Towards a Holistic Approach to Verification and Validation of Autonomous Cognitive Systems

ANGELO FERRANDO, LOUISE A. DENNIS, RAFAEL C. CARDOSO, and MICHAEL FISHER, The University of Manchester
DAVIDE ANCONA and VIVIANA MASCARDI, University of Genova

When applying formal verification to a system that interacts with the real world we must use a *model* of the environment. This model represents an *abstraction* of the actual environment, so it is necessarily incomplete and hence presents an issue for system verification. If the actual environment matches the model, then the verification is correct; however, if the environment falls outside the abstraction captured by the model, then we cannot guarantee that the system is well-behaved. A solution to this problem consists in exploiting the model of the environment used for statically verifying the system's behaviour and, if the verification succeeds, using it also for validating the model against the real environment via runtime verification. The paper discusses this approach and demonstrates its feasibility by presenting its implementation on top of a framework integrating the Agent Java PathFinder model checker. A high-level Domain Specific Language is used to model the environment in a user-friendly way; the latter is then compiled to trace expressions for both static formal verification and runtime verification. To evaluate our approach, we apply it to two different case studies, an autonomous cruise control system, and a simulation of the Mars Curiosity rover.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Runtime Verification, Model Checking, Autonomous Systems, Trace Expressions, MCAPL

ACM Reference Format:
Angelo Ferrando, Louise A. Dennis, Rafael C. Cardoso, Michael Fisher, Davide Ancona, and Viviana Mascardi. 2021. Towards a Holistic Approach to Verification and Validation of Autonomous Cognitive Systems. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2021), 44 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The development of autonomous systems to control physical objects such as cars, unmanned aircrafts or robots is undergoing a period of rapid development fuelled in part by advances in machine learning. Many such systems will operate in close proximity to humans, thus, these systems will need to be reliable and safe in their operations, and able to explain the decisions they make.

Satisfying these requirements can be difficult for systems that aim to pursue their goals successfully in dynamic and non-deterministic real-world environments. Such systems were named "autonomous cognitive systems" by Tjeerd

Authors' addresses: Angelo Ferrando, angelo.ferrando@manchester.ac.uk; Louise A. Dennis, louise.dennis@manchester.ac.uk; Rafael C. Cardoso, rafael.cardoso@manchester.ac.uk; Michael Fisher, michael.fisher@manchester.ac.uk, The University of Manchester, Manchester, United Kingdom, M13 9PL; Davide Ancona, davide.ancona@dibris.unige.it; Viviana Mascardi, viviana.mascardi@dibris.unige.it, University of Genova, Genova, Italy, 16146.

Andringa and Vincent C. Müller in a meeting of the 2nd European Network for Cognitive Systems, Robotics and Interaction[1] that took place in 2011.

According to Müller, "*the distinguishing feature of 'cognitive' artificial systems is not so much the use of higher level, traditionally called 'cognitive' features, but rather flexibility in pursuing the goals of the system*" [74]. However, he recognises that "*one way to achieve this flexibility is through higher level features*".

Safety, reliability and transparency have paramount importance for cognitive robotic systems too [72], based on a questionnaire developed as part of the RockEU2 Robotics Coordination Action[2], and answered by thirteen developers with industrial background and experience. The interviewed developers confirmed that "*cognitive robots will be able to operate reliably and safely around humans and they will be able to explain the decisions they make, the actions they have taken, and the actions they are about to take*" [94].

This paper addresses the issue of ensuring safety and reliability of autonomous cognitive systems by combining in a holistic way two techniques that are widely used for safety critical systems: model checking (MC [33, 34]) and runtime verification (RV [45, 46]). We contribute to moving a step forward in that direction in the following ways:

- mitigating the risk of making assumptions – during the model checking stage – on the environment where the system will operate, leading to a *structured abstraction* of the environment, by interleaving model checking with runtime verification;
- providing a Domain Specific Language, called Environment Assumptions for Autonomous Systems Language (EAASL), to model such assumptions in a user friendly way;
- providing a compiler from EAASL to a lower level formalism, trace expressions, that can be used both during the model checking and the runtime verification stages.

In this paper we use "autonomous system" [35], "cognitive agent" [96], and "autonomous cognitive systems" interchangeably, although in general they have different flavours. We also assume that the reasoning component of such agents is specified in a declarative way. An example of such agency is the well known Belief-Desire-Intention (BDI) [24, 86] model. BDI agents use these mental attitudes to reason and react to changes in the world. Although in this paper we validate our approach using a BDI language, it could also be applied to other similar autonomous cognitive systems.

Dennis et al. [40] advocate the use of model-checking for verifying declarative decision-making components within such autonomous systems. These components take input from sensors (potentially including images classifiers developed using machine learning) as *perceptions*, transform these into ground predicates (equivalent to propositions) which we will refer to as *environmental predicates* and then reason explicitly about such inputs in order to decide upon high-level courses of actions. This model-checking takes place statically in advance of system deployment. In order to reason over possible inputs the technique requires a model of the world. Dennis et al. recommend using the simplest environment model, in which any combination of the environment predicates that correspond to possible perceptions of the autonomous system is possible.

Consider an intelligent cruise control in an autonomous vehicle that perceives the environmental predicates safe, meaning it is safe to accelerate, at_speed_limit, meaning that the vehicle reached its speed limit, driver_brakes and driver_accelerates, meaning that the driver is braking/accelerating. During model-checking, each time the decision-making component would query its sensors for input, the model-checker instead generates all possible subsets of these

[1] http://www.eucognition.org/, accessed on December 2020.
[2] 2016-2018 RockEU2 - Robotics Coordination Action for Europe Two, https://www.eu-robotics.net/eurobotics/about/projects/rockeu2.html, accessed on December 2020.

four predicates. Each subset forms a branch of the state space to be explored during verification so that, ultimately, all possible combinations are verified.

This model is an *unstructured abstraction* of the world, as it makes no specific assumptions about the world behaviour and deals only with the possible incoming perceptions that the system may react to. Unstructured abstractions can lead to significant state space explosion. The state space explosion problem can be addressed by making assumptions about the environment. For instance, we might assume that the driver of a car will not both brake and accelerate at the same time. Therefore when we are statically verifying the agent, it should never be sent subsets of environmental predicates containing both driver_brakes and driver_accelerates as a possible input from perception on any path through the model as they do not correspond to situations that we believe that can happen in the actual environment. This *structured abstraction* of the world is grounded on assumptions that help prune the possible perceptions and hence control state space explosion. Structured abstractions have advantages over unstructured ones, provided that the assumptions they rely on are correct. Let us suppose that the cruise control system crashes if the driver is accelerating and braking at the same time. If the subsets of environmental predicates generated to verify it never contain both driver_brakes and driver_accelerates, then the static formal verification succeeds but if one real driver, for whatever reason, operates both the acceleration and brake pedals at the same time, the real system crashes.

In this paper, which extends our AAMAS'18 extended abstract [49] and RV'18 full paper [51], we propose an approach for exploiting the advantages of structured abstractions, while mitigating their risks. Our approach consists of modelling the structured abstraction in a formalism that can be used both for statically verifying the autonomous system's behaviour via model checking and for validating the model against the real environment by means of runtime verification. This RV could take place either during a testing phase in order to identify erroneous assumptions about the environment to be fed back to developers or after system deployment in conjunction with an appropriate failsafe system.

Differently from [49, 51], in this work the structured abstractions are defined using a novel Domain Specific Language (DSL) that we call EAASL. EAASL makes the definition of constraints more intuitive and easier to learn and maintain. As we are going to show, the introduction of a DSL did not change how our approach works, since it is compiled to the same formalism we used to generate the environment model and the runtime monitor. From an engineering perspective, we added an additional step of abstraction to make our approach more user-friendly, preserving the low-level implementation at the same time.

To demonstrate the feasibility of the proposed approach, we implemented it on top of the MCAPL framework developed by Dennis et al. [37, 41] (which provides a model-checker for cognitive agents) using trace expressions developed by Ancona et al. [5, 10, 11] as the formalism to generate both the environment model and the runtime monitor. We choose trace expressions instead of more widely used formalisms for model checking such as Linear Temporal Logic (LTL [80]) for three main reasons:

(1) we are familiar with the formalism, and this is relevant when choosing a development language or framework among many options that present similar features: being familiar with trace expressions helped us to develop the code described in this paper in less time and with less errors;
(2) trace expressions are supported by tools that allow system developers and testers to automatically generate runtime monitors, without writing any additional code: the ease of runtime monitor generation was a mandatory requirement for reaching our goal of implementing an integrated working approach to verification and validation, in an acceptable amount of time and with a limited effort;

(3) trace expressions are able to express and verify sets of traces that are context-free, being more expressive than LTL in the context of runtime verification [10]: while their expressive power has not been exploited in this paper, it opens up many possibilities in the future, as discussed in the conclusions.

The paper is organized as follows. Section 2 introduces the necessary background: the MCAPL framework, a running example, trace expressions, and the AJPF model checker. Section 3 presents EAASL, how a specification using such DSL can be compiled to a trace expression, and how trace expressions can be used to support both static and runtime verification inside MCAPL. Section 4 presents the experiments we carried out. Section 5 compares our proposal with related work and Section 6 concludes the paper and describes future work.

2 BACKGROUND AND RUNNING EXAMPLE

MCAPL: model checking BDI agents. The Belief-Desire-Intention (BDI) model, originally proposed by Bratman [24] as a philosophical theory of practical reasoning, inspired both architectures [87] and programming languages [23, 82, 85] for agents. BDI languages are based on *cognitive agency* [86]. Beliefs represent the agent's (possibly incorrect) information about its environment, desires represent the agent's long-term goals, and intentions represent the goals that the agent is actively pursuing. The MCAPL framework[3] [22, 37, 41] supports model checking of programs in BDI-style languages via the implementation of interpreters for those languages in Java. The framework implements *program model-checking* in which the *actual* program to be verified, not a model of it, is checked, and contains the Agent Java PathFinder (AJPF) model checker which customises the Java PathFinder[4] (JPF) model checker for Java bytecodes. We use the Engineering Autonomous Space Software (EASS) variant of Gwendolen [36], a language developed for programming agent-based autonomous systems and verifying them in AJPF. EASS assumes an architecture in which the cognitive agents are partnered with an *abstraction engine* that discretises continuous information from sensors in an explicit fashion [38, 39]. Model checking is used to demonstrate that the agent always tries to act in line with requirements and never *deliberately* chooses options that lead to states the agent believes to be unsafe.

Running Example: Autonomous Cruise Control. The (slightly simplified) EASS code in Example 1 is for an agent implementing intelligent cruise control in an autonomous vehicle. It uses standard syntactic conventions from BDI agent languages: +!g indicates the addition of a goal, g; +b indicates the addition of a belief, b; and −b indicates the removal of a belief. Plans follow the pattern trigger : guard ← body; with the trigger representing the addition of a goal or a belief (beliefs may be acquired via perceptions from the environment or as a result of internal deliberation); the guard states conditions about the agent's beliefs (or goals – not used in this example) which must be true for the plan to be selected for execution; and the body is a stack of *deeds* the agent performs in order to execute the plan. These deeds typically involve the addition or deletion of goals and beliefs, as well as *actions* (e.g. perf(accelerate), meaning "perform the action of accelerating") which indicate code delegated to non-rational parts of the system, such as low-level control of actuators or the environment model.

According to the operational semantics of Gwendolen [36], the agent moves through a *reasoning cycle* polling an external environment for perceptions; converting these into beliefs and creating intentions from new beliefs; selecting an intention for consideration; if the intention has no associated plan body, then the agent seeks a plan that matches the trigger event and places the body of this plan on the deed stack; the agent then processes the first deed, and places the intention at the end of the intention queue before again polling for perceptions. An intention may be suspended while

[3] https://github.com/mcapl/mcapl, accessed on December 2020.
[4] https://github.com/javapathfinder, accessed on December 2020.

it waits for some belief to become true. We use *b to indicate a deed that suspends processing of an intention until b is believed. Plan guards are evaluated using Prolog-style reasoning with *reasoning rules* of the form h :– body and literals drawn from agent's belief base. Negation is indicated with ~ and its semantics is negation by failure as in Prolog.

Example 1. *(Cruise Control Agent). When the car has an* initial goal *to be at the speed limit (line 8), +! at_speed_limit, it can accelerate if it believes it to be safe, that there are no incoming instructions from the human driver, and it does not already believe it is accelerating or is at the speed limit (line 11) — it does this by removing any belief that it is braking, adding a belief that it is accelerating, performing acceleration, then waiting until it no longer believes it is accelerating (line 12). If it does not believe it is safe, believes the driver is accelerating or braking, or believes it is already accelerating, then it waits for the situation to change (lines 13-16). If it believes it is at the speed limit, it maintains its speed having achieved its goal (which will be dropped automatically when it has been achieved).*

If new beliefs arrive from the environment that the car is at the speed limit (line 17), no longer at the speed limit (line 19), no longer safe (line 20), or the driver has accelerated or braked (line 22), then it reacts appropriately. Note that even if the driver is trying to accelerate, the agent only does so if it is safe.

```
:name:                                                                                  1
car                                                                                     2
                                                                                        3
:Reasoning Rules:                                                                       4
can_accelerate :- safe, ~ driver_accelerates, ~ driver_brakes;                          5
                                                                                        6
:Initial Goals:                                                                         7
at_speed_limit                                                                          8
                                                                                        9
:Plans:                                                                                 10
+! at_speed_limit: {can_accelerate, ~accelerating, ~at_speed_lim}                       11
    <- -braking, +accelerating, perf(accelerate), *~accelerating;                       12
+! at_speed_limit: {~safe} <- *safe;                                                    13
+! at_speed_limit: {driver_accelerates} <- *~driver_accelerates;                        14
+! at_speed_limit: {driver_brakes} <- *~driver_brakes;                                  15
+! at_speed_limit: {accelerating} <- *~accelerating;                                    16
+at_speed_lim: {can_accelerate, at_speed_lim}                                           17
    <- -accelerating, -braking, perf(maintain_speed);                                   18
-at_speed_lim: {~at_speed_lim} <- +! at_speed_limit;                                    19
-safe: {~driver_brakes, ~safe, ~braking} <- -accelerating, +braking,                    20
    perf(brake);                                                                        21
+driver_accelerates: {safe, ~driver_brakes, driver_accelerates, ~accelerating}         22
    <- +accelerating, -braking, perf(accelerate);                                       23
+driver_brakes: {driver_brakes, ~braking} <- +braking, -accelerating,                   24
    perf(brake);                                                                        25
```

The cruise control agent has to be connected to either a physical vehicle or a simulation. Similar EASS agents have been connected to both detailed simulations of ground vehicles and physical vehicles [38, 62]. Here we will consider embedding the agent within a multi-lane, multi-vehicle motorway (highway) simulation. The agent is connected to the simulator via a *Java environment* that communicates using sockets. The environment reads simulated speeds of the vehicles from the socket and publishes values for acceleration to the socket. The information from sensors is then passed on to an *abstraction engine* that converts it to discrete representations, shared with the agent as logical predicates. The agent accesses these *shared beliefs* as perceptions. Previously, the model of the combined behaviour of simulator, Java environment, and abstraction engine used for verification was unstructured: all the possible combinations of the

$$(\varepsilon\text{-empty})\frac{}{\varepsilon(\varepsilon)} \qquad (\varepsilon\text{-or-l})\frac{\varepsilon(\tau_1)}{\varepsilon(\tau_1 \vee \tau_2)} \qquad (\varepsilon\text{-or-r})\frac{\varepsilon(\tau_2)}{\varepsilon(\tau_1 \vee \tau_2)} \qquad (\varepsilon\text{-shuffle})\frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1 | \tau_2)}$$

$$(\varepsilon\text{-cat})\frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1 \cdot \tau_2)} \qquad (\varepsilon\text{-and})\frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1 \wedge \tau_2)} \qquad (\varepsilon\text{-cond})\frac{\varepsilon(\tau)}{\varepsilon(\vartheta \gg \tau)}$$

Fig. 1. Empty trace containment

$$(\text{prefix})\frac{}{\vartheta{:}\tau \xrightarrow{e} \tau} \; e \in \vartheta \qquad (\text{or-l})\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 \vee \tau_2 \xrightarrow{e} \tau_1'} \qquad (\text{or-r})\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \vee \tau_2 \xrightarrow{e} \tau_2'}$$

$$(\text{and})\frac{\tau_1 \xrightarrow{e} \tau_1' \quad \tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \wedge \tau_2 \xrightarrow{e} \tau_1' \wedge \tau_2'} \qquad (\text{shuffle-l})\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 | \tau_2 \xrightarrow{e} \tau_1' | \tau_2} \qquad (\text{shuffle-r})\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 | \tau_2 \xrightarrow{e} \tau_1 | \tau_2'}$$

$$(\text{cat-l})\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 \cdot \tau_2 \xrightarrow{e} \tau_1' \cdot \tau_2} \qquad (\text{cat-r})\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \cdot \tau_2 \xrightarrow{e} \tau_2'} \; \varepsilon(\tau_1) \qquad (\text{cond-t})\frac{\tau \xrightarrow{e} \tau'}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau'} \; e \in \vartheta \qquad (\text{cond-f})\frac{}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau} \; e \notin \vartheta$$

Fig. 2. Operational semantics of trace expressions

shared beliefs were explored. This is where our proposal for modeling structured abstractions as trace expressions and validating them via RV, as well as using them for model checking, can be applied (described in Section 3).

Trace expressions. Trace expressions are a specification formalism specifically designed for RV and constrain the ways in which a stream of events may occur. An *event trace* over a fixed universe of events $\mathcal{E}$ is a (possibly infinite) sequence of events from $\mathcal{E}$. The *juxtaposition*, $e\,u$, denotes the trace where e is the first event, and u is the rest of the trace. A trace expression denotes a set of event traces over $\mathcal{E}$. More generally, trace expressions are built on top of event types (chosen from a set $\mathcal{ET}$), rather than single events; an event type denotes a subset of $\mathcal{E}$. A *trace expression*, τ, represents a set of possibly infinite event traces, and is defined on top of the following operators:

- ε, the set containing only the empty event trace.
- $\vartheta{:}\tau$ (*prefix*), denoting the set of all traces whose first event e matches the event type ϑ ($e \in \vartheta$), and the remaining part is a trace of τ.
- $\tau_1 \cdot \tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of τ_1 with those of τ_2.
- $\tau_1 \wedge \tau_2$ (*intersection*), the intersection of traces τ_1 and τ_2.
- $\tau_1 \vee \tau_2$ (*union*), denoting the union of traces of τ_1 and τ_2.
- $\tau_1 | \tau_2$ (*shuffle*), denoting the union of the sets obtained by shuffling each trace of τ_1 with each trace of τ_2 (see [27] for a more precise definition).
- $\vartheta \gg \tau$ (*filter*), denoting the set of all traces such that their restrictions to events in ϑ are in τ.

