

The KGP Model of Agency for Global Computing: Computational Model and Prototype Implementation

A. Bracciali¹, N. Demetriou², U. Endriss³, A. Kakas², W. Lu⁴, P. Mancarella¹,
F. Sadri³, K. Stathis^{4,1}, G. Terreni¹, and F. Toni^{3,1}

¹ Dip. di Informatica, Università di Pisa

{braccia, paolo, terreni}@di.unipi.it

² Dept of Computer Science, Cyprus University

{demetriou, antonis}@cs.ucy.ac.cy

³ Dept of Computing, Imperial College London

{ue, fs, ft}@doc.ic.ac.uk

⁴ School of Informatics, City University London

{lue, kostas}@soi.city.ac.uk

Abstract. We present the computational counterpart of the KGP (**K**nowledge, **G**oals, **P**lan) declarative model of agency for Global Computing. In this context, a computational entity is seen as an agent developed using Computational Logic tools and techniques. We model a KGP agent by relying upon a collection of capabilities, which are then used to define a collection of transitions, to be used within logically specified, context sensitive control theories, which we call cycle theories. In close relationship to the declarative model, the computational model mirrors the logical architecture by specifying appropriate computational counterparts for the capabilities and using these to give the computational models of the transitions. These computational models and the one specified for the cycle theories are all based on, and are significant extensions of, existing proof procedures for abductive logic programming and logic programming with priorities. We also discuss a prototype implementation of the overall computational model for KGP.

1 Introduction

Global Computing (GC) and its applications rely upon computing environments that are composed of *autonomous* computational entities whose activity is not centrally controlled but is *decentralised* instead. Decentralisation results either because global control is impossible or at times impractical, or because the entities are created or controlled by different owners. The computational entities may be mobile, due to the movement of the physical platforms or by movement of the entities from one platform to another. In other words, the environment in which the entities are situated is *open* and evolves over time. For instance, in a typical GC application it might be required to allow for the introduction

and deletion of computational entities. The internal structure and behaviour of these entities may also be *heterogeneous* and may vary over time.

Programming the behaviour of a computational entity that is situated in a GC environment is a non-trivial task. One of the problems is that such an entity should be in a position to operate with *incomplete information* about the environment. Incompleteness might arise from the entities having newly joined the environment of an application and having only a partial view over the status of that application. Incompleteness might also arise from the autonomy of the entities and their unwillingness to disclose information about themselves. Moreover, incompleteness might sometimes be caused by the fact that information in a GC environment becomes rapidly out of date. Thus, a GC entity needs to be able to discover relevant information or other entities in the dynamically evolving environment.

If the ultimate goal of GC research is to provide a solid scientific foundation for the design of GC systems, we will need to lay the groundwork for achieving effective principles for building and analysing such systems. In trying to achieve this goal, within the GC project SOCS we interpret the GC vision as follows. Entities in GC systems are defined via *Computational Logic* (CL), as understood in [26, 29, 27], which is used to define their internal organisation, reasoning and their mutual interactions. We call the entities *computees*, standing for *agents in CL*¹. One important feature of computees is that they are able to *reason* by using CL tools and techniques. We call the systems composed of such entities *societies* (see [7]) as they are characterised by “social rules” for computees to interact and operate in the presence of each other.

In order to interact freely, computees can use *high-level communication*, as understood in multi-agent systems. Computees may be *heterogeneous* as far as *behaviour* is concerned, provided by CL-based cycle theories allowing a highly modular and flexible specification of control. Cycle theories allow to render computees adaptable to dynamically changing environments and allow to characterise, via different cycle theories, heterogeneously behaved computees.

Computees also need to adapt their internal state as the environments in which they are situated evolve. A number of CL techniques have been developed for addressing tasks such as temporal reasoning in a changing environment, hypothetical reasoning for dealing with incomplete information, hypothetical reasoning for planning, hypothetical reasoning to achieve communication, argumentation for decision-making, 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. Our model for computees integrates (extensions of) a number of existing CL techniques, in order to achieve the enhanced performance which is required by the GC vision.

We call our model *KGP*, since computees’ internal state consists of a knowledge base (*K*), from which they reason, goals (*G*) that they need to achieve,

¹ In this paper, we will use the terms computees and agents interchangeably.

and plans (P) for their goals, consisting of actions that may be physical, sensing or communicative. Computees pursue their goals while being alert to the environment and adapt their goals and plan to any changes that they perceive.

The paper is organised as follows. In section 2 we summarise the main features of the KGP model and give some of the technical details underlying it. In sections 3–5 we provide the computational models of some components of the KGP model and state their soundness wrt their formal specification. The overall computational model is built bottom-up, mirroring the hierarchical and modular structure of the abstract model. Section 3 also gives some background on the CL techniques that we have employed to define the KGP model, namely Abductive Logic Programming (ALP) and Logic Programming with Priorities (LPP), as well as the proof procedures (for ALP and for LPP) from which we have built the computational counterpart of the KGP model, in a bottom-up fashion. In section 6 we describe the prototype implementation of KGP agents, namely the SOCS-iC (for SOCS individual Computee) component of the PROSOCS platform [36]. Section 7 concludes.

2 KGP Model: Recap

Here we briefly summarise the KGP model for computees, see [18, 17] for any additional details. This model relies upon

- an internal (or mental) state,
- a set of reasoning capabilities, supporting planning, temporal reasoning, identification of preconditions of actions, reactivity and goal decision,
- a sensing capability,
- a set of transition rules, defining how the state of the computee changes, and defined in terms of the above capabilities,
- a set of selection functions, to provide appropriate inputs to the transitions,
- a cycle theory, for deciding which transitions should be applied when, and defined using the selection functions.

The model is defined in a modular and hierarchical fashion.

Internal State. This is a tuple $\langle KB, Goals, Plan, TCS \rangle$, where:

- KB is the knowledge base of the computee, and describes what the computee knows (or believes) of itself and of the environment. KB consists of modules supporting different reasoning capabilities:
 - KB_{plan} , for Planning,
 - KB_{pre} , for the Identification of Preconditions of actions,
 - KB_{TR} , for Temporal Reasoning,
 - KB_{GD} , for Goal Decision,
 - KB_{react} , for Reactivity, and
 - KB_0 , for holding the (dynamic) knowledge of the computee about the external world in which it is situated (including past communications).

Syntactically, KB_{plan} , KB_{react} and KB_{TR} are abductive logic programs with constraint predicates (see section 3.1), KB_{pre} is a logic program (see section 3.1), KB_{GD} is a logic program with priorities (see section 3.2), and KB_0 is a set of logic programming facts, and it is (implicitly) included in all the other modules.

- *Goals* is the set of properties that the computee wants to achieve, each one explicitly time-stamped by a time variable. Goals may also be equipped with a temporal constraint (belonging to *TCS*) bounding the time variable and defining when the goals are expected to hold. Goals may be *mental* or *sensing*. Both can be observed to hold (or not to hold) via the Sensing capability. In addition, mental goals can be brought about actively by the computee by its Planning capability and its actions.
- *Plan* is a set of actions scheduled in order to satisfy goals. Each is explicitly time-stamped by a time variable and possibly equipped with a temporal constraint, similarly to *Goals*, but defining when the action should be executed. Actions are partially ordered, via their temporal constraints. Each action is also equipped with the preconditions for its successful execution, determined by the Identification of Preconditions capability. Actions may be *physical*, *communicative*, or *sensing*. We assume that actions are atomic and do not have a duration. Actions can be seen as special kinds of goals which are directly executable.
- *TCS* is a set of constraint atoms (referred to as *temporal constraints*) in some given underlying constraint language with respect to some structure \mathfrak{R} equipped with a notion of constraint satisfaction $\models_{\mathfrak{R}}$ (see section 3.1). We assume that the constraint predicates include $<, \leq, >, \geq, =, \neq$. Temporal constraints refer to time constants, namely numbers, and time variables, namely distinguished variables which can be instantiated to time constants. These constraints bind the time of goals in *Goals* and actions in *Plan*. For example, they may specify a time window over which the time of an action can be instantiated, at execution time.

Goals and actions are uniquely identified by their associated time variable, which is implicitly existentially quantified within the overall state.

To aid revision and partial planning, *Goals* and *Plan* form a *tree*². The tree is augmented by calls to the Goal Decision, Planning and Reactivity capabilities. The tree is given implicitly by associating with each goal and action its parent. *Top-level* goals and actions are children of the root of the tree, which, by convention, is the special symbol \perp . Actions always occur as leaves.

² In the full model [5], we actually have two trees, the first containing *non-reactive* goals and actions, the second containing *reactive* goals and actions. All the top-level non-reactive goals are either assigned to the computee by its designer at birth, or they are determined by the Goal Decision capability. All the top-level reactive goals and actions are determined by the Reactivity capability. Here for simplicity we overlook the distinction amongst the two trees.

