal determination through the Goal Introduction and Goal Revision transitions, based on a theory of needs and personality. We also have argumentation as a way of specifying preferences over goals.

- The intention and commitment determination component – derives the agent's intended and committed goals and plans. Although in KGP we have Plan Introduction and Plan Revision to deal with committed plans, we do not have in the model the DESIRE distinction between intended and committed goals and plans. In other words, in KGP the agent commits to all plans and goals without any further distinction, but uses the notion of preferences to choose between goals that are, for example, more urgent.

Although we have seen that KGP does not distinguish between intended and committed goals and plans, it is important for the discussion that follows to say that the intention and commitment determination component introduces the goals and/or plans it intends to pursue before committing to the specific selected goals and/or plans. In order to do that DESIRE uses two main sub-components, as follows.

- The goal determination component – which consists of two additional sub–components, the intended goal determination and the committed goal determination, in order to determine the intended and committed goals of the agent respectively. The intended goal determination contains the goals that the agent intends to pursue; in this component different agents also hold different strategies specifying under which conditions a goal needs to be revised. In the committed goal determination, on the other hand, a number of intended goals are selected to become goals to which the agent commits; again different agents have different strategies for selecting committed goals. In both cases the components hold selection strategies specified as meta–knowledge. In KGP terms the union of intended and committed goals is a superset of the KGP goals, while the committed goal determination can be thought of as a combination of the Goal Decision capability as it is used in the goal transitions and goal selection functions of the cycle theory.

- The plan determination component – which consists of two additional sub-components, the intended plan determination and the committed plan determination, in order to determine the intended and committed plans of the agent respectively. In the component intended plan determination plans are generated dynamically, combining primitive actions and predefined plans known to the agent (stored in an implementation, for example, in a library). On the

basis of knowledge of the quality of plans, committed goals, beliefs and desires, a number of plans become intended plans. The component committed plan determination determines which of these plans should actually be executed. In other words, to which plans an agent commits. If no plan can be devised to reach one or more goals to which an agent has committed, the result is communicated back to the goal determination component. There is no equivalent of intended plan determination in KGP, however, the committed plan determination can be obtained in KGP terms by a combination of the Planning capability with the Plan Introduction and Plan Revision transitions of the cycle theory.

The global reasoning strategy specified by task control knowledge in the model is that some chosen desires (depending on knowledge in the component intended goal determination, existing beliefs and specific agent characteristics) become intentions, and some selected intentions (depending on knowledge in the component committed goal determination and specific agent characteristics) are translated into committed goals to the agent itself and to other agents. The agent then reasons about ways to achieve the committed goals on the basis of knowledge about planning in the component committed plan determination, resulting in the construction of a committed plan. This plan is transferred to one or more of the other high–level components (e.g. world interaction manager) for execution.

Task control knowledge of the component own process control determines the following process:

1. initially all links within the component own process control are awakened, and the component belief determination is activated.

2. Once the component belief determination has succeeded in reaching all possible conclusions (specified in the evaluation criterion goals) desire determination is activated and belief determination is made continually active (awake).

3. Once the component desire determination has succeeded in reaching all possible conclusions (specified in the evaluation criterion desires), the component intention and commitment determination is activated and desire determination is made continually active (awake). In addition, the desires in which the agent may want to believe (wishful thinking) are transferred to the component belief determination.

4. Intended and committed goals and plans are determined by the components goal determination and plan determination. Each of these two components first determines the intended goals and/or plans it wishes to pursue before committing to a specific goal and/or plan. In the component goal determination commitments to goals are generated in two stages. In the component intended goal determination, based on beliefs and desires, but also on preferences between goals, specific goals become intended goals. Different agents have different strategies to choose which desires will become intentions.

Differences in agent characteristics can be expressed in the (meta-)knowledge specified for intended goal determination. For each intended goal a condition is specified that expresses the adequacy of the goal, i.e., that the goal is not subject to revision. As soon as it has been established that the intention has to be dropped, the intended goal becomes inadequate, so this condition no longer holds, which in turn leads to the retraction of the intended goal on the basis of the revision facilities built-in in the execution environment of DESIRE. These characteristics are similar to KGP, but the difference is that in KGP we use the transitions

for Goal Introduction and Goal Revision, and argumentation for the reasoning required about goals.

DESIRE also allows different agents to have different strategies to select committed goals, and these different strategies can be expressed in the (meta-)knowledge specified for the component committed goal determination. In KGP it is again the transitions of Goal Introduction and Goal Revision that are changing the commitments to current goals, selected modularly by a separate goal decision theory.

Another similarity between DESIRE and KGP is that they both allow for dynamic plan generation. DESIRE uses the component plan determination where commitments to goals are analysed and commitments to plans are generated in two stages. In the component intended plan determination plans are generated dynamically, combining primitive actions and predefined plans known to the agent (stored in an implementation, for example, in a library). KGP on the other hand, uses the Planning capability, which is called in Plan Introduction to generate partial plans.

One major difference between DESIRE and KGP is that in DESIRE if a plan has been devised, execution of a plan includes determining, at each point in time which actions are to be executed. However, in KGP action execution is determined by when the Action Execution transition will be called by the cycle theory. However, similar to DESIRE, where during plan execution, monitoring information can be acquired by the agent through observation and/or communication, KGP too support for this to happen through the sensing capabilities and the observation transitions. Both models allow for plans to be adapted on the basis of observations and communication, but also on the basis of new information on goals to which an agent has committed.

One issue that it is unclear with DESIRE is that nowhere in the specification of the system's control one can find a notion similar to that of the KGP interrupt.

Finally, the formal specification of DESIRE is based on a many–sorted predicate logic [DKT95], which distinguishes between object-level and meta-level descriptions of components. In KGP we have logic programming with extensions, where as we said in the comparison with BDI, we have not included meta–level reasoning. The dynamics of the overall compositional system in DESIRE is modelled through temporal models based on temporal logic [BTWW95], a feature that is beyond the scope of the KGP model.

12.6 Computational logic-based approaches

12.6.1 IMPACT

The principal goal of the IMPACT (Interactive Maryland Platform for Agents Collaborating Together) project [AEK$^+$99, ES98, ESP99, ES99, ESR00, SBD$^+$00], has been to develop both a logic-based theory as well as a software implementation that facilitates the creation, deployment, interaction, and collaborative aspects of software agents in a heterogeneous, distributed environment. IMPACT provides a set of servers (yellow pages, thesaurus, registration, type and interface) that facilitate agent inter-operability in an application independent manner. It also provides an Agent Development Environment for creating, testing, and deploying agents.

Unlike the assumption we make in the KGP model, 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. To see how this is achieved in IMPACT, we need to look closer at the structure of IMPACT agents. Each one of these agents consist of the following components:

- **Application Program Interface** (API): provides a set of functions which may be used to manipulate the data structures managed by the agent in question. This component consists of a set of procedures that enable external access and utilisation of the system, without requiring detailed knowledge of system internals such as the data structures and implementation methods used.

- **Service Description**: specifies the set of services offered by the agent.

- **Message Manager**: manages the incoming and outgoing messages of the agent.

