Runtime Self-Checking via Temporal (Meta-)Axioms for Assurance of Logical Agent Systems

Stefania Costantini and Giovanni De Gasperis[1]

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

Abstract. This paper deals with assurance of logical agent systems via runtime self-monitoring and checking. We adopt temporal-logic-based special constraints to be dynamically checked at a certain (customizable) frequency. These constraints are based upon a simple interval temporal logic particularly tailored to the agent realm, A-ILTL ('Agent-Oriented Interval LTL', LTL standing as customary for 'Linear Temporal Logic').

1 Introduction

Certification and assurance of agent systems constitute crucial and far-from-trivial issues, as agents represent a particularly complex case of dynamic, adaptive and reactive software systems. Certification is aimed at producing evidence indicating that deploying a given system in a given context involves the lowest possible level of risk of adverse consequences (which level of risk can be considered sufficiently "low" depends upon the application at hand). Assurance is related to dependability, i.e., to ensuring (or at least obtaining a reasonable confidence) that system users can rely upon the system. The issue is nicely discussed in [1], where it is noted that:

> The term [assurance] is used in a broad (and somewhat imprecise) sense. Where there is a clear specification (which is not always the case!) then we can use the two standard terms "verification" and "validation". Verification in this context refers to checking whether software meets its specification, and validation refers to checking whether the specification meets the user's requirements.

It is widely acknowledged that industrial adoption of agents systems finds a serious obstacle in the stakeholders lack of confidence about reliability of runtime behavior of such systems. Citing [2],

> ... the use of adaptive systems for greater resilience create situations where runtime verification and monitoring could be particularly valuable. ... Within suitable new frameworks, some of the evidence required for certification can be achieved by runtime monitoring - by analogy with runtime verification, this approach can, somewhat provocatively, be named "runtime certification".

In this paper, we propose methods for runtime monitoring of agent systems. These methods are not in alternative but rather complementary to the many existing verification and testing methodologies.

Pre-deployment assurance and certification techniques for agent systems include verification and testing. Since we do not have room for an extensive illustration we can provide just few pointers to recent literature, so we invite the reader to refer to the recent book [3] and to the references therein. Most verification methods rely upon model-checking, and some (e.g., [4]) upon theorem proving. Among recent interesting work about agent systems (pre-deployment) assurance we particularly mention [1] which proposes (though in a preliminary way) a method that alternates the application of testing with formal verification techniques applied within a "Shallow Scope", i.e, with a limited scope of variable values. The outcome of each phase should be taken as a guidance for the other phase. Thus, different techniques are used in synergy so as to improve the overall level of assurance. About fault detection and recovery a particularly interesting work is that of [5], that opens a new promising direction in model-checking-based verification techniques. This approach allows for CTL specifications that express injection and eventual recovery from a fault.

For formalizing and implementing runtime self-checking in logical agents while coping with unanticipated circumstances, we propose temporal-logic-based special constraints to be dynamically checked at a certain (customizable) frequency. These constraints are based upon a simple interval temporal logic particularly tailored to the agent realm, A-ILTL ('Agent-Oriented Interval LTL', LTL standing as customary for 'Linear Temporal Logic'). In this setting, properties can be defined that should hold according to events that have happened and to events which are supposed to happen or not to happen in the future. This also considering partially specified event sequences, unexpected events or event order. The adoption of an interval logics allows for the specification of time-bounded properties: it makes it possible to specify that some property should occur within a certain time frame or before/after a certain time, where the interval can also be conditionally defined. A-ILTL constraints are contextual, i.e., they can be specified in a general form and each time they are checked they are instantiated (via suitable preconditions) to the present agent's state.

In [2], it is advocated that for adaptive systems (of which agents are clearly a particularly interesting case) assurance methodologies should whenever possible imply not only detection but also recovery from software failures. In fact, though (at least in principle) a certified software should not fail, in practice serious software-induced incidents have been observed in certified critical systems. In [2] examples are produced concerning airplane and air traffic control, where failures are often due on the one hand to incomplete specifications and on the other hand to unpredictability of the environment.

In [6], which discusses medical robotic applications in human telesurgery, it is emphasized how such systems should be *fail safe* in the sense that, in the event of failure, should proactively respond so as to limit harm to other devices or danger to users.

Our methods in fact provide the possibility of correcting and/or improving agent's functioning: the behavior can be corrected whenever an anomaly is detected, but can also be improved whenever it is acceptable, yet there is room for getting a better perfor-

mance. Counter measures can be object-level, i.e., related to the application, or meta-level, e.g., replacing (as suggested in [2]) a software component by a diverse alternate.