Reasoning Capabilities. These are:

- Planning, which generates partial plans for sets of goals. It provides (temporally constrained) sub-goals and actions designed for achieving the input goals.
- Reactivity, which reacts to perceived changes in the environment, by replacing (some) goals in *Goals* and actions in *Plan* with (possibly temporally constrained) goals and actions.
- Goal Decision, which revises the top-most level goals of the computee, adapting the computee’s state to changes in its own preferences and in the environment. Differently from Reactivity, it only modifies the top-level goals, it does not add actions to *Plan* and it does not depend upon the current *Goals* and *Plan*.
- Identification of Preconditions for action execution.
- Temporal Reasoning, which reasons about the evolving environment, and makes predictions about properties (fluents) holding in the environment, based on the partial information the computee acquires.

Sensing Capability. In addition to the reasoning capabilities above, the computee is equipped with a Sensing capability which links it to its environment, by allowing to observe that properties hold or do not hold, and that other agents have executed actions in the past.

Transitions. The state of a computee evolves by applying transition rules, which employ capabilities and the constraint satisfaction $\models_{\mathcal{R}}$. The transitions are:

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

Cycle. The behaviour of a computee is given by the application of transitions in sequences, repeatedly changing the state of the computee. 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 (see section 3.2), defining preference policies over the order of application of transitions, which may depend on the environment and the internal state of a computee. This provision of a declarative control for computees in the form of cycle theories is a highly novel feature of the model, which could, in principle, be imported into other agent systems.

In the remainder of this section, we give some details on the state of computees. Details on the other components of the model (capabilities, transitions, selection functions and cycle) will be illustrated when describing their computational models, in sections 3–5. Here, note that alternative choices for reasoning capabilities and transitions would have been possible, and the model could be extended to incorporate further such capabilities and further transitions.

Vocabularies. We assume (possibly infinite) vocabularies of *time constants* (e.g., the set of all natural numbers), *time variables* (indicated with t, t', s, \dots), *fluents* (indicated with f, f', \dots), *action operators* (indicated with a, a', \dots), and *names of computees* (indicated with c, c', \dots). Given a fluent f , f and $\neg f$ are referred to as *fluent literals*. We use l, l', \dots to denote fluent literals. Moreover, given a fluent literal l , by \bar{l} we denote its complement, namely $\neg f$ if l is f , f if l is $\neg f$.

We assume that the set of fluents is partitioned in two disjoint sets: *mental fluents* and *sensing fluents*. Intuitively, mental fluents represent properties that the computee itself is able to plan for so that they can be satisfied, but can also be observed. On the other hand, sensing fluents represent properties which are not under the control of the 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 a request for some (given) resource is accepted and that some (given) connection is active.

We also assume that the set of action operators is partitioned into three disjoint sets: *sensing*, *physical*, and *communication 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(c₁, c₂, request(r1), d, t)* may be a communication action expressing that computee c_1 is requesting from computee c_2 the resource r_1 within a dialogue with identifier d , at time t .

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

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

Top-level goals are goals of the form $G = \langle l[t], \perp \rangle$. As an example, we may have a top-level goal G of the form $\langle \text{problem_fixed}(p2, t), \perp \rangle$ and a subgoal G' of G of the form $\langle \text{get_resource}(r1, t'), G \rangle$, with $TCS = \{5 \leq t \leq 10, 5 \leq t' < t\}$, meaning that to fix problem $p2$ within a certain time interval, the computee needs to have (or acquire) a resource $r1$ within an appropriate other time interval.

Mental (sensing) goals are goals whose fluent is mental (sensing, respectively).

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

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

As an example, we may have an action $\langle \text{tell}(c_1, c_2, \text{request}(r1), d, t''), G', \{\} \rangle$ within the state of some computee c_1 , where G' is given above, $5 \leq t'' < t'$ also belongs to TCS , and c_2 is the name of some other computee.

(Non-)Sensing actions are actions whose operator is a (non-)sensing one.

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

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

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

- $\text{executed}(a[t], \tau)$ where $a[t]$ is a timed operator and τ is a time constant, meaning that action a has been executed at time $t = \tau$ by the computee.
- $\text{observed}(l[t], \tau)$ where $l[t]$ is a timed fluent literal and τ is a time constant, meaning that the property l has been observed to hold at time $t = \tau$.

- $observed(c, a[\tau'], \tau)$ where c is 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 means that the given computee has observed at time τ that computee c has executed the action a at time τ' ($\tau' \leq \tau$).

Note that assertions in KB_0 of the third kind are variable-free. These are intended, e.g., to represent reception of communication from other computees. Instead, assertions of the first two kinds refer explicitly to a time variable t . This representation with explicit variables allows us to instantiate implicitly the time variable of (executed) actions and (observed) goals, via $\Sigma(S)$ (see below), while keeping the time variable explicitly as an identifier for actions and goals. As a consequence, the time variables in KB_0 are not properly speaking variables as such.

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

Valuation of Time Variables and Temporal Constraints.

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

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

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). Thus, 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 any temporal constraint Tc in a state S will always take Σ into account. Namely, any ground valuation for the temporal variables in Tc must agree with Σ on the temporal variables assigned to in Σ .

3 Computational Models for Capabilities

The reasoning capabilities of Planning, Reactivity, Identification of Preconditions and Temporal Reasoning are specified within the framework of Abductive Logic Programming (ALP), and the reasoning capability of Goal Decision is specified within the framework of Logic Programming with Priorities (LPP). Their computational models rely upon proof procedures for ALP and LPP (as appropriate). In this section, we briefly recall ALP and LPP, and summarise the concrete proof procedures used for the computational model of the reasoning capabilities. Finally, we give detailed specification and computational model for Planning (chosen as the representative ALP-based capability) and for Goal

Decision. The remaining capabilities, of Identification of Preconditions and Reactivity, are briefly mentioned. We also state the soundness results for these capabilities, building upon the soundness results for the underlying procedures. For details and proofs see [5].

3.1 ALP-Based Capabilities

Background: Abductive Logic Programming with Constraints. An *abductive logic program* with constraints is a tuple $\langle \mathfrak{R}, P, A, I \rangle$ where:

- \mathfrak{R} is a structure consisting of a domain $D(\mathfrak{R})$ and a set of constraint predicates including equality, together with an assignment of relations on $D(\mathfrak{R})$ for each constraint predicate. The structure is equipped with a notion of \mathfrak{R} -satisfiability. Given (a set of) constraints C , $\models_{\mathfrak{R}} C$ stands for C is \mathfrak{R} -satisfiable, and $\sigma \models_{\mathfrak{R}} C$, for some grounding σ of the variables of C over $D(\mathfrak{R})$, stands for C is \mathfrak{R} -satisfied by σ .
- P is a *normal logic program with constraints*, namely a set of rules of the form $H \leftarrow L_1 \wedge \dots \wedge L_n$ with H atom, L_i literals, and $n \geq 0$. Literals can be positive, namely atoms, or negative, namely of the form *not* B , where B is an atom, or constraint atoms over \mathfrak{R} . The negation symbol *not* indicates *negation as failure* [8]. All variables in H, L_i are implicitly universally quantified, with scope the entire rule. If $n = 0$, the rule is called a *fact*.
- A is a set of *abducible predicates* in the language of P . Atoms whose predicate is abducible are referred to as *abducible atoms* or simply as *abducibles*.
- I is a set of *integrity constraints*, that is, a set of sentences in the language of P . All the integrity constraints in the KGP model have the implicative form $L_1 \wedge \dots \wedge L_n \Rightarrow A_1 \vee \dots \vee A_m$ ($n \geq 0, m > 0$) where L_i are literals (as in the case of rules)³, A_j are atoms (possibly the special atom *false*). All variables in the integrity constraints are implicitly universally quantified from the outside.

Given an abductive logic program $\langle \mathfrak{R}, P, A, I \rangle$ and a formula (*query*) Q , which is an (implicitly existentially quantified) conjunction of literals in the language of the abductive logic program, the purpose of abduction is to find a (possibly minimal) set of (ground) abducible atoms Γ which, together with P , “entails” (an appropriate ground instantiation of) Q , with respect to some notion of “entailment” that the language of P is equipped with, and such that the extension of P “satisfies” I (see [19] for possible notions of integrity constraint “satisfaction”). Here, the notion of “entailment” depends on the semantics associated with the logic program P (there are many different possible choices for this [19]), appropriately combined with the notion of \mathfrak{R} -satisfiability, as in Constraint Logic Programming [16]. We will refer to such a combined semantics as $\models_{LP(\mathfrak{R})}$.

