# Verifying and Validating Autonomous Systems: Towards an Integrated Approach

Angelo Ferrando[1], Louise A. Dennis[2], Davide Ancona[1], Michael Fisher[2], and Viviana Mascardi[1(✉)]

[1] Università di Genova, Genova, Italy
{angelo.ferrando,davide.ancona,viviana.mascardi}@dibris.unige.it
[2] Liverpool University, Liverpool, UK
{L.A.Dennis,MFisher}@liverpool.ac.uk

**Abstract.** 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, but 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 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. Trace expressions are used to model the environment for both static formal verification and runtime verification.

**Keywords:** Runtime verification · Model checking
Autonomous systems · Trace expressions · MCAPL

## 1 Introduction

Static formal verification of autonomous systems that interact with the real world requires a model of the world to successfully accomplish the verification process. In [23] we recommended using the simplest environment model, in which any combination of the environmental predicates that correspond to possible perceptions of the autonomous system is possible. Consider an intelligent cruise control in an autonomous vehicle that can perceive the environmental predicates

Work supported by EPSRC as part of the Verifiable Autonomy research project [EP/L024845] and the FAIR-SPACE [EP/R026092], ORCA [EP/R026173], and RAIN [EP/R026084] Robotics and AI Hubs.


C. Colombo and M. Leucker (Eds.): RV 2018, LNCS 11237, pp. 263–281, 2018.
https://doi.org/10.1007/978-3-030-03769-7_15

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. In order to formally verify the behaviour of the cruise control agent, we might randomly supply subsets of {safe, at_speed_limit, driver_brakes, driver_accelerates}: the generation of each subset causes branching in the state space exploration during verification so that, ultimately, all possible combinations are explored.

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 obviously 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 a car can not both brake and accelerate at the same time: subsets of environmental predicates containing both driver_brakes and driver_accelerates should not be supplied to the agent during the static verification stage, as they do not correspond to situations that we believe likely 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 [31], we propose an approach for exploiting the advantages of structured abstractions, while mitigating their risks. Our proposal consists in 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 (RV). If performed during a testing stage, RV of the actual environment against its structured abstraction allows the developer to identify situations not foreseen in the initial assumptions. He/she can revise them, generate a new structured abstraction, re-verify it via model checking, re-validate it via RV once again, reaching in the end a "safe" abstraction. If RV takes place after system deployment and assumption violations are detected, mechanisms for handing control to a human, a failsafe system, or for performing ad hoc reasoning about the current system safety should be invoked. To demonstrate the feasibility of the proposed approach, we implemented it on top of the MCAPL framework developed by Dennis, Fisher, et al. [21,24] (which provides a model-checker for rational agents) using trace expressions developed by Ancona, Ferrando, Mascardi, et al. [3,10,11] as the single formalism to generate both the environment model and the runtime monitor. We choose trace expressions instead of more widely used formalisms for model checking like Linear Temporal Logic (LTL [39]) because of their

expressive power. In our previous work [10], we demonstrated that trace expressions are able to express and verify sets of traces that are context-free. When working in a RV scenario, trace expressions are more expressive than LTL. In this paper we keep the presentation as simple as possible and do not stress the potential of such expressive power. However, this power opens up interesting scenarios discussed in the conclusions.

## 2 Related Work

The growing popularity of model checking in industry is due to the possibility of transforming domain-specific input models familiar to the developers into "under the hood models" invisible to them and amenable to model checking using existing techniques [36]. The idea behind this work is similar: we use trace expressions as the front-end formalism suitable for modeling behaviour patterns in systems made up of autonomous entities [4,5,30] and we transform them into under the hood models suitable for both model checking and runtime verification (RV). The main difference is that trace expressions are not domain-specific, and although initially devised for modeling protocols in multiagent system (MASs), they have been successfully adopted for specifying different kinds of behavioural patterns, including interactions among objects in Java-like programs [7] and Internet of Things applications developed with Node.js [12]. This is both a strength and a weakness: a customised formalism for different domains would make it more usable by domain experts, at the cost of some loss in generality.

