Temporal Meta-Axioms in Logical Agents

Stefania Costantini[1] and **Panagiota Tsintza**[1]

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

Abstract. This paper deals with run-time detection and possible correction of erroneous and/or anomalous behavior in agents defined in datalog- or prolog-based logic languages. In particular, we augment our previous approaches by allowing an agent to explicitly observe and record its past behavior so as to be able to decide its best actions, detect anomalous behavior and try to dynamically correct detected problems.

1 Introduction

Agents, in their interaction with the environment, are subject to modify themselves and evolve according to both external and internal stimuli, the latter due to the proactive and deliberative capabilities of the agent themselves.

In previous work, we defined semantic frameworks for datalog-based and prolog-based logical agents that account for: (i) the kind of evolution of reactive and proactive agents due to directly dealing with stimuli, that are to be coped with, recorded and possibly removed [1,2]; and (ii) the kind of evolution related to adding/removing rules from the agent knowledge base. These frameworks have been integrated into an overall framework for logical evolving agents (cf. [3,4]), where every agent is seen as the composition of a base-level (or object-level) program and one or more meta-levels. In this model, updates related to recoding stimuli are performed in a standard way, while updates involving the addition/deletion of sets of rules, related to learning, belief revision, etc. are a consequence of meta-level decisions.

As agent systems are more widely used in real-world applications, the issue of verification is becoming increasingly important (see [5] and the many references therein). The motivation of the work presented in this paper is that, though agents behavior is affected by their interaction with the external world, in most practical cases the actual arrival order of events 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 somewhat unpractical.

Properties that one wants to verify often depend upon which events have been observed by an agent up to a certain point, and which others are supposed to occur later. We believe that static verification should be integrated with dynamic self-checking aimed at detecting properties violation and trying to restore an acceptable or desired state of affairs.

The definition of frameworks such as the one that we propose here, for checking agent behavior during its life based on its experience, has not been widely treated up to

now. In the quest quest for agent platforms whose component entities would be capable of exhibiting a correct and rigorous behavior with respect to expectations, developers have frequently applied model-checking techniques (cf. [6] and Section 4 for a discussion of applications in the agent realm) that are based upon abstract models of an agent system. On the one hand however, not all properties can be statically verified. On the other hand, it would be useful to devise methods for reinstating a correct behavior at run-time in case unwanted situations should occur. Thus, we believe that the introduction of forms of dynamic (self-)checking are necessary and can be useful.

In this paper, we propose a method for checking the agent behavior correctness during the agent activity, based on maintaining information on its past "experiences" and behavior. If augmented by time stamps, these records (that we call *past events*) constitute in a way the *history* of the agent activity. The set of past events evolves in time, and past events constitute the ground upon which the agent can check its own behavior with respect to what has happened in the external environment (more precisely, with respect to what the agent has perceived about what has happened). To perform such checks, we introduce a new kind of constraints, that we call *Evolutionary LTL Expressions*, that according to what has happened and to what is supposed to happen in the future define properties that should hold and what should be done if they are violated. Evolutionary LTL Expressions are based upon specifying: (i) a sequence of events that are supposed to have happened; by using a notation obtained from regular expressions, one does not need to completely specify a finite sequence, but is allowed to define partially specified sequences of unlimited length; (ii) a temporal-logic-like expression defining a property that should hold; (iii) a sequence of events that are supposed to happen in the future, without affecting the property; (iv) a sequence of events that are supposed to happen in the future; (v) optionally, "repair" actions to be undertaken if the property is violated. The interval temporal logic adopted for defining properties has been introduced in previous work. Evolutionary LTL Expressions are a novel feature that we introduce in this paper, and that we have (prototypically) implemented and experimented.

Our approach can be adopted in any logic agent-oriented framework, and we believe that, more generally, evolutionary LTL expressions can be adopted as a programming construct in event-based extensions to datalog and prolog. In particular, one such language is DALI [7,8], which is an agent-oriented extension to prolog. We have implemented the proposed approach in DALI, where we have performed the experiments presented below.