Formally and concretely, given a query Q , a set Δ of (possibly non-ground) abducible atoms, and a set C of (possibly non-ground) constraints, the pair (Δ, C) is an *abductive answer (with constraints)* for Q , with respect to an abductive logic program with constraints $\langle \mathfrak{R}, P, A, I \rangle$, iff for all groundings σ for

³ If $n = 0$, then L_1, \dots, L_n represents the special atom *true*.

the variables in Q, Δ, C such that $\sigma \models_{\mathfrak{R}} C$, it holds that (i) $P \cup \Delta\sigma \models_{LP(\mathfrak{R})} Q\sigma$, and (ii) $P \cup \Delta\sigma \models_{LP(\mathfrak{R})} I$. Here, $\Delta\sigma$ plays the role of Γ in the earlier informal description of abductive answer.

Such notion can be extended to take into account an initial set of (possibly non-ground) abducible atoms Δ_0 and an initial set of (possibly non-ground) constraint atoms C_0 , so that an abductive answer for Q , with respect to $\langle \mathfrak{R}, P, A, I \rangle$, Δ_0, C_0 , is a pair (Δ, C) such that $\Delta \cap \Delta_0 = \{\}$, $C \cap C_0 = \{\}$, and $(\Delta \cup \Delta_0, C \cup C_0)$ is an abductive answer for Q , with respect to $\langle \mathfrak{R}, P, A, I \rangle$ (in the earlier sense).

In the sequel, for simplicity, we will omit \mathfrak{R} from abductive logic programs.

In ALP (with constraints), abductive answers are computed via *abductive proof procedures*, which typically extend SLD-resolution, providing the computational backbone underneath most logic programming systems, in order to check and enforce integrity constraint satisfaction, the generation of abducible atoms, and the satisfiability of constraint atoms (if any). There are a number of such procedures in the literature, e.g. the A-system [24]. To provide a computational counterpart to (the abductive tasks in) the KGP model, we propose and adopt the CIFF proof procedure [11, 12], extending the IFF proof procedure [13] for the purposes of the SOCS project. This procedure is summarised next. Full details are given in [11].

CIFF: A Proof Procedure for ALP with Constraints. CIFF extends IFF by dealing with constraints and non-allowed abductive logic programs, by tackling the issue of allowedness *dynamically*, i.e. at runtime, rather than adopting a static and overly strict set of allowedness conditions as in IFF [13]. To this end, the CIFF procedure includes a *dynamic allowedness rule* which is triggered whenever the procedure encounters a particular formula it cannot manipulate correctly due to a problematic quantification pattern.

In defining CIFF, we assume the availability of a sound and complete constraint solver, that we use as a *black box* component of the procedure. We do not make any assumption on the language of constraints, except for assuming that it includes a relation symbol for equality and it is closed under complements.

Input. Given an abductive logic program $\langle P, A, I \rangle$, the *input* to the CIFF procedure consists of a *query* Q , the set of *integrity constraints* I , and, in the background, a *theory* Th , which is a set of iff-definitions obtained by completing [8] the non-abducible, non-constraint predicates in the language of $\langle P, A, I \rangle$. Thus, the set of *abducibles* A is implicitly the set of predicates for which there is no definition in Th . The iff-definitions in Th have the following form:

$$p(X_1, \dots, X_k) \leftrightarrow D_1 \vee \dots \vee D_n$$

Negative literals are treated as implications (e.g. *not* $q(X, Y)$ is treated as $q(X, Y) \Rightarrow \text{false}$).

Output. There are three possible *outputs* of the CIFF procedure: (1) the procedure succeeds and produces an abductive answer to the query Q ; (2) the procedure fails, thereby indicating that there is no abductive answer to the query

Q ; and (3) the procedure reports that computing an abductive answer for the query Q is not possible, because a critical part of the input is not allowed.

Proof Rules. The CIFF procedure generates outputs from inputs by repeatedly applying a number of proof rules to the input. During this process, CIFF manipulates, essentially, formulas that are either atoms or implications or disjunctions of atoms and implications. Such formulas are referred to individually as *goals*. *Implications* are obtained by manipulating integrity constraints and (the rewriting of) negative literals. The theory Th is kept in the background and is only used to *unfold* defined predicates as they are being selected. Disjunction may thus be generated, and *splitting* will be applied to give rise to different branches in the proof search tree. The root of this tree is the original input $(Q \wedge I)$. *Nodes* of this tree are sets (conjunctions) of goals. CIFF repeatedly manipulates a selected node, via the proof rules, by rewriting goals in the node, adding new goals to the node, deleting superfluous goals from it, or adding *false* to the node (and thus effectively deleting the node completely). For a full description of the proof rules, see [11]. Here, we only mention:

- *Case analysis for constraints:* Replace any goal of the form $Con \wedge A \Rightarrow B$, where Con is a constraint not containing any universally quantified variables, by $[Con \wedge (A \Rightarrow B)] \vee \overline{Con}$. There is a similar case analysis rule for equalities.
- *Constraint solving:* Replace any node containing an unsatisfiable set of constraints (as atoms) by *false*.
- *Dynamic allowedness rule:* Label nodes with problematic quantification patterns as *undefined*.

A node containing *false* is called a *failure node*. If all leaf nodes in a search proof tree are failure nodes, then the derivation leading to that tree is said to be *failed* (the intuition being that there exists no answer to the query in question). A node to which no more proof rules can be applied is called a *final node*. A final node that is not a failure node and which has not been labelled as *undefined* is called a *success node*.

Answer Extraction. An *extracted answer* from a final success node N is a pair $\langle \Delta, C \rangle$, where Δ is the set of abducible atoms in N and C is obtained from the set of constraint atoms, equalities and disequalities in N . Below, we will use the following notations:

- $\langle P, A, I \rangle, Q \vdash_{CIFF} (\Delta, C)$ to stand for (Δ, C) is the answer extracted from a final success node obtained from the initial goal $Q \wedge I$;
- $\langle P, A, I \rangle, Q, \Delta_0, C_0 \vdash_{CIFF} (\Delta, C)$ to stand for $\langle P, A, I \rangle, Q \wedge \Delta_0 \wedge C_0 \vdash_{CIFF} (\Delta \cup \Delta_0, C \cup C_0)$ and $\Delta \cap \Delta_0 = \{\}$ and $C \cap C_0 = \{\}$;
- $\langle P, A, I \rangle, Q \vdash_{CIFF} fail$ to stand for: there is a failed derivation for Q ;
- $\langle P, A, I \rangle, Q \vdash_{CIFF} flounder$ to stand for: there is a search proof tree for Q with no success leaf nodes and at least one undefined leaf node.

Soundness Results. We have shown that CIFF is *sound* [11, 12], and in particular the following result:

Theorem 1. (CIFF Soundness of Success) *Given a query Q and initial Δ_0, C_0 :
if $\langle P, A, I \rangle, Q, \Delta_0, C_0 \vdash_{CIFF} (\Delta, C)$
then (Δ, C) is an abductive answer for Q , with respect to $\langle P, A, I \rangle, \Delta_0, C_0$.*

Specification and Computational Model for the ALP-Based Capabilities. KB_{plan} , KB_{react} , KB_{TR} , and KB_{pre} are all specified within the framework of the event calculus (EC) for reasoning about actions, events and changes [28]. Below, we give the abductive logic program KB_{plan} and the logic program KB_{pre} . KB_{react} is an extension of KB_{plan} , incorporating additional integrity constraints representing reactive rules. KB_{TR} is another variant of the EC, sharing a common kernel with KB_{plan} . Both KB_{react} and KB_{TR} are fully described in [5]. KB_{TR} is also given in [6].

Abductive Event Calculus for KB_{plan} and KB_{pre} . In a nutshell, the EC allows to write meta-logic programs which "talk" about object-level concepts of *fluents*, *events* (that we interpret as action *operations*), and *time points*. The main meta-predicates of the formalism are: $holds_at(F, T)$ (a fluent F holds at a time T), $clipped(T_1, F, T_2)$ (a fluent F is clipped - from holding to not holding - between times T_1 and T_2), $declipped(T_1, F, T_2)$ (a fluent F is declipped - from not holding to holding - between times T_1 and T_2), $initially(F)$ (a fluent F holds from the initial time, say time 0), $happens(O, T)$ (an operation O happens at a time T), $initiates(O, T, F)$ (a fluent F starts to hold after an operation O at time T) and $terminates(O, T, F)$ (a fluent F ceases to hold after an operation O at time T). Roughly speaking, in a planning setting the last two predicates represent the cause-effects links between operations and fluents in the modelled world. We will also use a meta-predicate $precondition(O, F)$ (the fluent F is one of the preconditions for the executability of the operation O).

The EC allows to represent a wide variety of phenomena, including operations with indirect effects, non-deterministic operations, and concurrent operations [33]. A number of abductive variants of the EC have been proposed to deal with planning problems, e.g. see [32]. Here, we propose a novel variant, somewhat inspired by the \mathcal{E} -language [21], to allow situated agents to generate partial plans in a dynamic environment.