"Enabling sufficiently precise yet tractable verification" with models – be they explicit or under the hood – of the real environment is a main issue [46]. Developing "safe" structured abstractions of the environment (also named "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 [38]. The Bandera Environment Generator [46] 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 the same as 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 and implement 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 any system and environment; the automatic generation of the environment model is instead inherently domain-dependent, and the Bandera Environment Generator is in fact customized for model checking Java programs. The approach of Dhaussy et al. [27] 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 – corresponding 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. In a similar way, in [25] Desai et al. 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 [26], check the model and extract the assumptions deriving from such abstraction. Despite sharing the same purpose, our work is not committed to any specific case study and trace expressions are more expressive than STL specifications [35] used in [25]. Besides CDL, hybrid automata [2,32] 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 lively and promising research field [37,45], we are not aware of proposals where the same hybrid automaton model undergoes both a model checking and a RV process.

Investigation of model checking for MASs dates back to 1998 [13] and has continued to generate much follow up work, for instance the Model Checking Agent Programming Languages project which involves two authors of this paper (http://cgi.csc.liv.ac.uk/MCAPL/, [15,24]), and works by Lomuscio and Raimondi [34,41]. 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 run-time conformance of an actual conversation with respect to a given interaction protocol [47]. Similar work has been performed using commitments [18]. A more recent strand is related to the exploitation of trace expressions for MAS RV and monitoring, along with their ancestor formalism [6]. None of the contributions above tackles 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 "in general" and, more specifically, for autonomous systems and MASs.

## 3 Background and Running Example

**MCAPL: Model Checking BDI Agents.** The belief-desire-intention (BDI) model, originally proposed by Bratman [16] as a philosophical theory of the practical reasoning, inspired both architectures [43] and programming languages [14,40,44] for agents. BDI languages are based on *rational agency* [42]. 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 [21,24] 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 (JPF) model checker for Java bytecodes (https://babelfish.arc.nasa.gov/trac/jpf). We use the "Engineering Autonomous Space Software" (EASS) variant of GWENDOLEN [20], a language developed for programming agent-based autonomous systems and verifying them in AJPF. EASS assumes an architecture in which the rational agents are partnered with an *abstraction engine* that discretises continuous information from sensors in an explicit fashion [19,22]. We adopt the methodology from [23] setting out the formal verification of rational agent components in autonomous systems. This uses model checking to demonstrate that the rational 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;. The trigger is the addition of a goal or a belief (beliefs may be acquired thanks to the operation of perception or as a result of internal deliberation); the guard states conditions about the agent's beliefs which must be true before the plan can become active; 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.

According to the operational semantics of GWENDOLEN [20], 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 performing perception. As an intention may be suspended while 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. All of this is part of the standard Gwendolen semantics.

**Example 1** *(Cruise Control Agent). When the car has an* initial goal *to be at the speed limit, +! 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 — 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. 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. If it believes it is at the speed limit, it maintains its speed having achieved its goal (which will be dropped automatically, having been achieved).*

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

```
:Reasoning Rules:
can_accelerate :- safe, ~ driver_accelerates, ~ driver_brakes;

:Initial Goals:
at_speed_limit

:Plans:
+! at_speed_limit: {can_accelerate, ~accelerating, ~at_speed_lim}
    <- -braking, +accelerating, perf(accelerate), *~accelerating;
+! at_speed_limit: {~safe} <- *safe;
+! at_speed_limit: {driver_accelerates} <- *~driver_accelerates;
+! at_speed_limit: {driver_brakes} <- *~driver_brakes;
+! at_speed_limit: {accelerating} <- *~accelerating;
+at_speed_lim: {can_accelerate, at_speed_lim}
     <- -accelerating, -braking, perf(maintain_speed);
-at_speed_lim: {~at_speed_lim} <- +! at_speed_limit;
-safe: {~driver_brakes, ~safe, ~braking} <- -accelerating, +braking,
     perf(brake);
+driver_accelerates: {safe, ~driver_brakes, driver_accelerates, ~accelerating}
     <- +accelerating, -braking, perf(accelerate);
+driver_brakes: {driver_brakes, ~braking} <- +braking, -accelerating,
     perf(brake);
```

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 [22,33]. 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 *thin 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 rational agent as logical predicates. The rational agent accesses these *shared beliefs* as perceptions. Previously, the model of the combined behaviour of simulator, thin Java environment, and abstraction engine used for verification was unstructured: all the possible combinations of the 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, comes into play.