Trace expressions can be easily represented as Prolog terms. To support recursion without introducing an explicit construct, trace expressions are regular terms which can be represented by a finite set of syntactic equations, as happens in most modern Prolog implementations where unification supports cyclic terms. The semantics of trace expressions is specified by the transition relation $\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}$, where $\mathcal{T}$ denotes the sets of trace expressions. As customary, we write $\tau_1 \xrightarrow{e} \tau_2$ to mean $(\tau_1, e, \tau_2) \in \delta$. If the trace expression τ_1 specifies the current valid state of the system, then an event e is valid *iff* there exists a transition $\tau_1 \xrightarrow{e} \tau_2$; in such a case, τ_2 specifies the next valid state of the system after event e. Otherwise, the event e is not valid in τ_1. The rules for the transition functions are reported in Figure 2. While

in Figure 1, we report the semantics of the empty containment predicate used to denote when a trace expression can terminate.

To demonstrate how trace expressions work, let us consider the example

$$\tau = (\vartheta_1{:}\tau_1) \vee (\vartheta_2{:}\tau)$$
$$\tau_1 = (\vartheta_3{:}\epsilon) \mid (\vartheta_4{:}\epsilon)$$

where $\vartheta_1 = \{e_1\}$, $\vartheta_2 = \{e_2\}$, $\vartheta_3 = \{e_3\}$, and $\vartheta_4 = \{e_4\}$. To make the example easier, event types $\vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4$ are singleton; in general, they can contain multiple events such as in $\vartheta_5 = \{e_{5.1}, e_{5.2}, e_{5.3}\}$ where $e_{5.1}, e_{5.2}, e_{5.3}$ all match ϑ_5. In the example, the trace expression defining the expected behaviour of the system w.r.t. observed events e_1, e_2, e_3, e_4 is τ; τ_1 is an auxiliary trace expression, as trace expressions may be defined in terms of other trace expressions. τ is composed by a union of two trace expressions; which means only one of the two operands can be selected at a time. On the left, if an event matching the event type ϑ_1 (namely, the event e_1 since $\vartheta_1 = \{e_1\}$) is observed – or "consumed", using a common terminology in the RV field [45] – τ moves to τ_1. On the right, by consuming an event matching ϑ_2, τ moves to τ again (τ is cyclic). The trace expression τ_1 is not cyclic, and is composed by a shuffle of two trace expressions; which means we do not enforce any order on the events matching ϑ_3 and ϑ_4 (namely e_3 and e_4, respectively). On the left, one event matching ϑ_3 can be consumed; while on the right, an event matching ϑ_4 can be consumed. Both trace expressions combined by the shuffle end in the terminal state ϵ, meaning that the trace can terminate there.

The trace expression τ recognises the language of event traces

$$\{\ e_1e_3e_4, e_1e_4e_3, \tag{1}$$
$$e_2e_1e_3e_4, e_2e_1e_4e_3, \tag{2}$$
$$e_2e_2e_1e_3e_4, e_2e_2e_1e_4e_3, \tag{3}$$
$$\dots$$
$$e_2^n e_1e_3e_4, e_2^n e_1e_4e_3, \tag{4}$$
$$e_2^\omega\ \} \tag{5}$$

The traces at line (1) derive from matching the left operand of the union in τ ($e_1 \in \vartheta_1$), and then the two operands of the shuffle in τ_1, in any order (in fact we have one trace when we first go left by consuming e_3, and the other one, when we first go right by consuming e_4 in the shuffle). The traces at line (2) derive from consuming the right operand in the union in τ ($e_2 \in \vartheta_2$), which brings us back to τ (since τ is cyclic). After that, the trace continues as in line (1). The traces at line (3) derive from an additional step, where e_2 is consumed twice by expanding the right operand of the union in τ. Since τ is cyclic, the event e_2 can be consumed any finite number of times n before concluding with the same sequence $e_1e_3e_4$ or $e_1e_4e_3$, at line (4). Finally, since the right operand of the union can be consumed infinitely times, we also have the infinite trace e_2^ω, containing only e_2 events, at line (5).

An example involving the $\cdot$ operator is the following

$$\tau_2 = (\vartheta_6{:}\epsilon) \vee (\vartheta_7{:}\epsilon){\cdot}\tau_2$$

where $\vartheta_6 = \{e_6\}$ and $\vartheta_7 = \{e_7\}$. Union has the precedence over concatenation, so if the first event to be consumed is e_6, the first branch of the union is picked and τ_2 moves into $\epsilon \cdot \tau_2$, namely τ_2 again. The same happens if the first event is e_7. Upon observation of the first event, that may be either e_6 or e_7, the trace expression transition mechanism starts again from τ_2. The set of traces described by τ_2 consists in all the infinite traces that contain e_6 and/or e_7, in any possible ordering.

In these artificial examples events and event types carry no meaning, but in the real setting they do. In a "polite communication" scenario we might define the $greet$ event type as $\{cheer, hello, good\_morning, good\_afternoon\}$, meaning that any observed (or heard) utterance among cheer, hello, etc, is considered as a valid polite greeting. In an "abstract data type" context we might define the $safe\_queue\_op$ event type as $\{enqueue, empty\}$, meaning that the enqueue and empty operations on a queue are safe and can be used whatever the state of the queue, and the $unsafe\_queue\_op$ event type as $\{dequeue, first\}$, which raises an exception if the queue is empty and is, hence, unsafe. These definitions would help us define a safe pattern of operations on queues, or a polite conversation protocol, where we do not need to stick to a specific operation or utterance, but we can group them according to features relevant for the specification of safe (resp. polite) traces of events.

In this paper we will define and use domain specific events, which are suitable in the context of verification of autonomous systems. More specifically the events will be beliefs, determined by perceptions generated by the environment, and actions, performed by an agent to interact with the environment. As an example, a variant of τ_2, $Bel_j = (bel(belief_j) : \epsilon) \vee (not\_bel(belief_j) : \epsilon) \cdot Bel_j$, will be introduced in Section 3.2.1.

A Prolog implementation exists which allows a system's developer to use trace expressions for RV by automatically building a trace expression-driven monitor able to both observe events taking place in the environment, and execute the δ transition rules. If the observed event is allowed in the current state – which is represented by a trace expression itself – it is consumed and the δ transition function generates a new trace expression representing the updated current state. If when observing an event no δ transition can be performed, the event is not allowed in the current state. In this situation an error is "thrown" by the monitor. When a system terminates, if the trace expression representing the current state can halt (formally meaning that it contains the empty trace), the RV process ends successfully; otherwise an error is again "thrown" since the system should not stop here.

The time and space complexity of the monitor synthesised by a trace expression depend on the kind of trace expression we defined. In this paper, the trace expressions that we are going to use allow monitoring the system in linear time in terms of the length of the event trace analysed. This can be intuitively understood by observing that a trace expression, representing the current state of a trace expression, is never rewritten to a trace expression with a bigger size; where the size of a trace expression is determined by counting the number of operators, subterms and cycles. Because of this, to check if a trace expression accepts a specific event in its current state requires only constant time, since the size of the term is limited to a maximum number of subterms to analyse (which does not change by increasing the size of the event trace). For further information we refer the reader to [10].

AJPF Static Formal Verification. The EASS implementation provides a Java class supporting the creation of abstract environment models. Models can be created by overriding, in a *subclass*, the add_random_beliefs method of this class which is called when the agent requests an action execution or sleeps. This method should return a set of beliefs. These are then added to the environment's *percept base* which the agent polls as part of its reasoning cycle. Implementations of unstructured environments will randomly generate all possible sub-sets of the beliefs relevant to the agent. For static verification, therefore, we want to generate this subclass from our trace expression. Once the

abstract environment is created, AJPF performs model checking of LTL properties extended with modalities for BDI concepts. In normal operation, EASS abstraction engines communicate with the agent-based reasoning engine (the 'agent') by performing assert_belief and remove_belief actions. These actions are implemented by Java environments which also connect to sensors and simulators. There are four such actions: assert_belief(b) asserts a belief for all agents in the system and remove_belief(b) removes belief b from all beliefs. assert_belief(a, b) and remove_belief(a, b) alter the available beliefs for a specific agent a. Our runtime monitor needs to observe these events and any action performed by an agent.

3 RECOGNISING ASSUMPTION VIOLATIONS

In this section we discuss how trace expressions can be suitably adopted for specifying structured abstractions of the real world for use in AJPF. The idea is to generate *both* a suitable Java model for AJPF model checking *and* a runtime monitor from the same trace expression. The monitor can detect if the real (or simulated) environment violates the assumptions used during the static verification. Figure 3 gives an overview of this system. A EAASL specification is compiled into a trace expression τ, which is then used to generate an abstract environment model in Java used to verify an agent (Reasoning Engine) in AJPF (the dashed box on the right of the Figure). Once this verification is successfully completed, the verified agent is used with an abstraction engine, a Java environment, and the real world or external simulator. This is shown in the dashed box on the left of the Figure. If, at any point, the monitor observes an inconsistent event, then the abstraction used during verification was incorrect. Depending on the development stage reached different measures are possible, ranging from refining the trace expression and re-executing the verification-validation steps, to involving a human or a failsafe system in the loop.

Event Types for AJPF Environments. We have identified the assertion and removal of beliefs and the performance of actions as the "events of interest" in our Java environments. Our runtime monitor receives notification of all actions in the environment as events. It is possible to flexibly create a number of different event types (remember that an event type is a set of events) on top of this structure: $bel(b)$ and $not\_bel(b)$ are singleton sets and model events involving beliefs. They are defined as $bel(b) = \{assert\_belief(b)\}$ and $not\_bel(b) = \{remove\_belief(b)\}$. In the rest of the paper we also denote these event types as *opposite*. Given $bel(b)$, its opposite is $not\_bel(b)$, and vice versa. We coalesce these as event set $\mathcal{E}_b$ and define event types $action(any\_action)$ where $e \in action(any\_action)$ iff $e \notin \mathcal{E}_b$; not_action where $e \in not\_action$ iff $e \in \mathcal{E}_b$; $action(A)$ where $e \in action(A)$ iff $e \notin \mathcal{E}_b$ and $e = A$; $not\_action(A)$ where $e \in not\_action(A)$ iff $e \in \mathcal{E}_b$ or $e \neq A$. We use a free variable A representing any possible action. $e \in \mathcal{E}_b$ and $e = A$ are mutually exclusive.

Representing Abstract Environment Models in AJPF. Abstract environment models in AJPF can be represented as automata where each state is represented as a set of environmental predicates which will be supplied to the agent when it polls for perception. The automaton states can be divided into two parts: *initial beliefs* and *beliefs that follow actions*. Initial Beliefs represent all the beliefs that may be asserted before the system starts executing. After an action is performed, more beliefs may be asserted. In the unstructured abstractions used by the "standard" AJPF system the initial beliefs, and the beliefs after each action, were generated at random. Essentially the automaton representing the abstract environment model consisted of a state for every possible subset of environmental predicates and each state could transition to any other state in the model. Transitions occurred each time the model was polled for perception and the model-checking process searched over all possible transitions. Any structured abstraction will be one that places constraints upon the possible transitions in the automaton. In order to capture constraints related to actions, we

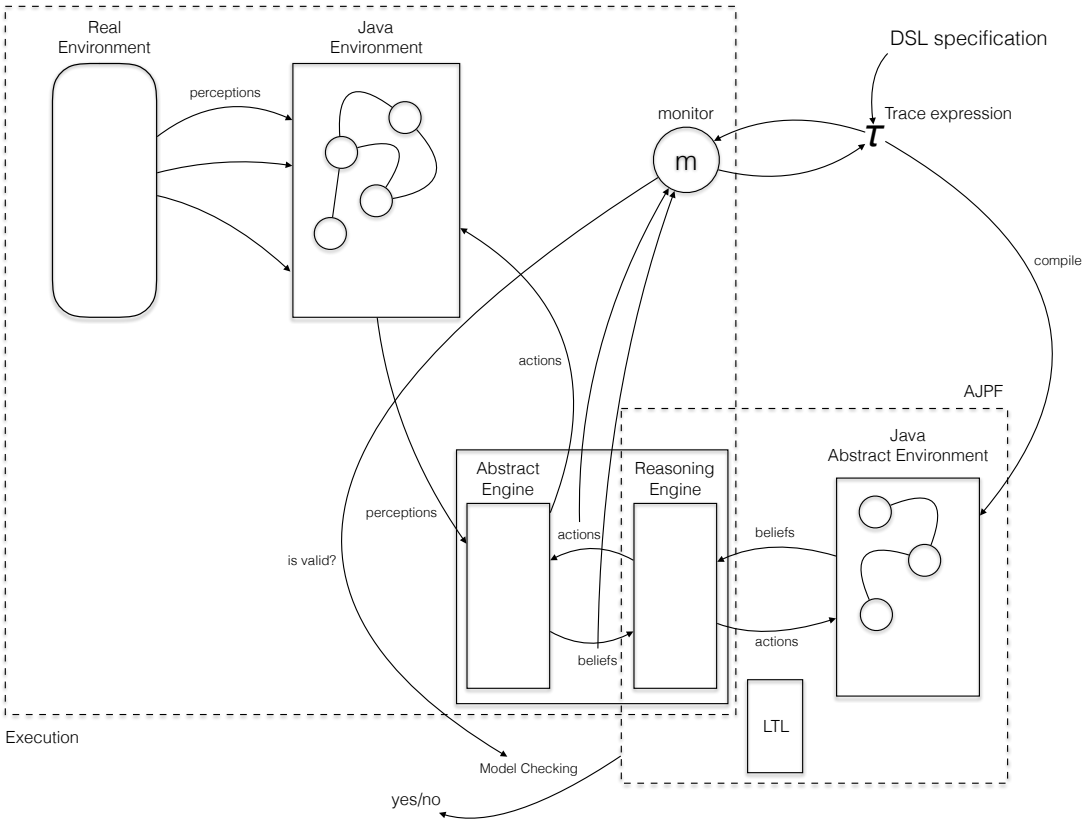

Fig. 3. General overview of the process.

also store the *last action performed by the agent* in each state of the abstract environment model (this is empty in the initial state).

3.1 EAASL, a Domain Specific Language to define Abstract Models

As mentioned before, by structuring the environment abstractions we can reduce the number of states to be analysed at static time by the model checker. In [51], we presented a first approach where we defined the abstract model using the trace expression formalism. As we already anticipated in the introduction, one of the reasons for the choice of this formalism is its ease of use to automatically generate runtime monitors. Nonetheless, its use as a formalism to represent the abstract model was not intuitive and difficult for non-expert users. As a result, we have added an intermediate step: a high-level formalism for specifying environment models in a user-friendly fashion. This high-level specification will then be automatically translated into the low-level formalism used for achieving both static and runtime verification of the agent – trace expressions.

The components of EAASL are:

3.1.1 Who is the agent involved in the verification? First we specify which agent is involved in the verification process by giving the agent's name:

```
agent:
    name
```

In Example 1, the agent is called `car`:

```
agent:
    car
```

3.1.2 Which beliefs are perceived by the environment? After defining the name of the agent we are interested in verifying, the next piece of information required is the set of beliefs the agent can gain through perceiving the environment. The set of beliefs is defined as follows:

```
beliefs:
    belief_1
    belief_2
    ...
    belief_n
```

These are the beliefs which are going to be perceived by the agent from the abstract model of the environment.

In Example 1, we have the following set of beliefs:

```
beliefs:
    safe
    driver_accelerates
    at_speed_limit
    driver_brakes
```

EAASL addresses the specification of structured abstractions of environments for autonomous cognitive systems, and for this reason has explicit statements for addition/removal of beliefs; it does not describe the *procedural behaviour* of an autonomous cognitive system, that is defined via plans and goals encapsulated within the agent's code, and not directly related with the external environment representation.

3.1.3 Which actions can the agent perform on the environment? Next, we specify the actions which can be performed by the agent in the abstract model of the environment. As with the beliefs, we report a set of the possible actions as follows:

```
actions:
    action_1
    action_2
    ...
    action_n
```

In this way, we can specify how the agent interact with the environment.

In Example 1, we could have the following set of actions:

```
actions:
    accelerate
    brake
```

3.1.4 How is the abstraction structured? The final step is the most important one. Until now we have just specified the agents involved, the beliefs perceived by the agent through the environment, and the actions. With this information, we could already define an abstract model of the environment, but without any constraint on how beliefs and actions interact. Practically speaking, the information provided so far only allows us to define an unstructured abstraction. But, what we are really interested in is structured abstractions, where we can reduce the number of possibilities by introducing constraints on how the beliefs and actions appear together.

When constraint. The first constraint we present is the *when* constraint that allows us to enforce when some beliefs can be generated.

We designed different versions of the constraint:

```
when agent believes belief_i it believes belief_j                    1
when agent believes belief_i it does not believe belief_j            2
when agent does not believe belief_i it believes belief_j            3
when agent does not believe belief_i it does not believe belief_j    4
```

The notation used to represent the *when* constraint is high-level and user-friendly. What the *when* constraint enforces is very intuitive. For example, in the first case (the others are similar) we say that the environment makes $belief_i$ perceptible to the agent only when $belief_j$ is already perceptible. Thanks to this constraint, we do not have to consider all the scenarios where $belief_i$ appears and $belief_j$ is not present. This constraint does not only concern the order in which beliefs are observed; in fact, it would not be enough to observe $belief_j$ before observing $belief_i$, but it would also be required that $belief_j$ keeps being satisfied as long as $belief_i$ is. The *when* constraint is used to restrict "when" a belief $belief_i$ can be observed; which in the first case is when $belief_j$ is currently satisfied[5]. In Example 1, we could have the following *when* constraint:

```
when car believes driver_accelerates it believes safe    1
```

This constraint states that a driver will only accelerate when it is safe. This constraint prevents the abstract environment from generating scenarios in which the car perceives that the driver is accelerating in an unsafe situation.