A-ILTL constraints are defined over formulas of any underlying logic language $\mathcal{L}$, and are rooted in the Evolutionary Semantics of agent programs [7]. We thus obtain a fairly general setting, that could be adopted in several logic agent-oriented languages and formalisms, such as, e.g., AgentSpeak (cf. [8, 9] and the references therein), DALI [10–12]), GOAL [13, 14], and 3APL [15, 16].

In this paper, we show how A-ILTL temporal constraints may be used to check for critical situations and to enforce suitable reaction patterns for achieving recovery. The novelty of the approach is in the following aspects. (i) A-ILTL temporal constraints constitute a device for run-time self-monitoring which can be completely integrated into agent programs and their semantics. I.e., there is no separate monitor which examines a "trace" of observations performed on the agent's behavior. (ii) Self-recovery/repair is encompassed in the approach. (iii) The semantic integration into the Evolutionary semantics is devised such that there is no need to implement a full temporal-logic inference engine, at least if keeping the expressions to be checked reasonably simple. (iv) Consequently, the complexity of check is reasonably low.

The paper is organized as follows. In Section 2 we recall the Evolutionary Semantics. In Sections 3- 4 we introduce the A-ILTL logic, also in relation to the Evolutionary Semantics. In Section 5 we illustrate A-ILTL constraints and show by means of examples how such constraints can be exploited for runtime monitoring and self-repair of agent systems. In Section 6 we briefly discuss the complexity related to run-time constraint checking. Finally, in Section 7 we discuss related work and propose some concluding remarks.

2 Evolutionary Semantics

The Evolutionary semantics (introduced in [7]) is meant at providing a high-level general account of evolving agents, trying to abstract away from the details of specific agent-oriented frameworks. We define, in very general terms, an agent as the tuple $Ag = < P_{\mathcal{A}}, E >$ where $\mathcal{A}$ is the agent name and $P_{\mathcal{A}}$ (that we call "agent program", but can be in turn a tuple) describes the agent according to some agent-oriented formalism $\mathcal{L}$. E is the set of the events that the agent is able to recognize or determine (so, E includes actions that the agent is able to perform), according to the specific agent-oriented framework.

Let H be the *history* of an agent as recorded by the agent itself (in a form that will depend upon the specific agent-oriented framework), i.e., H includes agent's perceptions and *memories*. For instance, in DALI the history consists of: the set Ev of external and internal events, that represent respectively events that the agent presently perceives of its environment, and events that the agent has raised by its own internal reasoning processes; the set Act of the actions that the agent is enabled to perform at its present stage of operation; the set P of most recent versions "past events", which include: previously perceived events, but also actions that the agent has performed (notice that elements of Ev and Act will be transferred into P at the next stage); the set PNV of previous instances of past events (e.g., P may contain the last measurements

of temperature while PNV may contain older ones), plus *past constraints* that specify interaction between P and PNV.

We assume that program $P_{\mathcal{A}}$ as written by the programmer is in general transformed into an initial agent program P_0 by means of an *initialization step*. When agent $\mathcal{A}$ is activated P_0 will go into execution, and will evolve according to events that either happen or are generated internally, to actions which are performed, etc., i.e., according to the evolution of H.

Evolution in this setting is represented via program-transformation steps, each one transforming P_i into P_{i+1} according to H_i, which is the partial history up to stage i. The choice of which elements of H_i do actually trigger an evolution step is part of the definition of a specific agent framework.

Thus, we obtain 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 semantics of P_i according to $\mathcal{L}$. Notice in fact that the approach is parametric w.r.t $\mathcal{L}$.

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* snapshot 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 [7], program transformation steps associated with DALI language constructs are defined in detail. They can easily be adapted to AgentSpeak [8, 9] as the two languages share a number of similarities. More generally however, in the specific agent setting under consideration an evolution step will occur at least whenever new events are perceived, reacted to, and recorded, and whenever an agent proactively undertakes measures to pursue its goals. An evolution step will possibly determine an update of the history, which is a part of the agent's *belief base*[1]. Thus, each evolution step affects the belief or "mental" state of an agent. The evolutionary semantics may express for instance the notion of *trace* of a GOAL agent [13, 14] where agent program P_i encompasses the agent's *mental state* and each evolution step, which in GOAL is called *computation step* is determined by a *conditional action*. For 3APL [15, 16], agent program P_i encompasses the agent's *initial configuration*, and the related sets GR of *goal rules*, PR of *plan rules*, IR of *interactive rules*; the evolutionary semantics corresponds to a 3APL agent *computation run*, and evolution steps are determined by the 3APL *transition system*.