**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 (over $\mathcal{E}$) 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*, $\tau$, represents a set of possibly infinite event traces, and is defined on top of the following operators:

- $\varepsilon$, 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 $\vartheta$ ($e \in \vartheta$), and the remaining part is a trace of $\tau$.
- $\tau_1 \cdot \tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of $\tau_1$ with those of $\tau_2$.
- $\tau_1 \wedge \tau_2$ (*intersection*), the intersection of traces $\tau_1$ and $\tau_2$.
- $\tau_1 \vee \tau_2$ (*union*), denoting the union of traces of $\tau_1$ and $\tau_2$.
- $\tau_1 | \tau_2$ (*shuffle*), denoting the union of the sets obtained by shuffling each trace of $\tau_1$ with each trace of $\tau_2$ (see [17] for a more precise definition).
- $\vartheta \gg \tau$ (*filter*), denoting the set of all traces contained in $\tau$, when "deprived" of all events that do not match $\vartheta$.

Trace expressions can be easily represented as Prolog terms. To support recursion without introducing an explicit construct, trace expressions are regular (a.k.a. rational or cyclic) 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 $\tau_1$ 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, $\tau_2$ specifies the next valid state of the system after event $e$. Otherwise, the event $e$ is not valid in $\tau_1$. The rules for the transition functions are presented in [10]. 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 $\delta$ 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 $\delta$ transition function generates a new trace expression representing the updated current state. If, on observing an event, no $\delta$ 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.

**AJPF Static Formal Verification.** The EASS implementation provides a Java class supporting the creation of abstract models. Unstructured abstractions can be created by overriding in a *subclass* its method add_random_beliefs which is called when the agent requests an action execution or sleeps. This method should generate a set of beliefs and add them to the environment's *percept base* which the agent then polls. It is assumed this implementation will randomly generate all possible sub-sets of the shared beliefs relevant to the agent. For static verification, therefore, we want to generate this subclass from our trace expression.

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 shared belief for all agents and remove_belief(b) removes shared belief $b$ from all agents. assert_belief (a, b) and remove_belief(a, b) alter the available beliefs for a specific agent a. For reasons of space we do not describe these further. Our runtime monitor needs to observe these events. We are also interested in any action performed by an agent, so our runtime monitor must also observe calls to the executeAction method that all EASS environments implement.

## 4 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 1 gives an overview of this system. A trace expression $\tau$ is used to generate an abstract model in Java used to verify an agent in AJPF (the dotted box on the right of the Figure). Once this verification is successfully completed, the verified agent is used with an abstraction engine, a thin Java environment, and the real world or external simulator. This is shown in the dotted 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 so far different measures will be 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 shared 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 (we remind 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 shared beliefs. They are defined as $bel(b) = \{assert\_belief(b)\}$ and $not\_bel(b) = \{remove\_belief(b)\}$. 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 $e = A$. Clearly, $e \in \mathcal{E}_b$ and $e = A$ are mutually exclusive.

**Representing Abstract Models in AJPF.** Abstract models in AJPF can be represented as automata. The automaton states can be divided into two parts: *initial beliefs* and *actions*. Initial Beliefs represent all the shared beliefs that may be asserted before the system starts executing. After an action is performed, more shared beliefs may be asserted. In the unstructured abstractions used by