We give $KB_{plan} = \langle P_{plan}, A_{plan}, I_{plan} \rangle$. P_{plan} consists of two parts: domain-independent rules and domain-dependent rules. The basic *domain-independent rules*, directly borrowed from the original EC, are:

$$\begin{aligned}
holds_at(F, T_2) &\leftarrow happens(O, T_1), initiates(O, T_1, F), \\
&T_1 < T_2, not\ clipped(T_1, F, T_2) \\
holds_at(\neg F, T_2) &\leftarrow happens(O, T_1), terminates(O, T_1, F), \\
&T_1 < T_2, not\ declipped(T_1, F, T_2) \\
holds_at(F, T) &\leftarrow initially(F), 0 \leq T, not\ clipped(0, F, T) \\
holds_at(\neg F, T) &\leftarrow initially(\neg F), 0 \leq T, not\ declipped(0, F, T) \\
clipped(T_1, F, T_2) &\leftarrow happens(O, T), terminates(O, T, F), T_1 \leq T < T_2 \\
declipped(T_1, F, T_2) &\leftarrow happens(O, T), initiates(O, T, F), T_1 \leq T < T_2
\end{aligned}$$

The *domain-dependent* rules define *initiates*, *terminates*, and *initially*, e.g.

$$\begin{aligned} \textit{initiates}(\textit{go}(X, L_1, L_2), T, \textit{at}(X, L_2)) &\leftarrow \textit{holds_at}(\textit{mobile}(X), T) \\ \textit{initiates}(\textit{go}(X, L_1, L_2), T, \textit{free}(L_1)) &\leftarrow \textit{holds_at}(\textit{mobile}(X), T) \\ \textit{terminates}(\textit{go}(X, L_1, L_2), T, \textit{at}(X, L_1)) &\leftarrow \textit{holds_at}(\textit{mobile}(X), T), L_1 \neq L_2 \\ \textit{terminates}(\textit{go}(X, L_1, L_2), T, \textit{free}(L_2)) &\leftarrow \textit{holds_at}(\textit{mobile}(X), T), L_1 \neq L_2 \\ \textit{initially}(\textit{at}(\textit{bob}, (1, 1))) & \end{aligned}$$

Namely, the operation *go* from one location L_1 to some other location L_2 initiates the agent (robot) X being at location L_2 and location L_1 being free and terminates X being at location L_1 and location L_2 being free, provided that X is mobile. Moreover, some agent *bob* is initially at location $(1, 1)$. The conditions for the rules defining *initiates* and *terminates* can be seen as preconditions for the effects of the operator *go* to take place. Preconditions for the executability of operators are specified within KB_{pre} , which contains a set of rules defining the predicate *precondition*, e.g.

$$\begin{aligned} \textit{precondition}(\textit{go}(X, L_1, L_2), \textit{at}(X, L_1)) \\ \textit{precondition}(\textit{go}(X, L_1, L_2), \textit{free}(L_2)) \end{aligned}$$

namely the preconditions of the operator $\textit{go}(X, L_1, L_2)$ are that X is at the initial location L_1 and that location L_2 X is moving to is free.

In order to accommodate (partial) planning we will assume that the domain-independent part in P_{plan} also contains the rules:

$$\begin{aligned} \textit{happens}(O, T) &\leftarrow \textit{assume_happens}(O, T) \\ \textit{holds_at}(F, T) &\leftarrow \textit{assume_holds}(F, T_1), T_1 \leq T_2, \textit{not_clipped}(T_1, F, T_2) \\ \textit{holds_at}(\neg F, T) &\leftarrow \textit{assume_holds}(\neg F, T_1), T_1 \leq T_2, \textit{not_declipped}(T_1, F, T_2) \end{aligned}$$

i.e. an operator can be made to happen and a fluent can be made to hold simply by assuming them, where *assume_happens* and *assume_holds* are the only predicates in A_{plan} in KB_{plan} . This supports partial planning as follows. Actions $\langle a[t], -, _ \rangle$ in the state amount to atoms *assume_happens*(a, t), thus, abducing an atom in the predicate *assume_happens* amounts to planning to execute the corresponding action. Moreover, goals $\langle l[t], _ \rangle$ in the state correspond to atoms *holds_at*(l, t) and *assume_holds*(l, t) (depending on whether they have already been planned for or not): thus, abducing atoms in the predicate *assume_holds* amounts to planning to further plan for the corresponding sub-goal.

I_{plan} contains the following domain-independent integrity constraints:

$$\begin{aligned} \textit{holds_at}(F, T), \textit{holds_at}(\neg F, T) &\Rightarrow \textit{false} \\ \textit{assume_happens}(O, T), \textit{precondition}(O, P) &\Rightarrow \textit{holds_at}(P, T) \\ \textit{assume_happens}(O, T), \textit{not_executed}(O, T), \textit{time_now}(T') &\Rightarrow T > T' \end{aligned}$$

namely a fluent and its negation cannot hold at the same time, when assuming (planning) that some action will happen, we need to enforce that each of its preconditions hold and that this action will be executable in the future.

To allow agents to draw conclusions from the contents of KB_0 , which represent the “narrative” part of the computee’s knowledge, the following *bridge rules* are also amongst the domain independent rules of P_{plan} :

$$\begin{aligned}
 \text{clipped}(T_1, F, T_2) &\leftarrow \text{observed}(\neg F, T), T_1 \leq T < T_2 \\
 \text{declipped}(T_1, F, T_2) &\leftarrow \text{observed}(F, T), T_1 \leq T < T_2 \\
 \text{holds_at}(F, T_2) &\leftarrow \text{observed}(F, T_1), T_1 \leq T_2, \text{not clipped}(T_1, F, T_2) \\
 \text{holds_at}(\neg F, T_2) &\leftarrow \text{observed}(\neg F, T_1), T_1 \leq T_2, \text{not declipped}(T_1, F, T_2) \\
 \text{happens}(O, T) &\leftarrow \text{executed}(O, T) \\
 \text{happens}(O, T) &\leftarrow \text{observed}(C, O, T)
 \end{aligned}$$

Note that we assume that the value of a fluent literal is changed according to observations only from the moment the observations are made, and actions by other agents have effects only from the time observations are made that they have been executed, rather than by the execution time itself. These choices are dictated by the rationale that observations can only be considered and reasoned upon from the moment the planning agent makes them.

Below, $KB_{plan}^\tau = \langle P_{plan} \cup \{time_now(\tau)\}, A_{plan}, I_{plan} \rangle$.

Planning. The planning capability \models_{plan}^τ is specified as follows⁴. Let $S = \langle KB, Goals, Plan, TCS \rangle$ be a state, and $G = \langle l[t], - \rangle$ be a mental goal in *Goals*. Let

$$\begin{aligned}
 - \Delta_0 &= \bigcup_{\langle a[t'], \dots \rangle \in Plan} \{assume_happens(a, t')\} \cup \\
 &\quad \bigcup_{\langle l'[t'], - \rangle \in Goals - \{G\}} \{assume_holds(l', t')\} \\
 - C_0 &= TCS \wedge \Sigma(S).
 \end{aligned}$$

Then, $S, G \models_{plan}^\tau (\mathcal{A}_s, \mathcal{G}_s, Tc)$ where

$$\begin{aligned}
 - \mathcal{A}_s &= \{a[t'] \mid assume_happens(a, t') \in \Delta\} \text{ and} \\
 - \mathcal{G}_s &= \{l'[t'] \mid assume_holds(l', t') \in \Delta\}
 \end{aligned}$$

for some (Δ, Tc) which is an abductive answer for $holds_at(l, t)$, wrt KB_{plan}^τ , Δ_0, C_0 . If no such abductive answer exists, then $S, G \models_{plan}^\tau \perp$, where \perp is used here to indicate failure.

The computational counterpart of \models_{plan}^τ , given by \vdash_{plan}^τ , is defined in terms of CIFF, as follows.

$$\begin{aligned}
 - S, G \vdash_{plan}^\tau (\mathcal{A}_s, \mathcal{G}_s, Tc) \text{ iff} \\
 \quad \bullet KB_{plan}^\tau, holds_at(l, t), \Delta_0, C_0 \vdash_{CIFF} (\Delta, Tc) \\
 \quad \bullet \mathcal{A}_s = \{a[t] \mid assume_happens(a, t) \in \Delta\} \\
 \quad \bullet \mathcal{G}_s = \{g[t] \mid assume_holds(l, t) \in \Delta\}. \\
 - S, G \vdash_{plan}^\tau \perp \text{ iff } KB_{plan}^\tau, holds_at(l, t), \Delta_0, C_0 \vdash_{CIFF} X \text{ and } X = fail \text{ or } \\
 X = flounder.
 \end{aligned}$$

Directly from theorem 1, we prove soundness of \vdash_{plan}^τ wrt \models_{plan}^τ , in the case of success.