The semantics presented in [17] for Reactive Answer Set Programming, based upon "incremental logic programs" and "online progression", brings some conceptual similarity with the (pre-existing) Evolutionary Semantics.

[1] Equivalently, according to the specific agent framework with its own terminology, one may talk of an agent's *knowledge base*

3 A-ILTL

For defining properties that are supposed to be respected by an evolving system, a well-established approach is that of Temporal Logic, and in particular of Linear-time Temporal Logics (LTL, cf., e.g., [18]). These 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"). In what follows, we use the standard notation for the best-known LTL operators.

An interval-based extension to the well-known linear temporal logic LTL is formally introduced in [19] where it is called A-ILTL for 'Agent-Oriented Interval LTL'. Though, as discussed in [19], several "metric" and interval temporal logic exist, the introduction of A-ILTL is useful in the agent realm because the underlying discrete linear model of time and the complexity of the logic remains unchanged with respect to LTL. This simple formulation can thus be efficiently implemented, and is nevertheless sufficient for expressing and checking a number of interesting properties of agent systems.

Formal syntax and semantics of A-ILTL operators (also called below "Interval Operators") are fully defined in [19]. A-ILTL expressions are (like plain LTL ones) 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 and A-ILTL operators (where however nesting of A-ILTL operators is not allowed), the semantics of a temporal formula is provided by a satisfaction relation: 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. A structure $\mathcal{M} = \langle \mathbb{N}, \mathcal{I} \rangle$ is a model of φ if $\mathcal{M}, i \models \varphi$ for some $i \in \mathbb{N}$.

Some among the A-ILTL operators are the following.

Definition 3. *Let $\varphi \in \mathcal{F}$ and let m, n be positive integer numbers.*

$F_{m,n}$ *(*eventually (or "finally") in time interval*). $F_{m,n}\varphi$ states that φ has to hold sometime 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 $\mathcal{M}, j \models \varphi$. Can be customized into F_m,* bounded eventually (or "finally")*, where φ should become true somewhere on the path from the current state to the (m)-th state after the current one.*

$G_{m,n}$ *(*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 $j \leq n$ $\mathcal{M}, j \models \varphi$. Can be customized into G_m,* bounded always*, where φ should become true at most at state s_m.*

$N_{m,n}$ *(*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 $\mathcal{M}, j \models \varphi$.*

4 A-ILTL and Evolutionary Semantics

In this section, we refine A-ILTL 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. 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 [20] 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 snapshots 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 is immediate to let $s_i = \rho_i$. This requires however a refinement: in fact, in a writing Op_m or $Op_{m,n}$ occurring in an agent program parameters m and n will not necessarily coincide with time instants of the above-defined timed state sequence. To fill this gap, in [19] a suitable approximation is introduced.

We need to adapt the interpretation function $\mathcal{I}$ of LTL to our setting. In fact, we intend to employ A-ILTL within agent-oriented languages, where 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 expressions of $\mathcal{L}$ (an expression is *ground* if it contains no variables, and each expression of $\mathcal{L}$ has a possibly infinite number of ground versions). A given agent program can be taken as standing for its (possibly infinite) ground version, as it is customarily done in many approaches. 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 5. *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, 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}.*

A-ILTL properties will be verified at run-time, and thus they act as *constraints* over the agent behavior[2]. In an implementation, verification may not occur at every state (of the given interval). Rather, sometimes properties need to be verified with a certain frequency, that can be specifically tuned to the various cases. 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, composed of formulas $\{\tau_1, \ldots, \tau_l\}$. Agent evolution can be considered to be "satisfactory" if it obeys all these properties.

Definition 6. *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 [21]. 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.

A-ILTL rules may imply asserting and retracting rules or sets of object rules ("modules"). In this setting, assert and retract can be considered as special A-ILTL operators, for which a formal semantics is provided (cf. [19]).