Before constraint. The second constraint we designed is *before*. With the *before* constraint we enforce the order of the appearance of beliefs/actions.

The constraint can assume different forms:

```
agent believes belief_i before believing belief_j                    1
agent believes belief_i before not believing belief_j                2
agent does not believe belief_i before believing belief_j            3
agent does not believe belief_i before not believing belief_j        4
agent performs action_i before believing belief_j                    5
agent performs action_i before not believing belief_j                6
```

Let us consider the first case. We say that *agent* has to perceive $belief_i$ before it can perceive $belief_j$. Thanks to this constraint, we can filter out all the scenarios where the environment sends the *agent* the perception $belief_j$ before $belief_i$. Note that, for this constraint, *not believing*, is interpreted as the removal of a belief/perception. That is, "*agent* believes $belief_i$ before not believing $belief_j$" is to be interpreted as "*agent* must perceive $belief_i$ before the perception that $belief_j$ holds is removed".

In Example 1, we could have the following *before* constraint:

[5]Which means being observed in the past, and not having observed its negation until now.

```
car believes driver_accelerates before believing at_speed_limit
```

This constraint states that the environment has to delay the generation of `at_speed_limit` until after the belief `driver_accelerates` is believed. Intuitively, we are just saying that to reach the speed limit, the car has to have accelerated.

In contrast to the *when* constraint, the *before* constraint cares only about the order of beliefs/actions. When we state that a belief $belief_i$ has to be believed before a belief $belief_j$, we only mean that $belief_i$ has to be believed some time before $belief_j$ is believed. After that $belief_i$ does not have to continue to be believed until $belief_j$ occurs.

Cause constraint. The last constraint is *cause* that enforces the direct effect of an action performed by the agent on the environment.

There are two versions of the constraint:

```
the action action_i causes agent to believe belief_j
the action action_i causes agent to not believe belief_j
```

Let us consider the first case: it means that an action $action_i$ performed on the environment by *agent* causes the generation of $belief_j$. Thanks to this constraint a cause-effect relation among the action $action_i$ and the belief $belief_j$ is created. This filters out all the scenarios where by performing $action_i$ a set of environmental predicates which does not contain $belief_j$ is generated.

In Example 1 there are no *cause* constraints; examples of its use will be introduced in Section 4.

Before going on with the presentation of how to compile EAASL into trace expressions, we would like to focus on the principles that drove the design of EAASL. As we have shown in this section, EAASL is focused on the definition of constraints in the context of autonomous cognitive systems. This can be noticed by looking at the notion of *agents* and *beliefs* which is widely used in EAASL. With respect to other DSLs in literature (see Section 6 for more details), EAASL was conceived to represent constraints on autonomous cognitive systems. The constraints we can define restrain when an agent can believe in something, how the agent's beliefs are related to each other, and finally, what is the cause–effect relation between the actuation of an action and its outcome in the agent's mind. As we are going to see, all these constraints are necessary to help us in limiting the state space derived by what an agent can do inside the environment.

3.2 Compiling EAASL specifications to Trace expressions

Starting from a configuration file following the syntax presented in Section 3.1, we developed a compiler (in SWI-Prolog[6]) which automatically generates the corresponding trace expression. We want to represent the abstract model of the real world as a set of possibly cyclic trace expressions modelled in Prolog, which are going to be compiled from EAASL. We use regular expression syntax: as parentheses are used for grouping in trace expressions, we adopt [and] to represent groupings within a regular expression; similarly, since | is a trace expression operator, we use || to indicate alternatives within the regular expression. Here, e? indicates zero or one occurrences of the element e. As we use Prolog, variables are represented by terms starting with an upper case letter (e.g., $Action_i$) and constants are represented by terms starting with a lower case letter (e.g., b_i, $action_i$). $\Big|_{i=1}^{n}$ indicates one or more trace expressions composed via the trace expression shuffle operator, |. Similarly, $\bigvee_{i=1}^{n}$ composes expressions using $\vee$ and $\bigwedge_{i=1}^{n}$ composes expressions using $\wedge$. Variables with the same name will be unified.

[6] https://www.swi-prolog.org, accessed on December 2020.

We now show how each part of the EAASL specification can be mapped to its trace expression representation. At the end of the section, we are going to show how all these parts are then combined together to create the final trace expression corresponding to the abstract model of the environment

3.2.1 Beliefs and Actions. Considering a generic list of beliefs, $belief_1, \ldots, belief_n$, as shown in Section 3.1.2, the compiler automatically generates the following trace expression:

$$Beliefs = \Big|_{i=1}^{n} Bel_i \tag{6}$$
$$Bel_1 = (bel(belief_1) : \epsilon) \vee (not\_bel(belief_1) : \epsilon) \cdot Bel_1 \tag{7}$$
$$\ldots \tag{8}$$
$$Bel_n = (bel(belief_n) : \epsilon) \vee (not\_bel(belief_n) : \epsilon) \cdot Bel_n \tag{9}$$

Fig. 4. Trace expressions for beliefs.

$$Beliefs = (Safe \mid DriverAccelerates \mid AtSpeedLimit \mid DriverBrakes) \tag{10}$$
$$Safe = ((bel(safe){:}\epsilon) \vee (not\_bel(safe){:}\epsilon)) \cdot Safe \tag{11}$$
$$DriverAccelerates = ((bel(driver\_accelerates){:}\epsilon) \vee (not\_bel(driver\_accelerates){:}\epsilon)) \cdot DriverAccelerates \tag{12}$$
$$AtSpeedLimit = ((bel(at\_speed\_limit){:}\epsilon) \vee (not\_bel(at\_speed\_limit){:}\epsilon)) \cdot AtSpeedLimit \tag{13}$$
$$DriverBrakes = ((bel(driver\_brakes){:}\epsilon) \vee (not\_bel(driver\_brakes){:}\epsilon)) \cdot DriverBrakes \tag{14}$$

Fig. 5. Trace expressions for the beliefs in the running example.

In Figure 4, at line 6 the compiler creates the trace expression through a shuffle of n different trace expressions each representing the events of adding or removing one of the beliefs in the **beliefs:** part of the EAASL specification; for each $belief_j$ there, one such "add or remove" pattern $Bel_j = (bel(belief_j) : \epsilon) \vee (not\_bel(belief_j) : \epsilon) \cdot Bel_j$ is generated into the compiled trace expression, and then they are all combined using a shuffle operator because we do not care about the order. To note that the resulting trace expression is cyclic and not terminating and – because of this – any infinite trace involving events matching $bel(belief_j)$ and $not\_bel(belief_j)$, in any order, is valid w.r.t. it. We used the shuffle operator in here because we do not care about the order on which the beliefs are actually generated and we want to consider all the possible interleavings. The lines from 7 to 9 denote the addition/removal of beliefs $belief_1$ to $belief_n$. Any given belief, $belief_i$ may appear in the set of perceptions sent to the belief base ($bel(belief_i)$), or disappear ($not\_bel(belief_i)$). In this trace expression, all the beliefs listed are valid appearing/disappearing in any possible order.

Considering the four beliefs listed in Example 1, we generate the trace expression reported in Figure 5 for the running example. The beliefs used in the Example 1 are mapped into four different terms (line 10) that are independent from one another and combined using the shuffle operator. In each term the occurrence of the corresponding belief is handled. For instance, in line 11, we may find the definition of the term handling the *safe* belief, which is where the addition/removal

of the belief can be observed. The other beliefs are defined following the same principle (lines 12-14). The constraints over the beliefs are going to added later on depending on the constraints inserted in EAASL.

The compilation of the set of actions in EAASL, $action_1, \ldots, action_n$, generates the trace expression in Figure 6:

$$Actions = \Big|_{i=1}^{n} Act_i \tag{15}$$
$$Act_1 = (action(action_1) : Act_1) \tag{16}$$
$$\ldots \tag{17}$$
$$Act_n = (action(action_n) : Act_n) \tag{18}$$

Fig. 6. Trace expressions for actions.

The trace expression that is obtained allows the generation of actions in any possible order and with no constraints.

Considering the two actions listed in Example 1, we generate the trace expression reported in Figure 7 for the running example. At line 19 we can find the term representing the possible combinations of the actions used in Example 1. Since their order is not relevant (at this level), the combination is obtained using the shuffle operator. The two terms defined at line 20 and 21 handle the occurrence of the two actions *accelerate* and *brake*.

$$Actions = (Accelerate \mid Brake) \tag{19}$$
$$Accelerate = action(accelerate){:}Accelerate \tag{20}$$
$$Brake = action(brake){:}Brake \tag{21}$$

Fig. 7. Trace expressions for the actions in the running example.

3.2.2 The when constraint. The *when* constraint forces two beliefs to be (or not) satisfied at the same time in the abstract environment. Which means that a belief can be added/removed only when the other belief has already been added/removed, but the two events of adding/removing the two beliefs are separated and happen at different time. In fact, trace expressions semantics is based on sequence of events (not sets). Because of this, the trace expression resulting from a *when* constraint has to keep track of the occurrences of addition/removal of the two beliefs in order to check if each addition/removal is consistent with the current agent's state. For instance, if a belief b_1 can be satisfied only **when** a belief b_2 is satisfied, this is translated into a trace expressions which has to keep track of addition/removal of belief b_2, because as soon as b_1 is added, we need to know that b_2 has been previously added, and it has not been removed since.

Considering the general case introduced in Section 3.1.4, we generate the trace expression in Figure 8.

Supposing we have specified n *when* constraints, in Figure 8 $B_{j,1}$ and $B_{j,2}$ are set according to which pattern of the *when* constraint is used. For instance, if the pattern selected is the first one:

when *agent* **believes** $belief_1$ it **believes** $belief_2$ 1

we would have $B_{j,1} = bel(belief_1)$, $NB_{j,1} = not\_bel(belief_1)$, $B_{j,2} = bel(belief_2)$ and $NB_{j,2} = not\_bel(belief_2)$. Intuitively, this constraint states that every time *agent* believes *belief*, we have that *agent* also believes *belief'*. If the pattern was instead:

when *agent* **does not believe** $belief_1$ it **believes** $belief_2$ 1

$$Whens = \bigwedge_{j=1}^{n} FilterEventType_j \gg [When_j^1 \parallel When_j^4 \parallel When_j^7] \tag{22}$$

$$When_j^1 = ((((B_{j,1}{:}\epsilon) \vee (B_{j,2}{:}\epsilon)) \cdot When_j^1) \vee (NB_{j,1}{:}When_j^2)) \tag{23}$$

$$When_j^2 = ((((B_{j,2}{:}\epsilon) \vee (NB_{j,1}{:}\epsilon)) \cdot When_j^2) \vee (NB_{j,2}{:}When_j^3) \vee (B_{j,1}{:}When_j^1)) \tag{24}$$

$$When_j^3 = ((B_{j,2}{:}When_j^2) \vee (((NB_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot When_j^3)) \tag{25}$$

$$When_j^4 = ((B_{j,1}{:}When_j^5) \vee (B_{j,2}{:}When_j^4) \vee (NB_{j,2}{:}When_j^6)) \tag{26}$$

$$When_j^5 = ((((B_{j,1}{:}\epsilon) \vee (B_{j,2}{:}\epsilon)) \cdot When_j^5) \vee (NB_{j,1}{:}When_j^4)) \tag{27}$$

$$When_j^6 = ((B_{j,2}{:}When_j^4) \vee (((NB_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot When_j^6)) \tag{28}$$

$$When_j^7 = ((B_{j,2}{:}When_j^8) \vee (((NB_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot When_j^7)) \tag{29}$$

$$When_j^8 = ((B_{j,1}{:}When_j^9) \vee (NB_{j,2}{:}When_j^8) \vee (((B_{j,2}{:}\epsilon) \vee (NB_{j,1}{:}\epsilon)) \cdot When_j^8)) \tag{30}$$

$$When_j^9 = ((((B_{j,1}{:}\epsilon) \vee (B_{j,2}{:}\epsilon)) \cdot When_j^9) \vee (NB_{j,1}{:}When_j^8)) \tag{31}$$

Fig. 8. Trace expressions for the *when* constraint: $B_{j,i}$ must be the "opposite operation" of $NB_{j,i}$.

we would have $B_{j,1} = not\_bel(belief_1)$, $NB_{j,1} = bel(belief_1)$, $B_{j,2} = bel(belief_2)$ and $NB_{j,2} = not\_bel(belief_2)$. Intuitively, this constraint states that every time *agent* does not believe *belief*, we have that *agent* believes *belief'*. And so on for the other patterns.

$B_{j,i}$ and $NB_{j,i}$ are event types, and they must meet the condition that if $B_{j,i} = bel(b_{j,i})$ then $NB_{j,i} = not\_bel(b_{j,i})$ and vice versa. $FilterEventType_j$ is an event type which denotes only the events involved in $When_j^x$. Its purpose is to filter out any events that are not constrained by $When_j^x$, and matches $bel(b_{j,1})$, $not\_bel(b_{j,1})$, $bel(b_{j,2})$ and $not\_bel(b_{j,2})$. It ensures that the trace expression can move to the next state without getting stuck.

The equations from (23) to (31) capture the *when* constraint that $B_{j,1}$ has to occur only when $B_{j,2}$ is satisfied. The constraint either starts in the state described by $When_j^1$, $When_j^4$ or $When_j^7$ depending upon whether both the constrained belief events are satisfied in the initial state ($When_j^1$), only $B_{j,2}$ is ($When_j^4$), or none of them are ($When_j^7$). These information have to be passed to the compilation process in order to correctly translate a EAASL *when* constraint to its corresponding trace expression. Of course, the case where only $B_{j,1}$ and not $B_{j,2}$ is satisfied in the initial state is forbidden because it violated the *when* constraint. This aspect will become clearer when we present the final trace expression deriving from the combination of all the constraints.

Each equation represents a state of the formal specification where we know if $B_{j,1}$ or $B_{j,2}$ are satisfied or not (having the opposite for their counterparts $NB_{j,1}$ and $NB_{j,2}$). Since we are talking about beliefs, we can start in three different initial scenarios, depending on which beliefs are initially satisfied. If $B_{j,1}$ and $B_{j,2}$ are both initially satisfied, we start in $When_j^1$. We stay in such state as long as the two beliefs keep being satisfied, because their satisfaction does not change the knowledge we have of the system. We move to $When_j^2$ only when we know that $B_{j,1}$ is not satisfied anymore ($NB_{j,1}$ has been observed). It is important to notice that we cannot observe $NB_{j,2}$ in $When_j^1$, because it would be a violation of the *when* constraint. In fact, by letting $NB_{j,2}$ to be observed in $When_j^1$, we would be in a situation where $B_{j,1}$ is satisfied, but $B_{j,2}$ is not. After moving to state $When_j^2$, we know that $B_{j,1}$ is not satisfied, and $B_{j,2}$ is. As before, if we observe $B_{j,2}$ or $NB_{j,1}$ we do not change state, because these information do not change our knowledge of the system, since we already know that. If we observe $NB_{j,2}$, we move to $When_j^3$, which represents the state where neither $B_{j,1}$ nor

$$
\begin{aligned}
When_1 &= (bel(safe) : When_2) \vee \\
&\quad ((not\_bel(driver\_accelerates){:}\epsilon) \vee (not\_bel(safe){:}\epsilon)) \cdot When_1 && (32)\\
When_2 &= (bel(driver\_accelerates){:}When_3) \vee (not\_bel(safe){:}When_1) \\
&\quad \vee ((bel(safe){:}\epsilon) \vee (not\_bel(driver\_accelerates){:}\epsilon)) \cdot When_2 && (33)\\
When_3 &= ((bel(driver\_accelerates){:}\epsilon) \vee (bel(safe){:}\epsilon)) \cdot When_3 \\
&\quad \vee (not\_bel(driver\_accelerates){:}When_2) && (34)
\end{aligned}
$$

Fig. 9. Trace expressions for the *when* constraint in the running example.

$B_{j,2}$ are satisfied. Finally, if we observe $B_{j,1}$, from $When_j^2$ we go back to $When_j^1$, because by being in $When_j^2$ we infer that $B_{j,2}$ is still satisfied, and by observing $B_{j,1}$, we conclude that both are satisfied; which is the scenario represented by $When_j^1$. Differently from $When_j^1$, there is no belief which is not allowed in $When_j^2$. This is due to the fact that $B_{j,2}$ is independent from $B_{j,1}$ satisfaction[7]. After moving to $When_j^3$ instead, we know that both $B_{j,1}$ and $B_{j,2}$ are not satisfied. In such state, if we observe $B_{j,2}$, we can go back to $When_j^2$, where we represent the state where only $B_{j,2}$ is satisfied; otherwise, by observing $NB_{j,1}$ and $NB_{j,2}$, we stay in the same state, since nothing new happened. Similarly to $When_j^1$, also in $When_j^3$ we have a belief which is not allowed to be observed, $B_{j,1}$. This depends again on the *when* constraint, since both $When_j^1$ and $When_j^3$ have $B_{j,2}$ not satisfied. The same reasoning can be followed to explain the construction of $When_j^i$, with $4 \leq i \leq 9$. The only difference is the knowledge we have in the initial states.

Considering our running example (Example 1), we reported in Figure 9 the trace expressions corresponding to the *when* constraint[8] showed in Section 3.1.4. In Figure 9, we have $When_1$ as initial state because we are assuming that the *driver_accelerates* and *safe* beliefs are not initially satisfied; $When_1$ can be traced back to $When_j^7$ in Figure 8, which denotes the case when $B_{j,1}$ and $B_{j,2}$ are both initially not satisfied. As we mentioned previously, we may have multiple scenarios where different beliefs are initially satisfied; because of this, we need a way to represent such information inside the formal specification. This is obtained following the template presented in Figure 8, where depending on which pattern is followed, it is possible to customise the trace expression to correctly handle the beliefs' addition/removal.

