Self-checking Logical Agents

Stefania Costantini[1]

Dip. di Ingegneria e Scienze dell'Informazione (DISIM), Università di L'Aquila, Coppito 67100, L'Aquila, Italy stefania.costantini@di.univaq.it

Abstract. This paper presents a comprehensive framework for run-time self-checking of logical agents, by means of temporal axioms to be dynamically checked. These axioms are specified by using an agent-oriented interval temporal logic defined to this purpose. We define syntax, semantics and pragmatics for this new logic, specifically tailored for application to agents. In the resulting framework, we encompass and extend our past work.

1 Introduction

Agent systems are more and more widely used in real-world applications: therefore, the issue of verification is becoming increasingly important, as discussed for instance in [1] (and in the many references therein).

According to [2], given representations of an agent, an environment, and a task we wish the agent to carry out in this environment, verification tries to determine whether the agent will carry out the task successfully. In particular, given the specification of agent Ag, environment Env and property ψ, the "verification problem" can be defined as the decision problem related to establishing whether ψ is reached in every run of Ag. As discussed in [2], the complexity of ψ will affect the complexity of this problem. As [2] points out, ψ can be either an *achievement goal*, i.e., a desirable state the the agent wants to reach, or a *maintainance goal*, related to undesirable states that the agent wishes to avoid. In other words, and taking time into account, two kinds of temporal properties can be distinguished: *liveness* properties concern the progress that an agent makes and express that a (good) state eventually will be reached; and *safety* properties express that some (bad) state will never be entered.

Static verification of agent programs and systems (see Section 4), i.e., verification performed prior to agent activation, can be accomplished through model-checking techniques [3], abstract interpretation [4] or theorem proving. This paper presents instead an approach to dynamic (run-time) verification of agent systems. Our motivation is that agents behavior is affected by their interaction with the external world, so in most practical cases the actual arrival order of events and thus the actual agent evolution is unforeseeable. Often, the set of possible events is so large that computing all combinations would result in a combinatorial explosion, thus making "a priori" (static) verification techniques only partially applicable. Moreover, the set of in-coming events may be only partially known in advance, at least if one admits that agents should learn, in the sense of enlarging over time their set of behaviors.

Therefore, we believe that static verification should be integrated with dynamic self-checking, basically aimed at detecting violations of wished-for properties. A crucial

point of our proposal is that, in case of violation, agents should try to restore an acceptable or desired state of affairs by means of run-time *self-repair*. Even in case desired properties are fulfilled, by examining relevant parameters of its own activities an agent might apply forms of *self-improvement* so as to perform better in the future.

Self-repair and improvement are seen in the direction of overcoming or at least alleviating "brittleness", that can be intended as the propensity of an agent to perform poorly or fail in the face of circumstances not explicitly considered by the agent's designer. In opposition to brittleness, [5] mentions *Versatility* as the ability of being trained to perform previously unanticipated tasks. [6,7] introduce the concept of *Perturbation Tolerance*, where a perturbation is any unanticipated change, either in the world or in the system itself, that impacts an agents performance. To achieve Perturbation Tolerance, [6,7] define a time-based *active logic* and a *Metacognitive Loop* (MCL), that involves a system monitoring, reasoning and meta-reasoning about and if necessary altering its own behavior. [7] presents a number of domain-specific implementations of MCL, to demonstrate that MCL is a general-purpose methodology for building flexible (non-brittle) programs in specific domains, and discusses the perspective of domain-independent implementation of MCL.

We agree with [6] on the fact that "self-consciousness" and self-modification are key aspects for building flexible and adaptable agents. In fact, we propose a comprehensive framework for checking the agent behavior correctness during the agent activity, aimed at self-repair and improvement (within this framework we encompass and extend pieces of our previous work [8,9,10]). We define an interval temporal logic devised to this aim, and particularly tailored to the agent realm, called A-ILTL (for Agent-oriented Interval Temporal Logic), for which we provide a full semantics. Based on A-ILTL formulas, we introduce two new kinds of constraints to be added to agent programs and to be checked dynamically, that we call, respectively, *A-ILTL Rules* and *A-ILTL Expressions*. These constraints are meant to be automatically attempted at a certain (customizable) frequency, where priorities can be established among properties to be verified. A-ILTL rules and expressions, also according to what has happened and to what is supposed to happen or not to happen in the future, define properties that should hold and what should be done if they are violated or if they are fulfilled, so that an agent can repair/improve its own behavior.

Our approach is very general, and thus could be adopted in several logic agent-oriented languages and formalisms. In particular, one such language is DALI [11,12], which is an agent-oriented extension to prolog that we have defined and developed in previous work (cf. [13] for a comprehensive list of references about DALI, while the DALI interpreter is publicly available at [14]). We have experimented with DALI in a variety of applications (see, e.g., [15,16,17,18]), from which we have drawn experience and inspiration for the present work. We have added A-ILTL rules and expressions to DALI, where we also have prototypically implemented the approach. DALI is in itself an 'active' agent-oriented logic programming language with certain features, in particular the "internal events", going towards flexible self-adaptability. In fact, internal events allow a DALI agent to react to internal conditions whenever they occur. By means of internal events DALI agents can take initiatives, adopt goals and intentions, execute plans and manipulate and revise their knowledge on their own accord, independently of

the environment. Therefore, the new features fit gracefully in this setting. In Section 5 we will show by means of a small but complete sample application how A-ILTL rules and expressions can be exploited in a DALI agent.

It can be useful to remark that in the proposed framework agent's state and behavior is checked (and possibly corrected, improved or re-arranged) during agent's functioning not by means of rules as usually intended, but by means of special constraints which are checked automatically with frequencies and priorities customizable according to the specific requirements of the application at hand. This helps to alleviate the problem mentioned in [19] that in rule-based systems "every item which is added to memory via a rule must be maintained by other rules ..." thus unavoidably resulting, in their opinion, in brittleness of these system. Brittleness and inflexibility are in fact often attributed to rule-based systems due to their supposed over-commitment to particular courses of action, that our approach intends to loosen.

The paper is structured as follows. In Section 2 we provide the reader with some notions concerning our declarative semantics of evolving agents, and we introduce the A-ILTL temporal logic and its semantics. In Section 3 we introduce A-ILTL Rules in various forms, and provide their semantics. We compare the proposed approach with other approaches to agent verification in Section 4. In Section 5 we provide an example demonstrating how the various elements of the proposed approach can be put at work in synergy. Finally, in Section 6 we conclude.

2 Agent Evolution

2.1 Evolutionary Semantics

In this paper we will refer to the declarative semantics introduced in [8], aimed at declaratively modeling changes inside an agent which are determined both by changes in the environment, that we call *external events*, and by the agent's own self-modifications, that we call *internal events*. The key idea is to understand these changes as the result of the application of program-transformation functions that perform changes to the agent program. E.g., the internal event corresponding to the decision of the agent to embrace a goal triggers a program transformation step, resulting in a version of the program where a corresponding plan is "loaded" so as to become executable.

An agent in this framework is defined as the tuple $Ag = < P_{Ag}, E, I, A >$ where Ag is the agent name and P_{Ag} (that we call "agent program") describes the agent behavioral rules in some agent-oriented language $\mathcal{L}$. E is the set of the external events, i.e, events that the agent is capable to perceive and recognize: let $E = \{E_1, \ldots, E_n\}$ for some n. I is the internal events set (distinguished internal conclusions): let $I = \{I_1, \ldots, I_m\}$ for some m. A is the set of actions that the agent can possibly perform: let $A = \{A_1, \ldots, A_k\}$ for some k. Let $\mathcal{Y} = (E \cup I \cup A)$. In set $\mathcal{Y}$, a postfix (to be omitted if irrelevant) indicates the kind of event. I.e., X_E is an external event, X_A is an action and X_I an internal event.

Program P_{Ag} written by the programmer is transformed into the initial agent program P_0 by means of an (optional) *initialization step*, that may possibly rewrite the

program in an intermediate language and/or load a "virtual machine" that supports language features and/or extract control information, etc. Thus, P_0 can be simply a program (logical theory) or can have additional information associated to it.

Later on, P_0 will evolve according to events that happen and actions which are performed, through corresponding program-transformation steps (each one transforming P_i into P_{i+1}, cf. [8]), thus producing a Program Evolution Sequence $PE = [P_0, \ldots, P_n, \ldots]$. The program evolution sequence will imply a corresponding Semantic Evolution Sequence $ME = [M_0, \ldots, M_n, \ldots]$ where M_i is the semantic account of P_i according to the semantics of $\mathcal{L}$. Notice in fact that the approach is parametric w.r.t $\mathcal{L}$.

The choice of a specific $\mathcal{L}$ will influence the following key points: (i) when a transition from P_i to P_{i+1} takes place, i.e., which are the external and internal factors that determine a change inside the agent; (ii) which kind of transformations are performed; (iii) which semantic approach is adopted, i.e., how M_i is obtained from P_i.

Let H be the *history* of an agent as recorded by the agent itself, and contains events that happened and actions that have been performed by the agent (seen as a particular kind of event), each one time-stamped so as to indicate when they occurred. In particular, we introduce a set P of current "valid" past events that describe the current state of the world [1], and a set PNV where to store previous ones if still useful. Thus, the history H is the couple $\langle P, PNV \rangle$. In practice, H is dynamically updated with new events that happen: as soon as event X is perceived by the agent, it is recorded in P in the form ${X_P}^Y : T_i$, $Y \in \{E, A, I\}$. In [9] we have defined *Past Constraints*, which allow one to define when and upon which conditions (apart from arrival of more recent ones) past events should be moved into PNV.

Definition 1 (Evolutionary semantics). *Let Ag be an agent. The evolutionary semantics ε^{Ag} of Ag is a tuple $\langle H, PE, ME \rangle$, where H is the history of Ag, and PE and ME are respectively its program and semantic evolution sequences.*

The next definition introduces the notion of instant view of ε^{Ag}, at a certain stage of the evolution (which is in principle of unlimited length).

Definition 2 (Evolutionary semantics snapshot). *Let Ag be an agent, with evolutionary semantics ε^{Ag} = $\langle H, PE, ME \rangle$. The* snaphot at stage i *of ε_i^{Ag} is the tuple $\langle H_i, P_i, M_i \rangle$, where H_i is the history up to the events that have determined the transition from P_{i-1} to P_i.*

In [8] we have coped in detail with evolutionary semantics of DALI language, specifying which program transformation steps are associated with DALI language constructs. We hope however to have convinced the reader about the fact that the approach is in principle applicable to many other agent-oriented languages.