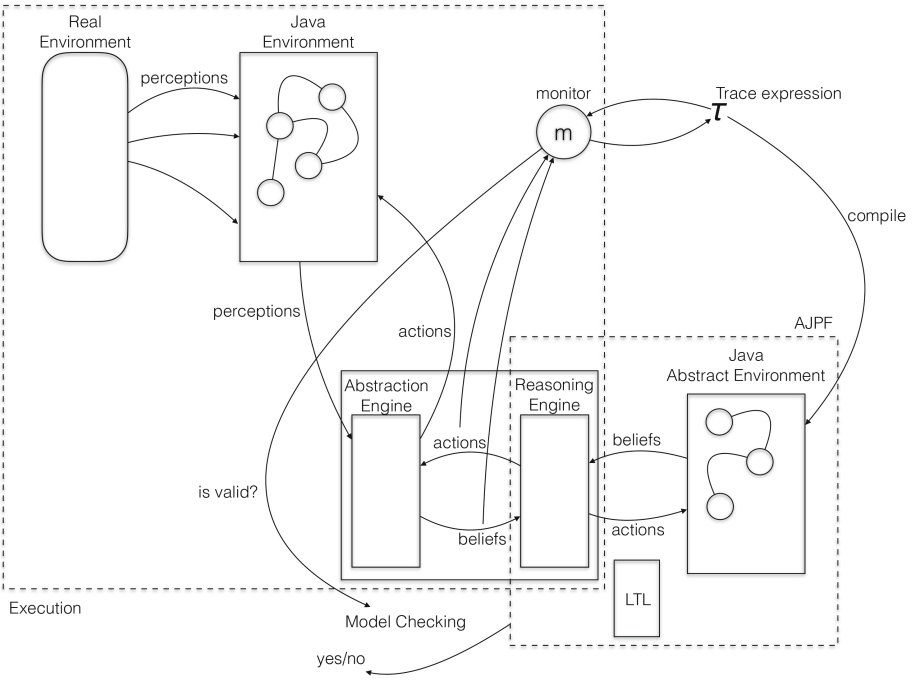


**Fig. 1.** General view.

the "standard" AJPF system the initial beliefs, and the beliefs after each action, were generated at random. Any structured abstraction will be one that places constraints upon the possible transitions in the automaton.

**Representing Abstract Models as Trace Expressions.** We represent an abstract model of the real world as a set of possibly cyclic trace expressions modelled in Prolog. The basic structure of the Prolog code is given in Fig. 2. We abuse 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. Occurrences of $Pre$ in (1) and (2) are intended to unify, and the variable names used in these positions in any instantiation of this template should be the same. $Pre$ is needed to model (optional) constraints on the beliefs that can be observed before the first action takes place, and the trace expression cycle ($Cyclic$) starts.

$$Protocol = Pre{\cdot}(Cyclic\,[\wedge Constrs]?) \tag{1}$$

$$Pre = [\,|_{i=1}^{n}\, bel(b_i){:}\epsilon]\;\|\;[not\_action : Pre{\vee}\epsilon] \tag{2}$$

$$Cyclic = SingleStep{\cdot}Cyclic \tag{3}$$

$$SingleStep = \bigvee_{i=1}^{m} Action_i{\cdot}AddBelEv \tag{4}$$

$$AddBelEv = not\_action : AddBelEv{\vee}\epsilon \tag{5}$$

$$Action_i = action(action_i){:}ProtocolBel \tag{6}$$

$$ProtocolBel = |_{i=1}^{k}\, (bel(b_i){:}\epsilon{\vee}not\_bel(b_i){:}\epsilon{\vee}\epsilon) \tag{7}$$

**Fig. 2.** Trace expression template for generating abstract environments. Indexes $k$, $n$, $m$ are not bound: they will be replaced by actual numbers when the template will be instantiated.

The template in Fig. 2 represents an unstructured abstraction in which any subset of the beliefs, $b_i$ in (7) can occur after an action. *Protocol* (1) is the main body of our trace expression. *Pre* (2) represents all events that can be generated before the first action of an agent. *Cyclic* (3) is the trace expression that describes the behaviour once the agent starts performing actions. *SingleStep* (4) represents a single action step. It is the union of the trace expressions that describe the possible results of each action the agent may take followed by *AddBelEv* which describes additional belief events after the immediate results of the action – for instance if the agent sleeps and other agents are acting. *Action* (6) consists of an action event followed by *ProtocolBel* (7) which describes the possible belief events. Any given belief, $b_i$ may appear in the shared belief base ($bel(b_i)$), disappear ($not\_bel(b_i)$) or its status may be unchanged ($\epsilon$).

Figure 2 contains an optional variable *Constrs*. If present this provides *constraints* that structure the abstraction. The template for constraints is shown in Fig. 3. *Constrs* consists of an intersection of trace expressions of the form $FilterEventType_j{\gg}C_j^x$. It appears at the top level of the trace expression in an intersection ($\wedge$) with the repeating *Cyclic* step. This allows us to put constraints on belief events without considering at which action step they occur. In this way,

$$Constrs = \bigwedge_{j=1}^{o} FilterEventType_j{\gg}[C_j^1\;\|\;C_j^2] \tag{8}$$

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

$$C_j^2 = (((NB_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)){\cdot}C_j^2) \vee (B_{j,1}{:}C_j^1) \vee (B_{j,2}{:}C_j^3) \tag{10}$$

$$C_j^3 = (((B_{j,2}{:}\epsilon) \vee (NB_{j,1}{:}\epsilon)){\cdot}C_j^3) \vee (NB_{j,2} : C_j^2) \tag{11}$$

**Fig. 3.** Trace expressions for Constrs: $B_{j,i}$ must be the "opposite operation" of $NB_{j,i}$.

each time a constrained belief event is observed in a *SingleStep*, we can keep track of the fact. $B_{j,i}$ and $NB_{j,i}$ are event types, and they must meet the condition (not modeled in Fig. 3) 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 $C_j^x$. Its purpose is to filter out any events that are not constrained by $C_j^x$, 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.