The paper is structured as follows. In Section 2 we provide the reader with some background, concerning our declarative semantics of evolving agents, and temporal logic (extended to interval temporal logic). In Section 3 we introduce *Evolutionary LTL Expressions*, that define properties that are supposed to hold, given events that are supposed to have occurred in the past, and events that are supposed to occur or not to occur in the future. These expressions are actually constraints to be verified at run-time. We compare the proposed approach with other approaches to verification in Section 4. We introduce and discuss the problem of the experimental evaluation of the approach in Section 5. Finally, in Section 6 we conclude.

2 Background

In this paper we will refer to the declarative semantics introduced in [1], 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.

As a typical situation, perception of an external event will have an effect on the program which represents the agent: for instance, the event will be stored as a new fact. This transforms the program into a new program, that will procedurally behave differently than before, e.g., by possibly reacting to the event. Or, the internal event corresponding to the decision of the agent to undertake an activity triggers a more complex program transformation, resulting in a version of the program where the corresponding *intention* is "loaded" so as to become executable.

We abstractly formalize an agent 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. 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 $\mathcal{Y} = (E \cup I \cup A)$.

According to this semantic account, one will have an initial program $P_0 = P_{Ag}$ which, according to events that happen and actions which are performed, passes through corresponding program-transformation steps (each one transforming P_i into P_{i+1}, cf. [1]), and thus gives rise to a Program Evolution Sequence $PE = [P_0, ..., P_n, ...]$. The program evolution sequence will imply a corresponding Semantic Evolution Sequence $ME = [M_0, ..., M_n, ...]$ where M_i is the semantic account of P_i.

The different languages and different formalisms in which an agent can possibly be expressed 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 in 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.

We also believe it useful to perform an *Initialization step*, where the program P_{Ag} written by the programmer is transformed into a corresponding program P_0 by means of some sort of knowledge compilation. This initialization step can be understood as a rewriting of the program in an intermediate language and/or as the loading of a "virtual machine" that supports language features. This stage can on one extreme do nothing, on the other extreme it can perform complex transformations by producing "code" that implements language features in the underlying logical formalism. P_0 can be simply a program (logical theory) or can have additional information associated to it.

Let H be the history of an agent, that is, a set of events that happened and actions performed by the agent, 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 state of

the world as perceived by the agent[1], and a set PNV where we store all previous ones. Thus, the history H is the tuple $\langle P, PNV \rangle$. In practice, H is dynamically augmented with new events that happen. Given set $\mathcal{Y}$ of events, if one is interested in identifying the kind of event, a postfix (to be omitted if irrelevant) can provide this indication. I.e., X_E is an external event, X_A is an action and X_I an internal event. As soon as 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 [2] 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 its program and semantic evolution sequence.*

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 $\varepsilon^{Ag} = \langle H, PE, ME \rangle$. 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.*

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 Pnueli [9], for a survey the reader can refer to [10]), and in particular Linear-time Temporal Logic (LTL), that implicitly quantifies universally upon all possible paths. LTL logics are called linear because, in contrast to branching time logics, they evaluate each formula with respect to a vertex-labeled infinite path $s_0 s_1 \ldots$ 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 connectives composed of a number of unary and binary connectives referring to future-time and past-time. In prior work (see e.g., [11]), we introduced an extension to temporal logic based on *intervals*, where states in which a 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.

3 Checking the behavior of Evolving Agents

In this section we introduce Evolutionary LTL Expressions, which are aimed at checking an agent's behavior at run-time under partially specified sequences of past and future events. This extends our past work [11] where we defined meta-axioms with repair to be undergone upon violation of given conditions, but without reference to event sequences. These expressions are based upon specifying: (i) a sequence of events that are supposed to have happened; by using a notation obtained from regular expressions, one does not

[1] The agent is aware of the world through perceptions, that necessarily report a past (though hopefully recent) state.

need to completely specify a finite sequence, but is allowed to define partially specified infinite sequences; (ii) a temporal-logic-like expression defining a property that should hold; (iii) a sequence of events that are supposed to 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.