2.2 Interval Temporal Logic A-ILTL

For defining properties that are supposed to be respected by an evolving system, a well-established approach is that of Temporal Logic (introduced in Computer Science by

[1] An agent can describe the state of the world only in terms of its perceptions, where more recent remembrances define the agent's approximation of the current state of affairs.

Pnueli [20], for a survey the reader can refer to [21]), and in particular of Linear-time Temporal Logic (LTL). LTL logics are called 'linear' because, in contrast to branching time logics, they evaluate each formula with respect to a vertex-labeled infinite path (or "state sequence") $s_0 s_1 \ldots$ where each vertex s_i in the path corresponds to a point in time (or "time instant" or "state"). LTL enriches an underlying propositional logic language with a set of temporal unary and binary connectives referring to future time and past time. In what follows, we use the following notation for the best-known LTL operators: X stands for 'next state', or next time; F stands for 'eventually', or 'sometime'; G stands for 'always', N stands for 'never'.

LTL expressions are interpreted in a discrete, linear model of time. Formally, this structure is represented by $\mathcal{M} = \langle \mathbb{N}, \mathcal{I} \rangle$ where, given countable set Σ of atomic propositions, interpretation function $\mathcal{I} : \mathbb{N} \mapsto 2^{\Sigma}$ maps each natural number i (representing state s_i) to a subset of Σ. Given set $\mathcal{F}$ of formulas built out of classical connectives and of LTL operators, the semantics of a temporal formula is provided by the satisfaction relation $\models : \mathcal{M} \times \mathbb{N} \times \mathcal{F} \rightarrow \{true, false\}$. For $\varphi \in \mathcal{F}$ and $i \in \mathbb{N}$ we write $\mathcal{M}, i \models \varphi$ if, in the satisfaction relation, φ is true w.r.t. $\mathcal{M}, i$. We can also say (leaving $\mathcal{M}$ implicit) that φ *holds* at i, or equivalently in state s_i, or that state s_i satisfies φ. For atomic proposition $p \in \Sigma$, we have $\mathcal{M}, i \models p$ iff $p \in \mathcal{I}(i)$. The semantics of $\models$ for classical connectives is as expected, and the semantics for LTL operators is as reported in [21]. A structure $\mathcal{M} = \langle \mathbb{N}, \mathcal{I} \rangle$ is a model of φ if $\mathcal{M}, i \models \varphi$ for some $i \in \mathbb{N}$. Similarly to classical logic, an LTL formula φ can be satisfiable, unsatisfiable or valid and one can define the notions of entailment and equivalence between two LTL formulas.

In prior work (see e.g., [22]) we informally introduced an extension to temporal logic based on *intervals*, called A-ILTL for 'Agent-Oriented Interval LTL', that we report, formalize and extend here. Via A-ILTL operators the time point or the time interval in which a given temporal formula is supposed to hold are explicitly stated. E.g., $G_{m,n}$ (*always in time interval*) states that formula φ should become true at most at state s_m and then hold at least until state s_n. Intervals can have an upper bound or can be unlimited, in the sense that only the lower bound is provided.

The introduction of A-ILTL is in our opinion useful despite the fact that, since the seminal work of [23], several "metric temporal logics" (MTL) have been defined (cf., e.g., [24,25] and the references therein). These logics are able to express "metric", or quantitative time constraints. This is important and necessary as in many systems and applications there are properties which are not required to hold forever or somewhere in time, but in specific time instants or intervals: in fact, a system can be required to performed a certain task *at* or *by* a certain time, or for a certain duration. MTL logics however have been mainly devised for applications involving real-time and hybrid systems, with an underlying 'dense' (continuous) model of time, based on real numbers. Consequently, as pointed out in [25], general results about expressiveness, decidability and complexity are lacking as these properties turn out to be sensitive to slight differences in the semantics or in the choice of operators. In contrast, A-ILTL is defined as a simple extension of LTL, then still relying upon an underlying discrete linear model of time. We impose strong limitations upon nesting of operators, so as to avoid having to explicitly cope with the representation of time intervals and their interactions. However,

as we will see in the rest of the paper this simple formulation is sufficient for expressing and checking a number of interesting properties of agent systems.

Formal syntax and semantics of "core" A-ILTL operators (also called below "Interval Operators") are defined as follows.

Definition 3. *Set $\mathcal{F}$ of A-ILTL formulas is built out out of classical connectives and of LTL operators and of the following A-ILTL operators, where m, n are positive integer numbers (with $m \leq n$), and φ and ψ are LTL formulas (i.e., nesting of A-ILTL operators is not allowed).*

- $C(i)$ *(*current state*). $C(i)$ is true if s_i is the current state. I.e., $\mathcal{M}, i \models C(i')$ iff $i = i'$. From this operator we obtain the shorthand expression now, where $now = t : C(t)$.*
- p_i *(p* at i*). Proposition p holds at time i. This notation transposes a propositional letter into a "timed" version. I.e., $\mathcal{M}, i \models p(i)$ if $\mathcal{M}, i \models p$. The short form p_{now} is a shorthand, where $\mathcal{M}, i \models p_{now}$ if $\mathcal{M}, i \models p_i \wedge C(i)$.*
- $p_{\langle i \rangle}$ *(p* since i*). Proposition p holds since time i. This notation transposes a propositional letter into a "timed" version, but considers a peculiar feature of agents, where due to the interaction with the environment the agent modifies its knowledge base. There is no need to change the overall semantic framework, as function $\mathcal{I}$ may account for such changes. I.e., $\mathcal{M}, i \models p\langle i \rangle$ if $\mathcal{M}, i \models p$ and $\forall i' < i$, $\mathcal{M}, i' \not\models p$. The short form $p_{\langle now \rangle}$ is a shorthand, where $\mathcal{M}, i \models p_{\langle now \rangle}$ if $\mathcal{M}, i \models p_{\langle i \rangle} \wedge C(i)$.*
- X_m *(*future m-state*). $X_m\varphi$ holds if φ is true in the $(m+1)$-th state after the current state. I.e., $\mathcal{M}, i \models X_m\varphi$ if $\mathcal{M}, i' \models \varphi$, $i' = i + m$. This operator is relative to current state $C(i)$, in fact it can hold or not hold depending on state i that is considered. Therefore a more suitable form is $X_m(i)$, where the reference state is explicitly stated.*
- F_m *(*bounded eventually (or "finally")*). $F_m\varphi$ holds if φ is true somewhere on the path from the current state to the (m)-th state after the current one. I.e., $\mathcal{M}, i \models F_m\varphi$ if there exists j such that $j \geq i$ and $j \leq i + m$ and $\mathcal{M}, j \models \varphi$. This operator is relative to current state $C(i)$, in fact it can hold or not hold depending on state i that is considered. Therefore a more suitable form is $F_m(i)$, where the reference state is explicitly stated.*
- $F_{m,n}$ *(*eventually (or "finally") in time interval*). $F_{m,n}\varphi$ states that φ has to hold somewhere on the path from state s_m to state s_n. I.e., $\mathcal{M}, i \models F_{m,n}\varphi$ if there exists j such that $j \geq m$ and $j \leq n$ and $\mathcal{M}, j \models \varphi$.*
- G_m *(*bounded always*). $G_m\varphi$ states that φ should become true at most at state s_m. It is different from LTL G in that the state where the property should start to hold is explicitly indicated. I.e., $\mathcal{M}, i \models G_m\varphi$ if for all j such that $j \geq m$ $\mathcal{M}, j \models \varphi$.*
- $G_{\langle m \rangle}$ *(*bounded strong always*). $G_{m,n}\varphi$ states that φ should become true just at state s_m, while it was not true at previous states. I.e., $\mathcal{M}, i \models G_{\langle m \rangle}\varphi$ if for all j such that $j \geq m$ $\mathcal{M}, j \models \varphi$ and $\forall i' < m$, $\mathcal{M}, i' \not\models \varphi$.*
- $G_{m,n}$ *(*always in time interval*). $G_{m,n}\varphi$ states that φ should become true at most at state s_m and then hold at least until state s_n. I.e., $\mathcal{M}, i \models G_{m,n}\varphi$ if for all j such that $j \geq m$ and $j \leq n$ $\mathcal{M}, j \models \varphi$.*

$G_{\langle m,n\rangle}$ *(*strong always in time interval*). $G_{\langle m,n\rangle}\varphi$ states that φ should become true just in s_m and then hold until state s_n, and not in s_{n+1}. I.e., $\mathcal{M}, i \models G_{\langle m,n\rangle}\varphi$ if for all j such that $j \geq m$ and $j \leq n$ $\mathcal{M}, j \models \varphi$, and $\forall j' < m$, $\mathcal{M}, j' \not\models \varphi$, and $\mathcal{M}, j'' \not\models \varphi$, $j'' = n + 1$.*

N^b_m *(*never before*). $N^b_m\varphi$ states that φ should not be true in any state prior than s_m, i.e., $\mathcal{M}, i \models N^b_m\varphi$ if there not exists j such that $j < m$ and $\mathcal{M}, j \models \varphi$.*

N^a_m *(*never after*). $N^a_m\varphi$ states that φ should not be true in any state after s_m, i.e., $\mathcal{M}, i \models N^a_m\varphi$ if there not exists j such that $j > m$ and $\mathcal{M}, j \models \varphi$.*

$N_{m,n}$ *(*never in time interval*). $N_{m,n}\varphi$ states that φ should not be true in any state between s_m and s_n, i.e., $\mathcal{M}, i \models N_{m,n}\varphi$ if there not exists j such that $j \geq m$ and $j \leq n$ and $\mathcal{M}, j \models \varphi$.*

$E_{m,f}$ *(*bounded sometimes*). $E_{m,n}\varphi$ states that φ has to be true one or more times starting from state s_m, with frequency f. I.e., $\mathcal{M}, i \models E_{m,f}\varphi$ if $\mathcal{M}, m \models \varphi$ and $\mathcal{M}, i \models E_{m',f}\varphi$, $m' = m + f$.*

$E_{m,n,f}$ *(*sometimes in time interval*). $E_{m,n}\varphi$ states that φ has to be true one or more times between s_m and s_n, with frequency f. I.e., $\mathcal{M}, i \models E_{m,n,f}\varphi$ if $\mathcal{M}, i \models \varphi$ whevener $m + f \geq n$, or otherwise if $\mathcal{M}, m \models \varphi$ and $\mathcal{M}, i \models E_{m',n}\varphi$ with $m' = m + f$.*

Other A-ILTL operators (also referring to the past) can be defined (see [22], where a preliminary version of A-ILTL, called I-METATEM, was presented) but we do not report them here for the sake of brevity. There is no need to change the notion of model reported above for LTL: in fact, drawing inspiration from the LTL treatment of 'next state' and 'sometime', an A-ILTL formula holds in a state if it can be checked to hold in the states included in the interval the formula refers to. In this sense, it is the whole state sequence that implies truth or falsity of an A-ILTL formula. However, it is easy to see that for most A-ILTL formulas $Op\varphi$ there is a *crucial state* where it is definitely possible to assess whether the formula holds or not in given state sequence, by observing the sequence up to that point and ignoring the rest. The crucial state is for instance $i+m$ for $X_m(i)$, some j with $j \geq i$ and $j \leq m$ for $F_m(i)$, etc. It corresponds to the upper bound of the *interval of interest* of operator Op, which is the interval $[v, w]$ of the first and last states where the inner formula φ must be checked, according to the semantic definition of the operator. Sometimes w can be ∞. If $w = \infty$ then there is no crucial state, as it is the case for instance for N^a_m.