Theorem 2. (*Planning Soundness of Success*)

If $\langle KB, Goals, Plan, TCS \rangle, G \vdash_{plan}^\tau (\mathcal{A}_s, \mathcal{G}_s, Tc)$,
 then $\langle KB, Goals, Plan, TCS \rangle, G \models_{plan}^\tau (\mathcal{A}_s, \mathcal{G}_s, Tc)$.

⁴ In the full model, we consider planning for multiple goals concurrently. Here, for simplicity we present the case of planning for single goals only.

Identification of Preconditions. This capability is specified as follows: given a timed action operator $a[t]$, KB , $a[t] \models_{pre} Cs$ iff

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

The computational counterpart \vdash_{pre} of this capability can be obtained by computing \models_{LP} suitably. Trivially, if a sound and complete realisation of \models_{LP} is used, the resulting \vdash_{pre} is sound and complete with respect to \models_{pre} .

Reactivity. The specification of \models_{react}^τ and the provision of \vdash_{react}^τ are very similar to those of \models_{plan}^τ and \vdash_{plan}^τ , and are omitted here for lack of space.

Temporal Reasoning. The temporal reasoning capability \models_{TR} is invoked by other components of the KGP model (namely the Goal Decision capability, the Goal Revision transition and some of the selection functions, see section 4) to prove or disprove that a given (possibly temporally constrained) fluent literal holds (wrt the given theory KB_{TR}). We briefly summarise the specification of \models_{TR} (see [5, 6] for the details).

- KB_{TR} is another variant of the EC, similar to KB_{plan} , and, analogously, is divided into a *domain-independent part*, a *domain dependent part*, and a *narrative part* KB_0 , assumed not to contain “inconsistent” observations. The set of abducibles in KB_{TR} has *assume.holds* as its only abducible.
- \models_{TR} is invoked by the Goal Decision capability to prove that a fluent literal $l[t]$, referring to a time constant t , holds wrt KB_{TR} , denoted as $KB \models_{TR} l[t]$. \models_{TR} is invoked by the revision transitions and selection functions to prove that a fluent literal $l[t]$, referring to a time variable t constrained by some Tc , holds wrt KB_{TR} , denoted as $KB \models_{TR} l[t] \wedge Tc$.
- \models_{TR} is understood *skeptically*, as follows. $KB \models_{TR} l[t] \wedge Tc$ (where Tc may be empty) iff (i) there exists an abductive answer for *holds_at*(l, t) $\wedge Tc$, given KB_{TR} , and (ii) there exists no abductive answer for *holds_at*(\bar{l}, t) $\wedge Tc$, given KB_{TR} , namely the fluent literal can be proven abductively, and its complement cannot.

The computational counterpart of \models_{TR} , given by \vdash_{TR} , is given by first providing a transformed version KB'_{TR} of KB_{TR} , and then by appropriate calls to the CIFF proof procedure, as follows (all details can be found in [6]).

- The transformation of KB_{TR} into KB'_{TR} relies upon the intuition that changes may happen only at significant time points, called *oases*, when events occur, while in the remaining time intervals, called *deserts*, nothing changes. Hence, it is possible to check for the validity of the query fluent literals and the integrity constraints in KB_{TR} with respect to oases only. The transformed KB'_{TR} addresses to some extent computational issues of viable realisation and scalability.

- \vdash_{TR} is defined as follows, given $l[t] \wedge Tc$ (with Tc possibly empty):
- $KB \vdash_{TR} l[t] \wedge Tc$ iff
 - $KB'_{TR}, holds_at(l, t) \wedge Tc \vdash_{CIFF} (\Delta, C)$, for some (Δ, C) , and
 - $KB'_{TR}, holds_at(l, t) \wedge Tc \vdash_{CIFF} fail$.
 - $KB \vdash_{TR}^{fail} l[t] \wedge Tc$ iff
 - $KB'_{TR}, holds_at(l, t) \wedge Tc \vdash_{CIFF} fail$, or
 - $KB'_{TR}, holds_at(l, t) \wedge Tc \vdash_{CIFF} (\Delta, C)$, for some (Δ, C) , and
 - $KB'_{TR}, holds_at(l, t) \wedge Tc \vdash_{CIFF} (\Delta', C')$, for some (Δ', C') .

The following result of soundness directly follows from theorem 1 and from the equivalence between KB_{TR} and KB'_{TR} .

Theorem 3. (*Temporal Reasoning Soundness*)

Given $l[t] \wedge Tc$ (with Tc possibly empty):

- If $KB_{TR} \vdash_{TR} l[t] \wedge Tc$ then $KB_{TR} \models_{TR} l[t] \wedge Tc$.
- If $KB_{TR} \vdash_{TR}^{fail} l[t] \wedge Tc$ then $KB_{TR} \not\models_{TR} l[t] \wedge Tc$.

3.2 LPP-Based Capability: Goal Decision

Background: Logic Programming with Priorities. For the purposes of this paper, a *logic program with priorities*, referred to as \mathcal{T} , consists of four parts:

- (i) a low-level part P , consisting of a logic program; each rule in P is assigned a name, which is a term; e.g., one such rule could be

$$n(X) : p(X) \leftarrow q(X, Y), r(Y)$$
 with name $n(X)$;
- (ii) a high-level part H , specifying conditional, dynamic priorities amongst rules in P ; e.g., one such priority could be

$$h(X) : m(X) \succ n(X) \leftarrow c(X)$$
 to be read: if (some instance of) the condition $c(X)$ holds, then (the corresponding instance of) the rule named by $m(X)$ should be given higher priority than (the corresponding instance of) the rule named by $n(X)$.
- (iii) an auxiliary part A , defining predicates occurring in the conditions of rules in P, H and not in the conclusions of any rule in P ;
- (iv) a notion of incompatibility which, for our purposes, can be assumed to be given as a set of rules defining the predicate *incompatible*, e.g.

$$incompatible(p(X), p'(X))$$
 to be read: any instance of the literal $p(X)$ is incompatible with the corresponding instance of the literal $p'(X)$. We assume that incompatibility is symmetric, and refer to the set of all incompatibility rules as I .

Any concrete LPP framework is equipped with a notion of entailment, that we denote by \models_{pr} , defined differently by different approaches to LPP, wrt some given underlying logic programming semantics \models_{LP} . Intuitively, $\mathcal{T} \models_{pr} \alpha$ iff α is the conclusion (wrt \models_{LP}) of a sub-theory of $P \cup A$ which is “preferred” wrt $H \cup A$ in \mathcal{T} over any other sub-theory of $P \cup A$ that derives a conclusion incompatible with α (wrt I). For example, in [30, 25, 22], \models_{pr} is defined via argumentation (see the next section).

The concrete framework for LPP that we adopt within the computational counterpart of the KGP model is that of Logic Programming without Negation as Failure (*LPwNF*) [9] suitably extended to deal with dynamic preferences [22]. Other concrete frameworks that could be used for LPP instead are, for instance, those presented in [30, 25].

LPwNF: An Argumentation-Based Framework for LPP. In this section, we summarise the main features of *LPwNF* and the notion of preference reasoning \models_{pr} , given with respect to an argumentation-based formulation of *LPwNF*.

LPwNF is a concrete LPP framework, whereby the various components of a theory \mathcal{T} , as defined earlier, are as follows:

- (i) The low-level part P consists of labelled propositional rules of the form $label : l \leftarrow l_1, \dots, l_n$, where l, l_1, \dots, l_n are atoms a or explicitly negative literals $\neg a$. The underlying semantics, \models_{LP} , is given by the single inference rule of modus ponens. Non-ground rules are represented via all their ground instances in the given Herbrand universe of the program.
- (ii) The high-level part H consists of propositional rules of the form $label : l \succ l' \leftarrow l_1, \dots, l_n$, where l_1, \dots, l_n are atoms or explicitly negative literals and l, l' are labels of rules in P .
- (iii) The auxiliary part A is a set of propositional rules of the form $l \leftarrow l_1, \dots, l_n$.
- (iv) The notion of incompatibility includes $incompatible(p, \neg p)$, for all atoms p , and $incompatible(r \succ s, s \succ r)$, for all labels of rules r and s .

We realise the notion of preference reasoning \models_{pr} for *LPwNF* through argumentation. Argumentation has recently been shown to be a useful framework for formalising non-monotonic reasoning and other forms of reasoning (see e.g. [4, 10, 20, 31, 30]). In general, an argumentation framework is a pair (Th, At) where Th is a theory in some background (monotonic) logic, equipped with an entailment \models_{Th} , and At is a binary *attacking relation* on the subsets of Th , i.e. $At \subseteq 2^{Th} \times 2^{Th}$. The subsets of Th form the arguments of the framework and At is therefore an attacking relation between arguments. We will write Δ *attacks* Δ' iff $(\Delta, \Delta') \in At$.

The semantics of an argumentation framework is based upon the following notion of *admissible* argument. An argument $\Delta \subseteq Th$ is admissible iff

- Δ does not attack itself,
- for all arguments $\Delta' \subseteq Th$, if Δ' attacks Δ , then Δ (counter-)attacks Δ' .