5 A-ILTL for Monitoring Liveness and Safety Properties

In this section we illustrate the usefulness of A-ILTL constraints for defining and verifying liveness and safety properties in agent systems. In software engineering, *liveness* properties concern the progress that an agent makes and express that a (good) state eventually will be reached, while *safety* properties express that some (bad) state will never be entered. This implies that liveness is concerned with the evolution of a system, while in general safety is not: notice in fact that, paradoxically, doing nothing prevents bad states from being reached. Notice however that in our setting we restricted ourselves to monotonic state sequences based upon the evolutionary semantics, so that our agents evolve by definition. Notice that, if violated, liveness properties are violated in infinite

[2] By abuse of notation we will indifferently talk about A-ILTL rules, expressions, or constraints.

time (a good state not yet reached might be in principle reached in the future) while safety properties are violated in finite time, in case a "bad" state is reached. It is widely acknowledged (cf., e.g., [22]) that any property can be expressed as a conjunction of a safety and a liveness property. In agents, "bounded" liveness is often more interesting than "pure" liveness: in fact, sometimes it does not suffice that a certain state might be reached in an indefinite future, as agents are situated real-time working entities that operate with limited computational resources and within deadlines. Bounded liveness properties are equivalent to safety properties that are violated whenever the desirable state is not reach withing the deadline. However, expressing such properties in the form of liveness properties is often more intuitive. A-ILTL operators can be defined either on finite intervals and then, to any practical extent, they define safety properties, or to infinite intervals (with no upper bound) thus defining liveness properties.

We employ in the examples a *pragmatic* form for A-ILTL expressions related to logic agent-oriented languages. In particular, we represent an A-ILTL expression in the form $OP(m, n; k)\,\varphi$ where: m, n define the time interval where (or since when, if n is omitted) expression $OP\,\varphi$ is required to hold, and k (optional) is the frequency (in terms of states, or time instants) for checking whether the expression actually holds.

For instance, $EVENTUALLY(m, n; k)\,\varphi$ states that φ should become true at some point between time instants (states) m and n.

In rule-based logic programming languages, we may reasonably restrict φ to be a conjunction of literals. In pragmatic A-ILTL formulas, φ must be ground when the formula is checked. In fact, we allow variables to occur in an A-ILTL formula, to be instantiated via a *context* χ (we then talk about *contextual A-ILTL formulas*). Notice that, for the evaluation of φ and χ, we rely upon the procedural semantics of the 'host' language.

In the following, a contextual A-ILTL formula τ will implicitly stand for the ground A-ILTL formula obtained via evaluating the context. In [19] it is specified how to *operationally* check whether such a formula holds. This by observing that A-ILTL operators defined over finite intervals there is a *crucial state* where it is definitely possible to assess whether a related formula holds or not in given state sequence, by observing the sequence up to that point and ignoring the rest.

In runtime self-checking, as discussed above, an issue of particular importance in case of violation of a property is that of undertaking suitable measures in order to recover or at least mitigate the critical situation. Actions to be undertaken in such circumstances can be seen as an internal reaction to criticalities. More effective reaction can be defined if complex reactive features are available in the underlying language. In non-trivial cases, the issue of runtime recovery has a significant intersection (that had not been identified so far) with "Complex Event Processing" (CEP), which is an emergent relevant new field of software engineering and computer science [23]. In fact, a lot of practical applications have the need to actively monitor vast quantities of event data to make automated decisions and take time-critical actions [24–27] (cf. also the Proceedings of the RuleML Workshop Series). Many of the current approaches to CEP are declarative and based on rules, and often on logic-programming-like languages and semantics: for instance, [24] is based upon a specifically defined interval-based Event Calculus [28]. In logical agents, [29–31] tackled the issue of complex reactivity, by

considering the possibility of choosing among different possible reactive patterns also by means of complex preferences. In the present paper, we show by means of examples how kinds of A-ILTL constraints exploiting complex reactivity can be useful in runtime recovery. For lack of space reactive patterns will be discussed informally in relation to examples.

Below is the general form of an A-ILTL constraint with a reactive component that we call *recovery pattern*.

Definition 7. *A reactive A-ILTL rule is of the form (where M, N, K can be either variables or constants)*

$$OP(M, N; K)\varphi :: \chi \div \rho$$

where:(i) $OP(M, N; K)\varphi :: \chi$ is a contextual A-ILTL formula, called the monitoring condition*, that should involve the observation of either external or internal events; (ii) ρ is called the* recovery component *of the rule, and it consists of a complex reactive pattern.*

Whenever the monitoring condition (automatically checked at frequency K) is violated (i.e., it does not hold) within given interval, then the recovery component ρ is executed. Syntax and semantics of reactive patterns usable in the recovery component will depend upon the underlying language $\mathcal{L}$. In the examples, we adopt a sample syntax suitable for logic-programming-based settings.

