Agent Modules and Dynamic Verification

Stefania Costantini, Pierangelo Dell’Acqua, Luís Moniz Pereira, Arianna Tocchio

Dip. di Informatica, Università di L’Aquila, Coppito 67010, L’Aquila, Italy

Dept. of Science and Technology - ITN, Linköping University, Norrköping, Sweden

Centro de Inteligência Artificial (CENTRIA), Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

RCRA 2010
Verification of Properties in Agents

- As agent systems are more widely used in real-world applications, the issue of verification is becoming increasingly important:
  - A priori verification: Model Checking & Theorem Proving
  - Dynamic verification: properties verified at runtime
- The two approaches are complementary rather than in competition
Verification of Properties in Agents

How to verify in agents that a property $\varphi$ holds?
Prove that $\varphi$ holds in any future state of the agent:

- One can verify $\varphi$ by explicitly examining all possible future states
- One can perform a run-time verification of $\varphi$
  and suitable counter-measures can be undertaken in case of violation
Motivations

- Given the evolving nature of learning agents, their behavior has to be checked from time to time and not only a priori.
- Model checking and other a priori approaches are static and need to re-check whenever the agent learns a new piece of information.
**Motivations**

- A priori full validation of agents’ behavior would have to consider all possible scenarios, but in most practical cases:
  - the actual arrival order of events is unforeseeable, and/or
  - the set of possible events is so large that computing all combinations would result in a **combinatorial explosion**

- These are the reasons why we propose a run-time control on agent behavior and evolution, for checking correctness during agents’ activity.
Past Work: I-METATEM logic

- Extends METATEM with intervals. Examples:
  - $U_{m,n}$: strong until in a time interval
  - $\hat{S}_{m,n}$: since in a time interval
Contextual A-IMETATEM rules with improvement

Example

\[ \text{goal}(\text{Goal}), \text{deadline}(\text{Goal}, T), \text{now}(T2), T2 \leq T \Rightarrow \]
\[ E(1, 12) \ (\text{not achieved}(\text{Goal}) \land \text{dropped}(\text{Goal}) \) : \]
\[ \text{inc\_comt}(T2) \]

If a goal not achieved is dropped sometime in the interval (1,12), then increase commitment level
Self-checking based upon agent History

- Agent’s behavior is affected by events perceived and in which order
- Actual arrival order of events is unforeseeable
- Large set of possible events: computing all combinations would result in a combinatorial explosion
- Therefore, other forms of dynamic self-checking required
Agent History: past events

- Each event or action $r$ recorded as $rP : T$, postfix $P$ standing for past, $T$ time-stamp

- Agent history $\langle P, PNV \rangle$ where $P$ last version of past events, $PNV$ “old” ones
Agent History: past events

- The agent re-elaboration of its experiences creates a particular view of the external world.
- $P$ constitutes a “snapshot” of the agent world as it is (present state of affairs)

- $PNV$ records of the agent world as it was in the past. Past events in $PNV$ can still play a role in reaching internal conclusions.
- Past events that are in $P$ will have to be moved into $PNV$ whenever overridden by most recent ones.
Agent History: past events

- When is an event $rP : T$ “overridden”?
- Trivially, whenever a more recent $rP : T_1$ arrives.
- But also, when something happens that “invalidates” $rP$.
- We propose Past Constraints to cope with this issue.
Sample Past Constraint:

\[ \text{food\_availableP} : T_1 \subseteq \text{eaten\_foodP} : T_2, \{T_2 > T_1\} \]

- \text{food\_availableP} : T_1 \text{ is removed from } P, \text{ i.e., it is no longer part of the agent’s current state of affairs, as soon as } \text{eaten\_foodP} : T_2 \text{ is recorded.}
- \text{food\_availableP} : T_1 \text{ is inserted into } PNV, \text{ and can be used in future reasoning.}
Past Constraints

- A Past Constraint has syntax:
  
  \[ X_{kP} : T_k, \ldots, X_{mP} : T_m \subseteq X_{sP} : T_s, \ldots, X_{zP} : T_z, \{C_1, \ldots, C_n\} \]

  where \( X_{kP} : T_k, \ldots, X_{mP} : T_m \) are the past events which are no longer valid whenever past events \( X_{sP} : T_s, \ldots, X_{zP} : T_z \) become known and conditions \( C_1, \ldots, C_n \) are true, i.e., as we will say, whenever the constraint holds.

- Given history \( H = \langle P, PNV \rangle \), set of past constraints \( PC \) and event \( X \), the result of \( H \star X \) is an updated history \( H' = \langle P', PNV' \rangle \) where: (i) \( P' = S \setminus F \) with \( S = H.P \cup \{X_P\} \) and \( F = PC(S) \); (ii) \( PNV' = H.PNV \cup F \).
Evolutionary LTL Expressions

- In order to cope with the many cases where agent’s evolution cannot be fully foreseen, we propose Evolutionary LTL Expressions.

- These temporal expressions take into account the events that have happened already and those that are expected to happen in the future and to be relevant to the property that we intend to check.
Evolutionary LTL Expressions

In many practical cases we are unable to provide a full sequence of the expected events, and sometimes we will be interested only in some of them.

Thus, in the definition of Evolutionary LTL Expressions, to be able to indicate the sets of past and future events in a more flexible way we admit the syntax of regular expressions with “wildcard” events.
Evolutionary LTL Expressions

- Evolutionary LTL Expressions are constraints, to be checked at run-time.
- Given events that have happened and events which are expected to happen, they establish actions to be undertaken so as to keep the agent’s state parameters under control.
Evolutionary LTL Expressions by examples

- There has been a supply of resource $r$ in the past, for quantity $s$.
- Consumption actions are expected to happen.
- No more consumption can take place if the available quantity of resource $r$ is scarce:

$$ X^+(\text{supply}_A(r, s)) $$
$$ N(\text{quantity}(r, V), V < th) X^+(\text{consume}_A^+(r, Q)) : \text{prevent}(\text{consume}_A(r, Q)) $$

- The distinguished predicate prevent to be implicitly added to the preconditions of every action
Evolutionary LTL Expressions by examples

- The available quantity of resource $r$ is scarce: self-limit consumption to small quantities

\[
X^+(supply_A(r, s)) \\
N(quantity(r, V), V < th + s) X^+(consume_A^+(r, Q)) : \\
allow(consume_A(r, Q), Q < th1)
\]

- The distinguished predicate allow should be a precondition of every action, that should be performed only if not prevented and allowed.
Conclusions

- Past Constraints and Evolutionary LTL Expressions have been implemented in the DALI language interpreter.

- Experiments are being performed to assess the trade-off between the overhead due to constraint-checking and the advantage gained from keeping the agent code “light”.

- In fact, constraints can be added to any agent program in an elaboration-tolerant fashion.
The End!

Thank you for your attention:)  
(questions by E-mail to the authors, please . . . )