To provide \models_{pr} for an *LPwNF* theory $\mathcal{T}=(P, H, A, I)$, we view the latter as a concrete argumentation framework (Th, At) as follows. The set of arguments Th is given by $P \cup H \cup A$, with \models_{Th} given by \models_{LP} . The attacking relation At is realised via a notion of *conflict* (using the notion of incompatibility I of \mathcal{T}) together with a notion of *strength* between arguments (using the preference rules in the H component of \mathcal{T}). Then, the preference semantics \models_{pr} of an *LPwNF* theory is given through argumentation in terms of the maximally admissible subsets of the corresponding argumentation framework. Usually, two variants of \models_{pr} are defined. Given an *LPwNF* theory \mathcal{T} , the corresponding argumentation framework (Th, At) and a formula F ,

- $\mathcal{T} \models_{pr}^{cred} F$ iff there is one maximal (with respect to set inclusion) admissible argument Δ of (Th, At) where F holds, i.e. $\Delta \models_{LP} F$;
- $\mathcal{T} \models_{pr}^{sk} F$ iff $\mathcal{T} \models_{pr}^{cred} F$ and, for any G s.t. $I \cup A \models_{LP} incompatible(F, G)$, it holds that $\mathcal{T} \not\models_{pr}^{cred} G$.

In the *KGP* model, the preference entailment, \models_{pr} , is given by the skeptical entailment \models_{pr}^{sk} in this section, when we define the Goal Decision capability, and by \models_{pr}^{cred} in section 5, when we define the operational trace of computees via cycle theories.

Computing Preferential Reasoning in *LPwNF*. The computational counterpart of \models_{pr} for *LPwNF* is realised via a proof procedure, referred to as GORGIAS [5]. Given a formula F , GORGIAS aims to construct an admissible argument, Δ , that derives F , under the background logic \models_{LP} .

GORGIAS is based on an existing proof theory for computing admissible arguments for abstract argumentation frameworks [23]. This proof theory is given in terms of derivations of trees, where nodes are arguments and each node is labelled as “attack” or “defence”. A defence node is followed by a set of children attack nodes, one for each of its possible minimal (counte-)attacks. An attack node is followed by a defence node containing a (counter-)attack against its parent. *Successful derivations* terminate with a tree whose root, Δ , is an admissible argument supporting, via \models_{LP} , the initially given formula F . The root node Δ_0 of the initial tree is computed by reducing the given formula F into a minimal set that concludes F via \models_{LP} .

GORGIAS specialises this abstract proof theory by incorporating the *LPwNF* specific way of computing attacks and counter-attacks. Any node N of the tree results from first choosing a “culprit” conclusion c of the parent node of N and then reducing (by resolution) some conflicting (incompatible) literal, \bar{c} , of c so that N minimally entails \bar{c} under the background logic \models_{LP} .

The GORGIAS proof procedure then provides the following derivability relations for *LPwNF*. Given a theory \mathcal{T} in *LPwNF* and a literal L , let (Th, At) be the corresponding argumentation framework. Then

- $\mathcal{T} \vdash_{pr}^{cred} L$ iff there exists a successful derivation of GORGIAS for L .
- $\mathcal{T} \vdash_{pr}^{sk} L$ iff $\mathcal{T} \vdash_{pr}^{cred} L$ and $\mathcal{T} \not\vdash_{pr}^{cred} \bar{L}$ for any \bar{L} such that $I \cup A \models_{LP} incompatible(L, \bar{L})$.

We can then show that for finite theories of *LPwNF*, these derivability relations $\vdash_{pr}^{sk}, \vdash_{pr}^{cred}$ based upon GORGIAS, are sound and complete, as follows.

Theorem 4. (*soundness and completeness of \vdash_{pr}^{cred} and \vdash_{pr}*) *Let \mathcal{T} be a finite theory of *LPwNF*. Then the derivability relation \vdash_{pr}^{cred} is sound and complete with respect to \models_{pr}^{cred} , provided that GORGIAS uses a sound and complete realisation of \models_{LP} . Hence the skeptical relation \vdash_{pr}^{sk} is also sound and complete with respect to \models_{pr}^{sk} , provided that GORGIAS uses a sound and complete realisation of \models_{LP} .*

Specification and Computational Model for Goal Decision. The computee Goal Decision capability, \models_{GD}^τ , selects, at a given instant, the top level goals to be pursued. These goals are preferred by the computee at the time of their selection, but this may change over time. This capability relies directly on the underlying preference reasoning within the *LPwNF* framework. It simply uses this form of reasoning with a specific *LPwNF* theory, KB_{GD} , in which the computee represents its goal preference policy.

The knowledge base KB_{GD} is written in four parts in the standard way as for any theory in *LPwNF*. The details of the specific form of its rules are omitted due to lack of space. The main specialised forms of sentences in KB_{GD} are the following. Statements of incompatibility in KB_{GD} , $incompatible(l_1, l_2)$, are amongst literals, l_1 and l_2 , referring to a subset of the fluents in the language of the computee separated out as the set of *goal fluents*. Rules in the basic (low-level) part of KB_{GD} have conclusions of the form $\langle g[t], Tg \rangle$ where g is a goal 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. The auxiliary part, A , of KB_{GD} is augmented with $KB_0 \cup KB_{TR}$ and so the conditions of the rules in KB_{GD} are evaluated by combining the background derivability \models_{LP} of the *LPwNF* framework with the Temporal Reasoning capability \models_{TR} . Finally, any rule in KB_{GD} may have in its body a special atom, denoted by $now(\tau)$ that refers to the (current) time τ at which the capability of goal decision is applied by the computee. We will denote by KB_{GD}^τ the knowledge base obtained by adding to (the auxiliary part of) KB_{GD} the atom $now(\tau)$.

The capability of \models_{GD}^τ is defined directly in terms of the preference, \models_{pr} , of *LPwNF* as follows. Given a state $\langle KB, Goals, Plan, TCS \rangle$, and a time point τ ,

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

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

$$KB_{GD}^\tau \models_{pr} \langle g_1[t_1], Tg_1 \rangle \wedge \dots \wedge \langle g_n[t_n], Tg_n \rangle$$

This means that a new set of goals $\mathcal{G}s$ is generated that is currently (skeptically) preferred under the preference policy represented in KB_{GD} and the current information in KB_0 , via the use of the Temporal Reasoning capability by \models_{pr} . Note that any two goals in $\mathcal{G}s$ are necessarily compatible with each other. Note also that \models_{GD}^τ may return an empty set of goals when there are no skeptically preferred goals at the time τ of application of this capability.

The derivability relation, \vdash_{GD}^τ , and the computational model for Goal Decision can be drawn directly from the general GORGAS proof procedure for *LPwNF* and the derivability relations \vdash_{pr}^{cred} and \vdash_{pr} that this provides, as presented earlier. A simple but relatively inefficient way to compute \vdash_{GD}^τ would then be to generate one by one skeptical goals, via \models_{pr} , adding the most recently generated goal to the previous goals and re-checking, again via \vdash_{pr} , that the whole set remains a skeptical conclusion. A more efficient algorithm for computing \vdash_{GD}^τ that exploits some of the special features of KB_{GD} relies only on

the credulous derivability relation of $LPwNF$ to *generate* in the first step a set of candidate goals and then, using checks of incompatibility, to *filter* from this the required goals.

Let us assume that the knowledge base, KB_{GD}^τ , for any given current time τ together with $KB_{TR} \cup KB_0$ added to the auxiliary part of KB_{GD} is such that only a finite number of goals can be derived from $\mathcal{T} = KB_{GD}^\tau \cup KB_{TR} \cup KB_0$ via the background logic \models_{LP} . We call this assumption the *goal finiteness* assumption. We also assume that the auxiliary part of \mathcal{T} is consistent. Then, the following soundness result holds, directly from theorem 4.

Theorem 5. (*Goal Decision Soundness*) *Let $\mathcal{T} = KB_{GD}^\tau \cup KB_{TR} \cup KB_0$ have goal finiteness property and a consistent auxiliary part. Suppose that a sound and complete realisation of \models_{LP} is used within \vdash_{pr}^{cred} . If $\mathcal{T} \vdash_{GD}^\tau Gs$ then $\mathcal{T} \models_{GD}^\tau Gs$.*

4 Computational Model for Transitions

The KGP model relies upon the state transitions GI, PI, RE, SI, POI, AOI, AE, GR, PR, as discussed in section 2. In [5], we have provided computational counterparts $\vdash_{GI}, \vdash_{PI}, \vdash_{RE}, \vdash_{SI}, \vdash_{POI}, \vdash_{AOI}, \vdash_{AE}, \vdash_{GR}, \vdash_{PR}$ for this transitions, defined via transition rules themselves, obtained from the specifications by replacing calls to capabilities appropriately by calls to their computational counterparts. Below, for one concrete transition (PI), we first summarise the formal specification, and then provide the computational counterpart.