- **Actions, Action Policies, and Constraints**: describe the set of actions that the agent can physically perform, an associated action policy that states the conditions under which the agent may, may not or must do some actions. The actions an agent can take, as well as its action policy, must be clearly stated in some declarative language. Furthermore, there might be constraints stating that certain ways of populating a data structure are invalid and that certain actions are not concurrently executable.

- **Meta–knowledge**: holds beliefs about the environment and other agents, used to produce action policies.

- **Temporal Reasoning**: supports an agent to schedule actions that take place in the future, which could be interpreted as the commitments of the agent.

- **Reasoning with Uncertainty**: allows the agent to take into account that a state can be uncertain. For example, based on its sensors, an agent may have uncertain beliefs about the properties of the environment's state, as well as uncertainty about how the environment is likely to change.

- **Security**: supports the designer of the agent to enforce security policies according to the application requirements of the agent.

If we ignore implementation components such as the API and the Message Manager, an IMPACT agent appears to be similar to a KGP one as far as they can both support actions, action policies and constraints. However, the formalism used to represent these notions in the two models differ in that in IMPACT the actions are STRIP like structures (with preconditions, an add and a delete list) while in KGP actions are data structures that abstract away from what is recorded in the KB (this is handled by the way observations are assimilated in KB_0). In addition, the IMPACT specification supports concurrent actions, which in KGP are supported in the execution of actions, planning for goals and sensing of fluents and preconditions. Concurrency at the level of KGP transitions is a subject for future extensions.

Another similarity between IMPACT and KGP is that in IMPACT integrity constraints have a logic-based interpretation like in KGP, however, they differ from the ones specified in KGP in that their syntax also allows the programmer to access data structures that represent existing programs/information sources, to allow for information integration using IMPACT agents. Apart from actions, action policies and integrity constraints, IMPACT also provides a richer language than that of KGP, in that it allows in the rules that describe actions to be specified using deontic concepts, by building on existing deontic systems that use operators about permission and obligation explicitly in the language.

One main difference between IMPACT and KGP is how the Temporal Reasoning capability is interpreted in an agent. In IMPACT this is supported through Temporal Agent Programs [DKS01] that rely on a simple interval-based logic [All84], and introduce a mechanism for specifying intermediate effects of an action. A sound, iterative, fix-point computational procedure that is also complete, and polynomial–time under certain conditions is also proposed. Compared with IMPACT, Temporal Reasoning in KGP is, as we have already discussed, based on an abductive Event Calculus that is extended to reason with incomplete information about fluents and inconsistencies arising from observations in the environment. Existing proof-procedures are then used to interpret the temporal logic programs that are required in this context.

It is important to note that the IMPACT support for Reasoning under uncertainty (using probabilities in the logic-based rules) and Security is not provided in KGP. In addition, there is no equivalent of the IMPACT meta–knowledge in KGP. We have also already seen that in KGP we can only have a much poorer model of other agents, in that we do not allow simulation of their reasoning capabilities, just simple beliefs about them. However, planning in IMPACT is complete, in the sense that the system does not support partial plans as we do with KGP. Also, reactivity in the sense of KGP is not supported in IMPACT. Moreover, communication in IMPACT is an implementation concept that allows an agent to use a message box that contains functions for sending and receiving messages. In KGP, however, communication is supported by a separate class of actions, called communicative actions, that are uniformly represented in communication policies described by integrity constraints, as we have shown in detail in section 10.

To summarise, the IMPACT project proposes a unifying approach for many different features of agent behaviour based on the adoption of computational logic as the underlying methodology for system development and analysis. On one hand, IMPACT is centered on the integration, based on agentification, of heterogeneous, possibly legacy, information sources, and in their co-operation in order to successfully accomplish a coordinated task. The notion of agentification, however, often makes it difficult to distinguish the implementation from the modelling of an agent. On the other hand, our work keeps the different concerns of the implementation separate from the modelling, and focuses on building autonomous agents by integrating existing logic programming techniques and their extensions in order to cope with highly dynamic nature of open and global computing environments. The different motivations of KGP and IMPACT make apparent the different ways agents are modelled in the two systems, most notably differences in the treatment of beliefs and actions, including the treatment of temporal reasoning, planning, goals, communication, proofs, and control.

12.6.2 $\mathcal{MINERVA}$

$\mathcal{MINERVA}$ is an agent architecture which exploits computational logic as a means for integrating diverse non-monotonic formalisms within a unique (dynamic) model, [LAP01b]. The basic architecture consists of a structured knowledge base encompassing both the knowledge of the agent, i.e. its representation of the environment and other agents, and (BDI-like) features, like capabilities, intentions, goals, and plans. The knowledge base is controlled by a varying set of modules, each of which is devoted to a specific task, like, for instance, a communicator, a sensing and reacting module as interface with the environment, a planner, a learner, and a scheduler module. All of these components update the agent's knowledge base.

At first glance, the architecture resembles many of the features that we have proposed in

KGP in order to model agent behaviour. However, unlike *KGP*, $\mathcal{MINERVA}$ relies on the Multidimensional Dynamic Logic Programming (MDLP) model [LAP01c], in order to represent the dynamic evolution of an agent's knowledge. MDLP is a non–monotonic LP–based model which models evolutions of a (generalised) Logic Program, and it is equipped with both a stable-model-based declarative semantics and an operational semantics.

MDLP is an extension of Dynamic Logic Programming (DLP) [ALP$^+$00], initially designed for modelling program evolutions over (linear) time. DLP contains "persistent by inertia" rules and caters for events that can change the program. DLP makes use of a logic program command language (LUPS) for specifying updates [APPP99]. In this language one can specify update rules for assert/retract under certain (event-based) conditions. In DLP, program evolutions are linear and the program at each state (instant) is defined by all the rules belonging to previous states which have not been overridden during the evolution of the program and all the newly added rules [Lei02].

MDLP extends DLP in allowing for a non-linear structure of programs evolutions, as a directed acyclic graph. In this context, the time–line relation becomes a 'dependency' relation which can model different situations, as a program can be considered as the (consistent) evolution of its (many) ancestor programs in the graph.

The agent architecture consists of a knowledge base (KB) and of specialised, concurrent, sub-agents. The KB is conceptually divided in modules which are based on MDLP for knowledge representation and LUPS for state transitions. KB modules consist of:

- **Object Knowledge Base** – which contains object-level knowledge about the world represented as an MDLP, as well as information about the society in which the agent is situated.
- **Capabilities** – which describe the actions that the agent is able to perform and their effects. Typically, the execution of an action is considered as an event that may trigger a KB update.
- **Intentions** – which consist of the actions which an agent has committed to according to its plans, and are subject to conditions and temporal constraints.
- **Goals** – which contain the goals that must be achieved by the agent. Each goal has a priority and a temporal constraint. Goals are managed by a Goal Manager which can update them.
- **Plans** – planning is supported by means of LUPS and abduction [LAPQ00], and consists of a (conditional) sequence of (timed) actions. Plans are generated by a Planner sub-agent and stored in KB for future reference. Plans in the KB are executed, together with Reactions (see below), by a Scheduler sub-agent.
- **Reactions** – which are a set of rules describing the (timely) reactive behaviour of an agent.
- **Internal Behaviour Rules** – describe the rules of the (reactive) behaviour of an agent which may affect the Object Knowledge Base. They are managed by a Reactor sub-agent, which is in charge of 'executing' the behaviour of an agent, but are relevant also for other sub-agents, like, for instance, the Planner, which must be aware of the actions that have been executed.