Each constraint represents a pairwise relationship between two belief events. These are captured by the three trace expressions in (9), (10) and (11) which describe the evolving behaviour of the four belief events of interest where $B_{j,i}$ is either the assertion or removal of $b_{j,i}$ and $NB_{j,i}$ is its converse. The three equations capture the constraint that if $B_{j,1}$ has occurred then $B_{j,2}$ can not occur until after $NB_{j,1}$ has been observed and vice versa. The constraint either starts in the state described by $C_j^1$ or $C_j^2$ depending upon whether only one of the constrained belief events is possible in the initial state ($C_j^1$) or both are ($C_j^2$).

**Abstract Model Generation.** Once we have created 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[1]. Our trace expression is defined according to the template in Figs. 2 and 3. Many parts of these trace expressions are not directly translated into Java; the sub-expressions relevant to the generation of abstract models are *Pre* (2), *SingleStep* (4) and *Constrs* (8). Note that the MCAPL framework provides support for constructing logical predicates and adding them to the belief base.

If *Pre* specifies particular initial beliefs then the subclass adds these to the agent's belief base at the start. *SingleStep* contains a union of trace expressions of the form $Action = action(action\_name)$:$ProtocolBel$. $ProtocolBel = |_{i=1}^{k}$ $(bel(b_i) \vee not\_bel(b_i) \vee \epsilon)$ defines the set of belief events that may occur. We define the set $\mathcal{B}(ProtocolBel)$ as $b_i \in \mathcal{B}(ProtocolBel)$ iff $(bel(b_i) \vee not\_bel(b_i) \vee \epsilon)$ is one of the interleaved trace expressions in $ProtocolBel$. For each $b_i \in \mathcal{B}(ProtocolBel)$ we define a predicate in the environment class and bind it to a Java field called $b_i$. *Constrs* constrains events by specifying mutual exclusion between some couples of them. For each *Action* trace expression we generate a corresponding *if statement* inside the add_random_beliefs method.

```
if(act.getFunctor().equals("action_name")) { translation(ProtocolBel, Constrs) }
```

We construct a set of mutually exclusive belief events, $\mathcal{M}_x(Constrs)$, from $Constrs$ where $(B_{j,1}, B_{j,2}) \in \mathcal{M}_x(Constrs)$ iff $FilterEventType_j \gg Constraint_j$ is one of the conjuncts of $Constrs$ and $C_j^1 = (((B_{j,1}{:}\epsilon) \vee (NB_{j,2}{:}\epsilon)) \cdot C_j^1) \vee (NB_{j,1} : C_j^2)$ and $C_j^3 = (((B_{j,2}{:}\epsilon) \vee (NB_{j,1}{:}\epsilon)) \cdot C_j^3) \vee (NB_{j,2} : C_j^2)$.

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

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

$$\mathcal{PB}(ProtocolBel, Constrs) = \{S \mid (\forall b_i \in \mathcal{B}(ProtocolBel).\ bel(b_i) \in S \vee not\_bel(b_i) \in S) \\ \wedge (\forall (B_1, B_2) \in \mathcal{M}_x(Constrs).\ B_1 \in S \leftrightarrow B_2 \notin S)\} \quad (12)$$

Say that $\mathcal{PB}(ProtocolBel, Constrs)$ contains $k$ sets of belief events, $S_j$, $0 \leq j < k$. We generate $translation(ProtocolBel, Constrs)$, as follows:

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

where random_int_generator is a special method that generates random integers in a way that optimises the model checking in AJPF. For each $S_j$ we generate

```
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.

## 5 Case Study and Experiments

Figures 4 and 5 show the trace expression modeling the cruise control agent from Example 1. *Pre* is reused for *AddBelEnv* since, in this case, they are the same trace expression. *SingleStep* contains only one branch which matches any action. *ProtocolBel* specifies that the possible belief events are the assertion and removal of *safe*, *at_speed_lim*, *driver_accelerates* and *driver_brakes*.

We have two constraints. Firstly we assume that the driver never brakes and accelerates at the same time. This establishes a mutual exclusion

$$Protocol = Pre{\cdot}(Cyclic \wedge Constrs) \quad (13)$$
$$Pre = ((not\_action{:}Pre) \vee \epsilon) \quad (14)$$
$$Cyclic = SingleStep{\cdot}Cyclic \quad (15)$$
$$SingleStep = action(any\_action){:}(ProtocolBel{\cdot}Pre) \quad (16)$$
$$Safe = ((bel(safe){:}\epsilon) \vee (not\_bel(safe){:}\epsilon) \vee \epsilon) \quad (17)$$
$$AtSpeedLimit = ((bel(at\_speed\_lim){:}\epsilon) \vee (not\_bel(at\_speed\_lim){:}\epsilon) \vee \epsilon) \quad (18)$$
$$Accel = ((bel(driver\_accelerates){:}\epsilon) \vee (not\_bel(driver\_accelerates){:}\epsilon) \vee \epsilon) \quad (19)$$
$$Brakes = ((bel(driver\_brakes){:}\epsilon) \vee (not\_bel(driver\_brakes){:}\epsilon) \vee \epsilon) \quad (20)$$
$$ProtocolBel = (Safe|AtSpeedLimit|Accel|Brakes) \quad (21)$$

**Fig. 4.** Trace expression for a Cruise Control Agent.

$$Constrs = (brake\_or\_accelerate \gg BrakeOrAccelerate) \wedge (accelerates\_or\_safe \gg CanBeUnsafe) \quad (22)$$

$$CanBeUnsafe = (bel(safe):AccelOrUnsafe) \vee (((not\_bel(safe):\epsilon) \vee (not\_bel(driver\_accelerates):\epsilon)) \cdot CanBeUnsafe) \quad (23)$$

$$AccelOrUnsafe = (bel(driver\_accelerates):CanAccel) \vee (((not\_bel(driver\_acclerates):\epsilon) \vee (bel(safe):\epsilon)) \cdot AccelOrUnsafe) \vee (not\_bel(safe):CanBeUnsafe) \quad (24)$$

$$CanAccel = (not\_bel(driver\_accelerates):AccelOrUnsafe) \vee (((bel(safe):\epsilon) \vee (bel(driver\_accelerates):\epsilon)) \cdot CanAccel) \quad (25)$$

$$BrakeOrAccelerate = (bel(driver\_accelerates):AccelOnly) \vee (((not\_bel(driver\_accelerates):\epsilon) \vee (not\_bel(driver\_brakes):\epsilon)) \cdot \vee BrakeOrAccelerate)(bel(driver\_brakes):BrakeOnly) \quad (26)$$

$$AccelOnly = (not\_bel(driver\_accelerates):BrakeOrAccelerate) \vee (((bel(driver\_accelerates):\epsilon) \vee (not\_bel(driver\_brakes):\epsilon)) \cdot AccelOnly) \quad (27)$$

$$BrakeOnly = (not\_bel(driver\_brakes):BrakeOrAccelerate) \vee (((bel(driver\_brakes):\epsilon) \vee (not\_bel(driver\_accelerates):\epsilon)) \cdot BrakeOnly) \quad (28)$$

**Fig. 5.** Trace expression for the Constraints on a Car where the driver only accelerates when it is safe to do so, and never uses both brake and acceleration pedal together.

between $bel(driver\_accelerates)$ and $bel(driver\_brakes)$. Initially either belief may appear. Secondly, we assume the driver only accelerates if it is safe to do so. This establishes a mutual exclusion between $bel(driver\_accelerates)$ and $not\_bel(safe)$. Initially we are in the state were we cannot observe $bel(driver\_accelerates)$. $brake\_or\_accelerate$ and $accelerates\_or\_safe$ are event types that match the relevant events for each constraint.

**MCAPL Runtime Verification.** 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 $\delta$ transitions) was easy using the JPL interface, http://jpl7.org, between Java and Prolog. In order to verify a trace expression $\tau$ 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 and reports any *violations* that occur during execution. AJPF's property specification language uses LTL extended with modalities for BDI concepts such as beliefs ($B(a, b)$ is interpreted as

meaning agent $a$ believes $b$). In this language $\Box$ means "it is always the case" and $\Diamond$ means"it is eventually the case".

We carried out experiments using the agent discussed in Example 1. When model checked using a typical hand-constructed unstructured abstraction, verification takes 4,906 states and 32:17 min 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. Obviously we would prefer a system in which the car is *forced* to start in a safe state but this would have complicated our example and discussion. To test our approach, we first used the trace expression in Fig. 4 with the omission of $Constrs$: this trace expression 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 took 4,906 states and 30:37 min: the behaviour was exactly the same as that for the unstructured model that had been created manually, and this helped validate that trace expressions following the template in Fig. 2 without constraints create unstructured abstractions that behave the same way as hand crafted ones.

We then investigated the effect of structuring the model using the trace expression in Fig. 5, which adds constraints to that in Fig. 4. With this abstraction (P1) takes 8:22 min to prove using 1,677 states – this has more than halved the time and the state space.

To illustrate how we cope with the risk that a structured abstraction may not reflect reality, we consider a version of the cruise control agent with slight variations. It is widely considered 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 the runtime verification engine fits in. The engine 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 (23) in Fig. 5.