In the above formulation, for simplicity we do not allow A-ILTL operators to be nested. An extension in this sense is possible, but one should then consider how temporal intervals interact (cf., e.g., [26] for a survey of many existing interval temporal logics), and this requires relevant modifications to the semantic approach that should explicitly cope with time intervals instead of time instants. Instead, A-ILTL operators can occur within LTL ones. We have purposedly defined a very simple interval logic that, as we will see below, is sufficient to our aims with no computational extra-burden. In fact, it is easy to get convinced that the addition of interval operators leaves the complexity of the resulting logic unchanged with respect to plain LTL.

We can employ A-ILTL formulas in *response formulas* (also called *response rules*) in the sense of [27]. They are of the form $p \Rightarrow q$, meaning that any state which satisfies p must be followed by a later state which satisfies q (see [27] for formal properties of

$\Rightarrow$). Assuming for instance a suitable encoding of time and date with corresponding operators for adding, say, minutes hours or days to a given date, one can write by means of A-ILTL formulas response rules such as the following, stating that upon an order received at date d, the corresponding product should be delivered within k days.

$$received\_order_{\langle d \rangle} \Rightarrow F_{d+k_{days}} deliver\_product$$

To express that orders must *always* be processed this way, we can exploit the corresponding LTL operator:

$$G(received\_order_{\langle d \rangle} \Rightarrow F_{d+k_{days}} deliver\_product)$$

The A-ILTL formula below states that one (maybe an e-mail system), if realizing to be expecting new mail, should from now on check the mailbox every 5 minutes (5_m).

$$G_{now}\, expect\_mail \Rightarrow E_{now,5_m}\, check\_mail$$

A-ILTL is particularly well-suited for the agent setting, where temporal aspects matter also from the point of view of when, since when and until when agent properties should hold.

Similar rules, but without intervals, can be expressed in the METATEM logic [28,29,30], whose language is based on classical propositional logic enriched by temporal connectives and on the direct execution of temporal logic statements: in the METATEM approach, the $\Rightarrow$ in a response rule such as $p \Rightarrow q$ is interpreted in an imperative fashion as 'Then Do'. I.e., a response rule is interpreted (similarly to a conditional in traditional imperative programming languages) as 'If p Then Do q'. In METATEM, the antecedent p encodes solely (without loss of generality) properties referring to the past, where the consequent q encodes what the agent should do whenever p holds. Intervals and frequency however add significant expressive power (in the pragmatic sense, not in reference to complexity) in practical agent settings. In fact, these features are important and sometimes crucial in many knowledge representation applications, including deontic defeasible logics (see, e.g., the temporal logic of violations presented in [31]).

2.3 Interval Temporal Logic and Evolutionary Semantics

In this section, we refine our Interval Temporal Logic so as to operate on a sequence of states that corresponds to the Evolutionary Semantics defined before. In fact, states in our case are not simply intended as time instants. Rather, they correspond to stages of the agent evolution, marked with the time when each stage has been reached. Time in this setting is considered to be local to the agent, where with some sort of "internal clock" is able to time-stamp events and state changes. We borrow from [32] the following definition of *timed state sequence*, that we tailor to our setting.

Definition 4. *Let σ be a (finite or infinite) sequence of states, where the ith state e_i, $e_i \geq 0$, is the* semantic snaphots at stage i ε_i^{Ag} *of given agent Ag. Let T be a corresponding sequence of time instants t_i, $t_i \geq 0$. A* timed state sequence for agent Ag *is the couple $\rho_{Ag} = (\sigma, T)$. Let ρ_i be the i-th state, $i \geq 0$, where $\rho_i = \langle e_i, t_i \rangle = \langle \varepsilon_i^{Ag}, t_i \rangle$.*

We in particular consider timed state sequences which are *monotonic*, i.e., if $e_{i+1} \neq e_i$ then $t_{i+1} > t_i$. In our setting, it will always be the case that $e_{i+1} \neq e_i$ as there is no point in semantically considering a static situation: as mentioned, a transition from e_i to e_{i+1} will in fact occur when something happens, externally or internally, that affects the agent.

Then, in the above definition of A-ILTL operators, it would be immediate to let $s_i = \rho_i$. This requires however a refinement: in fact, in our setting, A-ILTL operators are intended to be used within agent programs to define properties to be verified during agent evolution. In this kind of use, when writing Op_m or $Op_{m,n}$ parameters m and n are supposed to define the interval of time in which the property is expected to hold in the program definition: then, m and n will not necessarily coincide with the time instants of the above-defined timed state sequence. To fill this gap, we introduce the following approximation.

Definition 5. *Given positive integer number t, we introduce the following terminology. We indicate with $\check{\rho}_t$ state $\rho = \langle \varepsilon^{Ag}, \check{t} \rangle$ where $\check{t} \geq t$ and $\forall \rho' = \langle \varepsilon^{Ag'}, t' \rangle$ with $t' < \check{t}$ we have $t' < t$. We indicate with $\hat{\rho}_t$ state $\rho = \langle \varepsilon^{Ag}, \hat{t} \rangle$ where $\hat{t} \geq t$ and $\forall \rho'' = \langle \varepsilon^{Ag''}, t'' \rangle$ with $t'' > \hat{t}$ we have $t'' > t$.*

That is, if t represents a time instant not exactly coincident with an element of the time sequence T, $\check{\rho}_t$ is the state whose time component better approximates t "by defect", i.e., the upper bound of all states whose time component is smaller than t. Symmetrically, $\hat{\rho}_t$ is the state whose time component better approximates t "by excess", i.e., the lower bound of all states whose time component is greater than t. Notice that, given A-ILTL expression $Op_{m,n}\, \phi$ state $\check{\rho}_t$ is the first state where ϕ becomes 'observable' in the agent's semantics, i.e., the first state where ϕ is actually required to hold, and $\hat{\rho}_t$ is the last such state. Therefore in the following, by Op_m (resp. $Op_{\langle m \rangle}$) we will implicitly mean $Op_{\check{\rho}_m}$ and by $Op_{m,n}$ (resp. $Op_{\langle m,n \rangle}$) we will implicitly mean $Op_{\check{\rho}_m, \hat{\rho}_n}$.

We need to adapt the interpretation function $\mathcal{I}$ to our setting. In fact, we intend to employ A-ILTL within agent-oriented languages. In particular, we restrict ourselves to logic-based languages for which an evolutionary semantics and a notion of logical consequence can be defined. Thus, given agent-oriented language $\mathcal{L}$ at hand, the set Σ of propositional letters used to define an A-ILTL semantic framework will coincide with all ground[2] expressions of $\mathcal{L}$. Each expression of $\mathcal{L}$ has a (possibly infinite) number of ground versions, obtained by replacing variables with constants (from the alphabet of $\mathcal{L}$) in every possible way. A given agent program can be taken as standing for its (possibly infinite) ground version. This is customarily done in many approaches, such as for instance Answer Set Programming (see e.g., [33] and the references therein). Notice that we have to distinguish between logical consequence in $\mathcal{L}$, that we indicate as $\models_{\mathcal{L}}$, from logical consequence in A-ILTL, indicated above simply as $\models$. However, the correspondence between the two notions can be quite simply stated by specifying that in each state s_i the propositional letters implied by the interpretation function $\mathcal{I}$ correspond to the logical consequences of agent program P_i:

Definition 6. *Let $\mathcal{L}$ be a logic language. Let $Expr_{\mathcal{L}}$ be the set of ground expressions that can be built from the alphabet of $\mathcal{L}$. Let ρ_{Ag} be a timed state sequence for agent Ag,*

[2] An expression is ground if it contains no variables

and let $\rho_i = \langle \varepsilon_i^{Ag}, t_i \rangle$ be the ith state, with $\varepsilon_i^{Ag} = \langle H_i, P_i, M_i \rangle$. An A-ILTL formula τ is defined over sequence ρ_{Ag} if in its interpretation structure $\mathcal{M} = \langle \mathbb{N}, \mathcal{I} \rangle$, index $i \in \mathbb{N}$ refers to ρ_i, which means that $\Sigma = Expr_{\mathcal{L}}$ and $\mathcal{I} : \mathbb{N} \mapsto 2^{\Sigma}$ is defined such that, given $p \in \Sigma$, $p \in \mathcal{I}(i)$ iff $P_i \models_{\mathcal{L}} p$. Such an interpretation structure will be indicated with $\mathcal{M}^{Ag}$. We will thus say that τ holds/does not hold w.r.t. ρ_{Ag}.

In practice, run-time verification of A-ILTL properties may not occur at every state (of the given interval). Rather, sometimes properties need to be verified with a certain frequency, that can even be different for different properties. Then, we have introduced a further extension that consists in defining subsequences of the sequence of all states: if Op is any of the operators introduced in A-ILTL and $k > 1$, Op^k is a semantic variation of Op where the sequence of states ρ_{Ag} of given agent is replaced by the subsequence $s_0, s_{k_1}, s_{k_2}, \ldots$ where for each $k_r, r \geq 1$, $k_r \ mod \ k \ = 0$, i.e., $k_r = g \times k$ for some $g \geq 1$.