- An **internal clock**.

The architecture is modular in that it is composed of functional sub-agents, which may add and remove functionality to the single agent. This is similar to the modular specification in KGP.

In addition to the sub-agents already discussed, a $\mathcal{MINERVA}$ agent also consists of, among others, a *Sensor*, which perceives the external world and updates the KB, a *Dialoguer*, whose LUPS program can update KB and Goals according to the communications it performs with the other agents, and a *Goal Manager*, in charge of managing goals, possibly resolving conflicts among goals generated by different sub-agents.

Interestingly, many of the apparent similarities between $\mathcal{MINERVA}$ and KGP are really differences, if we look at the details. For example, what KGP uses as a sensing capability with transitions that support Passive and Active Observation in $\mathcal{MINERVA}$ it is a Sensor sub-agent, when the knowledge base of KGP is updated implicitly via the use of the Event Calculus $\mathcal{MINERVA}$ specifies the evolution via explicit rules, while in KGP communication is only a special kind of action in $\mathcal{MINERVA}$ there is a Dialoguer sub-agent, and the responsibilities of Goal Manager are incorporated in KGP via special transitions that rely upon capabilities incorporated in the cycle theories, with preferences specified via argumentation. Moreover, the KGP link to society via a series of components such as Action Execution, Active and Passive Observation to support communication, is achieved in $\mathcal{MINERVA}$ agent via the Dialoguer sub-agent that works by combining inter- and intra-agent viewpoints, as it has been demonstrated in [LAP01a].

There are also many actual similarities between $\mathcal{MINERVA}$ and KGP, namely, the notion of goals in both systems, the $\mathcal{MINERVA}$ intentions with the KGP actions, the notion of plans and reaction in both system. An apparent difference, however, between $\mathcal{MINERVA}$ and KGP is that it is not clear in $\mathcal{MINERVA}$ how the control of the agent is achieved, while KGP models control in an explicit and flexible manner, through the notion of cycle theories.

In conclusion, the $\mathcal{MINERVA}$ project appears to share with KGP many underlying assumptions for modelling an agent, most notably the use of Computational Logic and the exploitation of the twofold reading of agents in terms of a declarative and an operational semantics. Although the overall updating mechanisms for reasoning on changes appear based on a clear and expressive model, it is difficult, at this stage, to compare its expressiveness with that of KGP. At this level of comparison, however, it will be fair to say that the way evolution is modelled in $\mathcal{MINERVA}$ can be complementary to the way the KB of KGP is being revised; given a set of conditions on events that have happened, the former revises the KB with new rules while the latter implicitly changes it with new facts. Still, there are important differences between the two approaches, notably the commitment to different representation formalisms that rely on different assumptions and LP semantics.

12.6.3 GOLOG

GOLOG, after alGOl LOGic, [LRL$^+$97] is another approach to a logic-based modelling of multi-agent systems. As the name suggests, GOLOG is a language which tries to import the programming paradigm of a procedural language like Algol into the realm of logic. In particular, it is based on the situation calculus [MH69], which represents a sophisticated logic of actions.

GOLOG is provided with procedural constructs like sequencing, choice and iteration of situation transforming actions, and it has a computational implementation based on Logic Programming. An explicit representation of the dynamic world being modelled evolves according

to actions, which are characterised by (user supplied) axioms about their preconditions and effects. This allows programs to reason about the state of the world and consider the effects of various possible courses of action before committing to a particular behaviour. Given a representation of the word, the pre-/post- conditions for actions and the agent program, it is possible to prove that the course of actions, when executed, brings the world in a desired state.

A concurrency-based extension of the original language [GLL00] provides an high-level agent control based on facilities for prioritising the concurrent execution, interrupting the execution when certain conditions become true, and dealing with exogenous actions. This sort of high-level agent control constitutes an alternative to planning, being the courses of actions constrained by their concurrent synchronisations. A distinguishing feature with respect to other procedural formalisms for concurrency, which are valuable when modelling open systems, is the possibility of dealing with incompletely specified states, which represent a partially accessible environment.

This logic-based approach to modelling the evolutions of a dynamic word shares with the our work here many motivations, and in particular the interest in a representation of the agent's state which allows for reasoning about changes occurring over time. However, while GOLOG, and ConGOLOG, can be seen as models of a concurrent execution of different action-based programs, our work here is more oriented towards the concept of agents as autonomous and flexible entities which exhibit a richer set of features other than a concurrent execution of actions, like coordination of their tasks, capability of executing a negotiation dialogue, adoption of common policies ruling their behaviour, activities of knowledge and plan revision, failure recovery, as well as dealing with open-world settings such as environments envisaged in global computing.

To evaluate programs in an open world setting, an extension of Golog and ConGolog is specified in [GLS01], and is known as IndiGolog (Incremental deterministic Golog). This system allows a programmer to specify guarded action theories that allow a programmer to control the online and off–line execution of conditions. In the online execution case, a sensing capability affects the current state of the computation, which is obtained by incrementally executing programs represented as guarded theories. Such implementation is provably correct under certain conditions, and is reminiscent of the KGP combination of sensing capability combined with the knowledge revision obtained by event calculus theories.

12.6.4 Vivid Agents

We close our related work with *vivid agents* [SW00], a software-controlled system whose state comprises the mental components of knowledge, perceptions, tasks, and intentions, and whose behaviour is represented by means of action and reaction rules. The basic functionality of a vivid agent comprises a knowledge system (including an update and an inference operation) acting on a knowledge base specified with an Extended Logic Program [AP92], a perception (event handling) system, and the capability to represent and perform reactions and actions in order to be able to react to events, and to generate and execute plans.

Reactions may be immediate and independent from the current knowledge state of the agent but they may also depend on the result of deliberation. In any case, reactions are triggered by events which are not controlled by the agent. A vivid agent without the capability to accept explicit tasks and to solve them by means of planning and plan execution is called reagent. The tasks of reagents cannot be assigned in the form of explicit ('see to it that') goals at run time, but have to be encoded in the specification of their reactive behaviour at design time. The concept of vivid agents is not based on a specific logical system for the knowledge base of

the agent. Rather, it allows to choose a suitable knowledge system for each agent individually according to its domain and its tasks.

There are a number of similarities between vivid agents and KGP, for example the use of reactions and plans being described separately in the model, the use of transitions to specify formally the evolution of an agents state, and the use of a LP–based cycle to represent the control of the agent. However, the use of transitions in vivid agents are used for capturing the temporal behaviour of the agent, which in KGP is done via the specification of Event Calculus theories. Also, the notion of transitions is more general in KGP, where goal, plan and action representation and execution are supported via the integration of existing proof-procedures and their extensions. In addition, these proof procedures allow KGP to plan and specify the reasoning with preferences in order to manage a flexible control mechanism based on a cycle theory. Moreover, communication in vivid agent is specified at a low-level, while in KGP it is specified abstractly as part of the modelling of the actions.