Plan Introduction

This transition takes as input a state and a set of goals in the state (that have been selected by the *goal selection function*, see below) and produces a new state by calling the computee's Planning (\models_{plan}^τ , see section 3.1) and Identification of Preconditions (\models_{pre} , see section 3.1) capabilities. For simplicity, we will provide specification and computational counterpart of this transition in the case of a single input goal (see [5] for the general case of multiple input goals).

$$\text{Specification of (PI)} \quad \frac{\langle KB, Goals, Plan, TCS \rangle \quad G}{\langle KB, Goals', Plan', TCS' \rangle} \tau$$

where G is a goal selected for planning and

$$\begin{aligned} Goals' &= Goals \cup Subg(G) \\ Plan' &= Plan \cup Pplan(G) \\ TCS' &= TCS \cup Tc \end{aligned}$$

where the sets $Subg(G)$, $Pplan(G)$ and Tc are obtained as follows.

- (i) if G is a mental goal: let $\langle KB, Goals, Plan, TCS \rangle, G \models_{plan}^\tau X$. Then, either $X = \perp$ and $Subg(G) = Pplan(G) = Tc = \{\}$, or $X = (As, Gs, Tc)$ and $Subg(G) = \{\langle l[t], G \mid l[t] \in Gs \rangle\}$, $Pplan(G) = \{\langle a[t], G, P \mid a[t] \in As \text{ and } KB, a[t] \models_{pre} P \rangle\}$.

of transitions” according to certain requirements that the designer of the agent would like to impose on the operation of the agent, but still allowing the possibility that any (or a number of) sequences of transitions can actually apply in the “life” of an agent. Thus, whereas a fixed cycle can be seen as a restrictive and rather inflexible catalogue of allowed sequences of transitions (possibly under pre-defined conditions), a cycle theory identifies *preferred patterns* of sequences of transitions. In this way a cycle theory regulates in a flexible way the operational behaviour of the agent.

Cycle theory. Formally, a cycle theory \mathcal{T}_{cycle} consists of the following parts:

- An *initial* part $\mathcal{T}_{initial}$, that determines the possible transitions that the agent could perform when it starts to operate (*initial cycle step*). More concretely, $\mathcal{T}_{initial}$ consists of rules of the form

$$\mathcal{R}_{0|T}(S_0, X): *T(S_0, X) \leftarrow C(S_0, \tau, X), now(\tau)$$

with name $\mathcal{R}_{0|T}(S_0, X)$, and sanctioning that, if the conditions C are satisfied in the initial state S_0 at the current time τ , then the initial transition should be T , applied to state S_0 and input X , if required. Note that the conditions C determine the input X of the first transition T . Such inputs are determined by calls to the appropriate selection functions (see section 4). Note also that $C(S_0, \tau, X)$ may be *true*, and $\mathcal{T}_{initial}$ might simply indicate a fixed initial transition T_1 .

The notation $*T(S, X)$ in the head of these rules, meaning that the transition T can be potentially chosen as the next transition, is used in order to avoid confusion with the notation $T(S, X, S', \tau)$ that we have introduced earlier to represent the actual application of the transition T .

- A *basic* part \mathcal{T}_{basic} that determines the transitions (*cycle steps*) that may follow other transitions, and consists of rules of the form

$$\mathcal{R}_{T|T'}(S', X'): *T'(S', X') \leftarrow T(S, X, S', \tau), EC(S', \tau', X'), now(\tau')$$

with name $\mathcal{R}_{T|T'}(S', X')$, and sanctioning that, after the transition T has been executed, starting at time τ in the state S and ending at the current time τ' in the resulting state S' , and the conditions EC evaluated in S' at τ' are satisfied, then transition T' could be the next transition to be applied in the state S' with the (possibly empty) input X' , if required. The conditions EC are called *enabling conditions* as they determine when a cycle-step from the transition T to the transition T' can be applied. In addition, they determine the input X' of the next transition T' . Such inputs are determined by calls to the appropriate selection functions.

- A *behaviour* part $\mathcal{T}_{behaviour}$ that contains rules describing dynamic priorities amongst rules in \mathcal{T}_{basic} and $\mathcal{T}_{initial}$. Rules in $\mathcal{T}_{behaviour}$ are of the form

$$\mathcal{R}_{T|T'}(S, X') \succ \mathcal{R}_{T|T''}(S, X'') \leftarrow BC(S, X', X'', \tau), now(\tau)$$

with $T' \neq T''$, which we refer to via the name $\mathcal{P}_{T' \succ T''}^T$. Recall that $\mathcal{R}_{T|T'}(\cdot)$ and $\mathcal{R}_{T|T''}(\cdot)$ are (names of) rules in $\mathcal{T}_{basic} \cup \mathcal{T}_{initial}$. Note that, with an abuse of notation, T could be 0 in the case that one such rule is used to specify

a priority over the *first* transition to take place, in other words, when the priority is over rules in $\mathcal{T}_{initial}$. These rules in $\mathcal{T}_{behaviour}$ sanction that, at the current time τ , after transition T , if the conditions BC hold, then we prefer the next transition to be T' over T'' , namely doing T' has *higher priority* than doing T'' , after T . The conditions BC are called *behaviour conditions* and give the behavioural profile of the agent. These conditions depend on the state of the agent after T and on the parameters chosen in the two cycle steps represented by $\mathcal{R}_{T|T'}(S, X')$ and $\mathcal{R}_{T|T''}(S, X'')$. Behaviour conditions are *heuristic* conditions, which may be defined in terms of *heuristic selection functions* (see [17] for details). For example, the heuristic action selection function may choose those actions in the agent's plan whose time is close to running out amongst those whose time has not run out.

- An *auxiliary part* including definitions for any predicates occurring in the enabling and behaviour conditions, and in particular for selection functions (including the heuristic ones, if needed).
- An *incompatibility part*, including rules stating that all different transitions are incompatible with each other:

$$incompatible(*T(S, X), *T'(S, X'))$$

for all T, T' such that $T \neq T'$, and that different calls to the same transition but with different input items are incompatible with each other:

$$incompatible(*T(S, X), *T(S, X')) \leftarrow X \neq X'$$

Overall, these rules express that only one transition can be chosen at a time.

Hence, \mathcal{T}_{cycle} is an LPP-theory (see Section 3.2) where:

- (i) $P = \mathcal{T}_{initial} \cup \mathcal{T}_{basic}$, and
- (ii) $H = \mathcal{T}_{behaviour}$.

In the sequel, we will indicate with \mathcal{T}_{cycle}^0 the sub-cycle theory $\mathcal{T}_{cycle} \setminus \mathcal{T}_{basic}$ and with \mathcal{T}_{cycle}^s the sub-cycle theory $\mathcal{T}_{cycle} \setminus \mathcal{T}_{initial}$.

Operational Trace. A cycle theory \mathcal{T}_{cycle} is used to induce *cycle operational traces* of an agent, namely a (typically infinite) sequence of transitions

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

(where each of the X_i may be empty), such that

- S_0 is the given initial state;
- for each $i \geq 1$, τ_i is given by the clock of the system, such that $\tau_i < \tau_{i+1}$;
- (*Initial Cycle Step*) $\mathcal{T}_{cycle}^0 \wedge now(\tau_1) \models_{pr} *T_1(S_0, X_1)$;
- (*Cycle Step*) for each $i \geq 1$
 $\mathcal{T}_{cycle}^s \wedge T_i(S_{i-1}, X_i, S_i, \tau_i) \wedge now(\tau_{i+1}) \models_{pr} *T_{i+1}(S_i, X_{i+1})$
namely each (non-final) transition in a sequence is followed by the most preferred transition, as specified by \mathcal{T}_{cycle}^s . If the most preferred transition determined by \models_{pr} is not unique, we choose arbitrarily one.

Computational Counterpart of Operational Trace. Since the notion of operational trace is based upon \models_{pr} and transitions (and selection functions), its computational counterpart is obtained by replacing \models_{pr} with its computational counterpart \vdash_{pr} and transitions with their computational counterparts. Thus, a computational operational trace is a (possibly infinite) sequence of computational counterparts of transitions, of the form

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

where

- $\mathcal{T}_{cycle}^0 \wedge now(\tau_1) \vdash_{pr} *T_1(S_0, X_1)$ and,
- for each $i \geq 1$, $\mathcal{T}_{cycle}^s \wedge T^i(S_{i-1}, X_i, S_i, \tau_i) \wedge now(\tau_{i+1}) \vdash_{pr} *T_{i+1}(S_i, X_{i+1})$.

Trivially, computational counterparts of operational traces correspond to operational traces, thanks to the soundness results for each transition (similarly to theorem 6) and the soundness of \models_{pr} (see theorem 4).