According to the evolutionary semantics summarized above, time instants $s_0 s_1 \ldots$ concerning an agent's "life" can be understood in terms of the events that happen. In fact, at the i-th evolution step we have an history H_i, an agent program P_i and its intended semantics M_i, determined by events $E_1, \ldots, E_i$ occurred so far. The next evolution step will take place in accordance to the perception of next event E_{i+1}. Then, any property φ which holds w.r.t. ε_i^{Ag}, i.e. w.r.t. the agent evolutionary semantics up to step i, will keep holding until next event will determine a transition to the next snapshot. In other words, the agent understands the world only in terms of the events that it perceives. Therefore we can define a state s_i as

$$s_i = \varepsilon_i^{Ag} = \langle H_i, P_i, M_i \rangle.$$

that is, a state is taken to be the snapshot at stage i of the evolutionary semantics of the agent.

In model-checking, the aim is to establish if some LTL formula $Op\,\varphi$ or $\varphi\,Op\,\psi$ (where Op can be, for instance, N for "never", E for "eventually", etc.) can be established to be true, given a description of the system at hand from which the system evolution can be somehow elicited. In order to cope with the many cases where this evolution cannot be fully foreseen, we propose to exploit temporal logic operators in a different way than before, so as to take into account the events that have happened already and those that are expected to happen or not to happen in the future and that are relevant to the property that we intend to check.

Definition 3 (Evolutionary LTL Expressions). *Let τ be a temporal logic expression of the form $Op\,\varphi$ if operator Op is unary or $\varphi\,Op\,\psi$ if operator Op is binary, where Op can be an interval operator $Op_{z,x}$. The evolutionary version of τ, that we will call* Evolutionary LTL Expression*, is of the form*

$$\{E_{P_1}, \ldots, E_{P_l}\} \,:\, \tau \,::\, \{F_1, \ldots, F_m\} \,:::\, \{J_1, \ldots, J_r\}$$

with $l > 1$ and $m, r \geq 0$ where:

- *$\{E_{P_1}, \ldots, E_{P_l}\}$ denote the relevant events which are supposed to have happened (without assumption about their order); i.e., these events act as preconditions: whenever they happen, τ will be checked;*
- *$\{F_1, \ldots, F_m\}$ denote the events that are expected to happen in the future (without assumption about their order) without affecting τ;*
- *$\{J_1, \ldots, J_r\}$ denote the events that are expected* not *to happen in the future; i.e., whenever any of them should happen, τ is not required to hold any longer.*

The state until which τ is required to hold depends upon the current state s_n and the operator Op occurring in τ with the following special cases.

– *τ contains an interval operator $Op_{z,x}$: in this case, x is the state until which τ is required to hold;*
– *one of the J_i's happens at state s_w, $w < x$: then, τ is not required to hold after s_w.*

Notice that both the F_i's and the J_i's are optional Notice also that, in general, we cannot foresee when (and in which order) the expected relevant events F_i's will happen.

In many practical cases, we are unable to provide a full sequence of the expected events, and in particular to be aware of how many times they will occur. Sometimes, we will be interested in specifying the (partial) order in which they should occur. Thus, in the above definition, to be able to indicate the sets of past and future events in a more flexible way we admit in part the syntax of regular expressions [12]. We also extend this syntax, in order to specify a partial order among events.

Definition 4. *If E is an event, as customary in regular expressions E^* will indicate zero or more occurrences of E, and E^+ one or more occurrences. Given events E_1 and E_2, by $E_1 \bullet E_2$ we mean that E_1 must occur before E_2 and by $E_1 \bullet \bullet E_2$ we mean that E_1 must occur immediately before E_2 (i.e., the two events must be consecutive). Wild-card X, standing for any event, can be used.*

For instance, $E_1^+ \bullet X^* \bullet E_2, E_3 \bullet \bullet X \bullet \bullet E_4$ means that, after a certain (non-zero) number of occurrences of E_1 and, possibly, of some unknown event X, E_2 and E_3 can occur in any order, immediately followed by some unknown event X and, immediately afterwards, by E_4. 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 (in a reliable way) 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.