3.2.3 The before constraint. The *before* constraint (Figure 10) enforces the order between two beliefs. Its trace expression representation is simpler than the one necessary for the *when* constraint, because we do not need to check if the two beliefs are satisfied at the same time, but only if one of the two appeared before the other. In the following trace expression we use $FilterEventType_j$ as before to filter out the events which are not of interest for the constraint. Supposing we have specified n *before* constraints, the trace expression corresponding to the each constraint forbids the appearance of the second belief, as long as the first one has not yet been observed. After the first belief has been observed, we can move to *All* where any belief is accepted; this is due to the fact that once the first belief is satisfied, the constraint is satisfied by default, because in case $B_{j,2}$ will be observed in the future, we know we have already observed $B_{j,1}$ before.

Considering our running example (Example 1), we reported in Figure 11 the trace expressions corresponding to the *before* constraint[9] showed in Section 3.1.4.

[7] The constraint is over $B_{j,1}$, there are no restrictions for $B_{j,2}$.
[8] **when** car **believes** *driver_accelerates* it **believes** *safe*.
[9] car **believes** *driver_accelerates* **before believing** *at_speed_limit*.

$$Befores = \bigwedge_{j=1}^{n} FilterEventType_j \gg [Before_j^1 \parallel Before_j^2] \tag{35}$$

$$Before_j^1 = (B_{j,1}{:}All^1) \vee (((NB_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot Before_j^1) \tag{36}$$

$$Before_j^2 = (Act_{j,1}{:}All^2) \vee ((NAct_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon) \cdot Before_j^2) \tag{37}$$

$$All^1 = ((B_{j,1}{:}\epsilon) \vee (NB_{j,1}{:}\epsilon) \vee (B_{j,2}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot All^1 \tag{38}$$

$$All^2 = ((Act_{j,1}{:}\epsilon) \vee (NAct_{j,1}{:}\epsilon) \vee (B_{j,2}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot All^1 \tag{39}$$

$$\tag{40}$$

Fig. 10. Trace expressions for the *before* constraint : $B_{j,i}$ must be the "opposite operation" of $NB_{j,i}$, and $NAct_{j,1}$ accepts every action, except $Act_{j,1}$.

$$\begin{aligned} Before = {} & (bel(driver\_accelerates){:}All) \vee ((not\_bel(driver\_accelerates){:}\epsilon) \vee \\ & (not\_bel(at\_speed\_limit){:}\epsilon)) \cdot Before \end{aligned} \tag{41}$$

$$\begin{aligned} All = {} & ((bel(driver\_accelerates){:}\epsilon) \vee (not\_bel(driver\_accelerates){:}\epsilon) \\ & \vee (bel(at\_speed\_limit){:}\epsilon) \vee (not\_bel(at\_speed\_limit){:}\epsilon)) \cdot All \end{aligned} \tag{42}$$

Fig. 11. Trace expressions for running example's *before* constraint.

3.2.4 The cause constraint. The *cause* constraint (Figure 12) creates a cause-effect relation between an action and a belief. With this constraint we can enforce that each time an action is performed by the agent on the environment, a specific belief must be generated.

Supposing we have specified *n cause* constraints, the corresponding trace expression generated by compiling the *cause* constraint is as follows:

$$Causes = \bigwedge_{j=1}^{n} FilterEventType_j \gg [Cause_j^1] \tag{43}$$

$$Cause_j^1 = ((Act_{j,1}{:}Cause_j^2) \vee (NB_{j,1}{:}Cause_j^1) \vee (NAct_{j,1}{:}Cause_j^1)) \tag{44}$$

$$Cause_j^2 = (B_{j,1}{:}Cause_j^1) \tag{45}$$

Fig. 12. Trace expressions for *cause* constraint: $B_{j,i}$ must be the "opposite operation" of $NB_{j,i}$, and $NAct_{j,1}$ accepts every action, except $Act_{j,1}$.

The trace expression for the *cause* constraint is very intuitive; we start in a state where as long as we observe actions different from $Act_{j,1}$, we stay in the same state. But, upon reception of $Act_{j,1}$, the only possible next observation is the perception $B_{j,1}$, and nothing else. For instance, to observe another action instead of $B_{j,1}$ is forbidden.

Before moving on to the next section, where we will focus on how a trace expression can be compiled into a Java abstract model, it is important to discuss the kind of trace expressions that are generated by the EAASL compilation. One could question the difference between writing a trace expression and generating a trace expression from an EAASL specification. Specifically, we expect the trace expressions automatically generated from an EAASL specification to be more compact (less verbose) and less inclined to contain errors than the trace expressions directly written by the user.

$$Protocol = (Cyclic\ [\ \wedge Whens]?\ [\ \wedge Befores]?\ [\ \wedge Causes]?) \tag{46}$$
$$Cyclic = Actions \mid Beliefs \tag{47}$$
$$Actions = [\text{defined in (15)}] \tag{48}$$
$$Beliefs = [\text{defined in (6)}] \tag{49}$$
$$Whens = [\text{defined in (22)}] \tag{50}$$
$$Befores = [\text{defined in (35)}] \tag{51}$$
$$Causes = [\text{defined in (43)}] \tag{52}$$

Fig. 13. Trace expression template for generating abstract environments.

This claim is supported by the fact that when we generate a trace expression starting from an EAASL specification, we only create the terms that are needed, and nothing more. For each constraint in EAASL, we show how to extract its corresponding trace expression. No additional terms are created in such translation if not explicitly required at EAASL level. The same level of compactness can be achieved by directly writing the trace expression, but it is prone to adding errors (unwanted behaviours) and useless constraints (bigger terms to represent the same notion). A complete comparison between directly written and automatically generated trace expressions is however beyond the scope of this paper.

3.3 Combining all trace expressions into the global abstract model

Now that we have all the individual parts of the trace expression, the final step is the combination of the separate trace expressions into the global model that represents the abstract environment, as shown in Figure 13. The final protocol can be generated by combining the three classes of constraints, the beliefs and the actions trace expressions obtained as described in the previous section.

Cyclic (47) is the trace expression that describes the behaviour once the agent starts interacting with the model. It consists in a shuffle between the possible beliefs and actions that can be observed. As in the configuration step, if we stop right now we would have just an unstructured model. Each cycle, permissible beliefs and actions as defined by the trace expression constraints can be observed or generated (depending on the use context). The presence at the *Protocol* level of the three different constraints allows us to constrain the order of the events for each cycle.

3.4 Generating the Abstract Model

Once we have automatically generated a trace expression, we translate it into Java by implementing add_random_beliefs. We omit the involved low level details (e.g. constructing appropriate class and package names) but just focus on the core aspects[10]. Our trace expression is defined according to the template in Figure 13. Some parts of these trace expressions are not directly translated into Java; the sub-expressions relevant to the generation of abstract models are *Cyclic* (47), *Whens* (22), *Befores* (35) and *Causes* (43). Note that the MCAPL framework provides support for constructing logical predicates and adding them to the belief base.

Cyclic contains a shuffle of trace expressions involving the actions and beliefs of our interest. Regarding the beliefs, $Beliefs = |_{i=1}^{k}\ Bel_i$, with $Bel_i = (bel(belief_i):\epsilon) \vee (not\_bel(belief_i):\epsilon) \cdot Bel_i$, defines the set of belief events that may occur.

[10]Full source code can be found in the MCAPL distribution: https://github.com/mcapl/mcapl. Code for the examples is also available from the University of Liverpool data catalogue together with experimental data – https://www.doi.org/10.17638/datacat.liverpool.ac.uk/438

We define the set $\mathcal{B}(Beliefs)$ as $b_i \in \mathcal{B}(Beliefs)$ iff $(bel(b_i) \vee not\_bel(b_i) \vee \epsilon)$ is one of the interleaved trace expressions in *Beliefs*. For each $b_i \in \mathcal{B}(Beliefs)$ we define a predicate in the environment class and bind it to a Java field called b_i.

Whens is used to filter out from the set of all possible perception events, the set violating the *when* constraints.

We construct a set of belief events satisfying the *Whens* constraint, $\mathcal{M}_x(Whens)$, which contains for all the conjuncts in *Whens* the corresponding tuple $(B_{j,1}, B_{j,2})$.

The set of possible sets of belief events for our structured environment is:

$$\mathcal{PB}(Beliefs, Whens) = \{S \mid (\forall b_i \in \mathcal{B}(Beliefs).\ bel(b_i) \in S \vee not\_bel(b_i) \in S) \wedge (\forall (B_1, B_2) \in \mathcal{M}_x(Whens).\ B_1 \in S \implies B_2 \in S)\} \quad (53)$$

Say that $\mathcal{PB}(Beliefs, Whens)$ contains k sets of belief events, $S_j,\ 0 \leq j < k$. We generate a random integer between 1 and k:

```
int assert_random_int = random_int_generator(k);
```

where random_int_generator is a special method that generates random integers in a way that supports a number of AJPF features (in particular the ability to replay counter-example traces generated during model-checking) and for each S_j we generate a *if clause* in the code which adds S_j as a perception

```
if (assert_random_int == j) { add_percepts(S_j) }
```

Here $add\_percepts(S_j)$ adds b_i to the percept base for each $bel(b_i) \in S_j$. We do not need to handle the belief removal events, $not\_bel(b_i) \in S_j$, because AJPF automatically removes all percepts before calling add_random_beliefs.

The if statements that are generated allow us to represent the set of beliefs satisfying the *Whens* constraints. The other constraints we may have in the trace expression can modify these statements. More specifically, for each *Causes* constraint denoting that an action *action* causes a belief b, we update all the if statements having S_j not containing b in the following way:

```
if (assert_random_int == j && !act.getFunctor().equals(action)){
    add_percepts(S_j)
}
```

where `act.getFunctor()` gives the name of the last action performed.

In this way, every time a belief set does not contain b, we require the action to be different from *action*, because otherwise would be a violation of the *Causes* constraint.

The *Befores* constraint is slightly more complex. With it we can constrain the order of beliefs. Even though it is a weaker limitation than the one forced by *Whens*, it requires us to keep track of previous information. For each *Befores* constraint saying that a belief b_1 has to happen before a belief b_2; we modify all the if statements having S_j containing b_1, as follows:

```
if (assert_random_int == j){
    add_percepts(S_j)
    b1_observed = true;
}
```

And we modify all the if statements having S_j containing b_2 as follows:

```
if (assert_random_int == j && b1_observed){
    add_percepts(S_j)
}
```

In this way, we force the order on the two beliefs, since we cannot generate any set of beliefs containing $b2$, as long as we have not already observed $b1$. This translation process from trace expressions to Java is fully implemented in SWI-Prolog.

3.5 Verifying MCAPL at Runtime

The same trace expression obtained from the EAASL specification is used to synthesise a monitor to validate at runtime the consistence of the real environment with the abstracted one.

Since the MCAPL framework is implemented in Java, its integration with the trace expressions runtime verification engine or "monitor" (namely, the Prolog engine that "executes" the δ transition function, briefly introduced in Section 2) was done using the JPL interface[11], between Java and Prolog. We use pre-existing work to monitor Java applications using trace expressions expressed in Prolog [6, 7, 25, 26]. In order to verify a trace expression τ modelled in Prolog, we supply the runtime verification engine with Prolog representations of the events taking place in the environment. These are easily obtained from the abstraction engine and the Java environment that links to sensors and actuators. The Java environment reports instances of assert_shared_belief, remove_shared_belief and executeAction to the runtime verification engine which checks if the event is compliant with the current state of the modelled environment (namely, the state where τ moved via δ, based on the previous observed events), and reports any *violations* that occur during execution.

Imagine we have some program that has been verified in AJPF using a structured abstraction generated from the trace expression τ. We have an abstraction engine and Java environment that link to sensors and actuators. We link the Java environment to the existing Prolog implementation for runtime monitoring using trace expressions. This link reports instances of assert_shared_belief, remove_shared_belief and executeAction to a Monitor object that forwards these as events to the Prolog program. This monitor then reports any *violations* of the trace expression that occur during execution.

4 CASE STUDIES AND EXPERIMENTS

In this section we report the experimental results obtained for two different case studies. The first one is derived from Example 1, the cruise control agent. The second case study is more complex and involves the combination of multiple frameworks such as Robotic Operating System[12] (ROS), Gazebo[13] and ROSBridge[14].

The two case studies discussed in this section show how, by following the approach presented in this paper, we can answer the following research questions:

- Can the constraints on the environment for autonomous cognitive systems expressed by EAASL help reduce the complexity of the model checking problem?
- How can we assure that such constraints, besides avoiding the state explosion problem during the model checking stage, are also met by the real world? Namely, how do we close the reality gap?
- If a constraint is violated, is the system informed in a reasonable amount of time?

Both case studies have been run on a machine with the following specification: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 4 cores 8 threads, 16 GB RAM DDR4.

[11] http://jpl7.org, accessed on December 2020.
[12] https://www.ros.org/, accessed on December 2020.
[13] http://gazebosim.org/, accessed on December 2020.
[14] http://wiki.ros.org/rosbridge_suite, accessed on December 2020.

4.1 Cruise Control

Figure 14 shows a possible definition of the abstract model for the cruise control example using EAASL.

```
agent:
  car

beliefs:
  safe
  driver_accelerates
  at_speed_limit
  driver_brakes

actions:
  accelerate
  brake

constraints:
  when car believes driver_accelerates it believes safe
  car believes driver_accelerates before believing at_speed_limit
```

Fig. 14. EAASL configuration file for Cruise Control example.

Figure 15 shows the trace expression modeling the cruise control agent from Example 1. The single parts of such trace expression have been introduced singularly through Section 3.2. This trace expression has been automatically generated from the EAASL configuration file (Figure 14). When required, multiple beliefs and actions are combined into an event type allowing all the events so combined. For instance, in Figure 15, we combine beliefs $bel(driver\_accelerate)$, $not\_bel(driver\_accelerate)$, $bel(safe)$ and $not\_bel(safe)$, into the event type $accelerate\_or\_safe$. Such event type matches all the beliefs combined to create it. Generally, we use combinations of beliefs and actions as conditions for the $\gg$ operator to filter out events which are not interesting for the trace expression contained in the body of the filter (right operand of $\gg$). For instance, in equation (63), we use the $accelerate\_or\_safe$ event type to let only beliefs $bel(driver\_accelerate)$, $not\_bel(driver\_accelerate)$, $bel(safe)$ and $not\_bel(safe)$ pass inside $When_1$. All other events are filtered out, according to the semantics of the $\gg$ operator.

AJPF's property specification language uses LTL extended with modalities for BDI concepts. For example, beliefs such as $B(a, b)$ are interpreted as agent a believes b. In this language $\Box$ means "it is always the case" and $\Diamond$ means "it is eventually the case".

4.1.1 Can constraints help reduce the complexity of the model checking problem? For our first case study, we carried out experiments using the agent discussed in Example 1. First, we implemented an abstract environment model in Java (recall that these are the standard unstructured abstractions used by AJPF), where the verification takes 7,378 states and 12:07 minutes to verify that it is always the case that eventually the car believes it is safe or that it is in the process of braking:

$$\Box(B(car, safe) \rightarrow \Box(\Diamond(B(car, safe) \vee B(car, braking)))) \qquad \text{(P1)}$$

The condition $B(car, safe) \rightarrow$ at the start of the formula considers the possibility that the car never believes it is safe since braking is only triggered when the *safe* belief is removed. To test our approach, we first used the automatically generated trace expression in Figure 15 with the omission of $Constrs$. Without the constraints the model is equivalent to an unstructured abstraction, i.e., one where the percepts safe, at_speed_lim, driver_brakes, and driver_accelerates could all either be true or false at any moment. Verifying (P1) in an abstract model generated from this trace expression without the constraints took 7,378 states and 12:07 minutes. The behaviour was exactly the same as that for the environment model created in Java.

$$Protocols = (Beliefs \mid Actions) \wedge Constrs \tag{54}$$
$$Beliefs = (Safe \mid DriverAccelerates \mid AtSpeedLimit \mid DriverBrakes) \tag{55}$$
$$Safe = ((bel(safe):\epsilon) \vee (not\_bel(safe):\epsilon)) \cdot Safe \tag{56}$$
$$DriverAccelerates = ((bel(driver\_accelerates):\epsilon) \vee (not\_bel(driver\_accelerates):\epsilon)) \cdot DriverAccelerates \tag{57}$$
$$AtSpeedLimit = ((bel(at\_speed\_limit):\epsilon) \vee (not\_bel(at\_speed\_limit):\epsilon)) \cdot AtSpeedLimit \tag{58}$$
$$DriverBrakes = ((bel(driver\_brakes):\epsilon) \vee (not\_bel(driver\_brakes):\epsilon)) \cdot DriverBrakes \tag{59}$$
$$Actions = (Accelerate \mid Brake) \tag{60}$$
$$Accelerate = action(accelerate):Accelerate \tag{61}$$
$$Brake = action(brake):Brake \tag{62}$$
$$Constrs = (accelerate\_or\_safe \gg When_1) \wedge (accelerate\_or\_at\_speed \gg Before) \tag{63}$$
$$When_1 = (bel(safe) : When_2) \vee ((not\_bel(driver\_accelerates):\epsilon) \vee (not\_bel(safe):\epsilon)) \cdot When_1 \tag{64}$$
$$When_2 = (bel(driver\_accelerates):When_3) \vee (not\_bel(safe):When_1) \vee ((bel(safe):\epsilon) \vee (not\_bel(driver\_accelerates):\epsilon)) \cdot When_2 \tag{65}$$
$$When_3 = ((bel(driver\_accelerates):\epsilon) \vee (bel(safe):\epsilon)) \cdot When_3 \vee (not\_bel(driver\_accelerates):When_2) \tag{66}$$
$$Before = (bel(driver\_accelerates):All) \vee ((not\_bel(driver\_accelerates):\epsilon) \vee (not\_bel(at\_speed\_limit):\epsilon)) \cdot Before \tag{67}$$
$$All = ((bel(driver\_accelerates):\epsilon) \vee (not\_bel(driver\_accelerates):\epsilon) \vee (bel(at\_speed\_limit):\epsilon) \vee (not\_bel(at\_speed\_limit):\epsilon)) \cdot All \tag{68}$$

Fig. 15. Automatically generated trace expression for the Cruise Control Agent.

We then investigated the effect of structuring the model using the complete trace expression (i.e., with the constraints). With this abstraction (P1) takes 4:33 minutes to prove using 2947 states – one third of the model checking time and the size of the state space.