A-ILTL formulas to be associated to given agent can be defined within the agent program, though they constitute an additional but separate layer. In fact, their semantics is defined as seen above on the agent evolution ρ_{Ag}. In the next sections we will review and extend previous work on such rules, and we will provide their semantics in terms of the above-defined framework. Rules properly belonging to the agent program P_i will be called *object rules*, and the set of object rules composing P_i can be called *object layer*. In this sense, $\mathcal{L}$ can be called *object language*. The set of A-ILTL formulas associated to given agent, that represent properties which agent evolution should hopefully fulfil, can be called *check layer* and will be composed of formulas $\{\tau_1, \ldots, \tau_l\}$. Agent evolution can be considered to be "satisfactory" if it obeys all these properties.

Definition 7. *Given agent Ag and given a set of A-ILTL expressions $\mathcal{A} = \{\tau_1, \ldots, \tau_l\}$, timed state sequence ρ_{Ag} is* coherent *w.r.t. $\mathcal{A}$ if A-ILTL formula $G\zeta$ with $\zeta = \tau_1 \wedge \ldots \wedge \tau_n$ holds.*

Notice that the expression $G\zeta$ is an *invariance property* in the sense of [34]. In fact, coherence requires this property to hold for the whole agent's "life". In the formulation $G_{m,n}\zeta$ that A-ILTL allows for, one can express *temporally limited coherence*, concerning for instance "critical" parts of an agent's operation. Or also, one might express forms of *partial* coherence concerning only some properties.

An "ideal" agent will have a coherent evolution, whatever its interactions with the environment can be, i.e., whatever sequence of events arrives to the agent from the external "world". However, in practical situations such a favorable case will seldom be the case, unless static verification has been able to ensure total correctness of agent's behavior. Instead, violations will occasionally occur, and actions should be undertaken so as to attempt to regain coherence for the future. Also, some properties will not have to be checked anyway, but only upon occurrence of certain situations. In the following sections, we will introduce two kinds of A-ILTL *rules* (that we also call A-ILTL *expressions*), we will explain their usefulness and provide their semantics.

A-ILTL rules may include asserting and retracting object rules or sets of object rules ("modules"). Thus, below we provide a semantic account of such operations, that is easily defined w.r.t. the Evolutionary Semantics. The need of modifying the agent's knowledge base has been widely discussed with respect to EVOLP [35,36]. It has also

been discussed in [37], about agents learning by "being told" from other trusted agents, by exchanging sets of rules. In the present setting, we will assume to assert/retract ground (sets of) rules. However, in general this assumption can be relaxed as one can resort to *reified* form of rules, where variables are provisionally represented by constants (cf. [38] and the references therein). In this setting, we consider $assert_m(\rho)$ and $retract_m(\rho)$ as special A-ILTL operators, with the following semantics. In particular, a rule asserted at state s_i will be entailed by next state. Symmetrically, after retract a rule will no longer be entailed, with the provision that only existing rules can be retracted.

Definition 8. *Let ρ be a rule expressed in language $\mathcal{L}$ and ρ_{Ag} be a timed state sequence for given agent. The set of A-ILTL operators is enriched by the following.*

$assert_m(\rho)$ *(*assert rule*). $assert_m(\rho)$ holds if ρ belongs to the agent program in the next state. I.e., $\mathcal{M}, i \models assert_m(\rho)$ if $\mathcal{M}, m' \models \rho$, $m' = m + 1$.*

$retract_m(\rho)$ *(*retract rule*). $retract_m(\rho)$ holds if ρ belongs to the agent program in current state, and will no longer belong to it in the next state. I.e., $\mathcal{M}, i \models retract_m(\rho)$ if $\mathcal{M}, m \models \rho$ and $\mathcal{M}, m' \not\models \rho$, $m' = m + 1$.*

3 A-ILTL rules and meta-rules in Agent Programs

There can be different ways of exploiting A-ILTL and in general temporal logic in agent-oriented languages. The METATEM programming language [28,29,30], for instance, is directly based upon the METATEM logic: temporal operators are interpreted as modalities, and semantics is provided accordingly. This semantics is the 'core' of an executable 'imperative' language. In fact, in the authors view rules such as response rules are 'imperative' in the sense that they imply performing actions. We embrace a different position, on the one hand because of the complexity (it is well-known that model-checking for LTL is PSPACE-complete, see [39]) but on the other hand because, as outlined in previous section, we believe that A-ILTL expressions might constitute a check layer to be added to agent programs, whatever the specific formalism in which agents are expressed.

How should A-ILTL expressions be checked? As mentioned, the fact that all expressions associated to an agent program are valid is an invariance property that should hold all along (or at least in specific, "critical" intervals). However, agents evolution is discrete, so A-ILTL expressions can at most be checked at each stage of the evolutionary semantics. Moreover, both the specific expression and the particular application may require checks to be performed at a certain frequency. As seen before, we explicitly associated a frequency with the operator E: in fact, this frequency defines it very nature. However, for the other operators we preferred to just define timed sub-sequences, so as to defer the specification of frequency to the particular implementation setting rather than introducing it into the logic. For instance, the expression (where, following a prolog-like syntax, predicates are indicated with lower-case initial letter and variables in upper-case):

$$G\ within\_range(Temperature)$$

should be clearly checked much more frequently if supervising a critical appliance than if supervising domestic heating. Thus, following the famous statement of [40] that "Algorithm = Logic + Control", we assume to associate to each agent program specific

control information including the frequency for checking A-ILTL operators. Specifically, in the formulation below we associate the frequency to an (optional) additional parameter of each operator.

The representation of A-ILTL operators within a logic agent-oriented programming language can be, e.g., the one illustrated in Table 1, that we have adopted in DALI, where m and n denote the time interval and k is the frequency. We will call this syntax *practical* or also *pragmatic* syntax. When not needed, the argument corresponding to frequency can be omitted. A plain LTL operator OP can be expressed by omitting all arguments. The operator OP on the right column of each line is called *pragmatic* A-ILTL operator and is said to *correspond* to A-ILTL operator Op on the left column.

A-ILTL Op^k	**OP(m,n;k)**
now	$NOW(t)$
${X_m}^k$	$NEXT(m;k)$
F_m^k	$EVENTUALLY(m;k)$
${F_{m,n}}^k$	$EVENTUALLY(m,n;k)$
${G_m}^k$	$ALWAYS(m;k)$
${G_{\langle m \rangle}}^k$	$ALWAYS\_S(m;k)$
${G_{m,n}}^k$	$ALWAYS(m,n;k)$
${G_{\langle m,n \rangle}}^k$	$ALWAYS\_S(m,n;k)$
${N_m^b}^k$	$NEVER\_B(m;k)$
${N_m^a}^k$	$NEVER\_A(m;k)$
${N_{m,n}}^k$	$NEVER(m,n;k)$
$E_{m,k}$	$SOMETIMES(m;k)$
$E_{m,n,k}$	$SOMETIMES(m,n;k)$

Table 1. A-ILTL operators

In the following, we refer to rule-based logic programming languages like DALI, where A-ILTL formulas occur in the agent program of which they constitute the check layer. For simplicity, in this context we restrict φ to be a conjunction of literals. Formulas built out of pragmatic A-ILTL operators with such a restriction on φ are called *pragmatic A-ILTL formulas* (though with some abuse of notation and when clear from the context we will often omit the adjective). In pragmatic A-ILTL formulas, φ must be ground when the formula is checked. However, similarly to negation-as-failure (where the negated atom can contain variables, that must however have been instantiated by literals evaluated previously), we allow variables to occur in an A-ILTL formula, to be instantiated via a *context* χ. From the procedural point of view, χ is required to be evaluated in the first place so as to make the A-ILTL formula ground. Notice that, for the evaluation of φ and χ, we rely upon the procedural semantics of the 'host' language $\mathcal{L}$. For prolog and DALI, (extended) resolution procedures [41] guarantee, with some peculiarities, correctness and, under some conditions, completeness w.r.t. declarative semantics. Below, with some abuse of notation we talk about A-ILTL formulas both in a theoretical and practical sense, in the latter case referring to pragmatic A-ILTL opera-

tors with the above restriction on φ. Whenever discussing A-ILTL formulas (and, later, A-ILTL expressions and rules) we will implicitly refer to timed state sequence ρ_{Ag} for given agent, and by 'state(s)' we mean state(s) belonging to this sequence. In practice, this state sequence will develop along time according to agent's activities, so during agent operation only the prefix of the state sequence developed up to the present time can be "observed" in order to evaluate A-ILTL formulas.

Definition 9. *Let $OP(m, n; k)\varphi$ be a pragmatic A-ILTL formula. The corresponding* contextual A-ILTL formula *has the form $OP(M, N; K)\varphi :: \chi$ where:*

- *M, N and K can be either variables or constants and φ is a conjunction of literals;*
- *χ is called the* evaluation context *of the rule, and consists of a conjunction of literals;*
- *each of the M, N and K which is a variable and each variable occurring in φ* must *occur in an atom (non-negated literal) of χ.*

In the following, a contextual A-ILTL formula will implicitly stand for the ground A-ILTL formula obtained via evaluating the context. We have to establish how to *operationally* check whether such a formula τ holds. In fact, during agent operation one cannot observe the entire state sequence. In all points preceding the interest interval (as defined in previous section) there is no harm in assuming that τ holds. Within the interest interval, τ can be provisionally assumed to hold if the inner formula φ holds in all points up to now. When the crucial state (which is the upper bound of the interest interval) is reached, τ can be definitely established to hold or not.

Definition 10. *Given operator $OP(m, n)$ corresponding to A-ILTL operator $Op_{m,n}$ (resp. operator $OP(m)$ corresponding to Op_m), an A-ILTL formula $OP(m, n)\varphi$ (resp. $OP(m)\varphi$, we disregard frequency here)* operationally holds *w.r.t. state s_i if, given interval of interest $[v, w]$ of $Op_{m,n}$ (resp. Op_m), one of the following conditions hold:*

- *$i < v$;*
- *$i \geq v$ and $i \leq w$, i.e., i is in the interest interval, and φ holds (according to the semantics of Op) in all states of sub-interval $[v, i]$;*
- *$i \geq w$, i.e., i is the crucial state or i is beyond the crucial state, and $Op\varphi$ holds.*

In the next sections, whenever saying that an A-ILTL formula τ holds we implicitly mean (unless differently specified or clear from the context) that τ holds operationally. For uniformity, the above formulas will be called A-ILTL *rules*, though as we will see below they act as *constraints* that are required to be fulfilled, otherwise there is an anomaly in the agent's operation. In what follows we will discuss how to manage such anomalies.