Evolutionary LTL Expressions are easily given a semantics in the context of the Evolutionary Semantics of an agent. Therefore, a given Evolutionary LTL Expression can be evaluated w.r.t. a state, i.e. (as seen above), a snapshot ε_i^{Ag} and will hold whenever H_i encompasses the sequence of events that have been supposed to have occurred (i.e., they occur in H_i in the given order) and τ holds (i.e., i belongs to the given interval and the formula is true w.r.t. i). According to agent's activities the agent will evolve to a new snapshot, in particular when a new event happens, either expected or unwanted. In the new snapshot, the given expression can be re-evaluated. This constitutes a "punctual" evaluation of the expression. An expression involving $Op_{z,x}$ can be finally deemed to hold whenever either the interval has expired and τ has been true in all points, or an unwanted event (one of the J_is) has occurred. Instead, an expression can be deemed *not* to hold (or, as we often say, to be *violated* as far as it expresses a wished-for property) whenever τ is false in some point.
In Definition 3 we do not require the E_{P_i}'s, the F_i's and the J_i's to be ground terms. Instead, we admit each of them to contain variables if we are not interested in precisely specifying some of their parameters. For instance, the expression $consume_A{}^+(r, Q)$ (where postfix A as specified before indicates an action) stands for a sequence of events where the action of consuming (some resource r) occurs at least once. Each action will refer to an unspecified quantity Q. All variables occurring in evolutionary LTL

expressions are implicitly universally quantified, as customary in datalog- and prolog-like logic languages. An evolutionary LTL expression could be for instance (where N stands for the LTL operator "never"):

$$supply_P{}^+(r,s) \ : \ N(quantity(r,V), V < th) \ :: \ consume_A{}^+(r,Q)$$

stating that, after having provided a supply of resource r for a certain total quantity, the agent is expected to consume unknown quantities of the resource itself. The expression defines a constraint requiring that the available quantity of resource r must remain over a certain threshold th. Evolutionary LTL expressions are in fact supposed to act as constraints to be verified at run-time whenever new events are perceived. Notice in fact that the "supply" event is supposed to occur at least once, possibly an unlimited number of times. Interval-LTL formula $\tau = N(quantity(r,V), V < th)$ will be (re-)evaluated after each occurrence. A violation will happen whenever τ does not hold.

As another example, consider the following 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.

$$checkup_P(Car) : t_1 \ : \ G_{t_1,t1+6months} work\_ok(Car) \ :: long\_trip^+(Car) \ ::: \\ accident(Car)$$

In the above example we have used the operator $G_{t_1,t2}$, $t2 = t1 + 6months$ which is an interval LTL operator, G standing for "always".

We may notice the similarity between evolutionary LTL expressions and event-calculus formulations. The Event Calculus has been proposed by Kowalski and Sergot [13] 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 above LTL expression, past event $checkup_P(Car) \ : \ t_1$ initiates a fluent which is actually an interval LTL 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 LTL 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.

In our opinion, evolutionary LTL expressions might in perspective be considered as a useful programming construct in datalog and prolog (and their extensions). In fact, also by means of interval LTL operators, propositions holding over intervals in a dynamic setting can be expressed in a direct and compact way. Below we state when an evolutionary LTL expression is defined to hold.

Definition 5. *An evolutionary LTL expression $\mathcal{E}$, of the form*

$$\{E_{P_1}, \ldots, E_{P_l}\} \,:\, \tau \,::\, \{F_1, \ldots, F_m\} \,:::\, \{J_1, \ldots, J_r\}$$

holds *in state s_i whenever (i) some of the $E_{P_1}, \ldots, E_{P_l}$ has happened, some (or none) of the $F_1, \ldots, F_m$ has happened, none of the $J_1, \ldots, J_r$ has happened, and τ holds or (ii) some of the $J_1, \ldots, J_r$ has happened.*

As said before, whenever an unwanted event (one of the J_is) should happen, τ is not required to hold any longer (though it might). The proposition below formally allows for dynamic run-time checking of evolutionary LTL expressions. In fact, it states that, if a given expression holds in a certain state and is supposed to keep holding after the first expected event has happened, then checking this expression amounts to checking the modified expression where: (i) the occurred event has become a past event, and (ii) subsequent events are still expected.