This answers our research question, indeed by adding constraints on the environment we managed to reduce the model checking time. This is achieved by pruning states that are not considered valid since we are assuming specific situations will never occur. In this case, for instance, we are removing the states where the agent would have believed to have reached the speed limit without the driver previously accelerating.

4.1.2 How can we assure the abstraction's constraints are met by the real world? In autonomous driving, it is important that an autonomous vehicle *should not* be able to override the actions of a driver. In our previous example the vehicle violates this rule – it would only let the driver accelerate if it was safe to do so, and it would brake *whenever* it detected unsafe conditions even if the driver was currently trying to accelerate. We adapted the program, removing these restrictions. This modified program could *not* be verified in the unstructured model because our property is *not*

actually true in that model – if the driver continually accelerates in an unsafe situation then the car can *never* brake. However, it is true in the structured model which assumes that the driver never accelerates if the situation is unsafe. When we run this program in our simulator it is indeed possible to cause a crash by accelerating in unsafe conditions. This is where using runtime verification can be advantageous. The monitor logs an exception at the moment when the unsafe acceleration takes place. It generates the error message shown below and also shows the current state of the trace expression, which is the equivalent of (65) in Figure 15.

```
*** DYNAMIC TYPE-CHECKING ERROR ***
Message event(abstraction_car0, assert_shared(driver_accelerates))
cannot be accepted in the current state

S_8=(bel(safe):S_6)\/((not_bel(safe):epsilon)\/
    (not_bel(driver_accelerates):epsilon))*S_8])
```

This identifies the system as now being in an unverified state, as this acceleration has violated the trace expression. The example shows how we have addressed the development of a principled mechanism for creating structured abstractions in a way that allows us to provide at least some guarantee of the validity of our results.

This answers our research question, indeed the automatically synthesised monitor is capable of catching violations of the abstract environment's restrictions. This helps close the reality gap between what is assumed at design time (in the first research question to reduce the model checking time) and how the real system behaves. Thanks to the addition of a monitor we can control the real system, and when an event violating the abstract environment's constraints is observed, the monitor can raise an error informing the system about it.

4.1.3 If a constraint is violated, is the system informed in a reasonable amount of time? We have shown the time required to model check the agent in AJPF using structured and unstructured abstractions. We have also shown how the addition of assumptions on the model of the environment helped reducing the time required for performing model checking of LTL properties, by pruning states we assume will never occur. Next, present an evaluation of the monitoring process applied to the system at runtime. More specifically, we evaluate two aspects: monitor response time, and memory usage.

In Figure 16, we report the response time of the monitor for the cruise control case study. On the x-axis, we have the length of the analysed trace; on the y-axis, we have the time the monitor requires to intercept and analyse a single event of the trace. The monitor response time does not suffer from increasing the trace length. The average time it requires to intercept and analyse an event (the red line in the graph) is around 0.5 milliseconds. Our results corroborate the findings found in [10], that the runtime verification problem using trace expressions requires linear time with respect to the trace length.

This answers our research question, indeed the runtime verification problem can be solved in linear time with respect to the trace length. These results show that the use of a monitor is not invasive, and allow its use in scenarios where the resources of the system are limited.

In Figure 17, we report the memory usage of the cruise control system execution, with and without a monitor. As we can see, the presence of a monitor running in parallel with the system does not affect negatively the memory usage, which is almost unaltered. This allows us to show that increasing the length of the trace to be analysed the monitor does not require an increased amount of memory. This observation can also be proved by considering the kind of trace expressions used to represent abstract environments. Such trace expressions can be cyclic, but can never increase in

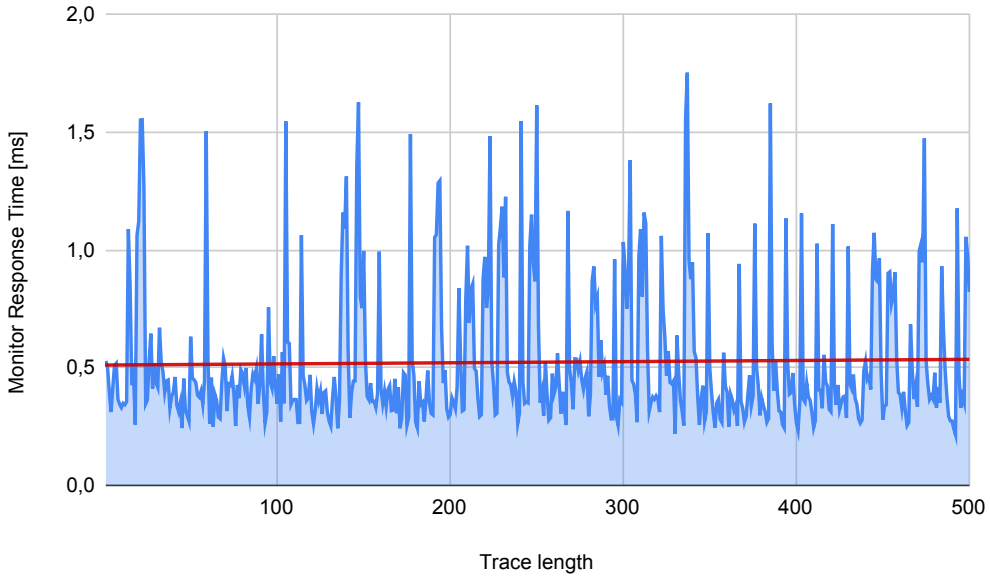

Fig. 16. Monitor response time for a sample trace observed during execution of the cruise control case study.

size[15]; thus, we can always find a finite and constant number T for which at any state, the trace expression will always have a number N of possible event types to check, s.t. $N \leq T$.

4.2 Simulation Mars Curiosity Rover

To further demonstrate the feasibility of our approach, we introduce another challenging case study where our framework has been successfully applied. This case study involves a simulation of the Mars Curiosity Rover whose main objectives include recording image data and collecting soil/rock data.

Even though in the original mission the software used in Curiosity was not ROS-based, a ROS simulation in Gazebo has been developed[16] using official data and 3D models of Curiosity and Mars terrain that were made available by NASA.

ROS [83] is an open-source set of software libraries and tools to build robotic applications. It is modular, supported by a large community, and highly compatible with many types of robots. We used the existing integration of the Curiosity rover simulation with MCAPL[17] [28], where the ROS simulation is extended to allow the rover to be guided by an agent coded in Gwendolen. Such integration has been possible thanks to the ROSBridge package, which has been used to establish the communication between the Gwendolen agent and the Curiosity rover in ROS. In this implementation, the actions performed by the agent are directly mapped into ROS messages published on the corresponding topics (communication channels) used by Curiosity; conversely, to obtain the action results and the perceptions coming from the sensors the agent has to subscribe to the relevant ROS topic.

[15] According to trace expressions operational semantics, no new terms can be generated by expanding a term.
[16] https://bitbucket.org/theconstructcore/curiosity_mars_rover/src/master/, accessed on December 2020.
[17] https://github.com/autonomy-and-verification-uol/gwendolen-ros-curiosity, accessed on December 2020.

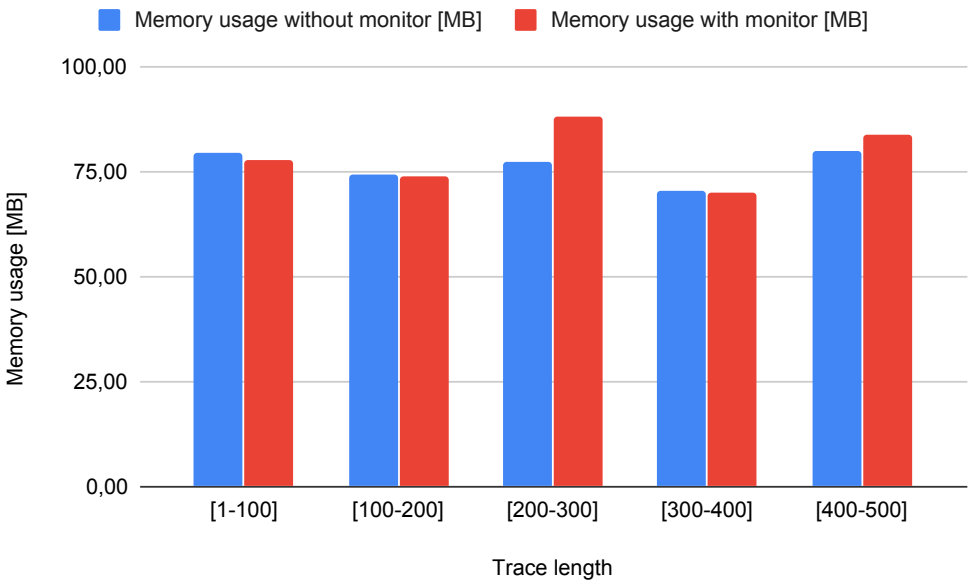

Fig. 17. Memory usage of the cruise control process with and without a monitor.

Besides being a challenging case study to evaluate our approach in practice, we chose this scenario for two main reasons. First, the Gwendolen interface with ROS[18] [29]: by showing the applicability of our approach to this scenario, we show that this could also be used for many other robotic applications developed in ROS. Second, its highly dynamic and unpredictable environment: our approach is focused on recognising assumptions violations in the abstract model used in the static verification process. Depending on the scenario, the environment can be more easily and safely abstracted, but, in complex, unpredictable and error-prone scenarios it is necessary to have a validation process at runtime.

In the following, we present an example of the code used by the agent controlling the Curiosity rover to patrol an area on the Mars surface.

Example 2. *(Simulation Mars Curiosity Rover). We consider a patrolling mission, where the Curiosity has to patrol four waypoints (a, b, c, and d). At the start of the simulation, the rover waits for the initialisation of the low-level arm, mast and wheels control modules. The belief +actuator_ready(Mode) is added, with Mode varying on the set {arm, mast, wheels}. When all three of them are correctly set and ready for the mission, the goal start is added. The corresponding plan performs two actions sequentially, first the agent asks the rover to turn right of a certain amount, then it asks to extend the mast (control_mast(open)); this allows the Curiosity to take pictures using the camera that is mounted on top of the mast. After the completion of each control_wheels action (used to make the rover move), the agent receives the movement_completed belief. Each time, depending on where the agent believes to be, a different plan is selected and that makes the agent patrol*

[18] https://github.com/autonomy-and-verification-uol/gwendolen-rosbridge, accessed on December 2020.

a different waypoint on the Mars surface. In some of these steps the actions control_mast and control_arm are performed in order to simulate interaction with the environment (like taking pictures and moving the arm in different positions to manipulate soil or rock samples).

```
:name:
curiosity

:Initial Beliefs:
ready(0)
patrol(firstturn)

:Plans:

+actuator_ready(Mode) : { B ready(0) } ←
    -actuator_ready(Mode), -ready(0), +ready(1);
+actuator_ready(Mode) : { B ready(1) } ←
    -actuator_ready(Mode), -ready(1), +ready(2);
+actuator_ready(Mode) : { B ready(2) } ←
    -actuator_ready(Mode), -ready(2), +!start [perform];

+!start [perform] : {⊤}
← control_wheels(right,45),
   control_mast(open);

+movement_completed : { B patrol(firstturn) } ←
    -movement_completed, -patrol(firstturn), +patrol(halfforward), control_wheels(forward,35);
+movement_completed : { B patrol(halfforward) } ←
    -movement_completed, -patrol(halfforward), +patrol(finalturn), control_wheels(right,45);
+movement_completed : { B patrol(finalturn) } ←
    -movement_completed, -patrol(finalturn), +patrol(a), +patrol(turn), control_wheels(forward,50);
+movement_completed : { B patrol(a), B patrol(turn)  } ←
    -movement_completed, -patrol(turn), control_wheels(right,45), control_mast(close), control_arm(open);
+movement_completed : { B patrol(a), ~B patrol(turn) } ←
    -movement_completed, -patrol(a), +patrol(b), +patrol(turn), control_wheels(forward,50);
+movement_completed : { B patrol(b), B patrol(turn)  } ←
    -movement_completed, -patrol(turn), control_wheels(right,45);
+movement_completed : { B patrol(b), ~B patrol(turn) } ←
    -movement_completed, -patrol(b), +patrol(c), +patrol(turn), control_wheels(forward,100);
+movement_completed : { B patrol(c), B patrol(turn)  } ←
    -movement_completed, -patrol(turn), control_wheels(right,45), control_mast(open), control_arm(close);
+movement_completed : { B patrol(c), ~B patrol(turn) } ←
    -movement_completed, -patrol(c), +patrol(d), +patrol(turn), control_wheels(forward,100);
+movement_completed : { B patrol(d), B patrol(turn)  } ←
    -movement_completed, -patrol(turn), control_wheels(right,45);
+movement_completed : { B patrol(d), ~B patrol(turn) } ←
    -movement_completed, -patrol(d), +patrol(a), +patrol(turn), control_wheels(forward,100);
```

As in the cruise control example, we have verified several LTL properties. We report one of them here (*cur* is the abbreviation for *curiosity*):

$$\Box((B(cur, patrol(a)) \wedge B(cur, mast(open)) \wedge B(cur, arm(close))) \rightarrow (\Diamond(B(cur, patrol(b)) \wedge B(cur, mast(close)) \wedge B(cur, arm(open))))) \quad \text{(P2)}$$

With (P2) we check that each time the agent believes it is patrolling position a, with the mast open and the arm closed, then, eventually it will believe it is patrolling position b, with the mast closed and the arm open instead.

As in the previous case study, in order to verify the properties in AJPF, we have to create an abstract model of the environment first. Here, the EAASL specification is useful to describe more complex abstractions in a higher-level.

```
agent:
  curiosity

beliefs:
  movement_completed
  actuator_ready(wheels)
  actuator_ready(mast)
  actuator_ready(arm)
  mast(open)
  mast(close)
  arm(open)
  arm(close)

actions:
  control_wheels
  control_mast
  control_arm

constraints:
 curiosity believes actuator_ready(wheels) before believing movement_completed
 curiosity believes actuator_ready(mast) before believing movement_completed
 curiosity believes actuator_ready(arm) before believing movement_completed
 when curiosity believes mast(open) it does not believe mast(close)
 when curiosity believes arm(open) it does not believe arm(close)
 the action control_mast(open) causes curiosity to believe mast(open)
 the action control_mast(close) causes curiosity to believe mast(close)
 the action control_arm(open) causes curiosity to believe arm(open)
 the action control_arm(close) causes curiosity to believe arm(close)
```

Fig. 18. EAASL configuration file for the Mars Curiosity Rover example.

Figure 18 shows our EAASL specification for the Mars Curiosity Example. At the start we specify the name of the agent involved in the verification, curiosity, and the list of beliefs which are generated by the environment. Then, we have the list of actions which can be performed by the agent in the environment: the three types of control action for interacting with the wheels, the mast and the arm of the Mars rover. Finally, we have the list of constraints that we want to enforce in the abstract model. The first three are *before* constraints, which are used to enforce the order between actuator_ready and movement_completed. This means that all the rover components have to be ready before being allowed to observe movement_completed. After that, we have two *when* constraints where we enforce exclusivity between mast(open) and mast(close) (respectively for arm(open) and arm(close)). With these two constraints, we can remove all the scenarios from the abstract model where the agent believes that the mast is both open and close (the same for the arm). Finally, the last four lines are *cause* constraints, which we use to create cause-effect relations between performing a controlling action (e.g. control_mast(open)) and the belief we expect to be generated by the abstract environment (e.g. mast(open)) as result.

4.2.1 Can constraints help reduce the complexity of the model checking problem? In contrast to the specification for the cruise control example in Figure 14, there are many beliefs and actions available to the agent in this example. This highlights an important aspect of the use of unstructured models, that is the state space explosion problem (common in model checking). We are going to need the constraints not only to reduce the execution time of the model checker (as in the cruise control example), but simply to make the process feasible. Without constraining the environment, AJPF is not actually capable of completing the model-checking execution because of the huge number of states to check. By adding the constraints and consequently removing a large set of states, AJPF is then able to complete the analysis.

The trace expression deriving from the compilation of the high-level specification presented in Figure 18 is not shown because of its size. The full trace expression can be found in the Appendix. We have performed the verification of the (P2) LTL property with both the unstructured and structured abstraction models. Where the unstructured model corresponds to compiling the specification in Figure 18 without the list of constraints, and the structured one is compiled using the full specification.

When we generate an unstructured environment from our specification (via a trace expression) and attempt to verify the agent using this environment, the verification process runs out of memory and crashes due to state space explosion. Using the structured abstraction, due to its many constraints we can reduce the number of states to be analysed by the model checker, and the verification process completes successfully. That is, (P2) is considered satisfied by the GWENDOLEN agent executing in the structured environment model. The final number of states analysed is 10,438, and the time required to analyse them is 51 seconds.

In this case study, we show how adding constraints while abstracting the environment can make the model checking problem feasible. By reducing the size and complexity of the model of the environment we can verify the LTL property of interest, empirically answering the research question.

4.2.2 How can we assure the abstraction's constraints are met by the real world? As in the cruise control case study, here we also need to close the reality gap induced by adding constraints while abstracting the environment. This can be done by validating the structured abstraction at runtime, through the monitoring of the trace expression generated by the EAASL high-level specification. Using EAASL we synthesise the abstract model of the environment used by AJPF (static verification), and then, the monitor (runtime verification) that we use to validate the conformance of the real ROS environment against this abstraction.

We force a violation of our constraints at runtime to show how the monitor works. For this example, we cause a protocol violation by forcing a failure in one of the control_mast action executions in the ROS/Gazebo simulation. More specifically, considering the constraints we listed in the specification, we can introduce a ROS-side failure upon a control_mast(open) causing the mast to fail to open and so the corresponding mast(open) belief is never asserted. Even though such a violation has been manually induced, it represents a typical error determined by a failure in the real world. In this case, we are assuming that each time the action control_mast(open) is performed, the rover succeeds in opening the mast component, producing the mast(open) belief. But some technical difficulty causing the mast to not open could really happen in a real scenario. To assume that a control_mast(open) action will never fail can help to reduce the size of the model to statically verify, but we need a runtime monitor to close the gap and control that this will always be the case.