3.1 A-ILTL Rules with Repair and Improvement

There can be the case where an A-ILTL expression, checked at a certain stage of the agent evolution, does not hold (we say that it is *violated*). What to do upon violation? In static checking, the outcome can indicate a violation, and the agent program should be modified so as to remove the anomalous behavior. But, at run-time, no such correction

is possible, and there is in general no user intervention. However, the agent may try to *repair* itself, by self-modifications to its goals and commitments or even to its code, by adding/removing (sets of) rules. Even when, on the contrary, an A-ILTL expression holds, actions may be undertaken as an *improvement*. Take for instance the example of one who wants to lose some weight by a certain date. If (s)he fails, then (s)he should undertake a new diet, with less calories. But if (s)he succeeds before the deadline, then a normocaloric diet should be resumed.

Definition 11. *An A-ILTL rule with a repair/improvement is a rule the form:* $OP(M, N; K)\varphi :: \chi \div \eta \div \xi$*, where:*

- $OP(M, N; K)\varphi :: \chi$ *is a contextual A-ILTL rule, called the* monitoring condition*;*
- η *is called the* repair action *of the rule, and it consists of an atom* η*;*
- ξ *(optional) is called the* improvement action *of the rule, and it consists of an atom* η*.*

Whenever the monitoring condition $OP(M, N; K)$ of an A-ILTL rule is violated, the repair action η is attempted. The repair action is specified via an atom that is 'executed' in the sense that it gives way to an inference process as provided by host language $\mathcal{L}$ (in the prolog terminology, that here we adopt, the atom is a 'goal'). If instead the monitoring condition succeeds, in the sense that the specified interval is expired and the A-ILTL formula holds or, in case of the operator 'eventually', if φ holds within given interval, then the improvement action, if specified, can be 'executed'.

The above-mentioned example can be formalized as follows:

$$\begin{aligned} &EVENTUALLY(May{-}15{-}2012, June{-}10{-}2012)\, lose\_five\_kilograms \\ &\quad \div new\_stricter\_diet(June{-}10{-}2012, June{-}30{-}2012) \\ &\quad \div resume\_normal\_diet \end{aligned}$$

An A-ILTL rule with improvement/repair should of course give way to specified actions whenever the involved A-ILTL formula can be deemed to hold/not to hold i.e., in our terminology, as soon as its 'critical state' is reached. Formally:

Definition 12. *Let* ρ_{Ag} *be a timed state sequence for agent* Ag *and let* $\alpha = \tau \div \eta \div \xi$ *be an A-ILTL rule with repair/improvement occurring in* Ag*'s agent program, where* τ *is a contextual A-ILTL formula.* α *is* fulfilled *in* ρ *if, given crucial state* ρ_k *for* Op*, one of the following conditions hold: (i)* τ *does not hold, and* $\rho_{k+1} \models \eta$*; (ii)* τ *holds, and* $\rho_{k+1} \models \xi$*.*

3.2 Evolutionary A-ILTL Expressions

It can be useful in many applications to define properties to be checked upon arrival of partially known sequences of events. In general in fact, it is not possible to fully establish in advance which events will arrive and in which order. Moreover, restricting the agent "perception" only to known events or to an expected order heavily limits the ability of the agent to improve its behavior in time, e.g. via forms of learning. This is

our motivation for introducing a new kind of A-ILTL rules, that we call Evolutionary A-ILTL Expressions (first introduced in a preliminary form in [42,43]).

These expressions are based upon specifying: (i) a sequence of past events that may have happened; (ii) an A-ILTL formula defining a property that should hold; (iii) a sequence of events that might happen in the future, without affecting the property; (iv) a sequence of events that are supposed *not* to happen in the future, otherwise the property will not hold any longer; (v) optionally, "repair" actions to be undertaken if the property is violated.

To be able to indicate in a flexible way sequences of events of any (unlimited) length we admit a syntax inspired to regular expressions [44].

Definition 13. *If E is an event, E^* will indicate zero or more occurrences of E, and E^+ one or more occurrences. Given events E_1 and E_2, by $E_1, E2$ we mean that they may occur in any order; by $E_1 \bullet\bullet E_2$ we mean that E_1 must occur before E_2 (with possibly a sequence of unspecified events in between); by $E_1 \bullet E_2$ we mean that E_1 must occur immediately before E_2 (i.e., the two events must be consecutive). Wild-card X, standing for unspecified event, can be used* [3]*. Given set of events $\mathcal{E}v = \{E_1, \ldots, E_k\}$, $k \geq 0$, let* an event sequence for (or corresponding to) *$\mathcal{E}v$, indicated with $\mathcal{S}^{\mathcal{E}v}$, be a sequence defined in the above way on events in $\mathcal{E}v$. Event list $Z_1, \ldots, Z_r$, $r > 0$,* satisfies *$\mathcal{S}^{\mathcal{E}v}$ if all the Z_is occur in $\mathcal{S}^{\mathcal{E}v}$ following the specified order.*

For instance, $E_1^+ \bullet\bullet E_2, E_3 \bullet X \bullet E_4$ means that, after a certain (non-zero) number of occurrences of E_1 and, possibly, of some unknown event, E_2 and E_3 can occur in any order. They are followed by one unknown event X and, immediately afterwards, by E_4. List E_1, E_3 satisfies the above sequence, as both events occur in it in given order, while list E_3, E_1 does not, as the order is not correct.

Definition 14 (Evolutionary LTL Expressions). *Let $\mathcal{E}vp = \{E_{P_1}, \ldots, E_{P_l}\}$, $l > 1$, be a set of past events, and $\mathcal{F} = \{F_1, \ldots, F_m\}$, $\mathcal{J} = \{J_1, \ldots, J_r\}$, $m, r \geq 0$, be sets of events. Let $\mathcal{S}^{\mathcal{E}vp}$, $\mathcal{S}^{\mathcal{F}}$ and $\mathcal{J}^{\mathcal{J}}$ be corresponding event sequences. Let τ be a contextual A-ILTL formula $Op\,\varphi :: \chi$. An* Evolutionary LTL Expression *ϖ is of the form $\mathcal{S}^{\mathcal{E}vp} : \tau ::: \mathcal{S}^{\mathcal{F}} :::: \mathcal{J}^{\mathcal{J}}$ where:*

- *$\mathcal{S}^{\mathcal{E}vp}$ denote the sequence of relevant events which are supposed to have happened, and in which order, for the rule to be checked; i.e., these events act as preconditions: whenever one or more of them happen in given order, τ will be checked;*
- *$\mathcal{S}^{\mathcal{F}}$ denote the events that are expected to happen in the future without affecting τ;*
- *$\mathcal{J}^{\mathcal{J}}$ denote the events that are expected* not *to happen in the future; i.e., whenever any of them should happen, ϖ is not required to hold any longer, i.e., these can be called are "breaking events".*

[3] Notice that, for an agent, an event "occurs" when the agent perceives it. This is only partially related to when events actually happen in the environment where the agent is situated. In fact, the order of perceptions can be influenced by many factors. However, either events are somehow time-stamped externally (by a reliable third-party) whenever they happen, so as to certify the exact time of their origin (as sometimes it may be the case), or an agent must rely on its own subjective experience.

The state until which ϖ is required to hold is the critical state of the operator Op occurring in τ provided that if one of the J_i's happens at intermediate state s_w, then ϖ is not required to hold after s_w.

Notice that both the F_i's and the J_i's are optional, and that we do not require the E_{P_i}'s, the F_i's and the J_i's to be ground terms: variables occurring in them indicate values in which we are not interested.

An Evolutionary LTL Expression can be evaluated w.r.t. a state s_i, which contains (in the component ε_i^{Ag}) the history H_i of the agent, i.e., the list of past events: in fact, within an agent, an event has happened if it occurs as a past event in H_i. The expression holds, also in presence of expected or breaking events, if the inner contextual A-ILTL formula τ holds, or if a breaking event has occurred (as in this case τ is no longer required to hold). Notice that H_i satisfies each of the event sequences in the definition of an A-ILTL Expression ϖ provided that H_i includes zero or more elements of the sequence, the specified order. Formally:

Definition 15. *An Evolutionary A-ILTL Expression ϖ, of the form specified in Definition 14,* holds *in state s_i whenever (i) H_i satisfies $\mathcal{S}^{\mathcal{E}vp}$ and $\mathcal{S}^{\mathcal{F}}$, but not $\mathcal{J}^{\mathcal{J}}$, and τ holds or (ii) H_i satisfies $\mathcal{J}^{\mathcal{J}}$.*

Definition 16. *An Evolutionary A-ILTL Expression ϖ, of the form specified in Definition 14, is* violated *in state s_i whenever H_i satisfies $\mathcal{S}^{\mathcal{E}vp}$ and $\mathcal{S}^{\mathcal{F}}$, but not $\mathcal{J}^{\mathcal{J}}$, and τ does not hold.*

Definition 17. *An Evolutionary A-ILTL expression ϖ, of the form specified in Definition 14, is* broken *in state s_i whenever H_i satisfies $\mathcal{S}^{\mathcal{E}vp}$, $\mathcal{S}^{\mathcal{F}}$ and $\mathcal{J}^{\mathcal{J}}$, and τ does not hold.*

Operationally, an Evolutionary A-ILTL Expression can be finally deemed to hold if either the critical state has been reached and τ holds, or an unwanted event (one of the J_is) has occurred. Instead, an expression can be deemed *not* to hold (or, as we say, to be *violated* as far as it expresses a wished-for property) whenever τ is false at some point without breaking events.

The following is an example of Evolutionary A-ILTL Expression stating that, after a car has been submitted to a checkup, it is assumed to work properly for (at least) six months, even in case of (repeated) long trips, unless an accident occurs.

$$\begin{aligned} &checkup_P(Car) : T \ : \ ALWAYS(T, T + 6_{months}) \ work\_ok(Car) \\ &\quad ::: long\_trip^{+}(Car) \\ &\quad :::: accident(Car) \end{aligned}$$

As said before, whenever an unwanted event (one of the J_is) should happen, ϖ is not required to hold any longer (though it might). The proposition below formally allows for dynamic run-time checking of Evolutionary A-ILTL Expressions. In fact, it states that, if a given expression holds in a certain state and is supposed to keep holding after some expected events have happened, then checking this expression amounts to checking the modified expression where: (i) the occurred events are removed from event sequences, and (ii) subsequent events are still expected.