13 Evaluation

This document reports of the work carried out within Workpackage 1 (WP1): *"A logic-based model for computees"* of the SOCS project during its first year. As such it aims to address the objectives and requirements set out for this workpackage. In this section we examine, with respect to a given set of evaluation criteria, to what extent the KGP model (at its current state of development) achieves these objectives.

The evaluation criteria for WP1 have been proposed in the self assessment deliverable D3 [LMM$^+$03] of the project, where, in particular, the main objective of WP1 to provide a formal logic-based model for computees has been analysed within the Global Computing (GC) vision. The criteria are divided into two groups. The first group relates directly to the basic requirements of GC that SOCS aims at addressing. The second group of criteria are derived from the criteria of the first group as properties that will help the model address the requirements set out by this first group, and are motivated by SOCS' specific approach. These criteria are set out as follows:

First Group of GC Related Criteria

ADAPTABILITY – The model should allow the computee to be autonomous and adapt its operation to a changing environment.

TOLERANCE TO PARTIAL INFORMATION – The model should allow computees to be tolerant to partial information, by continuing to operate and take decisions despite the incompleteness of the available information. Also, the model should cope with the increase of information available to the computees over time.

OPENNESS– Societies are dynamic, in that computees might join and leave them at all times. As a result of this dynamicity, the computees already within the dynamically changing societies need to adjust their decisions to reflect the new state of the environment, due to the addition/deletion of other computees (**OPENNESS (1)**). Moreover, the computees that move and join new societies, should adjust their decisions and behaviour to take into account the different expertise and the different social rules available in the new society (**OPENNESS (2)**).

DISTRIBUTION – The model should allow an individual computee to be aware of other computees that could potentially help it to achieve some of its goals.

HETEROGENEITY – The model should support a variety of different types of heterogeneous computees in order to be able to address in a decentralised and distributed way the various requirements of GC.

Second Group of General Model Properties

MODULARITY of the representation of knowledge and structure of computees. Clear identification of independent components and parameters of the model of computees. Provision for the modular placement of a computee in and out of a society.

COMPUTATIONAL VIABILITY of the model. Each component of the model must have a clear computational counterpart that is viable within the current state of the art or well specified extensions of this. The overall model should be amenable to an abstract design and a well defined abstract architecture.

FORMALITY of the model. It should be possible to identify formal notions of alternative behaviour of a computee and link these formally to the internal structure and operation of the computee.

RELATED WORK – Investigate thoroughly related work in the field of modelling agents in multi-agent systems, with particular emphasis on models which are based upon logic in general and computational logic in particular.

Let us examine how the KGP model fairs under these (general) evaluation criteria examining each one of these separately. For each such criterion we will describe which elements of the model address it and to what extent they do so. We will also indicate which of the specific criteria, in terms of which each one of these general criteria is analysed in the self assessment deliverable D3 [LMM$^+$03], are addressed fully or partially by the KGP model. Specific criteria which are not addressed at all by the current computee model will be mentioned explicitly.

ADAPTABILITY – The model builds in explicitly features of autonomy and adaptability. The capability of Goal Decision allows the computee to decide its own goals and the model is built assuming that the computee has its own control on the goals that it will try to satisfy. The particular CL framework used in the model with its high-order preferences in the policy for goal decision accommodates directly the possibility to adapt these decisions to new circumstances. Similarly, the cycle theory policy which is given in the same framework allows for changes in the operation of a computee under different external and internal circumstances. In addition, the KGP model contains the capability of Reactivity which addresses directly the issue of being able to adapt to new information. This allows the computee to adapt its current plans and goals as a reaction to new input not available at the time of planning.

Also the Plan and Goal Revision transitions of a computee enable this to abandon goals and plans that have become un-achievable or that are already achieved, thus again adapting to a changing environment.

Hence the combination of the Goal Decision and Reactivity capabilities with their respective transitions and the Plan and Goal Revision transitions ensure that the specific criteria of *Reactivity, Adjustment/Abandonment, Suspension/Introduction*, as described in the evaluation deliverable D3, are fully met.

With respect to the problem of adapting to the societies that the computee belongs to as it moves from one to the other, the modular specification of the KGP model allows the

computee to reason with the policies of the relevant society as indicated in the knowledge base (e.g in KB_{react}) of the computee. This provides a transparent adaptation of the computee to the environment of the new society as required by the specific criterion of *Adoption of social goals*.

The specific criteria *Evolution and Discovery/Learning* have not been addressed in the current form of the computee model although as alluded to in section 11 these issues can be accommodated for by extensions of the model.

DEALING WITH PARTIAL INFORMATION – One of the main underlying reasoning ability of the computees in the KGP model is that of hypothetical reasoning (abduction) formalised within Abductive Logic Programming. Abductive reasoning is particularly suited for handling partial or incomplete information. Hence the Planning capability of the KGP model which is based on abduction allows the generation of plans in the absence of complete information. Abductive reasoning is also integrated within the Temporal Reasoning capability so that the computee is able to draw conclusions even in the face of incomplete information. Hence their conclusions are conditional on hypotheses about the unknown part of the computee's environment. This then addresses directly the specific criterion of *Conditional Decisions* as described in the deliverable D3.

In all these cases the abductive hypotheses on which the plan or the conclusions rest need to be linked to the operation of the computee. For this reason the KGP model contains transitions, such as Sensing Introduction and Active Observation Introduction, that are specifically designed to enable the computee to execute its plans and monitor their results in the face of incomplete information. In particular, using such information gathered from its environment the Temporal Reasoning capability of the computee allows it to recognise when actions that have been executed have failed to produce their desired effects. Thus this combination of hypothetical reasoning and sensing the environment enables the computee to operate in a partially unknown setting and to adjust its decisions and operation as more information is made available. This covers the specific evaluation criteria of *Adjustment and Correction* and partially the criterion of *Explanation*.

In addition, the flexibility and modularity of the cycle theory allows the specification of computees that a re better suited to environments where information is more or less easily available. For example, in an environment with less available information, computees can attempt to execute their actions despite lack of information regarding the preconditions of these actions.

OPENNESS – The KGP model can deal with computees joining and leaving a society, as discussed in section 10.3. As mentioned above under Adaptability the model allows the modular use of relevant policies in the computee's knowledge as the computee changes society. This can affect transparently any new decisions of the computee and ensure conformance within the new societies. Existing decisions of goals and plans by the computee need to be re-evaluated with the aim to adjust these to the new societies rather than replace them entirely with new ones. Currently, the KGP model facilitates this to some extent via the Reactivity capability and transition.

In particular, each society will specify different relative roles between its member computees which need to be taken into account when the computee joins this society. The underlying reasoning with logic programming rules and priorities for Goal Decision and

the Planning capability provide a way to formalise roles and context within the *KGP* model.

Similarly, when a computee joins a new society it becomes aware of specific (current) expectations (WP2 in D5 has developed in detail a specific theory for these expectations for societies of computees) of this society. This would then affect the Goal Decision and Planning of the computee as the expectations introduce new potential goals for the computee and impose new constraints on its actions. The precise mechanisms for this need to be developed alongside with the further development of the society organisation in WP2.