In this scenario, if the mast fails to open, we have a violation of the *cause* constraint:

```
the action control_mast(open) causes curiosity to believe mast(open)   1
```

and generates the following violation through the monitor

```
*** DYNAMIC TYPE-CHECKING ERROR ***
Message event(curiosity, action(control_wheels(right, 45)))
cannot be accepted in the current state

_S18 = (bel(mast(open)):_S18)
```

A violation occurs after performing control_mast(open) and not receiving back the belief mast(open). In the current state the only acceptable event is mast(open), which is the expected result of the previous action, but, because of the

failure on ROS-side we never received it. Thus, the next following action, control_wheels(right, 45) (according to the implementation showed in Example 2), is considered wrong because it is violating the cause-effect relation.

By adding a monitor at runtime looking for environment's constraints violations, we can identify if a violation happens and inform the system accordingly. This answers the research question since the presence of the monitor at runtime helps assuring the system will behave as intended. As long as the monitor does not find a violation at runtime, we will know that the real system is holding to our expectations.

4.2.3 If a constraint is violated, is the system informed in a reasonable amount of time? We carried out experiments to evaluate the monitor response time and memory usage in the Curiosity rover case study.

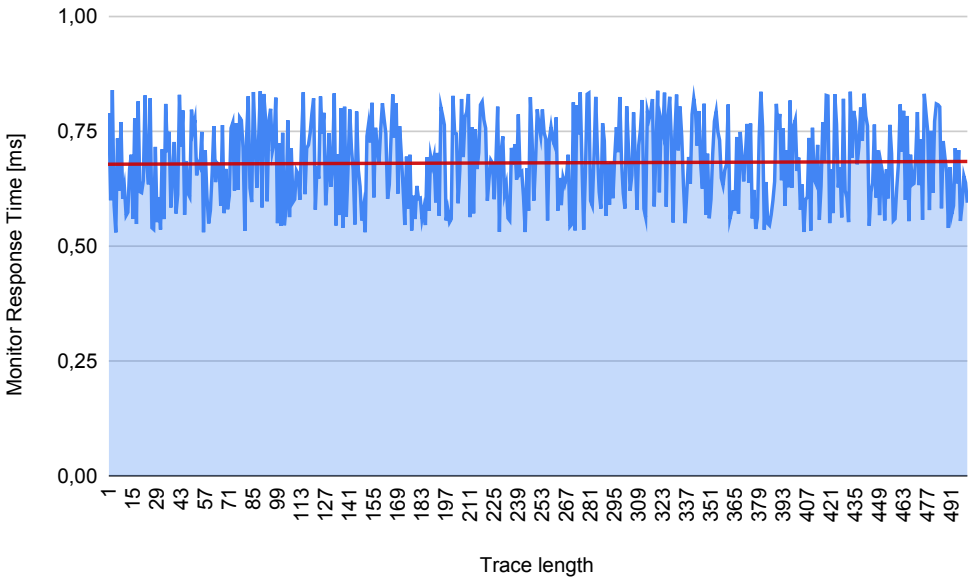

Fig. 19. Monitor response time for a sample trace observed during execution of the Curiosity case study.

In Figure 19, we report the monitor response time per event for a sample trace. As observed in the previous case study, the time required by the monitor to intercept and analyse an event does not depend on the length of the trace. The average response time (the red line) is around 0.68 milliseconds. In comparison to the cruise control case study, in the Curiosity case study the machine used for the evaluation had more computations to perform, and the trace expression generated by the EAASL high-level definition is slightly more complex. This resulted in slightly higher average response time for the monitor.

These results empirically show the feasibility of using a monitor at runtime even in the more complex scenario of the Curiosity rover. Moreover, in contrast to the cruise control case study, this scenario has more resource constraints, since the Curiosity rover has a predetermined computational power and memory, which cannot be changed on the fly; once deployed, it has to complete its missions using its fixed resources.

In Figure 20, we show the memory usage for the Curiosity case study. It is important to note that the process that was profiled to report memory usage was restricted to MCAPL; it did not include ROS and Gazebo processes (which

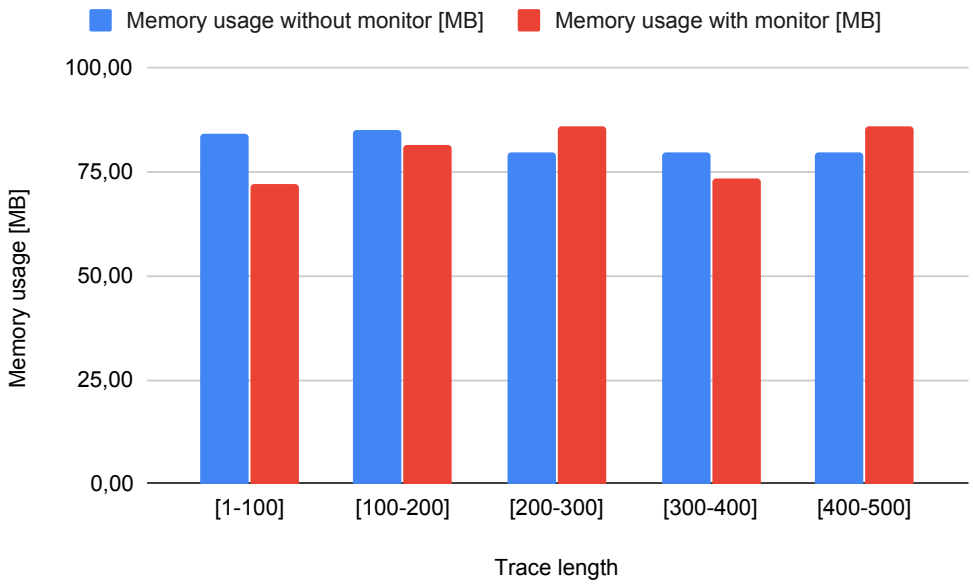

Fig. 20. Memory usage of the Curiosity process with and without a monitor.

take a lot more memory, but are not influenced by our approach). Similar to the previous case study, we observe that the presence of a monitor does not affect significantly the amount of memory required by the process execution. We can also observe that the memory usage is not influenced by increasing the length of the trace (x-axis), or more specifically, that the monitor requires constant memory.

4.3 Discussion

In this section we presented two case studies used to show the feasibility of our approach. Specifically, we focused on three different research questions to show how our work can reduce the complexity of model checking, close the reality gap deriving by restrictions added at design time, and quickly respond to constraint violations at runtime.

The two case studies cover (a) problems where a structured abstraction of the system is needed (for different reasons, from performance to pure feasibility); and (b) problems where we cannot trust the system will always behave as expected.

The choice of the two case studies was guided by their intrinsic need to abstract the environment in order to perform static verification. In a different scenario, where no abstraction of the environment or no constraints on an unstructured abstraction were needed, our approach would not be necessary; the model checking problem would be feasible on the full state space of the environment or its unstructured abstraction. This would mean that no reality gap would have been introduced. We considered two case studies which represent two typical scenarios where abstracting (and constraining) is not optional. In the first case study, we presented a typical scenario where the advantage of abstracting is in reducing model checking time. This confirmed our expectations and has shown how our approach closes the entire loop, from abstracting at design time, to validating at runtime through monitors. In the second case study, we could

stress even further the advantage of abstracting the environment, since the model checking problem without constraints on the environment was not feasible. Moreover, the second case study tackles a challenging scenario involving a robotic application, and has shown how our approach could handle the increased complexity.

We considered different restrictions in order to reduce the complexity of model checking. EAASL is expressive enough to represent all the constraints that were needed in our case studies. In Section 6, we comment more on possible future developments on the expressiveness of EAASL.

5 RELATED WORK AND COMPARISON

As observed by Fisher, Mascardi, et al. in their recent paper on certification of reliable autonomous systems [52], formal methods are a suitable technique for design, specification, validation, and verification of a wide variety of systems, including autonomous and robotic ones. This opinion is shared by many authors [30, 61, 93].

In some formal methods, such as model checking and theorem proving [68], both the system under verification and the properties to be verified are modelled using a rigorous mathematical or logical language. Other methods model the properties to check using rigorous formalisms and languages, and check the property against the *real* system under certification. These methods include some static analysis approaches [79] and runtime verification.

Combining static and runtime verification methods raises many issues due to the fact that properties are checked against a model in the first case, and against a real running system in the second. To the best of our knowledge, very few attempts to carry out such a combination exist.

In a position paper dating back to 2014, Hinrichs et al. suggested to "model check what you can, runtime verify the rest" [57]. Their work presented several realistic examples where such mixed approach would give advantages, but no technical aspects were addressed. Desai et al. [42] present a framework to combine model checking and runtime verification for robotic applications. They represent the discrete model of their system using the *P* language [43], check the model and extract the assumptions deriving from such abstraction. Despite sharing the same purpose, our work is applied to a different research area (cognitive autonomous systems) and trace expressions are more expressive than STL specifications [69] that were used in [42]. Kejstová et al.[63] extended an existing software model checker, DIVINE [17], with a runtime verification mode. The system under test consists of a user program in C or C++, along with the environment. The model checker operates in two modes: in *run* mode, a single execution of the program is explored in the standard execution order; in *verify* mode, the standard model checking algorithm is applied. This extension to DIVINE, besides being a prototype with many limitations recognised by the authors themselves, operates on C or C++ programs. Other blended approaches exist, such as a verification-centric software development process for Java making it possible to write, type check, and consistency check behavioural specifications for Java before writing any code [97]. Although it integrates a static checker for Java and a runtime assertion checker, it does not properly integrate model checking and RV. Both the Java approach and the extension to DIVINE are targeted to specific programming languages, and hence are not comparable with our approach.

Other methods such as software testing – which are generally not considered to be formal – are widely adopted in industry. The 'Software Testing Services Market by Product, End-users, and Geography – Global Forecast and Analysis 2019-2023' [60] foresees that the software testing services market will grow at a compounded average growth rate of over 12% during the period 2019-2023.

Many approaches for testing software or models exist. For example, Menghi et al. [71] propose an automated approach to translate requirements of Cyber Physical Systems specified in a logic-based language into test oracles (mechanisms

to automatically determine whether a test has passed or failed) specified in Simulink[19] – a graphical programming environment for modeling, simulating and analysing multi-domain dynamic systems. This and similar approaches would be worth exploring because of the well known connections between software testing and runtime verification [13, 47]: a really holistic approach would integrate software testing as a further stage of the process depicted in Section 3.

Next, we analyse related work according to the stages of the process presented in Figure 3: works that introduce formalisms for expressing properties of systems and perform static and/or runtime verification; works that define DSLs for specifying temporal properties; works addressing the development of safe structured abstractions for model checking; and works on model checking and RV of Multi-Agent Systems (MAS).

5.1 Formalisms for Specifying Temporal Properties

Temporal properties to be automatically verified, either statically or dynamically, are often represented using temporal logic. There are many surveys on temporal logic, starting from [56, 81] and moving to more recent works such as [20, 77, 95], which take time and intervals into account. The most well known and widely used variant of temporal logic is LTL, which is often applied in model checking [15].

When used for RV, LTL has some major limitations. Let us consider the following sequence of states, where ϕ and ψ are different formulae:

$$\phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi$$

Does the sequence satisfy $\Diamond\psi$? Given that the sequence is finite, an answer based on Pnueli's semantics [81], which takes infinite sequences or paths into account, is hard to provide: $\Diamond\psi$ is in fact satisfied if, from some time point $j \geq 0$ on, ψ becomes true. If the sequence is only a prefix of a (possibly) longer sequence, where the next states are unknown (i.e., they must still be observed), then it might be the case that $\Diamond\psi$ is true, if in some successive observed state ψ became true. Given that runtime verification aims to monitor the behaviour of a system and raises errors only when these errors actually take place, the correct answer to the question above *in a runtime verification setting* would be *it might, or it might not....* In other words, the verdict is *inconclusive*.

LTL_3 has been proposed by Bauer, Leucker, Schallhart [19] in order to make LTL suitable for runtime verification. LTL_3 is defined on finite traces and its semantics is based on three truth values: $true$, $false$ and $inconclusive$.

Timed Linear Temporal Logic [88] is a variant of LTL, and Metric Temporal Logic (MTL [64]) is an example of Timed Linear Temporal Logic. It extends LTL by constraining the until operator to an interval I that can be open, closed or half open: $I \subseteq \mathbb{R}_{\geq 0}$ whose left and right arguments belong to $\mathbb{N} \cup \{\infty\}$.

Metric Interval Temporal Logic (MITL [3]) shares both the syntax and the semantics with MTL, being a sort of restricted MTL. The difference is that the constraint on time intervals can be imposed to what is called *punctuality*. Let $\bigcup_I$ be the *until* operator and $I = [a, b]$ with $a, b \in \mathbb{R}_{\geq 0}$. The constraint $b > a$ is set in MITL, so that $I = [a, a]$ is not a valid interval.

Other variants of LTL often used in RV (see Sect 2.1 of [46]) are Mission-time Linear Temporal Logic (MLTL [65, 66]) and First-Order Linear Temporal Logic (FOLTL [18]), whereas in the cyber-physical systems domain, Timed Regular Expressions (TRE [14]), Signal Temporal Logic (STL [70]), Extended Signal Temporal Logic (xSTL [76]) and Signal First-Order Logic (SFO [16]) are also used. TRE specifies discrete behaviors augmented with timing information, whose expressive power is equivalent to the timed automata of Alur and Dill [2]. xSTL integrates TRE within STL, and SFO combines first-order logic with linear-real arithmetic and unary function symbols interpreted as piecewise-linear signals.

[19] https://www.mathworks.com/products/simulink.html, accessed on December, 2020.

An offline monitoring procedure for SFO has been developed that has polynomial complexity in the size of the input trace and the specification, for a fixed number of quantifiers and function symbols.

Trace expressions, initially devised for modeling interaction protocols in MAS, have been successfully adopted for specifying different kinds of behavioural patterns, including interactions among objects in Java-like programs [8] and Internet of Things applications developed with Node.js [12]. There are many differences between trace expressions and the formalisms mentioned in this section here. First, to the best of our understanding of these formalisms, none is more expressive than context free grammars, as traces expressions are. Second, trace expressions have been conceived with online RV in mind; the differences between trace expressions and LTL in the RV setting have been discussed by Ancona et al. [10]; similar arguments may be adapted to formalisms that extend LTL, and to formalisms for which only offline monitoring procedures exist. Third, trace expressions come with many implemented extensions ranging from parametric trace expressions [11] to timed ones [32]. Implementations exist for executing or verifying a specification in most formalisms above; this is true also for trace expressions, whose monitor is fully implemented in SWI-Prolog[20] and available to the research community[21].

Given the current definition of EAASL, we could have selected most of the temporal formalism mentioned above as the target for the compilation from EAASL. While it is true that trace expressions have been shown to be more expressive than standard LTL, and that they come with many already implemented extensions to cope with parameters, time, etc, these features have not been exploited because EAASL's expressive power is limited. *Yet.* In fact, trace expressions are suitable for being the target of the compilation even if, or when, EAASL will become more complex, powerful, expressive. In the future, we will be able to reuse the work we already did, and just add the compilation of new linguistic elements down to trace expressions, ready to support them. This would not be true if we opted for a less expressive target language since the beginning of the EAASL engineering.

Extensibility is one strong motivation for our choice.

5.2 DSL for User Friendly Specification of Temporal Properties

The direct use of any of the languages and formalisms mentioned in Section 5.1 requires a high level of experience from the user. For this reason, many languages exist that simplify the specification of temporal properties by making assumptions on the domain, by providing a graphical interface, or by restricting the language expressive power. They share with EAASL the goal of making the writing and understanding of properties easier for human users.

While most of them include patterns for stating that something should take place before or after something else, none addresses the specification of structured abstractions of environments for autonomous cognitive systems, and none has explicit statements for addition/removal of beliefs.

In PROPEL [90], property pattern templates are represented both as finite state automata and using a disciplined natural language notation. In both cases, the events that are specified, along with their relationships, are actions and responses. As an example, the sentence in natural language "after the elevator button is pushed, the elevator closes its doors" is translated into the PROPEL disciplined natural language notation "one or more occurrences of *button-push* eventually result in only one occurrence of *door-close*, *button- push* may occur zero times, *door-close* cannot occur before the first *button-push* occurs. The behavior above is repeatable".

Structured natural language is also used in FRETISH [54] that incorporates features from existing research and from NASA applications. FRETISH underlying semantics is determined by the types of four fields: scope, condition, timing,

[20] https://www.swi-prolog.org/, accessed on December, 2020.
[21] https://github.com/RMLatDIBRIS/monitor, accessed on December, 2020.

DSL	Target low-level formalism	Explicit addition/removal of beliefs	Can be manually introduced?
PROPER	Finite State Automaton	✗	✓
FRETISH	Future/Past Time LTL	✗	✓
VISPEC	MTL	✗	✓
RML	Trace expressions	✗	✓
ProMoBox	LTL	✗	✓
EAASL	Trace expressions	✓	not necessary

Table 1. DSLs for the specification of temporal properties.

and response. Each combination of field types defines a template from which future and past-time MTL formulas can be constructed. A sentence such as "When roll AP (Auto Pilot) is not engaged, the command to the roll actuator shall be zero" is translated into FRETISH as "RollAP shall always satisfy !ap_engaged $\implies$ roll_act_cmd = 0.0". So far, FRETISH does not support responses that involve ordering of actions or conditions that persist for some time interval.

VISPEC [59] is a graphical tool designed for the development and visualisation of formal specifications by users that do not have training in formal logic. Specifications in the fragment of MTL formulas called Safety MTL can be visually represented such as $\phi_1 = \Box_{[0,36]}(rpm < 4000)$, which states that in the next 36 seconds the engine speed should always be less than 4000, and $\phi_2 = \Diamond_{[0,39]}(speed > 100)$, which states that eventually, within the next 39 seconds the vehicle speed will go over 100. One valuable aspect of the paper presenting VISPEC is that it also presents a usability study that involved users familiar/not familiar with system requirements. The result was that users who have little to no mathematical training in formal logic could nevertheless develop formal specifications. Extending EAASL with a graphical interface would further simplify the specification of properties in our framework.