Proposition 1. *Given evolutionary LTL expression $\mathcal{E} = \{E_{P_1}, \ldots, E_{P_l}\} \,:\, \tau \,::\, \mathcal{F} \,:::\, \mathcal{J}$, where $\mathcal{F} = \{F_1, \ldots, F_m\}$ and $\mathcal{J}$ is the list of unexpected events, assume that $\mathcal{E}$ holds at state s_n and that it still holds after the occurrence of event $F_i \in \mathcal{F}$ at state s_v ($v \geq n$), and that none of the events in $\mathcal{J}$ has happened. Let $\mathcal{F}_1 = \mathcal{F} \setminus F_i$. Given $\mathcal{E}_{F_i} = \{E_{P_1}, \ldots, E_{P_l}, F_{iP}\} \,:\, \tau \,::\, \mathcal{F}_1 \,:::\, \mathcal{J}$ we have that for every state s_w with ($w \geq v$) $\mathcal{E}$ holds iff $\mathcal{E}_{F_i}$ holds.*

In prior work (see e.g., [4,11]), we introduced temporal logic rules with *repair*, where actions could be specified in order to cope with unwanted situations. We extend this approach to the present work. We distinguish between two cases: in the former, none of the unwanted events has happened, and nevertheless the inner interval LTL formula τ does not hold; the latter, where τ does not hold because some of the unwanted events has occurred (in this case however, according to previous definitions the overall expression still holds).

Definition 6. *An evolutionary LTL expression $\mathcal{E}$, of the form*

$$\{E_{P_1}, \ldots, E_{P_l}\} \,:\, \tau \,::\, \{F_1, \ldots, F_m\} \,:::\, \{J_1, \ldots, J_r\}$$

is violated *in state s_i whenever some of the $E_{P_1}, \ldots, E_{P_l}$ has happened, some (or none) of the $F_1, \ldots, F_m$ has happened, none of the $J_1, \ldots, J_r$ has happened, and τ does not hold.*

Definition 7. *An evolutionary LTL expression $\mathcal{E}$, of the form*

$$\{E_{P_1}, \ldots, E_{P_l}\} \,:\, \tau \,::\, \{F_1, \ldots, F_m\} \,:::\, \{J_1, \ldots, J_r\}$$

is broken *in state s_i whenever some of the $E_{P_1}, \ldots, E_{P_l}$ has happened, some (or none) of the $F_1, \ldots, F_m$ has happened, some of the $J_1, \ldots, J_r$ has happened, and τ does not hold.*

Whenever an evolutionary LTL expression is either violated or broken, a repair can be attempted with the aim of recovering the agent's state. LTL expressions will be "hosted" in some logic agent-oriented programming language $\mathcal{L}$. The repairs will be program fragments written in $\mathcal{L}$ to be executed upon violations.

Definition 8. *An evolutionary LTL expression with repair $\mathcal{E}^{\mathcal{R}}$ is of the form:*

$$\mathcal{E}|R_1||R_2$$

where $\mathcal{E}$ is an evolutionary LTL expression adopted in a language $\mathcal{L}$, and R_1, R_2 are program fragments written in $\mathcal{L}$. R_1 will be executed whenever $\mathcal{F}^{\mathcal{R}}$ is violated, and R_2 will be executed whenever $\mathcal{F}^{\mathcal{R}}$ is broken.

Going back to the example of the car, let us assume that, if before six months after a checkup we have a malfunctioning, then one is entitled to get free assistance; in case of an accident instead, one has (say) to call the police. Again, let's assume that postfix 'A' indicates actions.

$$checkup_P(Car) : t_1 \ : \ G_{t_1,t1+6months} work\_ok(Car) \ :: \ long\_trip^+(Car) \ :::$$

$$accident(Car) \,|\, call\_free\_assistanceA(Car, Address) \,||\, call\_policeA(Address)$$

Let us now assume that language $\mathcal{L}$ provides distinguished preconditions to actions, i.e., distinguished predicates defining preconditions that are implicitly added to every action. Let us assume that $prevent(Action)$, whenever true, prevents an action from being performed, and $allow(Action, Condition)$ allows an action to be performed only if the specified condition holds[2]. Then, we assume that an action can be performed only if not prevented and allowed.