The model thus addresses the criteria of *Conformance and Exploitation* and partially the criterion of *Adjustment*.

The model does not contain a specific mechanism for the computee to adopt a different cycle of operation with a new overall strategy of operation as it changes society. However, the resulting operation induced by a cycle theory can change implicitly in the new environment of a different society simply because the nature of cycle theories is such that these are sensitive to changing external information. This addresses in a limited way the criterion *Cycle Adjustment*.

DISTRIBUTION – The *KGP* model provides for the exchange of information between computees through sensing communication acts. Each computee would include in its Knowledge Base information pertaining to the expertise of the other computees in the environment. This information allows its Planning capability to generate plans that exploit the other computees in a collaborative way. The computee may also contain a preference policy to help it choose amongst several possibilities an appropriate computee with which to cooperate. In this way the achievement of a goal by a computee is distributed to a set of computees within its society. The model thus addresses the specific criterion of *Decentralisation* and partially that of *Emergence*.

The Planning capability needs to be refined further to distinguish which goals can be achieved favourably by the different members of the society (including the computee who is doing the planning reasoning) and to evaluate the suitability of the alternative computees. This refinement would again need to be developed alongside the further developments of the societies in WP2 and will address further the *Emergence* criterion.

The specific criterion of *Collectiveness* is not addressed directly by the current form of the model.

HETEROGENEITY – The *KGP* model allows for a high degree of heterogeneity in the different types of computees that can be defined within it. The highly modular structure of the model means that each computee could have its own subset of the set of capabilities and transitions, and the way these are used and combined together by each computee can give different characteristics to different computees. Cycle theories in the *KGP* model allow for this variety of synthesis of computees. The structure of cycle theories has been separated into parts where one component, the behaviour part, is designed for specifying explicitly high-level characteristics that we require from the operational behaviour of the computee. Modular changes in this part can result in a diverse spectrum of behaviour by the different computees. The model thus addresses fully the specific criterion of *Overall Behaviour*.

The internal knowledge base of each computee contains several independent modules with (preference) policies (e.g. goal decision policy or cooperation policy with other computees, etc). This gives another source of heterogeneity in the construction of a computee. A modular change in these parts of its knowledge base will result in different decisions of the computee and in turn in different operation. In particular, a computee can be given an individual personality by including in its knowledge base for goal decision specific type of personality policies with no extra mechanism required for this. Personality policies can be different amongst computees resulting in individual characteristics in the behaviour of the various computees. The model thus also covers well the specific criteria of *Expertise and Personality*.

It is important to stress that the high degree of heterogeneity of computees does not mean that the operation of a computee within a society is necessarily made more complex. The high level of autonomy that each computee has ensures that the effect of heterogeneity on the complexity of operation is not significant.

The second group of evaluation criteria permeate through the whole project. Hence they have been strong guiding principles in our development of the formal details of the KGP models.

MODULARITY – Modularity is one of the strongest general features of the KGP model. This stems from the high level specification framework that the use of computational logic provides and the well separated component structure of the model into knowledge bases, capabilities, transitions and cycle theory. The frameworks of Abductive Logic Programming and Logic Programming with Priorities enable us to represent the knowledge of the computee with high-level declarative statements that are independent of the way that they are used. This coupled with the declarative reasoning of abduction and preference (and the modular separation of the temporal constraint solving) on which all decisions and capabilities of the computee are based, means that the knowledge base of the computee can be changed independently of any other consideration.

Also as mentioned above in several places, the high-level specification of the societies and their properties together with the high degree of autonomy of the computees facilitates the modular placement of computees in and out of societies.

The clear separation of the various components of a computee also means that we can easily parameterise our computees. Components can be extended and altered independently from each other. In particular, we have shown how we can change in a highly modular way the cycle theory of a computee to obtain different behaviour characteristics and thus parameterise the overall behaviour of the computee.

COMPUTATIONAL VIABILITY – The components of the KGP model at different levels are all based on well understood computational logic frameworks using and extending these for the purposes of the model. We can then realize these components using the computational methods that these frameworks provide. Several extensions are required but these will not require fundamental new methods of computation. The main effort rather comes from the need to integrate together the computation of abduction, argumentation (for reasoning with rules and priorities) and constraint solving as required by the different components.

The modular structure of the KGP lends itself easily to an abstract design and architecture for the implementation of computees. Such an architecture can be drawn directly

from the model - a computee architecture - with three layers corresponding to capabilities, transitions and cycle theories (see figure 1 in section 2). At the bottom layers the capabilities exist as separate entities linked to the knowledge and external environment of the computee. The synthesis of these capabilities is designed at the middle layer of the transitions. At the top level of the architecture cycle theories control the synthesis of the transitions at the middle layer.

FORMALITY – The KGP model has a formal definition based on the underlying logic-based frameworks of abduction, argumentation (for preference reasoning) and constraint solving. The internal knowledge of a computee, its behaviour properties and the properties of the societies in which it operates are all given in terms of high-level declarative specifications in these logical frameworks. This facilitates the formal analysis and verification of properties of the computees. For example, we have already investigated [EMST03b] within these logical frameworks properties of conformance to society protocols showing the potential of this formal approach. Other formal properties of the behaviour of computees can also be captured declaratively within the KGP model, either by its decision making policies in its knowledge base or in its cycle theory policy and similar methods of their formal analysis could be applied.

RELATED WORK – Section 12 of this document gives a thorough comparison with the wide literature on logic-based and other multi-agent systems.

Summary of evaluation

Summarising the evaluation we see that the KGP model meets fully the following high-level Global Computing objectives, as derived by the SOCS project in its self-assessment document D3:

- computees should exhibit an autonomous and adaptable behaviour in the open and changing GC environment;
- the model should support different behaviours of computees, and the presence of behaviourally heterogeneous computees within societies;
- the model should allow for the interaction of computees within societies.

The KGP model defines computees as autonomous entities with diverse functionalities and operational behaviour. Its main novelty of approach stems from the attempt to synthesise together several advanced and powerful computational logic frameworks and techniques that until now have been examined mostly in isolation for particular problems of multi-agent systems. Its main strengths are the high degree of modularity that it gives to the structure of a computee and its formal declarative foundations. These together with the use of computational logic frameworks known to be easily realizable, will facilitate the construction of versatile computationally viable computees with verifiable properties of operation.

14 Conclusion

In this document we have proposed a model, called KGP, for autonomous computational entities that are to operate within societies (defined in the associated document of deliverable D5)

of such entities. This model shares its philosophy with earlier works on agents and multi-agent systems, notably that of BDI. It is though based on Computational Logic (CL), hence the name *computees* for these entities, and attempts to ensure that the model will be computationally realisable within the current state of the state-of-the-art technology for CL together with some specific extensions of this technology, and that properties of the model can be easily specified and verified formally.

Indeed, the main novelty of approach in our KGP model stems from this attempt to synthesise together several advanced and powerful CL frameworks and techniques that until now have been examined mostly in isolation for particular problems of multi-agent systems. These techniques include abduction and incremental planning, argumentation and decision making, temporal reasoning and updating of observational knowledge together with techniques of (time) constraint solving applied within these. Each one of these techniques has reached a level of maturity that it is now possible to integrate them together in an enhancing way to meet the complex requirements of the Global Computing setting that our computees need to satisfy. In addition, the use of CL as a basis for the KGP model gives this a sound theoretical underpinning which facilitates the formal analysis and verification of properties of computees defined within the model.