Consider for instance the example of a controller agent that has to keep the temperature in a certain time frame (say between 8 a.m. and 5 p.m.) in the range 19–21 (Celsius degrees). In this case, the measure $temperature$ of temperature implies sensing actions to be performed with a sampling period by the agent. If the condition is violated, a reaction should try to restore the wished-for situation. We assume in fact to be in a smart building, where the temperature is monitored by intelligent agents, and where each agent tries to select, in order to modify the temperature, the best suitable energy source: for instance, according to present circumstances, an agent might select the less expensive font of energy or, in case of a measure significantly different from wished-for values, the font which guarantees the most efficient correction. Notice that in the course of time different fonts of energy can be deemed to be the best choice. At each check (where in fact the A-ILTL constraints is dynamically checked at the specified frequency, or at a default frequency in case none is provided) we assume that the best choice can determined by means of an application-dependent decision procedure. So, in given interval, the monitoring condition will sometimes succeed (the temperature is within range, then nothing is done) and will sometimes fail. In the latter case, the font of energy S which is deemed more effective (in terms of cost and/or efficiency) *in that moment* is determined, and used in order to suitably affect the temperature and try to keep it within the specified range (where $modify\_temperature_G(S)$ is a goal, involving appropriate actions). In A-ILTL, this can be formalized as follows by exploiting complex preferences introduced in [32]. As there are no variables, context is omitted.

$$ALWAYS(8:00\ a.m.,\ 5:00\ p.m.; 10m)\ 19 \leq temperature \leq 21 \div modify\_temperature_G(S),$$
$$S\ IN\ \{external\_electricity, gas, solar\_panel\_electricity : most\_effective\}$$

The next example is a meta-statement expressing the capability of an agent to modify its own behavior. In case a goal G has not been achieved (in a certain context) because the allotted time has elapsed, then the recovery component implies replacing the planning module (assuming that more than one is available) and retrying the goal. We suppose that the possibility of achieving a goal G is evaluated w.r.t. a module M that represents the context for G (notation $P(G, M)$, P standing for 'possible'). Necessity and possibility evaluation within a reasonably complex framework has been discussed in [30]. In case the goal is still deemed to be possible but has not been achieved before a certain deadline, the reaction consists in substituting the present planning module and re-trying the goal.

$$NEVER\ goal(G),$$
$$eval\_context(G, M), P(G, M), timed\_out(G), not\ achieved(G) \div$$
$$replace\_planning\_module, retry(G)$$

It can be useful to define properties to be checked upon arrival of event sequences, of which however only relevant events (and their order) should be considered. To this aim we introduce a new kind of A-ILTL constraints, that we call *Evolutionary A-ILTL Expressions*. To define partially known sequences of any length, on the line of *dynamic logic* [33] we admit for event sequences a syntax reminiscent of regular expressions so as to specify irrelevant/unknown events, and repetitions. In particular, *event expressions* (and, analogously, *action expressions*) may be primitive events e, sequences of event expressions $e_1; e_2, \ldots$, zero or more iterations of an event expression e^*, or a choice among event expressions $e_1 + e_2 + \ldots$. We also admit "wild cards", i.e., variables (starting with uppercase) to stand for unknown events/actions.

Definition 8 (Evolutionary A-LTL Expressions). *Let $\mathcal{S}^{\mathcal{E}vp}$ be a sequence of past events, and $\mathcal{S}^{\mathcal{F}}$ and $\mathcal{J}^{\mathcal{J}}$ be sequences of events. 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: (i) $\mathcal{S}^{\mathcal{E}vp}$ denotes 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; (ii) $\mathcal{S}^{\mathcal{F}}$ denotes the events that are expected to happen in the future without affecting τ; (iii) $\mathcal{J}^{\mathcal{J}}$ denotes the events that are expected* not *to happen in the future; i.e., whenever any of them should happen, φ is not required to hold any longer, as these are "breaking events".*

An Evolutionary LTL Expression can be evaluated w.r.t. a state s_i which includes among its components the *history* of the agent, i.e., the list of past events perceived by the agent. A history H *satisfies* an event sequence S whenever all events in S occur in H, in the order specified by S itself.

Definition 9. *An Evolutionary A-ILTL Expression ϖ, of the form specified in Definition 8: (1)* holds *in state s_i whenever (i) history H_i satisfies $\mathcal{S}^{\mathcal{E}vp}$ and $\mathcal{S}^{\mathcal{F}}$ and does not include any event in $\mathcal{J}^{\mathcal{J}}$, and τ holds or (ii) H_i includes any event occurring in $\mathcal{J}^{\mathcal{J}}$ (the expression is* broken*); (2) is* violated *in state s_i whenever H_i satisfies $\mathcal{S}^{\mathcal{E}vp}$ and $\mathcal{S}^{\mathcal{F}}$ and does not include any event in $\mathcal{J}^{\mathcal{J}}$, 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 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 the occurrence of breaking events.