Proposition 1. *Given Evolutionary A-ILTL Expression of the form specified in Definition 14, assume that ϖ holds at state s_n and that it still holds after the occurrence of event $E \in \mathcal{E}vp$ and (possibly) of event $F \in \mathcal{F}$ at state s_v ($v \geq n$), and that none of the events in $\mathcal{J}$ has happened. Let $\mathcal{S}_1^{\mathcal{E}vp}$ and $\mathcal{S}_1^{\mathcal{F}}$ be modified event sequences obtained by respectively canceling E and F from $\mathcal{S}^{\mathcal{E}vp}$ and $\mathcal{S}_1^{\mathcal{F}}$ whenever they occur. Given $\varpi_1 = \mathcal{S}_1^{\mathcal{E}vp} : \tau ::: \mathcal{S}_1^{\mathcal{F}} :::: \mathcal{J}^{\mathcal{J}}$ we have that for every state s_w with ($w \geq v$) ϖ holds iff ϖ_1 holds.*

Whenever an Evolutionary A-ILTL expression is either violated or broken, a repair can be attempted with the aim of recovering the agent's state.

Definition 18. *An evolutionary LTL expression with repair ϖ^r is of the form:*

$$\varpi|\eta_1||\eta_2$$

where ϖ is an Evolutionary LTL Expression adopted in language $\mathcal{L}$, and η_1, η_2 are atoms of $\mathcal{L}$. η_1 will be executed (according to $\mathcal{L}$'s procedural semantics) whenever ϖ is violated, and η_2 will be executed whenever ϖ is broken.

4 Related Work

We may easily notice the similarity between Evolutionary A-ILTL Expressions and event-calculus formulations. The Event Calculus has been proposed by Kowalski and Sergot [45] as a system for reasoning about time and actions in the framework of Logic Programming. The essential idea is to have terms, called *fluents,* which are names of time-dependent relations. Kowalski and Sergot write $holds(r(x, y), t)$ which is understood as "fluent $r(x, y)$ is true at time t". Take for instance the default inertia law, stating when fluent f holds, formulated in the event calculus as follows:

$$holds(f, t) \leftarrow happens(e),\ initiates(e, f),\ date(e, t_s),\ t_s < t,\ not\, clipped(t_s, f, t)$$

The analogy consists in the fact that, in the sample A-ILTL expression of previous section, past event $checkup_P(Car) : t_1$ initiates a fluent which is actually an interval A-ILTL expression, namely $G_{t_1,t1+6months} work\_ok(Car)$, which would be "clipped" by $accident(Car)$, where a fluent which is clipped does not hold any longer. The Evolutionary A-ILTL Expression contains an element which constitutes an addition w.r.t. the event calculus formulation: in fact, $long\_trip^{+}(Car)$ represents a sequence of events that is expected, but by which the fluent should *not* be clipped if everything works as expected. Moreover, in Evolutionary A-ILTL Expressions one can specify a fluent to initiate and keep holding or terminate according not just to single events, but to complex event sequences of unlimited length.

Static verification of agent programs and systems (i.e., verification performed prior to agent activation) can be accomplished through model-checking techniques [3], abstract interpretation [4] (not commented here) or theorem proving.

About theorem proving, in [46] for instance, concerning the agent-oriented language GOAL, a temporal logic is defined to prove properties of GOAL agents. In general,

given a logical specification of an agent and its semantics, properties of that agent can be proved as theorems.

Model-checking is a method for algorithmically checking whether a program (intended as the "model" of a system) satisfies a specification, usually expressed by means of some kind of temporal logic. In mathematical terms, the method tries to decide if model M (expressed in some formal language), with initial state s, models a property p. Otherwise, a counterexample is usually generated. This ia done by exploring all possible state that given system can possibly reach. Model-checking techniques [3] have been originally adopted for testing hardware devices, their application to software systems and protocols is constantly growing [47,48], and there have been a number of attempts to overcome some known limitations of this approach.

The application of such techniques to the verification of agents is still limited by two fundamental problems. The first problem arises from the marked differences between the languages used for the definition of agents and those needed by verifiers (usually ad-hoc, tool-specific languages). Indeed, to apply static verification, currently an agent has to be remodeled in another language: this task is usually performed manually, thus it requires an advanced expertise and gives no guarantee on the correctness and coherence of the new model. In many cases (e.g., [48,49]) current research in this field is still focused on the problem of defining a suitable language that can be used to easily and/or automatically reformulate an agent in order to verify it through general model-checking algorithms. For example, [50] describes a technique to model-check agents defined by means of a subset of the AgentSpeak language, which can be automatically translated into PROMELA and Java and then verified by the model-checkers SPIN [47] and Java PathFinder [51], respectively, against a set of constraints which, in turn, are translated into LTL from a source language which is a simplified version of the BDI logic. [52] describes an approach that exploits bounded symbolic model-checking, in particular the tool MCMAS, to check agents and MAS (Multi-Agent Systems) against formulas expressed in the CTLK temporal logic.

The second obstacle is represented by the dynamic nature of agents, which are able to learn and self-modify over their life cycle, and by the extreme variability of the environment in which agents move. These aspects make it difficult to model agents via finite-state languages, which are typical of many model-checkers, and dramatically increase the resources (time, space) required for their verification (state explosion). This can be seen as a motivation for our approach, which defers at least part of the verification activity (namely, the part more dependent upon agent evolution) to run-time.

The literature reports fully-implemented promising verification frameworks (e.g., [50,52,53]), of which SCIFF [53,54,55] is not based upon model-checking. SCIFF is an abductive proof procedure inspired by the IFF proof procedure [56] by Fung and Kowalski. Unlike original IFF, SCIFF focusses on the externally observable agent behavior. so as to focalize on the interaction. Agents could be computational entities, reactive systems, peers in a distributed computer system, even human actors. This kind of detachment from specific features of an observed system is called "social approach" to agent interaction. Given a narrative of such an observed behavior (called a "history") the purpose of the SCIFF framework is (i) to define declaratively whether such a history

is "admissible", i.e., compliant to a specification, and (ii) to provide a computational proof-procedure to decide operationally about its compliance.

In the MAS domain, the SCIFF language has been used to define agent interaction protocols and, more generally, to describe the generation of expectations in the form of events, or "social goals", that express the social aim or outcome of some agent interaction. SCIFF allows one to model dynamically upcoming events, and specify positive and negative expectations, and the concepts of fulfilment and violation of expectations. SCIFF has a declarative semantics given in terms of Abductive Logic Programming, and is fully implemented. The implementation enjoys important properties, namely termination, soundness, and completeness w.r.t. the declarative semantics.

Reactive Event Calculus (REC) stems from SCIFF [57,58,59] and exploits the idea that, every time a new event (or set of events) is delivered to an agent, it must react by extending the narrative and by consequently extending and revising previously computed results. REC axiomatization can be based on Abductive Logic Programming (ALP), or alternatively on a lightweight form of Cached Event Calculus (CEC) [60], that exploits assert and retract predicates to cache and revise the maximal validity intervals of fluents. This latter semantics is suitable to deal with application domains where events are processed following the order in which they have been generated, like in business processes and (web) services.

Another example of social approach is the one based on commitments, firstly introduced by Singh in [61]. Commitments result from communicative actions, and capture the mutual obligations established between the interacting agents during the execution. [62] represents commitments as properties in the event calculus, and develop a scheme where to model the creation and manipulation of commitments as a result of performing actions.

All the above-mentioned approaches have relationships with the one presented in this paper. For instance, positive and negative expectations in SCIFF are similar to the Fs and Js in Evolutionary A-ILTL Expressions. However, our approach has its specific original features. The main one is that it is aimed at a single agent dynamically verifying itself under various respects, and not to the verification of interactions. Our focus is not on observable behavior to be confronted with expectations: rather, A-ILTL rules are aimed at expressing inherent agent properties. We drew inspiration from Cohen and Levesque work on rational agency (see, e.g., [63,64]). Let us consider one of their examples from [64], namely "I always want more money than I have": the following is a variation expressed as an Evolutionary A-ILTL Expression, where one always wants within one month ten percent more money than (s)he has at present. $have\_money_P(S)$: T is a past event that represents the last agent's reminiscence about how much money (s)he has (in fact, when time-stamp of a past event is a variable, the last version is obtained). The expression states that by time $T1$, which is T plus one month, (s)he intends to own a sum $S1$ greater by ten percent than S.

$have\_money_P(S):T$:
$EVENTUALLY(T1)\ have\_money(S1)\ ::\ S1 = S + 10\%S, T1 = T + 1_{month}$

In other approaches, for instance SCIFF or Reactive Event Calculus, one might to some extent specify properties similar to A-ILTL ones: this however would be easily

done only within agents defined in the related formalisms (respectively, abductive logic programming and event-calculus), and mechanisms for dynamically checking properties remain to be defined: in fact, these approaches can be adapted to dynamic checking when performed by a third party, where they have not been devised for self-checking. Moreover, the problem of how to apply these methodologies within other agent-oriented languages and frameworks has still not been considered. Our approach introduces a flexibility with respect to a pre-defined narrative, and the concept of repair and improvement is novel.

The approach of [6] introduces an *active logic* based on time, where they explicitly have inference rules such as:

$$\frac{i : \quad now(i)}{i+1 : now(i+1)}$$

to denote that the concept of "present time" evolves as time passes. They also have rules for 'timed' modus ponens (where the conclusion becomes true at next instant) and a frame axiom. Our A-ILTL logic has a primitive 'current state' operator that is represented as $NOW(t)$ in the practical syntax, and can occur in A-ILTL rules and expressions, where it is re-evaluated with the whole rule/expression at given frequency.

In addition to 'base level' systems providing fast reaction, in [6] a component supporting deliberation and re-consideration, capable of symbolic reasoning and meta-reasoning for self-monitoring and self-correction of the overall system, is advocated. They argue that a flexible, non-brittle system can be achieved by adding such a layer, where this oversight module executes a "Meta-Cognitive loop" (MCL). Active logic is in their view a suitable tool for designing MCL, that however should be kept simple and fast. In a way, our A-ILTL rules and expressions may be seen as composing a sort of MCL, where A-ILTL can be seen as the underlying active logic. Then, our approach (especially when applied in the DALI context) could be seen to some extent as an instance of theirs, with some differences. On the one hand, at the base level they envisage not necessarily symbolic reasoning modules, but also other kinds of non-symbolic or purely reactive systems. This of course is a design choice which is by no means incompatible with A-ILTL. On the other hand, reasoning in time is possible in DALI also at the base level. In fact, their example of making a date and meeting somewhere at a certain time might be fully represented by DALI "internal events", which provide rules to be re-evaluated at a certain frequency ('Is it time to go?') with a reaction that occurs when rule conditions are fulfilled ('If so, then go!').