One of the main strengths of the KGP model is the high degree of modularity that it gives to the structure of a computee. Again the use of CL has been instrumental in providing this. The high-level declarative specification, afforded by the use of CL, of the knowledge and operation of the computee has allowed us to have a clear separation of concerns in the model in terms of underlying types of reasoning, capabilities, transitions and cycle theories. The environment and social context of the computee is also modularly separated from the computee. This modular structure and separation of concerns (into capabilities, internal transitions and cycle control theory) in the KGP model will facilitate the design and development of versatile and formally verifiable architectures for individual computees operating within their societies.

With the development of the KGP model as given by this document the project is now well poised for its next two phases. In particular, a clear architecture for computees emerges directly out of the KGP model which can form the basis of the computational model for computees and their principled construction in Phase 2 (WP3 and WP4 in year 2) of the project.

Acknowledgements

We would like to thank the project reviewers, Thomas Eiter, Yves Demazeau and David Pearce, for useful comments and suggestions on an earlier version of this document, and in particular for their indication of relevant related work. We would also like to thank the internal reviewers, Evelina Lamma, Paola Mello and Paolo Torroni, for their comments and suggestions that we believe have helped us to greatly improve this document. Further, we would like to thank Nicolas Maudet, Pavlos Moraitis and Andrea Bracciali, for help in collecting and understanding related work. Finally, we would like to thank the whole SOCS consortium for helpful discussions.

References

[ABW78] K.R. Apt, H. Blair, and A. Walker. *Towards a theory of declarative knowledge*, chapter Logic and Databases. Plenum Press, New York, 1978.

[AEK+99] K. Arisha, T. Eiter, S. Kraus, F. Ozcan, R. Ross, and V. S. Subrahmanian. IMPACT: Interactive maryland platform for agents collaborating together. *IEEE Intelligent Systems*, 14(2):64–72, 1999.

[AGM85] C. E. Alchourron, P. Gardenfors, and D. Makinson. On the logic of theory change: Partial meet functions for contraction and revision. *Journal of Symbolic Logic*, 50:510–530, 1985.

[AKGP01] A. Artikis, L. Kamara, F. Guerin, and J. Pitt. Animation of open agent societies. In *Proceedings of the Information Agents in E-Commerce symposium, AISB convention, York, UK*, pages 99–109, 2001.

[AKM00] A. Michael A.C. Kakas and C. Mourlas. Aclp: Abductive constraint logic programming. *Journal of Logic Programming (special issue on Abductive Logic Programming)*, 44(1-3):129–177, 2000.

[All84] J. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–144, 1984.

[ALP+00] J.J. Alferes, J.A. Leite, L.M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic programming*, 45(1-3):43–70, 2000.

[AP92] J. J. Alferes and L. M. Pereira. On logic program semantics with two kinds of negation. In Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 574–588, Washington, USA, 1992. The MIT Press.

[APPP99] J.J. Alferes, L. M. Pereira, H. Przymusinka, and T. Przymusinki. Lups: a language for updating logic programs. In *Proceedings of LPNMR-99*, volume 1730 of *LNAI*. Springer, 1999.

[Apt90] K. R. Apt. Logic programming. In *Handbook of Theoretical Computer Science*, volume B, pages 493–574. Elsevier Science Publisher, 1990.

[BaJHHvdT01] J. Broersen, M. Dastani abd J. Hulstijn, Z. Huang, and L. van der Torre. The boid architecture: conficts between beliefs, obligations, intentions and desires. In *Proceedings of Fifth International Conference on Autonomous Agents (Agents2001)*, pages 9–16. ACM Press, Montreal, Canada, 2001.

[BBJ+02] R.H. Bordini, A. L. C. Bazzan, R. O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. Agentspeak(xl): Efficient intention selection in bdi agents via decision-theoretic task scheduling. In *First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, 15-19 July, 2002.

[BDKJT97] F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *Int Journal of Cooperative Information Systems*, 6(1):67–94, 1997.

[BDKT97] A. Bondarenko, P. M. Dung, R.A. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence*, 93:63–101, 1997.

[BDKTV97] F. M. T. Brazier, B. Dunin-Keplicz, J. Treur, and R. Verbrugge. Modelling internal dynamic behaviour of BDI agents. In *ModelAge Workshop*, pages 36–56, 1997.

[BIP88] M.E. Bratman, D.J. Israel, and M.E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4, 1988.

[BK82] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. L. Clark and S.-A. Tarnlund, editors, *Logic programming, vol 16 of APIC studies in data processing*, pages 153–172. Academic Press, 1982.

[BKJT95] F. Brazier, B. D. Keplicz, N. R. Jennings, and J. Treur. Formal specification of multi-agent systems: a real-world case. In *First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 25–32, San Francisco, CA, USA, 1995. AAAI Press.

[Bra87] M.E. Bratman. *Intensions, plans and practical reason*. Harvard University Press, 1987.

[BTWW95] F. M. T. Brazier, J. Treur, N. J. E Wijngaards, and M. Willems. Temporal semantics of complex reasoning tasks. In *Proc. of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, KAW'95*, pages 15/1–15/17. Calgary:SRDG Publications, 1995.

[Cla78] K. L. Clark. Negation as failure. In *Logic and Databases*. Plenum Press, 1978.

[Dav01] P. Davidsson. Categories of artificial societies. In A. Omicini, P. Petta, and R. Tolksdorf, editors, *Engineering Societies in the Agents World II*, volume 2203 of *LNAI*, pages 1–9. Springer-Verlag, December 2001. 2nd International Workshop (ESAW'01), Prague, Czech Republic, 7 July 2001, Revised Papers.

[DdBD+02] M. Dastani, F. S. de Boer, F. Dignum, W. van der Hoek, M. Kroese, and J. Ch. Meyer. Programming the deliberation cycle of cognitive robots. In *Proc. of 3rd International Cognitive Robotics Workshop (CogRob2002)*, Edmonton, Alberta, Canada, 2002.

[Den87] D.C. Dennet. *The Intensional Stance*. The MIT Press, 1987.

[DK95] Y. Dimopoulos and A. C. Kakas. Logic programming without negation as failure. In *Proc. ILPS'95, pp. 369-384*, 1995.

[DK03] N. Demetriou and A. C. Kakas. Argumentation with abduction. In *Proceedings of the fourth Panhellenic Symposium on Logic*, 2003.

[dKLW98] M. dÍnverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dmars. In M. Singh, A. Rao, and M. Wooldridge, editors, *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages*, pages 155–176. Springer-Verlag LNAI 1365, 1998.

[DKS01] J. Dix, S. Kraus, and V.S. Subrahmanian. Temporal agent programs. *Artificial Intelligence*, 127(1):87–135, 2001.