The following is an example of Evolutionary A-ILTL Expression that might occur in an agent program installed on an autonomous robot working on batteries, and able to check its own charge level. The robot moves in some environment to perform some task. The following A-ILTL axiom states that after a battery recharge (indicated as a past event, postfix 'P') at time T, the charge level should be sufficient for 6 hours despite a sequence of actions which can be considered to be 'normal' in relation to the robot's task. These actions may for instance involve moving around, cleaning rubbish, delivering packages, etc. Instead, the charge level can be expected to be low in case of extensive usage actions, for instance in case of an exceptional unexpected event that requires the robot to increase its activities (e.g., drying water in case of a flooding from a broken pipe). There is a classification of what should be intended by 'normal' and 'extensive' usage.

$$
\begin{array}{l}
recharge\_battery_P : T \ : \\
\quad ALWAYS(T, T + 6_{hour}) \ charge\_level(L), L > low \\
\quad ::: \ normal\_usage\_action(Act)* \ :::: \ extensive\_usage\_action(Act)*
\end{array}
$$

The above expression should be combined with another A-ILTL expression forcing recharge every six hours. The latter should state that if the last battery recharge $recharge\_battery_P$ has occurred at time T which is more than six hours different from present time now, then as a recovery the goal $recharge\_battery_G$ must be set. Achieving this goal may require, for instance, reaching the nearest recharge station. Notice that, in this case, we have used an A-ILTL constraint as a programming construct, which however has a role in terms of assurance since it forces the agent to respect a timing which is essential for the system good functioning.

$$
\begin{array}{l}
ALWAYS \\
\quad recharge\_battery_P : T, now - T > 6_{hour} \ \div \ recharge\_battery_G
\end{array}
$$

Whenever an Evolutionary A-ILTL expression is either violated or broken, a reaction can be attempted aiming at recovering a desirable or at least acceptable agent's state.

Definition 10. *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. η_1 and η_2 are called* countermeasures.

In previous example, whenever the robot detects a low level of charge, countermeasure η_1, taken in case of low battery under normal usage, may for instance imply alerting the user, as a fault either in the battery or in the recharge station can be hypothesized. Instead η_2, taken in case of low battery under exceptional usage, will simply imply the robot to resort to the recharge station. The overall expression will take the form:

$recharge\_battery_P : T$:
$ALWAYS(T, T + 6_{hour})\ charge\_level(L), L > low$
$\mid alert\_user\_possible\_fault_A \mid\mid recharge\_battery_G$

6 Complexity of Check and Discussion

In this section we synthetically analyze the complexity of checking A-ILTL expressions. For lack of space, we cannot provide a detailed account. We make the simplifying assumption that all expressions are checked at the same frequency: i.e., the agent devotes with a certain periodicity some amount time to perform the check. Here we evaluate this amount. Let us assume to have f A-ILTL expressions, and that the time for retrieving each expression from the computer memory is m. Thus, retrieving all expressions to be evaluated is $\mathcal{O}(f \star m)$. Let k be the number of the different A-ILTL operator occurring in the f expressions. Let if_eval be the time needed in order to understand whether each expression needs to be evaluated at the present state: this includes checking w.r.t. the crucial state and, in case of Evolutionary A-ILTL Expressions, checking the event sequence $\mathcal{S}^{\mathcal{E}vp}$ w.r.t. current agent's history. Let max_eval be the maximum time needed for the evaluation of each contextual A-ILTL formula $Op\,\varphi\ ::\ \chi$. Let $if\_viol\_or\_broken$ be the maximum time needed to state whether each Evolutionary A-ILTL Expressions is either violated or broken: this implies checking event sequences $\mathcal{S}^{\mathcal{F}}$ and $\mathcal{J}^{\mathcal{J}}$ w.r.t. current agent's history.

Therefore, the total time to be spent for checking all A-ILTL Expression (in the worst case, where all of them are of the Evolutionary kind, and each of them needs to be evaluated at the present state) can be estimated to:

$$\mathcal{O}((f \star m) + (f \star (if\_eval + max\_eval + if\_viol\_or\_broken)))$$

Then, for each expression which is either violated or broken, there will be a time spent in the recovery and countermeasure actions.

The relatively low complexity of check (which however requires to keep the number of A-ILTL expressions as low as possible, and to tune frequency carefully, according to the environment change rate) is due to the definition of A-ILTL in relation to the Evolutionary semantics: in fact, it is not needed to implement a temporal logic inference engine, rather to periodically check $Op\,\varphi\ ::\ \chi$. This in the case of simple non-nested A-ILTL expressions. Introducing more complex expressions is a subject of future work.