5 A Complete Example

In this section we propose and discuss a small but complete example of use of the various features that we have introduced so far for agent's dynamic self-checking. We follow DALI syntax, that we have partly introduced before and will further explain here. DALI syntax extends prolog syntax, that we suppose to be known to the reader (cf. e.g., [65,41]. The example concerns in particular an agent managing a bank cash machine.

The machine is supposed to be situated in a room. Customers enter and exit the room via a door. A sensor on the door delivers to the agent the two (external) events

$enter\_customer_E$ and $exit\_customer_E$ when a customer respectively enters or exits the room. Each external event is managed by a *reactive rule*, where the traditional prolog connective :- is replaced by new connective :>. This new connective indicates that whenever the event in the head is received (we can also say "perceived") by the agent, then the body is executed. The agent reacts to a customer entering or exiting by switching on and off the light. Atoms with postfix A indicate in fact actions. The two actions $switch\_on\_light_A$ and $switch\_off\_light_A$ for simplicity are supposed here non to have preconditions and to be always successful. After reaction, each external event is automatically recorded as a *past event*, time-stamped with the time of perception/reaction. So, e.g., past event $enter\_customer_P : T$ can be found in the agent's knowledge base whenever external event $enter\_customer_E$ has been perceived at time T (by convention, we take time of perception to coincide with time of reaction). Whenever such an event is perceived several times, time-stamp T refers to the last perception (though there is a management of versions, as illustrated in [9]). Whenever a customer enters the room, the agent expects the customer to exit within a reasonable time, say 5 minutes. Otherwise, something irregular may have happened, so it is better to alert a human operator. All the above is formalized via the following program fragment. The two reactive rules manage external events. The subsequent A-ILTL rule states that, if a customer does not follow the expected behavior, then (as a repair for the violation) the action $alert\_operator_A$ is executed. Precisely, the rule context (after the ::) specifies that, if a customer entered at time T (as recorded by past event $enter\_customer_P : T$), the limit time is set to $T1 = T + 5_m$. At most by this time, the user must $EVENTUALLY$ have gone out: i.e., past event $exit\_customer_P : T2$ must be found in the agent's knowledge base, recording the user's action of exiting at a certain time in given interval (in fact, in the context it is stated that $T2 > T$, $T2 \leq T1$). The frequency at which this rule is checked is explicitly set at 30 seconds.

$$
\begin{array}{l}
enter\_customer_E :> switch\_on\_light_A. \\
exit\_customer_E :> switch\_off\_light_A. \\
EVENTUALLY(T, T1, 30_s)\ exit\_customer_P : T2, :: \\
\qquad enter\_customer_P : T, T1 = T + 5_m, \\
\qquad T2 > T, T2 \leq T1 \\
\qquad \div\ alert\_operator_A.
\end{array}
$$

When in the room, a customer will presumably attempt to withdraw some money from her/his account. The customer inserting her/his data in the machine will result in delivering to the agent an external event such as $withdraw(Customer, Sum)_E$, where $Customer$ will be instantiated to the customer's code, and Sum to the amount of money that (s)he wants. The reactive rule reported below does the following: (i) finds the account number $Account$ related to the customer (fails if it does not exist); (ii) checks whether the customer is trustworthy (fails if withdrawal is for some security reason prevented); (iii) checks whether the amount is available on the account, and is within the daily and weakly maximum that can be provided; (iv) updates the customer account by subtracting the sum; (v) performs an action that will actually result in providing the money. Notice that checking trust is intended to be an action.

$$
\begin{aligned}
&withdraw(Customer, Sum)_E :> \\
&\qquad\qquad find\_account(Customer, Account), \\
&\qquad\qquad check\_trust_A(Customer), \\
&\qquad\qquad check\_sum(Sum), \\
&\qquad\qquad update(Account, Sum), \\
&\qquad\qquad give\_money_A(Account, Sum)
\end{aligned}
$$

The two actions occurring in the above reactive rule have preconditions. Preconditions to actions are expressed via rules where the new connective $:<$ appears. This means that the action in the head is enabled (via this rule) to be performed only if the body succeeds. Actually performing an action then implies the agent being connected to its environment by practical *actuator* devices. The atom in brackets which (optionally) occurs in such a rule indicates an alternative action to be performed if the body fails. The first rule states that money can be provided to the customer only if the corresponding amount is actually present in the cash machine (the available balance B, that for allowing the withdrawal must be greater than or equal to required sum, is recorded in the agent knowledge base by fact $machine\_content(B)$). If not, backup action $PrintErrorMsgIW_A$ will print an error message on the screen of the cash machine indicating that withdrawal is at the moment impossible. Before enabling actual withdrawal, balance B must be updated (by $update\_machine\_content(B, B1)$) by subtracting Sum. The second rule takes trust checking seriously: in fact, if the level of trust associated to a customer is less than a threshold, then the agent alerts a human operator. In practice, if trust is sufficiently high then action $check\_trust_A(Customer)$ simply succeeds, otherwise the backup action will be executed.

$$
\begin{aligned}
&give\_money_A(Sum) :< \\
&\qquad\qquad machine\_content(B), B \geq Sum, \\
&\qquad\qquad update\_machine\_content(B, B1), B1 = B - Sum \\
&\qquad\qquad \{PrintErrorMsgIW_A\}. \\
&check\_trust_A(Customer) :< \\
&\qquad\qquad trust(Customer, L), L > Threshold \\
&\qquad\qquad \{Alert\_Operator_A\}.
\end{aligned}
$$

Finally, the agent is responsible of providing the cash machine with money to be given to customers. The agent fills the machine with a standard quantity Q of money. This quantity is recorded in the agent's knowledge base by fact $standard\_quantity(Q)$. The filling is performed in practice by an action such as $fill\_machine_A(Q)$, executed at time T. Each action, similarly to external events, after execution is recorded in the agent's knowledge base as a past event: in this case, a past event will be added of the form $fill\_machine_P(Q) : T$. After filling the machine, the agent expects the money to be sufficient for some time, say 8 hours.

The following Evolutionary A-ILTL expression states that, after the action of filling the machine, for eight hours the machine should not get empty. Precisely, the time interval considered in the expression is $[T, T1]$ where, as specified in the evaluation context of the rule, $T1 = T + 8_{hours}$. Within this interval, the machine content (recorded

in the agent's knowledge base by fact ($machine\_content(B)$) must be $ALWAYS$, i.e., in all time instants of the interval, greater than a $minimum$ amount. This obviously also in case (as stated after the :::) of repeated withdrawals (whatever their number) though, as seen before, each withdrawal makes the machine content B decrease. In case of violation, i.e., in case the machine gets empty or almost (content less than a $minimum$ amount), the agent (as stated after the |) fills again the machine with standard quantity Q of money. This again by means of action $\mathit{fill\_machine}_A(Q)$, that will in turn become a past event with a new time-stamp corresponding to its completion. The agent however will also reconsider the standard quantity, as Q has proven to be insufficient, possibly updating it to a (presumably larger) new amount $Q1$. All this unless (as specified after the ::::) an exceptional condition, in this case robbery, occurs. If so, the repair action executed is (as specified after the ||) to call the police. Notice that an A-ILTL expression is checked at a certain frequency (in this case the default one). So, it can be the case that the condition is violated but the violation has not been detected yet. This case however is managed by rules coping with withdrawal: in particular, if money cannot be provided because it is insufficient, as seen before an error message will be displayed.

$$
\begin{array}{l}
\mathit{fill\_machine}_P(Q) : T \ : \ ALWAYS(T, T1) \\
\qquad (machine\_content(B), B > minimum) \ :: \ T1 = T + 8_{hours} \\
\qquad\qquad ::: \ withdraw(A, S)_A{}^+ \ | \\
\qquad\qquad\qquad\qquad \mathit{standard\_quantity}(Q), \mathit{fill\_machine}_A(Q), \\
\qquad\qquad\qquad\qquad \mathit{reconsider\_quantity}(Q, Q1) \\
\qquad\qquad :::: \ robbery \ || \ \mathit{call\_police}_A .
\end{array}
$$

6 Concluding Remarks

In this paper, we have presented a comprehensive framework for defining agent properties and run-time self-checking of such properties. Our methodology is intended as complementary to static verification techniques. To this aim, we have introduced A-ILTL rules and expressions, that allow one to define a number of useful properties that the evolution of an agent should fulfil, whatever the sequence of in-coming events. We have provided a complete semantic framework, adaptable to several practical settings. The approach is significantly different from related work, to which it could to some extent be usefully integrated. The approach has been prototypically implemented and experimented [43] in the context of the DALI language.

Acknowledgement

My mentor Gaetano Aurelio Lanzarone ('Elio' for his friends) died at the age of 66 in October 2011 after a long illness. He has been one of the pioneers of prolog and rule-based automated reasoning in Italy, and initiated me into this fascinating research area. My affection and gratitude to Elio will last forever. This paper is dedicated to him. I gratefully acknowledge past joint work and a long-lasting friendship and cooperation on the subjects discussed in the present paper with Pierangelo Dell'Acqua, Luís Moniz Pereira, Arianna Tocchio and Francesca Toni.