The evolutionary LTL expression with improvement listed below states that no more consumption can take place if the available quantity of resource r is scarce.

$$supply_P{}^+(r, s) \ : \ N(quantity(r, V), V < th) \ :: \ consume_A{}^+(r, Q) \,|$$

$$prevent(consume_A(r, Q))$$

We might instead adopt another (softer) constraint, that forces the agent to limit consumption to small quantities (say less than quantity q) if it is approaching the threshold (say that the remaining quantity is $th + f$, for some f).

$$supply_P{}^+(r, s) \ : \ N(quantity(r, V), V < th + f) \ :: \ consume_A{}^+(r, Q) \,|$$

$$allow(consume_A(r, Q), Q < q)$$

4 Related Work

Verification of agent programs can to some extent be performed statically (i.e., prior to agent activation), by checking the system against any possible input configuration, sequence of external events, etc., Static verification can be accomplished through model-checking techniques [6], abstract interpretation [14] or theorem proving (the latter two not commented here). Although model-checking techniques have been originally adopted for testing hardware devices, their application to software systems and

[2] These are supposed to be embedded procedures implemented within the interpreter of the language at hand to achieve the desired behavior of allowing/disallowing action execution.

protocols is constantly growing [15,16], 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., [16,17]) the 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.

However, the literature reports fully-implemented promising static verification frameworks (e.g., [18,19,20]). [20] is an extension of the SCIFF proof procedure to prove system properties at design time. [18] 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 [15] and Java PathFinder [21], 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. [19] 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 to the application of model-checking to agents and MAS 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 modeling agents through finite-state languages, which are typical of many model-checkers (e.g., [15,22]), 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 issue of run-time verification has been considered in recent work. The SCIFF proof procedure, based on abduction, allows for checking the compliance of a narrative of events w.r.t. a specification, by matching events w.r.t. expectations, ie., by performing a form of run-time monitoring (see, e.g., [23] and the references therein). [24] proposes a reactive event calculus to check *social commitments*, i.e., commitments made from an agent to another agent to bring about a certain property.

5 Experimental Evaluation and Discussion

We have implemented and we have been experimenting the proposed approach within the DALI multi-agent system (the DALI interpreter is freely available at http://www.di.univaq.it/stefcost/Sito-Web-DALI/WEB-DALI/). In this system, a frequency can be associated to each constraint by means of directives (a default frequency is otherwise provided). One can also establish since when and until when a constraint has to be checked.

An important aspect to be experimentally verified is whether constraints actually empower the agent ability to cope with events without introducing too much overhead, that would lead an agent to that "brittleness" that it is so important to avoid. For performing dynamic checking without specific constructs, one has build, within an agent program, a suitable cyclic structure to perform the needed checks. Our solution therefore, by exploiting the underlying basic cycle of the interpreter, avoids adding this further layer. In this sense, the overhead is in any case alleviated. Nonetheless, a crucial point to consider is that one, when adding new constraints and/or increasing the frequency of checks, has to experimentally verify that the addition/increment is "sustainable" by the interpreter. The experimental evaluation is in fact mandatory under this respect.

In the Appendix we present a basic experiment. Namely, we have implemented an agent that manages a FIFO queue, thus accepting requests of pushing and popping elements on/from the queue. This example is in our opinion significant because it models in an abstract way the very general and widely used architecture where an agent provides a service to a number of consumer. The instance size of our experiments (set by the user) coincides with the number of elements that are pushed/popped (at most 500). We establish the constraint, represented below in our formalism, that the queue must not contain any duplicated elements e_1 and e_2. The possible actions are $push_A(V, Q)$ that inserts a generic value V in the queue (as the e-th elements) and $pop_A(Q)$ that extracts the oldest element from the queue ($push_P(V, Q)$ is the past event recording this action having been performed). The constraint considers an unknown number of pushing actions happened in the past (and thus now recorded as past events) and can foresee an unknown number of popping actions.

$$push_P{}^{+}(V, Q) \ : \ N(in\_queue(e_1, V), in\_queue(e_2, V), e_1 \neq e_2) \ :: \ pop_A{}^{+}(Q)$$

In particular, in the Appendix we present in Section A the pure prolog code, and in Section B the DALI code. Then, in Figure 1 and Figure **??** we show the execution time of the two solutions at the growth of the instance size. In Figure **??** we show the difference in percentage between the execution times. It is possible to identify an initial "unstable" stage and then a trend that further consolidates with the growth of the instance size. In fact, what we can conclude beyond any doubt is that, when the number of events that we consider is small, the two solutions are more or less equivalent, the prolog one a bit better as it involves no overhead (while the DALI implementation of events management necessarily involves some). But, as soon as the instance size grows, the DALI solution becomes better and better in a very significant way, despite the overhead of the implementation.