7 Related Work and Concluding Remarks

In this paper, we have proposed A-ILTL runtime constraints for agents' self-checking and monitoring. We have shown how to express via these constraints a number of useful liveness and safety properties. We have provided a semantic framework general enough for accommodating a number of agent-oriented languages, so as to allow A-ILTL constraints to be adopted in different settings. This work has been influenced by [34, 19, 35, 36].

We may easily notice similarities between A-ILTL constraints and event-calculus formulations [28]. Also, approaches based on abductive logic programming such as,

SCIFF (cf. [37] and the references therein) allow one to model dynamically upcoming events, and specify positive and negative expectations, and the concepts of fulfillment and violation of expectations. Reactive Event Calculus (REC) stems from SCIFF [38] and adds more flexibility by reacting to new events by extending and revising previously computed results. However, these approaches have been devised for static or dynamic checking when performed by a third party. Event sequences, the concepts of violated and broken expressions, complex reaction patterns, and independence of the underlying logic are however distinguished features of the proposed approach.

A well-established line of work concerning the use of temporal logic in order to define run-time monitors is discussed in [39] and the references therein. However, this work is not related to agents, and does not concern self-checking: in fact, they propose a rule-based temporal language for defining "monitors" which examine either on-line or off-line some kind of "observable trace" generated by the program under check. There is no notion of recovery in case malfunctioning should be detected.

The proposed approach has been experimented in the context of energy management in smart buildings [40]. Such intelligent control is dynamic by nature, and must fulfill real-time requirements: in fact, each building has its own dynamical thermo-physical behavior and is immersed in a dynamic environment where weather events change its energy footprint in function of time. The outcome of the experiments is encouraging, in the sense that adopting agents equipped with the proposed features allows for not only general but also local (room-by-room or area-by-area) control of energy saving according to user comfort requirements and preferences.

Future work includes refining A-ILTL constraints to adapt to different self-checking issues and contexts. As suggested in [2], a very interesting line of investigation concerns automated synthesis of runtime constraints from specifications but also from test results, extracting invariants expressing correct or critical situations.

References

1. Winikoff, M.: Assurance of agent systems: What role should formal verification play? (2010)
2. Rushby, J.M.: Runtime certification. In Leucker, M., ed.: Runtime Verification, 8th International Workshop, RV 2008. Selected Papers. Volume 5289 of Lecture Notes in Computer Science. Springer (2008) 21–35
3. Dastani, M.M., Hindriks, K., Meyer, J.J.C., eds.: Specification and Verification of Multi-agent Systems. Springer US (2010)
4. Shapiro, S., Lesprance, Y., Levesque, H.: The cognitive agents specification language and verification environment (2010)
5. Ezekiel, J., Lomuscio, A.: Combining fault injection and model checking to verify fault tolerance in multi-agent systems. In Sierra, C., Castelfranchi, C., Decker, K.S., Sichman, J.S., eds.: 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Proceedings, Volume 1, IFAAMAS (2009) 113–120
6. Butner, S., Ghodoussi, M.: Transforming a surgical robot for human telesurgery. IEEE Transactions on Robotics and Automation **19**(5) (2003) 818–824
7. 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