[DKT95] B.M. Dunin-Keplicz and J. Treur. Compositional formal specification of multi-agent systems. In *Proc. of the ECAI'94 Workshop on Agent Theories, Architectures and Languages, Lecture Notes in AI*, volume 890, pages 102–117. Springer-Verlang, 1995.

[DST98] P. Dell'Acqua, F. Sadri, and F. Toni. Combining introspection and communication with rationality and reactivity in agents. In U. Furbach and L. Farinas del Cerro, editors, *Proc. JELIA'98, 6th European Workshop on Logics in Artificial Intelligence*, volume Springer Verlag LNAI 1489, pages 17–32, 1998.

[DST99] P. Dell'Acqua, F. Sadri, and F. Toni. Communicating agents. In *Proceedings ICLP Workshop on Multi-Agent Systems, Las Cruces, NM*, 1999.

[Dun91] P. M. Dung. Negation as hypothesis: An abductive foundation for logic programming. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 3–17. MIT Press, 1991.

[Dun95] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77:321–357, 1995.

[DvdHD02] M. Dastani, J. van der Ham, and F. Dignum. Communication for goal directed agents. In *Proceedings of the Agent Communication Languages and Conversation Policies Workshop (AAMAS'02)*, Bologna, Italy, 2002.

[EK89] K. Eshgi and R. A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Proceedings of the 6th International Conference on Logic Programming*, pages 234–255. MIT Press, 1989.

[EMST02] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Communication protocols for logic-based agents. In *UK Multi-Agent Systems (UKMAS) Annual Conference, Liverpool, Poster*, December 2002.

[EMST03a] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Aspects of Protocol Conformance in InterAgent Dialogue. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2003), Poster*, 2003.

[EMST03b] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Protocol conformance for logic-based agents. In *Proceedings IJCAI 2003 - To appear*, 2003.

[ES98] T. Eiter and V. S. Subrahmanian. Deontic action programs. In *Workshop on Foundations of Models and Languages for Data and Objects*, pages 37–54, 1998.

[ES99] T. Eiter and V. S. Subrahmanian. Heterogeneous active agents, II: Algorithms and complexity. *Artificial Intelligence*, 108(1–2):257–307, 1999.

[ESP99] T. Eiter, V. Subrahmanian, and G. Pick. Heterogeneous active agents i: Semantics. *Articial Intelligence*, 108(1–2):179–255, 1999.

[ESR00] T. Eiter, V. S. Subrahmanian, and T. J. Rodgers. Heterogeneous active agents, III: Polynomially implementable agents. *Artificial Intelligence*, 117(1):107–167, 2000.

[GI89a] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 972–978, 1989.

[GI89b] M. P. Georgeff and F. F. Ingrand. Monitoring and control of spacecraft systems using procedural reasoning. In *Workshop of the Space Operations-Automation and Robotics*, Houston, Texas, July 1989.

[GL86] M. P. Georgeff and A. L. Lansky. Procedural knowledge. In *Proceedings of the IEEE Special Issue on Knowledge Representation*, volume 74, pages 1383–1398, 1986.

[GL87] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, USA, July 1987. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th Int. Conf. on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GLL00] G. De Giacomo, Y. Lesperance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[GLS01] G. De Giacomo, H. J. Levesque, and S. Sardia. Incremental execution of guarded theories. *ACM Transactions on Computational Logic (TOCL)*, 2(4):495–525, October 2001.

[GRS88] A. Van Gelder, K. A. Ross, and J. S. Schlipf. Unfounded sets and the well-founded semantics for general logic programs. In *Proc. ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1988.

[HdBvdHM98] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. Ch. Meyer. Formal semantics for an abstract agent programming language. *Intelligent Agents IV (LNAI 1365)*, pages 215–229, 1998.

[HdBvdHM99a] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. Ch. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[HdBvdHM99b] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. Ch. Meyer. Semantics of communicating agents based on deduction and abduction. In *In IJCAI'99 Workshop on Agent Communication Languages*, 1999.

[HdL00] K. V. Hindriks, M. d'Inverno, and M. Luck. An formal architecture for 3APL. *ZB 2000*, pages 168–187, 2000.

[Hew91] C. Hewitt. Open information systems semantics for distributed artificial intelligence. *Artificial Intelligence*, 47(1-3):79–106, 1991.

[HM02] R. H.Bordini and . F. Moreira. Proving the asymmetry thesis principles for a bdi agent-oriented programming language. In *Proceedings of the Third International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-02)*, 1st August 2002.

[IGR92] F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34–44, 1992.

[Jia94] Y. Jiang. Ambivalent logic as the semantic basis fo metalogic programming. In P. Van Hentenrycki, editor, *Proceedings of the International Conference on Logic Programming*, pages 387–401. MIT Press, June 1994.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[K.87] Kunen K. Negation in logic programming. In *JOurnal of Logic Programming*, volume 4, pages 289–308, 1987.

[KD02] A. C. Kakas and M. Denecker. Abduction in logic programming. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond. Part I*, number 2407 in LNAI, pages 402–436. Springer Verlag, 2002.

[KKT93] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[KKT98] A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in AI and Logic Programming*, 5:235–324, 1998.

[KM90a] A. C. Kakas and P. Mancarella. Abductive logic programming. In V. Wiktor Marek, Anil Nerode, Dino Pedreschi, and V. S. Subrahmanian, editors, *Proceedings of the Workshop Logic Programming and Non-Monotonic Logic*, pages 49–61, Austin, TX, nov 1990.

[KM90b] A. C. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proceedings 9th European Conference on Artificial Intelligence*. Pitman Pub., 1990.

[KM91] A. C. Kakas and P. Mancarella. Preferred extensions are partial stable models. In *Proc. International Logic Programming Symposium ILPS91*, pages 85–102, 1991.

[KM95] A. Kakas and A. Michael. Integrating abductive and constraint logic programming. In *International Conference on Logic Programming, ICLP95*, 1995.

[KM97a] A.C. Kakas and R. Miller. Reasoning about actions, narratives and ramifications. *Electronic Transactions on Artificial Intelligence*, 1(4), 1997.

[KM97b] A.C. Kakas and R. Miller. A simple declarative language for describing narratives with ations. *Logic Programming*, 31, 1997.

[KM02] Antonis C. Kakas and Pavlos Moraitis. Argumentative agent deliberation, roles and context. In Jürgen Dix, João Alexandre Leite, and Ken Satoh, editors, *Computational Logic in Multi-Agent Systems: 3rd International Workshop, CLIMA'02, Copenhagen, Denmark, August 1, 2002, Proceedings*, number 93 in Datalogiske Skrifter (Writings on Computer Science), pages 35–48. Roskilde University, Denmark, 2002.

[KM03a] A. Kakas and L. Michael. On the qualification problem and elaboration tolerance. In *AAAI Spring Symposium on Logical Formalization of Commonsense Reasoning*, 2003.

[KM03b] A.C. Kakas and P. Moraitis. Argumentation based decision making for autonomous agents. In *Proc. of AAMAS'03, July 14–18, 2003, Melbourne, Australia.*, 2003.

[KMD94a] A. C. Kakas, P. Mancarella, and P. M. Dung. The acceptability semantics for logic programs. In *Proceedings of the 11th International Conference on Logic Programming*, 1994.