RML, the Runtime Monitoring Language[22] [53] is a rewriting-based and system agnostic DSL for RV which decouples monitoring from instrumentation by allowing users to write specifications and to synthesise monitors from them, independently of the system under scrutiny and its instrumentation. RML is compiled down into trace expressions, as EAASL does, but it serves a different purpose, being aimed at specifying traces of domain independent events rather than constraints on addition and removal of beliefs and on action execution.

Finally, ProMoBox [73] moves a step further in pursuing generality. It integrates the definition and verification of temporal properties in discrete-time behavioural Domain-Specific Modelling Languages (DSML) whose semantics can be described as a schedule of graph rewrite rules. With ProMoBox, the user models not only the system with a DSML, but also its properties, input model, runtime state and output trace. The DSML is thus comprised of five sub-languages, which share domain-specific syntax. The modelled system and its properties are transformed to Promela[23], and properties are verified with the Spin [58] model checker. Compared to ProMoBox, EAASL addresses one specific domain (autonomous cognitive systems) only. This makes its use unsuitable in many domains, but much easier in the one it has been conceived for.

In Table 1, we compare the DSLs analysed in this section. For each DSL, we consider the low-level formalism to which the high-level specification is compiled to, the presence of explicit addition/removal of beliefs, and the possibility of introducing beliefs in the DSL, if required.

[22] https://rmlatdibris.github.io/, accessed on December, 2020.
[23] http://spinroot.com/spin/Man/grammar.html, accessed on December, 2020.

5.3 Safe Structured Abstractions for Model Checking

"Enabling sufficiently precise yet tractable verification" with models – be they explicit or under the hood – of a real environment is a main research issue [91]. Developing "safe" structured abstractions of the environment ("environment models") for model checking that are sufficiently precise to enable effective reasoning yet not so over-restrictive that they mask faulty system behaviours has been understood as a significant challenge since the early 2000s [78].

The Bandera Environment Generator [91] is a toolset that automates the generation of environments to provide a restricted form of modular model checking of Java programs. Although the addressed problem is similar to ours, the approach is different. We do not automatically generate "safe by construction" trace expressions starting from observations of the environment. Rather, we manually design a trace expression encoding our assumptions and validate it against the real environment to empirically show that it is "safe". Although our approach requires a more accurate design stage and more manual work, it can be applied to more systems; the automatic generation of the environment model in the Bandera Environment Generator is instead inherently domain-dependent and is customised for model checking Java programs.

The approach of Dhaussy et al. [44] is closer to ours; the state space explosion is mitigated with requirements relative to scenarios which are verified instead of the full environment. In that work the context – which corresponds to our structured abstraction – is modelled with the domain-specific Context Description Language, CDL. The main difference is that CDL is less expressive than trace expressions (recursion and concatenation are not supported), and no methodology for checking the CDL specification against the real environment is discussed.

Besides CDL, hybrid automata [4, 55] are another widely adopted formalism for precise modelling of the real world. They do not solve the question of whether the model accurately captures the environment, and although RV of cyber-physical systems modelled with hybrid automata is a promising research field [75, 89], we are not aware of proposals where the same hybrid automaton model undergoes both a model checking and a RV process.

5.4 Model Checking and RV of MAS

The context of this paper is that of autonomous cognitive systems that can be to a large extent considered as MAS where agents are conceptualised and implemented in terms of mental, or cognitive, features. For this reason, we conclude the related work section with a brief overview of model checking and RV for MASs.

Investigation of model checking for MASs dates back to 1998 [21] and has continued to generate much follow up work. Besides MCAPL, we can cite for instance works by Lomuscio and Raimondi [67, 84]. Approaches to MAS RV complement these and include the proposals spun off from the SOCS project where the SCIFF computational logic framework [1] is used to provide the semantics of social integrity constraints. To model MAS interaction, expectation-based semantics specifies the links between observed and expected events, providing a means to test runtime conformance of an actual conversation with respect to a given interaction protocol [92]. Similar work has been performed using commitments [31]. None of the contributions above tackle the problem of recognising assumption violations in structured abstractions via RV for model checking autonomous systems immersed in a real environment. This makes our proposal original in the panorama of model checking both generally and, more specifically, for autonomous systems and MASs.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have shown how trace expressions can be used as a unifying formalism to generate both a structured abstraction for improving model checking efficiency and a runtime monitor to provide an additional route for guarantees

of the behaviour of a system that has been verified using an abstract model of the real world. Because of the complexity in writing trace expressions, we also presented EAASL, a high-level DSL which helps define trace expressions for generating abstract environment models more intuitively.

In previous work [50, 51], we explored the use of trace expressions to define structured abstractions. In this paper, we presented an extension of those works where a DSL is used instead of directly writing trace expressions specifications. One of the reasons that brought us to define EAASL was the feedback that we received in our previous work. The use of trace expressions was considered promising, but their complexity made the approach hard to access for non formal verification experts. This has been the main guideline we followed in this work, among exploring new and different case studies.

In particular, we have evaluated our approach by applying it to two case studies: cruise control, and the simulation of the Mars Curiosity rover. Our results in the first case study show that by using trace expressions to generate a structured abstraction model of the environment we were able to significantly decrease the number of states, and consequently reduce the time that is necessary to model check the system. Our results in the second case study demonstrate that using structured abstractions can also be the difference between successfully terminating the model checking process and getting stuck in a state space explosion. Moreover, the results in both case studies show that the response time of the runtime monitor does not increase with the length of the trace and that adding a runtime monitor causes no significant impact in memory usage.

The expressive power of trace expressions potentially paves the way to addressing more challenging scenarios.

Let us suppose that, at the end of each mission, the Mars Curiosity Rover must leave all the collected soil samples at the base station. Namely, at the beginning of each mission it is empty, and for every *collect_sample* (c) action performed during the mission, a corresponding *leave_sample* (l) action must be performed at the end. This pattern corresponds to $c^n l^n$ traces for any $n \in \mathbb{N}$. The parameter n is not fixed a priori, but defined only once the rover comes back to the station, the first *leave_sample* action is performed, and hence it becomes clear that no *collect_sample* actions should be performed until $n - 1$ *leave_sample* actions have. Traces like $c^n l^n$ cannot be modeled in LTL. So far, they cannot even be modeled in EAASL, but should EAASL be extended to cope with them, it could be easily compiled down into trace expressions, but not into LTL. Empirical results show that, in most cases, verifying whether a trace belongs to the language defined by a trace expression is linear in the length of the trace even when exploiting the full expressiveness of trace expressions: performance of RV should remain acceptable also in this scenario. In order to use expressive trace expressions for model checking, they should be over-approximated as model checking cannot be fed with properties expressed using context-free languages; this over-approximation is however possible, as shown in [48].

In future work, we aim to provide arguments (ideally proofs) that the behaviour of the abstract environments generated by the system genuinely expresses the behaviour specified by the trace expressions. We are also interested in expressing noise and uncertainty in beliefs potentially through some of our prior work in dealing with partially observable events [9].

Discovering a violation does not necessarily mean that the system is in danger. For example, in our cruise control case study braking and accelerating at the same time, even though they are tagged as a violation during the RV stage, might not cause the system to crash. Discriminating between safety-critical and acceptable violations is outside the scope of this paper, but it is a significant issue that deserves further exploration.

We also plan to apply our approach to a real case study. The scenario we have in mind is a cyber-physical system which must demonstrate its dependability in order to be acceptable to society and be trusted by its users. As an example, in a remote patient monitoring system where the program monitors a range of sensors, formal guarantees should

be provided that the system respects given medical guidelines before deployment (model checking stage). However such guarantees are likely to assume that the sensors perform within some range of behaviour so an RV stage should nevertheless be included to monitor sensor behaviour for issues and flag situations where the medical guidelines might be violated as a result.

ACKNOWLEDGMENTS

Work supported by the UK Research and Innovation Hubs for "Robotics and AI in Hazardous Environments": EP/R026084 (RAIN), EP/R026173 (ORCA), EP/R026092 (FAIR-SPACE). Fisher was funded by the Royal Academy of Engineering through a Chair in Emerging Technologies.

REFERENCES

[1] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. 2005. The *S*CIFF Abductive Proof-Procedure. In *AI*IA (Lecture Notes in Computer Science)*, Vol. 3673. Springer, 135–147.

[2] Rajeev Alur and David L. Dill. 1994. A Theory of Timed Automata. *Theor. Comput. Sci.* 126, 2 (1994), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8

[3] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. 1991. The Benefits of Relaxing Punctuality. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, Luigi Logrippo (Ed.). ACM, 139–152. https://doi.org/10.1145/112600.112613

[4] Rajeev Alur, Thomas A. Henzinger, Gerardo Lafferriere, and George J. Pappas. 2000. Discrete Abstractions of Hybrid Systems. *Proc. of the IEEE* 88, 7 (2000), 971–984.

[5] Davide Ancona, Matteo Barbieri, and Viviana Mascardi. 2013. Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In *Proc. of the 28th Annual ACM Symposium on Applied Computing, SAC '13*. ACM, 1377–1379.

[6] D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. 2015. Global Protocols as First Class Entities for Self-Adaptive Agents. In *Proc. of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015*. 1019–1029.

[7] Davide Ancona, Sophia Drossopoulou, and Viviana Mascardi. 2012. Automatic Generation of Self-monitoring MASs from Multiparty Global Session Types in Jason. In *Proc. of Declarative Agent Languages and Technologies X, DALT 2012*. 76–95.

[8] Davide Ancona, Angelo Ferrando, Luca Franceschini, and Viviana Mascardi. 2017. Parametric Trace Expressions for Runtime Verification of Java-Like Programs. In *Proc. of the 19th Workshop on Formal Techniques for Java-like Programs (FTFJP'17)*.

[9] Davide Ancona, Angelo Ferrando, Luca Franceschini, and Viviana Mascardi. 2018. Coping with Bad Agent Interaction Protocols When Monitoring Partially Observable Multiagent Systems. In *Proc. of Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection - 16th International Conference (LNCS)*, Vol. 10978. Springer, 59–71.

[10] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2016. Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification. In *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*.

[11] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2017. Parametric Runtime Verification of Multiagent Systems. In *Proc. of the 2017 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2017*. ACM, 1457–1459.

[12] Davide Ancona, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca. 2017. Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things. In *Proc. of the 1st workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM (EPTCS)*, Vol. 264. 27–42.

[13] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, et al. 2005. Combining test case generation and runtime verification. *Theoretical Computer Science* 336, 2-3 (2005), 209–234.

[14] Eugene Asarin, Paul Caspi, and Oded Maler. 2002. Timed regular expressions. *J. ACM* 49, 2 (2002), 172–206. https://doi.org/10.1145/506147.506151

[15] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

[16] Alexey Bakhirkin, Thomas Ferrère, Thomas A. Henzinger, and Dejan Nickovic. 2018. The first-order logic of signals: keynote. In *Proceedings of the International Conference on Embedded Software, EMSOFT 2018, Torino, Italy, September 30 - October 5, 2018*, Björn B. Brandenburg and Sriram Sankaranarayanan (Eds.). IEEE, 1. https://doi.org/10.1109/EMSOFT.2018.8537203

[17] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. 2013. DiVinE 3.0–an explicit-state model checker for multithreaded C & C++ programs. In *International Conference on Computer Aided Verification*. Springer, 863–868.

[18] David A. Basin, Felix Klaedtke, Samuel Müller, and Birgit Pfitzmann. 2008. Runtime Monitoring of Metric First-order Temporal Properties. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India (LIPIcs)*, Ramesh Hariharan, Madhavan Mukund, and V. Vinay (Eds.), Vol. 2. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 49–60. https://doi.org/10.4230/LIPIcs.FSTTCS.2008.1740

[19] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4 (2011), 14:1–14:64. https://doi.org/10.1145/2000799.2000800

[20] Pierfrancesco Bellini. 2001. Interval Temporal Logic for Real-Time Systems: Specification, Execution and Verification Processes. (2001). PhD. Thesis, University of Florence, Italy.

[21] Massimo Benerecetti, Fausto Giunchiglia, and Luciano Serafini. 1998. Model Checking Multiagent Systems. *J. Log. Comput.* 8, 3 (1998), 401–423.

[22] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael J. Wooldridge. 2006. Verifying Multi-agent Programs by Model Checking. *Autonomous Agents and Multi-Agent Systems* 12, 2 (2006), 239–256. https://doi.org/10.1007/s10458-006-5955-7

[23] R. H. Bordini, J. F. Hübner, and M. Wooldridge. 2007. *Programming multi-agent systems in AgentSpeak using Jason.* John Wiley and Sons.

[24] Michael E. Bratman. 1987. *Intention, Plans, and Practical Reason.* Cambridge, Mass., Harvard University Press.

[25] Daniela Briola, Viviana Mascardi, and Davide Ancona. 2014. Distributed Runtime Verification of JADE and Jason Multiagent Systems with Prolog. In *Proc. of the 29th Italian Conference on Computational Logic.* 319–323.

[26] D. Briola, V. Mascardi, and D. Ancona. 2014. Distributed Runtime Verification of JADE Multiagent Systems. In *Proc. of IDC.*

[27] Sabine Broda, António Machiavelo, Nelma Moreira, and Rogério Reis. 2018. Automata for regular expressions with shuffle. *Inf. Comput.* 259, 2 (2018), 162–173.

[28] R. C. Cardoso, M. Farrell, M. Luckcuck, A. Ferrando, and M. Fisher. 2020. Heterogeneous Verification of an Autonomous Curiosity Rover. In *NASA Formal Methods Symposium (NFM).*

[29] R. C. Cardoso, A. Ferrando, L. A. Dennis, and M. Fisher. 2020. An Interface for Programming Verifiable Autonomous Agents in ROS. In *European Conference on Multi-Agent Systems (EUMAS).*

[30] Ana Cavalcanti. 2017. Formal methods for robotics: RoboChart, RoboSim, and more. In *Brazilian Symposium on Formal Methods.* Springer, 3–6.

[31] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. 2009. Commitment Tracking via the Reactive Event Calculus. In *Proc. of the 21st International Joint Conference on Artifical Intelligence (IJCAI'09).* 91–96.

[32] Luca Ciccone, Angelo Ferrando, Davide Ancona, and Viviana Mascardi. 2019. Timed Trace Expressions. In *Proceedings of the 34th Italian Conference on Computational Logic, Trieste, Italy, June 19-21, 2019 (CEUR Workshop Proceedings)*, Alberto Casagrande and Eugenio G. Omodeo (Eds.), Vol. 2396. CEUR-WS.org, 229–241. http://ceur-ws.org/Vol-2396/paper13.pdf

[33] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2000. *Model Checking.* The MIT Press.

[34] Edmund M. Clarke and Bernd-Holger Schlingloff. 2001. Model Checking. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). Elsevier and MIT Press, 1635–1790.

[35] Arie A Covrigaru and Robert K Lindsay. 1991. Deterministic autonomous systems. *AI Magazine* 12, 3 (1991), 110–110.

[36] Louise A. Dennis. 2017. *Gwendolen Semantics: 2017.* Technical Report ULCS-17-001. University of Liverpool, Department of Computer Science.

[37] Louise A Dennis. 2018. The MCAPL Framework including the Agent Infrastructure Layer and Agent Java Pathfinder. *The Journal of Open Source Software* 3, 24 (2018).

[38] Louise A. Dennis, Jonathan M. Aitken, Joe Collenette, Elisa Cucco, Maryam Kamali, Owen McAree, Affan Shaukat, Katie Atkinson, Yang Gao, Sandor M. Veres, and Michael Fisher. 2016. Agent-Based Autonomous Systems and Abstraction Engines: Theory Meets Practice. In *Proc. of Towards Autonomous Robotic Systems: 17th Annual Conference, TAROS 2016.* 75–86.

[39] L. A. Dennis, M. Fisher, N. Lincoln, A. Lisitsa, and S. M. Veres. 2010. Declarative Abstractions for Agent Based Hybrid Control Systems. In *Proc. 8th Int. Workshop on Declarative Agent Languages and Technologies (DALT).* 96–111.

[40] Louise A. Dennis, Michael Fisher, Nicholas K. Lincoln, Alexei Lisitsa, and Sandor M. Veres. 2014. Practical verification of decision-making in agent-based autonomous systems. *Automated Software Engineering* (2014), 1–55.

[41] Louise A. Dennis, Michael Fisher, Matthew P. Webster, and Rafael H. Bordini. 2012. Model checking agent programming languages. *Automated Software Engineering* 19, 1 (2012), 5–63.

[42] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. 2017. Combining Model Checking and Runtime Verification for Safe Robotics. In *Proc. of the 17th International Conference on Runtime Verification, RV 2017 (LNCS)*, Vol. 10548. Springer, 172–189.

[43] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation 2013, PLDI 2013.* ACM, 321–332.

[44] Philippe Dhaussy, Jean-Charles Roger, and Frédéric Boniol. 2011. Reducing State Explosion with Context Modeling for Model-Checking. In *Proc. of the 13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011.* 130–137.

[45] Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. *Engineering Dependable Software Systems* 34 (01 2013), 141–175.

[46] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. 2018. A Taxonomy for Classifying Runtime Verification Tools. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science)*, Christian Colombo and Martin Leucker (Eds.), Vol. 11237. Springer, 241–262. https://doi.org/10.1007/978-3-030-03769-7_14

[47] Kevin Falzon and Gordon J. Pace. 2012. Combining Testing and Runtime Verification Techniques. In *Model-Based Methodologies for Pervasive and Embedded Software, 8th International Workshop, MOMPES 2012, Essen, Germany, September 4, 2012. Revised Papers (Lecture Notes in Computer Science)*, Ricardo Jorge Machado, Rita Suzana Pitangueira Maciel, Julia Rubin, and Goetz Botterweck (Eds.), Vol. 7706. Springer, 38–57. https://doi.org/10.1007/978-3-642-38209-3_3

[48] Angelo Ferrando. 2019. The early bird catches the worm: First verify, then monitor! *Sci. Comput. Program.* 172 (2019), 160–179. https://doi.org/10.1016/j.scico.2018.11.008

[49] Angelo Ferrando, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi. 2018. Recognising Assumption Violations in Autonomous Systems Verification. In *Proc. of the 2018 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2018*.

[50] Angelo Ferrando, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi. 2018. Recognising Assumption Violations in Autonomous Systems Verification. In *AAMAS*. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 1933–1935.