8. Rao, A.S.: Agentspeak(l): Bdi agents speak out in a logical computable language. In de Velde, W.V., Perram, J.W., eds.: Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Proceedings. Volume 1038 of Lecture Notes in Computer Science., Springer (1996) 42–55
9. Bordini, R.H., Hübner, J.F.: Semantics for the jason variant of agentspeak (plan failure and some internal actions). In Coelho, H., Studer, R., Wooldridge, M., eds.: ECAI 2010 - 19th European Conference on Artificial Intelligence, Proceedings. Volume 215 of Frontiers in Artificial Intelligence and Applications., IOS Press (2010) 635–640
10. 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)
11. 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)
12. Costantini, S., D'Alessandro, S., Lanti, D., Tocchio, A., al.: DALI web site, download of the interpreter (2012) Released: basic DALI features. For beta versions please ask the authors.
13. Hindriks, K.V.: Programming rationalagents in goal. In El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H., eds.: Multi-Agent Programming:. Springer US (2009) 119–157
14. Hindriks, K.: A verification logic for goal agents (2010)
15. Dastani, M., van Riemsdijk, B., Dignum, F., Meyer, J.J.C.: A programming language for cognitive agents goal directed 3apl. In Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: Programming Multi-Agent Systems, First International Workshop, PROMAS 2003, Selected Revised and Invited Papers. Volume 3067 of Lecture Notes in Computer Science., Springer (2004) 111–130
16. Dastani, M., van Riemsdijk, M.B., Meyer, J.J.C.: Programming multi-agent systems in 3apl. In Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: Multi-Agent Programming: Languages, Platforms and Applications. Volume 15 of Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer (2005) 39–67
17. Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. In Delgrande, J.P., Faber, W., eds.: Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Proceedings. Volume 6645 of Lecture Notes in Computer Science., Springer (2011)
18. Emerson, E.A.: Temporal and modal logic. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, vol. B. MIT Press (1990)
19. Costantini, S.: Self-checking logical agents. In: Proc. of LA-NMR 2012. Volume 911., CEUR Workshop Proceedings (CEUR-WS.org) (2012) Invited paper.
20. 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
21. Manna, Z., Pnueli, A.: Adequate proof principles for invariance and liveness properties of concurrent programs. Sci. Comput. Program. **4**(3) (1984) 257–289
22. Dederichs, F., Weber, R.: Safety and liveness from a methodological point of view. Inf. Process. Lett. **36**(1) (1990) 25–30
23. Chandy, M.K., Etzion, O., von Ammon, R.: 10201 Executive Summary and Manifesto – Event Processing. In Chandy, K.M., Etzion, O., von Ammon, R., eds.: Event Processing. Number 10201 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2011)
24. Paschke, A., Kozlenkov, A.: Rule-based event processing and reaction rules. In: RuleML. Volume 5858 of Lecture Notes in Computer Science., Springer (2009) 53–66

25. Etzion, O.: Event processing - past, present and future. Proceedings of the VLDB Endowment, PVLDB Journal **3**(2) (2010) 1651–1652
26. Paschke, A., Vincent, P., Springer, F.: Standards for complex event processing and reaction rules. In Olken, F., Palmirani, M., Sottara, D., eds.: RuleML America. Volume 7018 of Lecture Notes in Computer Science., Springer (2011) 128–139
27. Vincent, P.: Event-driven rules: Experiences in cep. In Olken, F., Palmirani, M., Sottara, D., eds.: RuleML America. Volume 7018 of Lecture Notes in Computer Science., Springer (2011) 11
28. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Generation Computing **4** (1986) 67–95
29. Costantini, S., Dell'Acqua, P., Tocchio, A.: Expressing preferences declaratively in logic-based agent languages. In: Proc. of Commonsense'07, the 8th International Symposium on Logical Formalizations of Commonsense Reasoning, AAAI Press (2007) Event in honor of the 80th birthday of John McCarthy.
30. Costantini, S.: Answer set modules for logical agents. In de Moor, O., Gottlob, G., Furche, T., Sellers, A., eds.: Datalog Reloaded: First International Workshop, Datalog 2010. Volume 6702 of LNCS. Springer (2011) Revised selected papers.
31. Costantini, S., De Gasperis, G.: Complex reactivity with preferences in rule-based agents. In Bikakis, A., Giurca, A., eds.: Rules on the Web: Research and Applications, RuleML 2012 - Europe, Montpellier, France, August 27-29, 2012. Proceedings. Volume 6826 of Lecture Notes in Computer Science., Springer (2012) 167–181
32. Costantini, S., Formisano, A.: Modeling preferences and conditional preferences on resource consumption and production in asp. J. Algorithms **64**(1) (2009) 3–15
33. Pratt, V.R.: Semantical considerations on floyd-hoare logic. In: 17th Annual IEEE Symposium on Foundations of Computer Science, Proceedings, IEEE Computer Society (1976)
34. 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)
35. Costantini, S.: Self-checking logical agents. In Gini, M.L., Shehory, O., Ito, T., Jonker, C.M., eds.: International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13, Proceedings, IFAAMAS (2013) 1329–1330
36. Costantini, S., Gasperis, G.D.: Meta-level constraints for complex event processing in logical agents. In: Informal Proc. of Commonsense 2013, 11th International Symposium on Logical Formalizations of Commonsense Reasoning. (2013)
37. 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)
38. 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
39. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from eagle to ruler. J. Log. Comput. **20**(3) (2010) 675–706
40. Caianiello, P., Costantini, S., Gasperis, G.D., Florio, N., Gobbo, F.: Application of hybrid agents to smart energy management of a prosumer node. In: Proc. of DCAI 2013, 10th International Symposium on Distributed Computing and Artificial Intelligence. Volume 217 of Advances in Intelligent and Soft Computing., Springer (2013) 597–607