6 Implementation

To realise the logical and computational aspects of the KGP model we have developed PROSOCS [36], a platform which allows us to deploy and test the functionality of KGP agents via the SOCSiC component of PROSOCS. Deployment of KGP agents using SOCSiC is based on an *agent template* whose design [35] builds upon previous work in multi-agent systems, in particular, the *head/body* metaphor described by [37] and [14], and the *mind/body* architecture introduced by [3] and more recently used by [15].

In the mind part of a PROSOCS agent, the ALP-based components of the KGP model are implemented in CIFF [11, 12] and the LPP-based components of the KGP model are implemented in GORGIAS [1]. Overall, we build the mind using SICStus Prolog [34] and the bidirectional Java-Prolog interface Jasper it provides; Jasper is used by the body to exchange information with the mind.

To implement the body of the agent we use Java *on top* of the Peer-to-Peer JXTA Project [39]. JXTA is suitable for the low-level functionality of a PROSOCS agent, such as interaction with the environment, and is provided in the form of an API (Application Programming Interface). By importing this API when we instantiate specific PROSOCS agents, we enable such agents to discover bodies of other PROSOCS agents (using JXTA's peer discovery protocols facilities for dynamic discovery in a GC network) as well as communicate with other agents (using JXTA's facilities for message transport and structuring via a pipe binding and resolver protocols).

To facilitate experimentation with KGP agents we have built interfaces, which allow us to animate an agent's behaviour while interacting with other agents in the context of a GC application. The aspects of the agent's behaviour that we animate, in the current state of the implementation, are: the computational trace in terms of the names of transitions being executed, the observations the agent

makes and the actions it executes, the internal state of the agent in terms of its knowledge, goals and plans. More details of the implementation of the agent template can be found in [2, 36].

7 Conclusions

In this paper we have summarised the declarative model of the KGP agents [18] and then given its computational counterpart and briefly described its implementation. The declarative model is based on computational logic, in particular on abductive logic programming and logic programming with priorities. It is modular, hierarchical and extensible. It specifies a collection of capabilities, uses them to define a collection of transitions, to be used within logically specified context sensitive cycle theories. In close relationship to the declarative model, the computational model mirrors the logical architecture by specifying appropriate computational counterparts for the capabilities and using these to give the computational models of the transitions. These computational models and the one specified for the cycle theories are all based on, and are significant extensions of, existing proof procedures for abductive logic programming and logic programming with priorities. This design has satisfied two of the main motivations of the KGP model, namely to reduce the gap between the logical and computational realisations, and to exploit and integrate (extensions of) existing computational logic tools and techniques.

The paper also reports a first prototype implementation of the KGP model based on SICStus Prolog, Java and JXTA [36]. The implementation reflects the modular and hierarchical architecture and is similarly extensible. The prototype has been used successfully on a number of small scenaria demonstrating, amongst others, the situatedness of agents and their adaptability and responsiveness to changes in their environment, as required by the GC challenge.

The KGP model does not currently include a number of features, some of which are subject of future work. It would be advantageous, for example, to include a more sophisticated way to assimilate and revise knowledge, for example using inductive logic programming. Another useful feature would be to allow agents to have in their knowledge, and to reason with, what they believe other agents believe. Such knowledge can be used, for example, in communication strategies and even possibly in planning. A particular issue included in the GC vision and not addressed in the KGP model to date is physical mobility, although we do cater, at the knowledge level, for agents moving from one society to another [38]. This issue will necessitate further work. Finally, we are currently working on identifying and verifying formally properties of behaviour of computees.

Acknowledgements

This work was funded by the IST programme of the EC, FET under the IST-2001-32530 SOCS project, within the GC proactive initiative. K. Stathis and F. Toni were also supported by the Italian MIUR programme “Rientro dei Cervelli”.

References

1. *Gorgias User Guide*: <http://www.cs.ucy.ac.cy/~nkd/gorgias/>, 2003.
2. M. Alberti, A. Bracciali, F. Chesani, U. Endriss, M. Gavanelli, W. Lu, K. Stathis, and P. Torroni. SOCS prototype. Technical report, SOCS Consortium, 2003. Deliverable D9.
3. J. Bell. A Planning Theory of Practical Rationality. In *Proceedings of AAAI'95 Fall Symposium on Rational Agency*, pages 1–4. AAAI Press, 1995.
4. 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.
5. A. Bracciali, N. Demetriou, U. Endriss, M. Gavanelli, A. C. Kakas, E. Lamma, P. Mancarella, P. Mello, P. Moraitis, F. Sadri, K. Stathis, G. Terreni, F. Toni, and P. Torroni. Computational model for computees and societies of computees. Technical report, SOCS Consortium, 2003. Deliverable D8.
6. A. Bracciali and A. Kakas. Frame consistency: Reasoning with explanations. In *Proceedings of the 10th International Workshop on "Non-Monotonic Reasoning" (NMR2004)*, Whistler BC, Canada, 2004.
7. F. Chesani, M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. The SOCS computational logic approach to the specification and verification of agent societies. 2004. This volume.
8. K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
9. Y. Dimopoulos and A. C. Kakas. Logic programming without negation as failure. In *Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon*, pages 369–384, 1995.
10. P. M. Dung. On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77:321–357, 1995.
11. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure for abductive logic programming with constraints. In *Proceedings JELIA04*. To appear.
12. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Abductive logic programming with CIFF: implementation and applications. In *Proceedings CILC2004, Convegno Italiano di Logica Computazionale*, 2004.
13. T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, Nov. 1997.
14. H. Haugeneder, D. Steiner, and F. McCabe. IMAGINE: A framework for building multi-agent systems. In S. M. Deen, editor, *Proceedings of the 1994 International Working Conference on Cooperating Knowledge Based Systems (CKBS-94)*, pages 31–64, DAKE Centre, University of Keele, UK, 1994.
15. Z. Huang, A. Eliens, , and P. de Bra. An Architecture for Web Agents. In *Proceedings of EUROMEDIA'01*. SCS, 2001.
16. J. Jaffar and M. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.
17. A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. A logic-based approach to model computees. Technical report, SOCS Consortium, 2003. Deliverable D4.
18. A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In *Proceedings ECAI2004*, 2004. To appear.

19. A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.
20. A. C. Kakas, P. Mancarella, and P. M. Dung. The acceptability semantics for logic programs. In *Proceedings of the Eleventh International Conference on Logic Programming, Santa Margherita Ligure, Italy*, pages 504–519, 1994.
21. A. C. Kakas and R. Miller. A simple declarative language for describing narratives with actions. *Logic Programming*, 31, 1997.
22. A. C. Kakas and P. Moraitis. Argumentation based decision making for autonomous agents. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2003)*, pages 883–890, Melbourne, Victoria, 2003. ACM Press.
23. A. C. Kakas and F. Toni. Computing argumentation in logic programming. *Journal of Logic and Computation*, 9:515–562, 1999.
24. A. C. Kakas, B. van Nuffelen, and M. Denecker. A-System: Problem solving through abduction. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 591–596, Seattle, Washington, USA, August 2001. Morgan Kaufmann Publishers.
25. R. Kowalski and F. Toni. Abstract argumentation. *Artificial Intelligence and Law Journal, Special Issue on Logical Models of Argumentation*, 4(3-4):275–296, 1996. Kluwer Academic Publishers.
26. R. A. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.
27. R. A. Kowalski. Problems and promises of computational logic. In *Proceedings of the Symposium on Computational Logic*, pages 1–36. Springer-Verlag, 1990.
28. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.
29. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd extended edition, 1987.
30. H. Prakken and G. Sartor. *A System for Defeasible Argumentation, with Defeasible Priorities*, pages 510–524. 1996.
31. H. Prakken and G. Sartor. Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-Classical Logics*, 7(1), 1997.
32. M. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 1055–1060, 1989.
33. M. Shanahan. *Solving the Frame Problem*. MIT Press, 1997.
34. SICStus Prolog user manual, release 3.8.4, 2000. Swedish Institute of Computer Science.
35. K. Stathis, C. Child, W. Lu, and G. K. Lekeas. Agents and Environments. Technical report, SOCS Consortium, 2002. IST32530/CITY/005/DN/I/a1.
36. K. Stathis, A. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: a platform for programming software agents in computational logic. In J. Müller and P. Petta, editors, *Proceedings of From Agent Theory to Agent Implementation (AT2AI-4 – EMCSR’2004 Session M)*, pages 523–528, Vienna, Austria, 2004.
37. D. E. Steiner, H. Haugeneder, and D. Mahling. Collaboration of knowledge bases via knowledge based collaboration. In S. M. Deen, editor, *CKBS-90 — Proceedings of the International Working Conference on Cooperating Knowledge Based Systems*, pages 113–133. Springer Verlag, 1991.

38. 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)*, volume 2577 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.
39. B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J. C. Hugly, and E. Pouyoul. Project JXTA-C: Enabling a web of things. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*, pages 282–287. IEEE Press, 2003.