References

1. Fisher, M., Bordini, R.H., Hirsch, B., Torroni, P.: Computational logics and agents: a road map of current technologies and future trends. Computational Intelligence Journal **23**(1) (2007) 61–91
2. Wooldridge, M., Dunne, P.E.: The computational complexity of agent verification. In: Intelligent Agents VIII, 8th International Workshop, ATAL 2001 Seattle, WA, USA, August 1-3, 2001, Revised Papers. Volume 2333 of Lecture Notes in Computer Science., Springer (2002)
3. Clarke, E.M., Lerda, F.: Model checking: Software and beyond. Journal of Universal Computer Science **13**(5) (2007) 639–649
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, ACM Press, New York, NY (1977) 238–252
5. Brachman, R.J.: (AA)AI more than the sum of its parts. AI Magazine **27**(4) (2006) 19–34
6. Anderson, M.L., Perlis, D.: Logic, self-awareness and self-improvement: the metacognitive loop and the problem of brittleness. J. Log. Comput. **15**(1) (2005) 21–40
7. Anderson, M.L., Fults, S., Josyula, D.P., Oates, T., Perlis, D., Wilson, S., Wright, D.: A self-help guide for autonomous systems. AI Magazine **29**(2) (2008) 67–73
8. Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In Baldoni, M., Endriss, U., Omicini, A., Torroni, P., eds.: Declarative Agent Languages and Technologies III, Third International Workshop, DALT 2005, Selected and Revised Papers. Volume 3904 of LNAI. Springer (2006) 106–123
9. Costantini, S.: Defining and maintaining agent's experience in logical agents. In: Informal Proc. of the LPMAS (Logic Programming for Multi-Agent Systems) Workshop at ICLP 2011, and CORR Proceedings of LANMR 2011, Latin-American Conference on Non-Monotonic Reasoning. (2011)
10. Costantini, S., Dell'Acqua, P., Pereira, L.M.: A multi-layer framework for evolving and learning agents. In M. T. Cox, A.R., ed.: Proceedings of Metareasoning: Thinking about thinking workshop at AAAI 2008, Chicago, USA. (2008)
11. Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf.,JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002)
12. Costantini, S., Tocchio, A.: The DALI logic programming agent-oriented language. In: Logics in Artificial Intelligence, Proc. of the 9th European Conference, Jelia 2004. LNAI 3229, Springer-Verlag, Berlin (2004)
13. Costantini, S.: The DALI agent-oriented logic programming language: References (2012) at URL http://www.di.univaq.it/stefcost/info.htm.
14. Costantini, S., D'Alessandro, S., Lanti, D., Tocchio, A.: Dali web site, download of the interpreter (2010) http://www.di.univaq.it/stefcost/Sito-Web-DALI/WEB-DALI/index.php, With the contribution of many undergraduate and graduate students of Computer Science, L'Aquila. For beta-test versions of the interpreter (latest advancements) please ask the authors.
15. Castaldi, M., Costantini, S., Gentile, S., Tocchio, A.: A logic-based infrastructure for reconfiguring applications. In Leite, J.A., Omicini, A., Sterling, L., Torroni, P., eds.: Declarative Agent Languages and Technologies, First International Workshop, DALT 2003, Revised Selected and Invited Papers. Volume 2990 of LNAI., Springer (2004) Hot Topics Sub-series.
16. Costantini, S., Mostarda, L., Tocchio, A., Tsintza, P.: Agents and security in a cultural assets transport scenario. In: Agents and Industry: Technological Applications of Software Agents, Proc. of WOA'07

17. Costantini, S., Mostarda, L., Tocchio, A., Tsintza, P.: Dalica agents applied to a cultural heritage scenario. IEEE Intelligent Systems, Special Issue on Ambient Intelligence **23**(8) (2008)
18. Bevar, V., Costantini, S., Tocchio, A., Gasperis, G.D.: A multi-agent system for industrial fault detection and repair. In Demazeau, Y., Müller, J.P., Rodríguez, J.M.C., Pérez, J.B., eds.: Advances on Practical Applications of Agents and Multi-Agent Systems - Proc. of PAAMS 2012. Volume 155 of Advances in Soft Computing., Springer (2012) 47–55 Related Demo paper "Demonstrator of a Multi-Agent System for Industrial Fault Detection and Repair", pages 237-240 of same volume.
19. SOAR-Research-Group: SOAR: A comparison with rule-based systems (2010) URL: http://sitemaker.umich.edu/soar/home.
20. Pnueli, A.: The temporal logic of programs. In: Proc. of FOCS, 18th Annual Symposium on Foundations of Computer Science, IEEE (1977) 46–57
21. Emerson, E.A.: Temporal and modal logic. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, vol. B. MIT Press (1990)
22. Costantini, S., Dell'Acqua, P., Pereira, L.M., Tsintza, P.: Runtime verification of agent properties. In: Proc. of the Int. Conf. on Applications of Declarative Programming and Knowledge Management (INAP09). (2009)
23. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems **2**(4) (1990) 255–299
24. Alur, R., Henzinger, T.A.: Logics and models of real time: A survey. In de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G., eds.: Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings. Volume 600 of Lecture Notes in Computer Science., Springer (1992) 74–106
25. Hirshfeld, Y., Rabinovich, A.M.: Logics for real time: Decidability and complexity. Fundam. Inform. **62**(1) (2004) 1–28
26. Goranko, V., Montanari, A., Sciavicco, G.: A road map of interval temporal logics and duration calculi. Journal of Applied Non-Classical Logics **14**(1-2) (2004) 9–54
27. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: Response. In: Time for Verification, Essays in Memory of Amir Pnueli. Volume 6200 of Lecture Notes in Computer Science., Springer (2010) 279–361
28. Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R.: MetateM: A framework for programming in temporal logic. In: Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. LNCS 430, Springer-Verlag (1989)
29. Fisher, M.: MetateM: The story so far. In Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: PROMAS. LNCS 3862, Springer (2005) 3–22
30. Barringer, H., Fisher, M., Gabbay, D.M., Gough, G., Owens, R.: MetateM: An introduction. Formal Asp. Comput. **7**(5) (1995) 533–549
31. Governatori, G., Rotolo, A.: Justice delayed is justice denied: Logics for a temporal account of reparations and legal compliance. In: CLIMA. Volume 6814 of Lecture Notes in Computer Science., Springer (2011) 364–382
32. Henzinger, T.A., Manna, Z., Pnueli, A.: Timed transition systems. In de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G., eds.: Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings. Volume 600 of Lecture Notes in Computer Science., Springer (1992) 226–251
33. Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation, Chapter 7. Elsevier (2007)
34. Manna, Z., Pnueli, A.: Adequate proof principles for invariance and liveness properties of concurrent programs. Sci. Comput. Program. **4**(3) (1984) 257–289

35. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002) 50–61
36. J.Alferes, J., Brogi, A., Leite, J.A., Pereira, L.M.: An evolvable rule-based e-mail agent. In: Procs. of the 11th Portuguese Intl.Conf. on Artificial Intelligence (EPIA'03). LNAI 2902, Springer-Verlag, Berlin (2003) 394–408
37. Costantini, S., Dell'Acqua, P., Pereira, L.M.: Conditional learning of rules and plans by knowledge exchange in logical agents. In: Proc. of RuleML 2011 at IJCAI. (2011)
38. Barklund, J., Dell'Acqua, P., Costantini, S., Lanzarone, G.A.: Reflection principles in computational logic. J. of Logic and Computation **10**(6) (2000) 743–786
39. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3) (1985) 733–749
40. Kowalski, R.A.: Algorithm = logic + control. Commun. ACM **22**(7) (1979) 424–436
41. Lloyd, J.W.: Foundations of Logic Programming. Springer (1987)
42. Costantini, S., Dell'Acqua, P., Pereira, L.M., Tocchio, A.: Ensuring agent properties under arbitrary sequences of incoming events. In: Proc. of 17th RCRA Intl. Worksh. on Experimental evaluation of algorithms for solving problems with combinatorial explosion. (2010)
43. Costantini, S., Tsintza, P.: Temporal meta-axioms in logical agents. In: Electr. Proc. of CILC 2012, Italian Conference of Computational Logic. Number 857 in CEUR Workshop Proceedings Series (2012)
44. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
45. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Generation Computing **4** (1986) 67–95
46. de Boer, F.S., Hindriks, K.V., van der Hoek, W., Meyer, J.J.C.: A verification framework for agent programming with declarative goals. J. Applied Logic **5**(2) (2007) 277–302
47. Holzmann, G.: The model checker spin. IEEE Transactions on Software Engineering (23) (199) 279–295
48. Bourahla, M., Benmohamed, M.: Model checking multi-agent systems. Informatica (Slovenia) **29**(2) (2005) 189–198
49. Walton, C.: Verifiable agent dialogues. J. Applied Logic **5**(2) (2007) 197–213
50. Bordini, R., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. Autonomous Agents and Multi-Agent Systems **12**(2) (2006) 239–256
51. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng.
52. Jones, A., Lomuscio, A.: Distributed bdd-based bmc for the verification of multi-agent systems. In: Proc. of the 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010). (2010)
53. Montali, M., Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verification from declarative specifications using logic programming. In: 24th Int. Conf. on Logic Programming (ICLP'08). Volume 5366 of Lecture Notes in Computer Science., Springer (2008) 440–454
54. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The sciff framework. ACM Trans. Comput. Log. **9**(4) (2008)
55. Montali, M., Chesani, F., Mello, P., Torroni, P.: Modeling and verifying business processes and choreographies through the abductive proof procedure sciff and its extensions. Intelligenza Artificiale, Intl. J. of the Italian Association AI*IA **5**(1) (2011)
56. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. J. Log. Program. **33**(2) (1997) 151–165

57. Chesani, F., Mello, P., Montali, M., Torroni, P.: A logic-based, reactive calculus of events. Fundam. Inform. **105**(1-2) (2010) 135–161
58. Torroni, P., Chesani, F., Mello, P., Montali, M.: A retrospective on the reactive event calculus and commitment modeling language. In Sakama, C., Sardiña, S., Vasconcelos, W., Winikoff, M., eds.: Declarative Agent Languages and Technologies IX - 9th International Workshop, DALT 2011, Revised Selected and Invited Papers. Volume 7169 of Lecture Notes in Computer Science., Springer (2012) 120–127
59. Bragaglia, S., Chesani, F., Mello, P., Montali, M., Torroni, P.: Reactive event calculus for monitoring global computing applications. In Artikis, A., Craven, R., Cicekli, N.K., Sadighi, B., Stathis, K., eds.: Logic Programs, Norms and Action - Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday. Volume 7360 of Lecture Notes in Computer Science., Springer (2012) 123–146
60. Chittaro, L., Montanari, A.: Efficient temporal reasoning in the cached event calculus. Computational Intelligence **12** (1996) 359–382
61. Singh, M.P.: Commitments in the architecture of a limited, rational agent. In Cavedon, L., Rao, A.S., Wobcke, W., eds.: Intelligent Agent Systems, Theoretical and Practical Issues, Based on a Workshop Held at PRICAI'96. Volume 1209 of Lecture Notes in Computer Science., Springer (1997) 72–87
62. Yolum, P., Singh, M.P.: Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. Ann. Math. Artif. Intell. **42**(1-3) (2004) 227–253
63. Levesque, H.J.: Comments on "knowledge, representation, and rational self-government". In Vardi, M.Y., ed.: TARK, Morgan Kaufmann (1988) 361–362
64. Cohen, P.R., Levesque, H.J.: Intention is choice with commitment. Artif. Intell. **42**(2-3) (1990) 213–261
65. Sterling, L., Shapiro, E.Y.: The Art of Prolog - Advanced Programming Techniques, 2nd Ed. MIT Press (1994)