```
*** 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.

## 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 model checking and a runtime monitor, providing a route for guarantees of the behaviour of a system that has been verified against an abstract model of the real world. Their expressive power would pave the way to addressing challenging scenarios where:

1. the behaviour of the system is modeled with a trace expression $\tau$ without expressive power limitations (for example, an expression representing the set of all $a^n b^n$ traces, for any $n \in \mathbb{N}$; this set of traces cannot be modeled in LTL) to allow specifications of complex environments;
2. $\tau$ is over-approximated by a Java model as shown in [28];
3. the model checking stage is performed using the generated over-approximating Java model;
4. the runtime verification stage uses $\tau$, with all its expressive power; 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: this means that – even when the highest modeling expressiveness of the formalism is exploited – performances of RV remain acceptable.

In the future, 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, also in case of noise and uncertainties in the formation of beliefs. We recently started working on partial observability of events [8], which is related to noise and uncertainty, and we plan to adapt and integrate the achieved results in the Verification and Validation framework presented in this paper. We also point out that discovering a violation does not necessarily mean that the system is in danger: for example, braking and accelerating at the same time – although tagged as a violation during the RV stage – might not cause the system to crash. Although discriminating between safety-critical violations and acceptable ones was out of the scope of this paper, it is a significant issues and deserves further exploration. We will also explore how to express a greater range of constraints in these models – for instance, the

constraint that some belief can only occur after some action is taken (e.g., that a car can only reach the speed limit after an acceleration has been performed).

From the practical side, we are currently designing a user friendly language for specifying trace expressions, as the current formalism is not easy to read and write for a human, and we will extend RIVERtools [9,29] to support the simplified notation. We also plan to apply our approach to a real case study. The scenario we have in mind is a cyberphysical 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 integrates sensory input, formal guarantees should be provided that the system respects given medical guidelines (model checking stage), and a RV stage looking at sensors perceptions should monitor that those guidelines are continuously met.

## References

1. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: The SCIFF abductive proof-procedure. In: Proceedings of the 9th Congress of the Italian Association for Artificial Intelligence, AI*IA 2005, pp. 135–147 (2005)
2. Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.J.: Discrete abstractions of hybrid systems. Proc. IEEE **88**(7), 971–984 (2000)
3. Ancona, D., Barbieri, M., Mascardi, V.: Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC 2013, pp. 1377–1379 (2013)
4. Ancona, D., Briola, D., Ferrando, A., Mascardi, V.: Global protocols as first class entities for self-adaptive agents. In: Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, pp. 1019–1029 (2015)
5. Ancona, D., Briola, D., Ferrando, A., Mascardi, V.: Runtime verification of fail-uncontrolled and ambient intelligence systems: a uniform approach. Intelligenza Artificiale **9**(2), 131–148 (2015)
6. Ancona, D., Drossopoulou, S., Mascardi, V.: Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In: Baldoni, M., Dennis, L., Mascardi, V., Vasconcelos, W. (eds.) DALT 2012. LNCS (LNAI), vol. 7784, pp. 76–95. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37890-4_5
7. Ancona, D., Ferrando, A., Franceschini, L., Mascardi, V.: Parametric trace expressions for runtime verification of Java-like programs. In: Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, FTFJP 2017 (2017)
8. Ancona, D., Ferrando, A., Franceschini, L., Mascardi, V.: Coping with bad agent interaction protocols when monitoring partially observable multiagent systems. In: Demazeau, Y., An, B., Bajo, J., Fernández-Caballero, A. (eds.) PAAMS 2018. LNCS (LNAI), vol. 10978, pp. 59–71. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94580-4_5
9. Ancona, D., Ferrando, A., Franceschini, L., Mascardi, V.: Managing Bad AIPs with RIVERtools. In: Demazeau, Y., An, B., Bajo, J., Fernández-Caballero, A. (eds.) PAAMS 2018. LNCS (LNAI), vol. 10978, pp. 296–300. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94580-4_24