6 Concluding Remarks

In this paper, we have presented an approach to detection and correction of run-time behavioral anomalies by using dynamic constraints. The runtime observation of actual anomalous behavior with dynamic possible correction of detected problems, as opposed to full prior classical program verification and validation on all inputs, can be a key

to bringing down the well-known computational burden of the agent behavior assurance problem. Suitable experiments, performed in the DALI language, have shown that the proposed approach is computationally effective, and constitutes and improvement also in terms of efficiency. However, as it happens in all the environments where static and dynamic checks are equally applicable, none of the two forms of verification may substitute the other: indeed, the only possible solution is given by a synergy aimed at creating an integrated, reliable and lightweight verification environment.

References

1. Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In Baldoni, M., Torroni, P., eds.: Declarative Agent Languages and Technologies. LNAI 3904. Springer-Verlag (2006) 106–123
2. 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)
3. Costantini, S., Tocchio, A., Toni, F., Tsintza, P.: A multi-layered general agent model. In: AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence. LNCS 4733, Springer-Verlag, Berlin (2007)
4. 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)
5. 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
6. Clarke, E.M., Lerda, F.: Model checking: Software and beyond. Journal of Universal Computer Science **13**(5) (2007) 639–649
7. 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)
8. 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)
9. Pnueli, A.: The temporal logic of programs. In: Proc. of FOCS, 18th Annual Symposium on Foundations of Computer Science, IEEE (1977) 46–57
10. Emerson, E.A.: Temporal and modal logic. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, vol. B. MIT Press (1990)
11. 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)
12. 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)
13. Kowalski, R., Sergot, M.: A logic-based calculus of events
14. 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

15. Holzmann, G.: The model checker spin. IEEE Transactions on Software Engineering (23) (199) 279–295
16. Bourahla, M., Benmohamed, M.: Model checking multi-agent systems. Informatica (Slovenia) **29**(2) (2005) 189–198
17. Walton, C.: Verifiable agent dialogues. J. Applied Logic **5**(2) (2007) 197–213
18. 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
19. 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)
20. 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). LNCS 5366, Springer Verlag (December 2008) 440–454
21. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng.
22. Dill, D., Drexler, A., Hu, A., Yang, C.: Protocol verification as a hardware design aid. In: Proc. of IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors, IEEE Computer Society Press (1992) 522–525
23. 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)
24. Torroni, P., Chesani, F., Mello, P., Montali, M.: Social commitments in time: Satisfied or compensated. In: DALT. Volume 5948 of Lecture Notes in Computer Science., Springer (2009) 228–243

A Pure Prolog Code

Below we report the code of the version of the Queue agent implemented in DALI (by Alessio Paolucci, that we thank), but using only prolog constructs. The single DALI feature that is exploited is related to the activation of the agent by means of a message. The test agent in fact gets active and performs a test session whenever it receives from the user a message with content $run\_pure\_test(Num\_Items)$ where Num_items specifies the number of elements to be pushed/popped on/from the queue.

When the agent receives event run_pure_test it reacts, thus invoking the $run\_pure\_testing$ goal with Times (=Size) as parameter, which starts the 'pushing' phase. The $pushing(Times)$ goal pushes an item (through $push\_item$ subgoal), when $Times > 0$, otherwise it ends.