[51] Angelo Ferrando, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi. 2018. Verifying and Validating Autonomous Systems: Towards an Integrated Approach. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science)*, Christian Colombo and Martin Leucker (Eds.), Vol. 11237. Springer, 263–281. https://doi.org/10.1007/978-3-030-03769-7_15

[52] Michael Fisher, Viviana Mascardi, Kristin Yvonne Rozier, Bernd-Holger Schlingloff, Michael Winikoff, and Neil Yorke-Smith. 2020. Towards a Framework for Certification of Reliable Autonomous Systems. *CoRR* abs/2001.09124 (2020). arXiv:2001.09124 https://arxiv.org/abs/2001.09124

[53] Luca Franceschini. 2019. RML: runtime monitoring language: a system-agnostic DSL for runtime verification. In *Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming, Genova, Italy, April 1-4, 2019*, Stefan Marr and Walter Cazzola (Eds.). ACM, 28:1–28:3. https://doi.org/10.1145/3328433.3328462

[54] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. 2020. Generation of Formal Requirements from Structured Natural Language. In *Requirements Engineering: Foundation for Software Quality - 26th International Working Conference, REFSQ 2020, Pisa, Italy, March 24-27, 2020, Proceedings [REFSQ 2020 was postponed] (Lecture Notes in Computer Science)*, Nazim H. Madhavji, Liliana Pasquale, Alessio Ferrari, and Stefania Gnesi (Eds.), Vol. 12045. Springer, 19–35. https://doi.org/10.1007/978-3-030-44429-7_2

[55] Thomas A. Henzinger. 1996. The Theory of Hybrid Automata. In *Proc. of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS)*. 278–292.

[56] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. 1991. Timed Transition Systems. In *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings (Lecture Notes in Computer Science)*, J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg (Eds.), Vol. 600. Springer, 226–251. https://doi.org/10.1007/BFb0031995

[57] Timothy L. Hinrichs, A. Prasad Sistla, and Lenore D. Zuck. 2014. Model Check What You Can, Runtime Verify the Rest. In *HOWARD-60: A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, Andrei Voronkov and Margarita V. Korovina (Eds.). EPiC Series in Computing, Vol. 42. EasyChair, 234–244. https://easychair.org/publications/paper/tq7

[58] Gerard J. Holzmann. 2004. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

[59] Bardh Hoxha, Nikolaos Mavridis, and Georgios E. Fainekos. 2015. VISPEC: A graphical tool for elicitation of MTL requirements. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2015, Hamburg, Germany, September 28 - October 2, 2015*. IEEE, 3486–3492. https://doi.org/10.1109/IROS.2015.7353863

[60] Industry Research. 2019. Software Testing Services Market by Product, End-users, and Geography – Global Forecast and Analysis 2019-2023. https://www.industryresearch.co/software-testing-services-market-by-product-end-users-and-geography-global-forecast-and-analysis-2019-2023-14620379

[61] Félix Ingrand. 2019. Recent Trends in Formal Validation and Verification of Autonomous Robots Software. In *3rd IEEE International Conference on Robotic Computing, IRC 2019, Naples, Italy, February 25-27, 2019*. IEEE, 321–328. https://doi.org/10.1109/IRC.2019.00059

[62] Maryam Kamali, Louise A. Dennis, Owen McAree, Michael Fisher, and Sandor M. Veres. 2017. Formal verification of autonomous vehicle platooning. *Science of Computer Programming* 148 (2017), 88–106. Special issue on Automated Verification of Critical Systems (AVoCS 2015).

[63] Katarína Kejstová, Petr Rockai, and Jiri Barnat. 2017. From Model Checking to Runtime Verification and Back. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings (Lecture Notes in Computer Science)*, Shuvendu K. Lahiri and Giles Reger (Eds.), Vol. 10548. Springer, 225–240. https://doi.org/10.1007/978-3-319-67531-2_14

[64] Ron Koymans. 1990. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems* 2, 4 (1990), 255–299. https://doi.org/10.1007/BF01995674

[65] Jianwen Li and Kristin Y. Rozier. 2018. MLTL Benchmark Generation via Formula Progression. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science)*, Christian Colombo and Martin Leucker (Eds.), Vol. 11237. Springer, 426–433. https://doi.org/10.1007/978-3-030-03769-7_25

[66] Jianwen Li, Moshe Y. Vardi, and Kristin Y. Rozier. 2019. Satisfiability Checking for Mission-Time LTL. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11562. Springer, 3–22. https://doi.org/10.1007/978-3-030-25543-5_1

[67] Alessio Lomuscio and Franco Raimondi. 2006. MCMAS: A Model Checker for Multi-agent Systems. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006*. 450–454.

[68] Donald W Loveland. 2016. *Automated Theorem Proving: a logical basis*. Elsevier.

[69] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals.. In *Proc. of Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, FORMATS/FTRTFT 2004 (LNCS)*, Yassine Lakhnech and Sergio Yovine (Eds.), Vol. 3253. Springer, 152–166.

[70] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal*

Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings (Lecture Notes in Computer Science), Yassine Lakhnech and Sergio Yovine (Eds.), Vol. 3253. Springer, 152–166. https://doi.org/10.1007/978-3-540-30206-3_12

[71] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C. Briand. 2019. Generating automated and online test oracles for Simulink models with continuous and uncertain behaviors. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 27–38. https://doi.org/10.1145/3338906.3338920

[72] Giorgio Metta and Angelo Cangelosi. 2012. *Cognitive Robotics.* Springer US, Boston, MA, 613–616. https://doi.org/10.1007/978-1-4419-1428-6_654

[73] Bart Meyers, Hans Vangheluwe, Joachim Denil, and Rick Salay. 2020. A Framework for Temporal Verification Support in Domain-Specific Modelling. *IEEE Trans. Software Eng.* 46, 4 (2020), 362–404. https://doi.org/10.1109/TSE.2018.2859946

[74] Vincent C. Müller. 2012. Autonomous Cognitive Systems in Real-World Environments: Less Control, More Flexibility and Better Interaction. *Cogn. Comput.* 4, 3 (2012), 212–215. https://doi.org/10.1007/s12559-012-9129-4

[75] Luan Viet Nguyen, Christian Schilling, Sergiy Bogomolov, and Taylor T. Johnson. 2015. Runtime Verification for Hybrid Analysis Tools. In *Proc. of the 6th International Conference on Runtime Verification, RV 2015 (LNCS).* 281–286.

[76] Dejan Nickovic, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. 2018. AMT 2.0: Qualitative and Quantitative Trace Analysis with Extended Signal Temporal Logic. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II (Lecture Notes in Computer Science)*, Dirk Beyer and Marieke Huisman (Eds.), Vol. 10806. Springer, 303–319. https://doi.org/10.1007/978-3-319-89963-3_18

[77] Joël Ouaknine and James Worrell. 2008. Some Recent Results in Metric Temporal Logic. In *Formal Modeling and Analysis of Timed Systems, 6th International Conference, FORMATS 2008, Saint Malo, France, September 15-17, 2008. Proceedings (Lecture Notes in Computer Science)*, Franck Cassez and Claude Jard (Eds.), Vol. 5215. Springer, 1–13. https://doi.org/10.1007/978-3-540-85778-5_1

[78] John Penix, Willem Visser, Eric Engstrom, Aaron Larson, and Nicholas Weininger. 2000. Verification of time partitioning in the DEOS scheduler kernel. In *Proc. of the 22nd International Conference on Software Engineering.* 488–497.

[79] Marco Pistoia, Satish Chandra, Stephen J. Fink, and Eran Yahav. 2007. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Syst. J.* 46, 2 (2007), 265–288. https://doi.org/10.1147/sj.462.0265

[80] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77).* IEEE Computer Society, Washington, DC, USA, 46–57.

[81] Amir Pnueli. 1986. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In *Current Trends in Concurrency, Overviews and Tutorials*, J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg (Eds.). Lecture Notes in Computer Science, Vol. 224. Springer, 510–584. https://doi.org/10.1007/BFb0027047

[82] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. 2005. Jadex: A BDI Reasoning Engine. In *Multi-Agent Programming: Languages, Platforms and Applications.* Multiagent Systems, Artificial Societies, and Simulated Organizations, Vol. 15. Springer, 149–174.

[83] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. 2009. ROS: an open-source Robot Operating System. In *Workshop on Open Source Software.* IEEE, Japan.

[84] Franco Raimondi and Alessio Lomuscio. 2007. Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *J. Applied Logic* 5, 2 (2007), 235–251.

[85] Anand S. Rao. 1996. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Agents Breaking Away: Proc. of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW).* 42–55.

[86] A. S. Rao and M. Georgeff. 1995. BDI Agents: From Theory to Practice. In *Proc. of the 1st Int. Conf. Multi-Agent Systems (ICMAS).* San Francisco, USA, 312–319.

[87] A. S. Rao and M. P. Georgeff. 1991. Modeling Agents within a BDI-Architecture. In *Proc. of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR&R).* 473–484.

[88] Jean-Francois Raskin. 1999. Logics, automata and classical theories for deciding real-time. (1999). PhD. Thesis, Facultés universitaires Notre-Dame de la Paix, Namur.

[89] A. Prasad Sistla, Miloš Žefran, and Yao Feng. 2012. Runtime Monitoring of Stochastic Cyber-physical Systems with Hybrid State. In *Proc. of the 2nd International Conference on Runtime Verification, RV 2011 (LNCS).* 276–293.

[90] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. 2002. PROPEL: an approach supporting property elucidation. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, Will Tracz, Michal Young, and Jeff Magee (Eds.). ACM, 11–21. https://doi.org/10.1145/581339.581345

[91] Oksana Tkachuk, Matthew B. Dwyer, and Corina S. Pasareanu. 2003. Automated Environment Generation for Software Model Checking. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003).* 116–129.

[92] Paolo Torroni, Pinar Yolum, Munindar P. Singh, Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, and Paola Mello. 2009. Modelling Interactions via Commitments and Expectations. In *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models.* IGI Global.

[93] Federico Vicentini, Mehrnoosh Askarpour, Matteo G Rossi, and Dino Mandrioli. 2019. Safety assessment of collaborative robotics through automated formal verification. *IEEE Transactions on Robotics* 36, 1 (2019), 42–61.

[94] Markus Vincze and David Vernon. 2017. Industrial Priorities for Cognitive Robotics, Deliverable D-3.1, RockEU2 - Robotics Coordination Action for Europe Two. (2017).

[95] Farn Wang. 2004. Formal verification of timed systems: A survey and perspective. *Proc. IEEE* 92, 8 (2004), 1283–1305.

[96] Michael Wooldridge and Nicholas R. Jennings. 1995. Intelligent agents: theory and practice. *Knowledge Eng. Review* 10, 2 (1995), 115–152. https://doi.org/10.1017/S0269888900008122

[97] Daniel M. Zimmerman and Joseph R. Kiniry. 2009. A Verification-Centric Software Development Process for Java. In *Proceedings of the Ninth International Conference on Quality Software, QSIC 2009, Jeju, Korea, August 24-25, 2009*, Byoungju Choi (Ed.). IEEE Computer Society, 76–85. https://doi.org/10.1109/QSIC.2009.18

A MARS CURIOSITY ROVER ENVIRONMENT TRACE EXPRESSION

Trace expression generated from the specification presented in Figure 18.

$$Protocols = (Beliefs \mid Actions) \wedge Constrs \tag{69}$$

$$\begin{aligned} Beliefs = (&MovementCompl \mid ActReadyWheels \mid \\ &ActReadyMast \mid ActReadyArm \mid MastOpen \mid \\ &MastClose \mid ArmOpen \mid ArmClose) \end{aligned} \tag{70}$$

$$\begin{aligned} MovementCompl = ((&bel(movement\_completed):\epsilon) \vee \\ &(not\_bel(movement\_completed):\epsilon)) \cdot MovementCompl \end{aligned} \tag{71}$$

$$\begin{aligned} ActReadyWheels = ((&bel(actuator\_ready(wheels)):\epsilon) \vee \\ &(not\_bel(actuator\_ready(wheels)):\epsilon)) \cdot ActReadyWheels \end{aligned} \tag{72}$$

$$\begin{aligned} ActReadyMast = ((&bel(actuator\_ready(mast)):\epsilon) \vee \\ &(not\_bel(actuator\_ready(mast)):\epsilon)) \cdot ActReadyMast \end{aligned} \tag{73}$$

$$\begin{aligned} ActReadyArm = ((&bel(actuator\_ready(arm)):\epsilon) \vee \\ &(not\_bel(actuator\_ready(arm)):\epsilon)) \cdot ActReadyArm \end{aligned} \tag{74}$$

$$\begin{aligned} MastOpen = ((&bel(mast(open)):\epsilon) \vee \\ &(not\_bel(mast(open)):\epsilon)) \cdot MastOpen \end{aligned} \tag{75}$$

$$\begin{aligned} MastClose = ((&bel(mast(close)):\epsilon) \vee \\ &(not\_bel(mast(close)):\epsilon)) \cdot MastClose \end{aligned} \tag{76}$$

$$\begin{aligned} ArmOpen = ((&bel(arm(open)):\epsilon) \vee \\ &(not\_bel(arm(open)):\epsilon)) \cdot ArmOpen \end{aligned} \tag{77}$$

$$\begin{aligned} ArmClose = ((&bel(arm(close)):\epsilon) \vee \\ &(not\_bel(arm(close)):\epsilon)) \cdot ArmClose \end{aligned} \tag{78}$$

$$Actions = (ControlWheels \mid ControlMast \mid ControlArm) \tag{79}$$

$$ControlWheels = action(control\_wheels):ControlWheels \tag{80}$$

$$ControlMast = action(control\_mast):ControlMast \tag{81}$$

$$ControlArm = action(control\_arm):ControlArm \tag{82}$$

Fig. 21. Trace expression for the Mars Curiosity Rover example (Part 1).

$$
\begin{aligned}
Constrs = {} & (mast\_open\_or\_close \gg When_1) \wedge (arm\_open\_or\_close \gg When_4) \wedge \\
& (mast\_open\_any\_action \gg Cause_1) \wedge (mast\_close\_any\_action \gg Cause_2) \wedge \\
& (arm\_open\_any\_action \gg Cause_3) \wedge (arm\_close\_any\_action \gg Cause_4) \wedge \\
& (actuator\_ready\_wheels\_move\_compl \gg Before_1) \wedge \\
& (actuator\_ready\_mast\_move\_compl \gg Before_2) \wedge \\
& (actuator\_ready\_arm\_move\_compl \gg Before_3) \qquad (83)
\end{aligned}
$$

$$
\begin{aligned}
When_1 = {} & (bel(mast(open)) : When_2) \vee \\
& ((not\_bel(mast(close)):When_1) \vee (bel(mast(close)):When_3)) \qquad (84)
\end{aligned}
$$

$$
\begin{aligned}
When_2 = {} & (((bel(mast(open)):\epsilon) \vee (not\_bel(mast(close)):\epsilon)) \cdot When_2) \\
& \vee (not\_bel(mast(open)):When_1) \qquad (85)
\end{aligned}
$$

$$
\begin{aligned}
When_3 = {} & ((bel(driver\_accelerates):\epsilon) \vee (bel(safe):\epsilon)) \cdot When_3 \\
& \vee (not\_bel(driver\_accelerates):When_2) \qquad (86)
\end{aligned}
$$

$$
\begin{aligned}
When_4 = {} & (bel(arm(open)):When_5) \vee (not\_bel(arm(close)):When_4) \\
& \vee (bel(arm(close)):When_6) \qquad (87)
\end{aligned}
$$

$$
\begin{aligned}
When_5 = {} & ((bel(arm(open)):\epsilon) \vee (not\_bel(arm(close)):\epsilon) \cdot When_5) \\
& (not\_bel(arm(open)):When_6) \qquad (88)
\end{aligned}
$$

$$
\begin{aligned}
When_6 = {} & (not\_bel(arm(close)):When_4) \vee \\
& ((not\_bel(arm(open)):\epsilon) \vee (bel(arm(close)):\epsilon) \cdot When_4) \qquad (89)
\end{aligned}
$$

$$
\begin{aligned}
Cause_1 = {} & (action(control\_mast(open)):bel(mast(open)):Cause_1) \vee \\
& (not\_bel(mast(open)):Cause_1) \vee (anyActionBut(action(control\_mast(open))):Cause_1) \qquad (90)
\end{aligned}
$$

$$
\begin{aligned}
Cause_2 = {} & (action(control\_mast(close)):bel(mast(close)):Cause_2) \vee \\
& (not\_bel(mast(close)):Cause_2) \vee (anyActionBut(action(control\_mast(close))):Cause_2) \qquad (91)
\end{aligned}
$$

$$
\begin{aligned}
Cause_3 = {} & (action(control\_arm(open)):bel(arm(open)):Cause_3) \vee \\
& (not\_bel(arm(open)):Cause_3) \vee (anyActionBut(action(control\_arm(open))):Cause_3) \qquad (92)
\end{aligned}
$$

$$
\begin{aligned}
Cause_4 = {} & (action(control\_arm(close)):bel(arm(close)):Cause_4) \vee \\
& (not\_bel(arm(close)):Cause_4) \vee (anyActionBut(action(control\_arm(close))):Cause_4) \qquad (93)
\end{aligned}
$$

$$
\begin{aligned}
Before_1 = {} & (bel(actuator\_ready(wheels)):Anything) \vee \\
& ((not\_bel(actuator\_ready(wheels)):\epsilon) \vee \\
& (not\_bel(movement\_completed):\epsilon)) \cdot Before_1 \qquad (94)
\end{aligned}
$$

$$
\begin{aligned}
Before_2 = {} & (bel(actuator\_ready(mast)):Anything) \vee \\
& ((not\_bel(actuator\_ready(mast)):\epsilon) \vee \\
& (not\_bel(movement\_completed):\epsilon)) \cdot Before_2 \qquad (95)
\end{aligned}
$$

$$
\begin{aligned}
Before_3 = {} & (bel(actuator\_ready(arm)):Anything) \vee \\
& ((not\_bel(actuator\_ready(arm)):\epsilon) \vee \\
& (not\_bel(movement\_completed):\epsilon)) \cdot Before_3 \qquad (96)
\end{aligned}
$$

Fig. 22. Trace expression for the Mars Curiosity Rover example (Part 2).