[KMD94b] A. C. Kakas, P. Mancarella, and P.M. Dung. The acceptability semantics for logic programs. In *ICLP94*, pages 504–519, 1994.

[KMM00] A. C. Kakas, A. Michael, and C. Mourlas. ACLP: Abductive Constraint Logic Programming. *Journal of Logic Programming*, 44(1-3):129–177, Jul 2000.

[Kow79] R. A. Kowalski. *Logic for Problem Solving*. North-Hollandg, 1979.

[Kow90] R. A. Kowalski. Problems and promises of computational logic. In *Proceedings Symposium on Computational Logic*, pages 1–36. Springer-Verlag, November 1990.

[KS86] R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[KS96a] R. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In *Proc. International Workshop on Logic with Databases*, pages 137–149. Springer Verlang, 1996.

[KS96b] R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In *Proc. International Workshop on Logic in Databases, San Miniato, Italy*, volume 1154 of *LNCS*. Springer-Verlag, 1996.

[KS99] R. A. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3/4):391–419, 1999.

[KS02] A.C. Kakas and F. Sadri. *Computational Logic: Logic Programming and Beyond*. Lecture Notes in Artificial Intelligence, Vol. 2407 and 2408, Springer Verlag, 2002.

[KST98] R. A. Kowalski, F. Sadri, and F. Toni. An agent architecture that combines backward and forward reasoning. In B. Gramlich and F. Pfenning, editors, *Proc. CADE-15 Workshop on Strategies in Automated Deduction*, pages 49–56, November 1998.

[KT99] A.C. Kakas and F. Toni. Computing argumentation in logic programming. *Journal of Logic and Computation*, 9:515–562, 1999.

[KTW98] R.A. Kowalski, F. Toni, and G. Wetzel. Executing suspended logic programs. *Fundamenta Informaticae*, 3(34):203–224, 1998. http://www-lp.doc.ic.ac.uk/UserPages/staff/ft/PAPERS/slp.ps.Z.

[KvD01] A.C. Kakas, B. van Nuffelen, and M. Denecker. A-System: Problem solving through abduction. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, pages 591–596, Seattle, Washington, USA, August 2001. Morgan Kaufmann.

[LAP01a] J. A. Leite, J. J. Alferes, and L. M. Pereira. On the use of multi-dimensional synamic logic programming to represent societal agents' viewpoints. In P. Brazdil and A. Jorge, editors, *Progress in Artificial Intelligence, 10th Portuguese International Conference on Artificial Intelligence (EPIA-01), vol. 2259 LNAI*. Springer, 2001.

[LAP01b] J. A. Leite, J.J. Alferes, and L.M. Pereira. Minerva — a dynamic logic programming agent architecture. In John-Jules Meyer and Milind Tambe, editors, *Pre-proceedings of the Eighth International Workshop on Agent Theories , Architectures, and Languages (ATAL-2001)*, pages 133–145, 2001.

[LAP01c] J.A. Leite, J.J. Alferes, and L. M. Pereira. Minerva - combining societal agents knowledge. Technical report, Dept. Informática, Universidade Nova de Lisboa, 2001.

[LAPQ00] J.A. Leite, J.J. Alferes, L. M. Pereira, and P. Quaresma. Planning as abductive updating. In *Proceedings of AISB'00 Symposium on AI Planning and Intelligent Agent s*, 2000.

[Lei02] J. A. Leite. *Evolving Knowledge Bases - Specification and Semantics*. PhD thesis, Universidade Nova de Lisboa, July 2002.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd extended edition, 1987.

[LMM+03] E. Lamma, P. Mello, P. Mancarella, A. Kakas, K. Stathis, and F. Toni. Self-assessment: parameters and criteria. Technical report, SOCS Consortium, 2003. Deliverable D3. Distribution restricted to the GC programme.

[LRL+97] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[MH69] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.

[PL90] P.R.Cohen and H.J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3), 1990.

[Prz91] T. Przymusinsky. Semantics of disjunctive logic programs and deductive databases. In *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases, LNCS 566 Springer Verlag*, pages 85–107, 1991.

[Rao96] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Rudy van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

[RG91] A. S. Rao and M. P. Georgeff. Asymmetry thesis and side-effect problems in linear-time and branching-time intention logics. In John Myopoulos and Ray Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 498–505, Sydney, Australia, 1991. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[RG92] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-92)*, pages 439–449, 1992.

[RG95] A. S. Rao and M. P. Georgeff. BDI agents: from theory to practice. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319, San Francisco, CA, USA, 1995. MIT Press.

[RG97] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 317–328. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[Sat92] T. Sato. Meta-programming through a truth predicate. In K. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 526–540. MIT Press, 1992.

[SBD+00] V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross. *Heterogenous Active Agents*. MIT-Press, 2000.

[Sha89] M.P. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings IJCAI 89*, pages 1055–1060, 1989.

[Sho93] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1), 1993.

[ST99] F. Sadri and F. Toni. Computational logic and multi-agent systems: a roadmap. *Compulog Newsletters*, 7, December 1999. Electronic version, URL: `http://www.cs.ucy.ac.cy/compulog/`.

[ST00] F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In E. Lamma and P. Mello, editors, *Proceedings AI*IA 99, 6th Congress of the Italian Association for Artifhcial Intelligence*, number 1792 in LNAI, pages 49–60. Springer Verlag, 2000.

[STT01] F. Sadri, F. Toni, and P. Torroni. Logic agents, dialogues and negotiation: an abductive approach. In *Proceedings AISB'01 Convention, York, UK*, March 2001.

[STT02a] F. Sadri, F. Toni, and P. Torroni. An abductive logic programming architecture for negotiating agents. In S. Greco and N. Leone, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA)*, volume 2424 of *LNCS*, pages 419–431. Springer Verlag, September 2002.

[STT02b] F. Sadri, F. Toni, and P. Torroni. Dialogues for negotiation: agent varieties and dialogue sequences. In *Intelligent Agents VIII, 8th International Workshop, ATAL 2001 Seattle, WA, USA, August 1-3, 2001 Revised Papers*, volume 2333 of *LNAI*, pages 405–421. Springer Verlag, 2002.

[SW00] M. Schroeder and G. Wagner. Vivid agents: Theory, architecture, and application. *International Journal for Applied Artificial Intelligence*, 14(7):645–76, August 2000.

[SZ90] D. Saccá and C. Zaniolo. Stable model semantics and non-determinism for logic programs with negation. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM Press*, pages 205–217, 1990.

[Tho95] S. R. Thomas. The PLACA agent programming language. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents*, Berlin, 1995. Springer Verlang.

[TK95] F. Toni and R.A. Kowalski. Reduction of abductive logic programs to normal logic programs. In *In proceedings of ICLP95*, 1995.

[TS02] F. Toni and K. Stathis. Access-as-you-need: a computational logic framework for flexible resource access in artificial societies. In *Proceedings of the Third International Workshop on Engineering Societies in the Agents World (ESAW'02)*, LNAI. Springer Verlag, 2002.

[WJ95] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engeneering Review*, 1995.

[WRR95] D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a concurrent agent-oriented language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 386–402. Springer-Verlag: Heidelberg, Germany, 1995.