$push\_item$ as first step retrieves the data structure $pqueue(Q)$. Then, it randomly generates an item, and checks if that item is already present in the queue. If so, then $push\_item$ fails, and this (and only this) item pushing is skipped. If the new item is not in the queue, then it is added in the head. The old queue is removed from memory ($retract(pqueue(Q))$), and the new one is stored in the knowledge base ($assert(pqueue(NQ))$. $popping$ is then invoked, to perform items removal from the queue. It makes use of pop_item, a goal that retrieves and unifies the queue through $clause(pqueue(Q), \_)$, and the extracts the head of the queue Q. After the 'popping' phase, the test ends.

```
run_pure_testE(Times):> run_pure_testing(Times).
run_pure_testing(Times):- pretty_start, now(StartTime), T1 is Times + 1,
                          nl, write('PUSHING...'), nl, pushing(T1),
                          nl, write('POPPING...'), nl, popping(T1),
                          now(EndTime), TestTiming is EndTime − StartTime,
                          nl, write('Time :'), write(TestTiming), nl, pretty_end.

pushing(0).
pushing(Times):-          push_item, T1 is Times − 1, pushing(T1).
push_item:-               clause(pqueue(Q), _),
                          random(1, 300, Item), not(exists_in_queue(Item, Q)),
                          nl, write('Pushing :'), write(Item),
                          append(Q, [Item], NQ), retract(pqueue(Q)),
                          assert(pqueue(NQ)), nl, write('Queue :'), write(NQ), nl.
push_item:-               assert(pqueue([])).
exists_in_queue(X, [X|_]):- true.
exists_in_queue(X, [_|Tail]):- exists_in_queue(X, Tail).


popping(0).
popping(Times):- pop_item, T1 is Times − 1, popping(T1).
pop_item:-       clause(pqueue(Q), _), nl, write('Popping :'),
                 Q = [H|T], write(H), retract(pqueue(Q)), assert(pqueue(T)),
                 nl, write('Queue :'), write(T), nl.
pop_item:-       assert(pqueue([])).
pretty_start:-   nl, write('Start testing...'), nl.
pretty_end:-     nl, write('Test finished...'), nl.
```

B DALI Code

The DALI implementation makes use of DALI advanced features, and in particular exploits actions, and the ability to remember what happened in the past (past actions). Basically, the power of the DALI infrastructure made us able to write the program in a very comfortable manner: each pushing is an action, so every time the action is performed, the DALI engine takes care, for the future, of this action as a past event. We exploited this feature so as to perform pushing actions in the present. Then, we use parameters of the action to take care of the items that have been pushed and of the order, through an index. When a pop needs to be performed, we use the power of DALI past events to retrieve actions previously performed, and thus retrieve the correct item from the head of the queue, in a very elegant manner. The DALI implementation allows us to concentrate on the problem, without focusing that much on the data structure, in a very strict declarative fashion.

Below we report the code of the proper version of the Queue agent implemented in DALI. The test agent gets active and performs a test session whenever it receives from the user a message with content $run\_dali\_test(Num\_Items)$ where Num_items specifies the number of elements to be pushed and popped on the queue.

```
run_dali_testE(Times):>  dali_test_startA, run_dali_testing(Times).
run_dali_testing(Times):< dali_test_startP.
run_dali_testing(Times):- dali_start_pushingA,
                          dali_pushing(Times), dali_end_pushingA,
                          dali_start_popA, dali_popping(Times), dali_end_popA,
                          dali_end_testingA.

dali_pushing(0):-         true.
dali_pushing(Times):-     dali_remember(Times), T1 is Times - 1, dali_pushing(T1).
dali_remember(Times):-    random(1, 300, Item),
                          not(clause(do_action(dali_push_queue(Item, _), _), _)),
                          get_push_index(PI),
                          dali_push_queueA(Item, PI).
get_push_index(I1):-      clause(push_index(Index), _),
                          I1 is Index + 1,
                          retract(push_index(Index)),
                          assert(push_index(I1)).
get_push_index(Index):-   assert(push_index(1)),
                          Index = 1.

get_pop_index(I1):-       clause(pop_index(Index), _),
                          I1 is Index + 1, retract(pop_index(Index))
                          assert(pop_index(I1)).
get_pop_index(Index):-    assert(pop_index(1)), Index = 1.
dali_popping(0):-         true.
dali_popping(Times):-     dali_forget(Times), V1 is Times, - 1, dali_popping(V1).
dali_forget(Dummy):-      get_pop_index(Index),
                          clause(do_action(dali_push_queue(Item, Index), _), _),
                          dali_pop_queueA(Item).
```

C Comparison

In Figure 1 below we show the execution time of the two solutions reported in previous sections at the growth of the instance size. It can be observed that as the instance size grows, the DALI solution becomes significantly better.

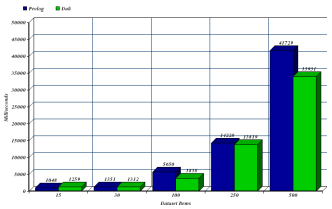

Fig. 1. x axis: instance size; y axis: execution time, blue bars pure prolog green bars DALI