10. Ancona, D., Ferrando, A., Mascardi, V.: 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 (2016)
11. Ancona, D., Ferrando, A., Mascardi, V.: Parametric runtime verification of multiagent systems. In: Proceedings of the 2017 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2017, pp. 1457–1459. ACM (2017)
12. Ancona, D., Franceschini, L., Delzanno, G., Leotta, M., Ribaudo, M., Ricca, F.: Towards runtime monitoring of Node.js and its application to the Internet of Things. In: Proceedings of the 1st workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM. EPTCS, vol. 264, pp. 27–42 (2017)
13. Benerecetti, M., Giunchiglia, F., Serafini, L.: Model checking multiagent systems. J. Log. Comput. **8**(3), 401–423 (1998)
14. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-agent Systems in AgentSpeak Using Jason. Wiley (2007)
15. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. Auton. Agents Multi-Agent Syst. **12**(2), 239–256 (2006)
16. Bratman, M.E.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge (1987)
17. Broda, S., Machiavelo, A., Moreira, N., Reis, R.: Automata for regular expressions with shuffle. Inf. Comput. **259**(2), 162–173 (2018)
18. Chesani, F., Mello, P., Montali, M., Torroni, P.: Commitment tracking via the reactive event calculus. In: Proceedings of the 21st International Joint Conference on Artifical Intelligence, IJCAI 2009, pp. 91–96 (2009)
19. Dennis, L.A., Fisher, M., Lincoln, N., Lisitsa, A., Veres, S.M.: Declarative abstractions for agent based hybrid control systems. In: Proceedings 8th International Workshop on Declarative Agent Languages and Technologies (DALT), pp. 96–111 (2010)
20. Dennis, L.A.: Gwendolen semantics: 2017. Technical report ULCS-17-001, University of Liverpool, Department of Computer Science (2017)
21. Dennis, L.A.: The MCAPL framework including the agent infrastructure layer and agent Java Pathfinder. J. Open Source Softw. **3**(24) (2018). https://doi.org/10.21105/joss.00617
22. Dennis, L.A., et al.: Agent-based autonomous systems and abstraction engines: theory meets practice. In: Alboul, L., Damian, D., Aitken, J.M.M. (eds.) TAROS 2016. LNCS (LNAI), vol. 9716, pp. 75–86. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40379-3_8
23. Dennis, L.A., Fisher, M., Lincoln, N.K., Lisitsa, A., Veres, S.M.: Practical verification of decision-making in agent-based autonomous systems. Autom. Softw. Eng., 1–55 (2014)
24. Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Autom. Softw. Eng. **19**(1), 5–63 (2012)
25. Desai, A., Dreossi, T., Seshia, S.A.: Combining model checking and runtime verification for safe robotics. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 172–189. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_11
26. Desai, A., Gupta, V., Jackson, E.K., Qadeer, S., Rajamani, S.K., Zufferey, D.: P: safe asynchronous event-driven programming. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation 2013, PLDI 2013, pp. 321–332. ACM (2013)

27. Dhaussy, P., Roger, J., Boniol, F.: Reducing state explosion with context modeling for model-checking. In: Proceedings of the 13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011, pp. 130–137 (2011)
28. Ferrando, A.: The early bird catches the worm: first verify, then monitor! (2016). presented at Vortex'16. Downloadable from http://trace2buchi.altervista.org/wp-content/uploads/2017/10/paper.pdf
29. Ferrando, A.: RIVERtools: an IDE for RuntIme VERification of MASs, and beyond. In: PRIMA Demo Track 2017. CEUR, Vol. 2056 (2017)
30. Ferrando, A., Ancona, D., Mascardi, V.: Monitoring patients with hypoglycemia using self-adaptive protocol-driven agents: a case study. In: Proceedings of Engineering Multi-Agent Systems - 4th International Workshop, EMAS, pp. 39–58 (2016)
31. Ferrando, A., Dennis, L.A., Ancona, D., Fisher, M., Mascardi, V.: Recognising assumption violations in autonomous systems verification. In: Proceedings of the 2018 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2018 (2018)
32. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS), pp. 278–292 (1996)
33. Kamali, M., Dennis, L.A., McAree, O., Fisher, M., Veres, S.M.: Formal verification of autonomous vehicle platooning. Sci. Comput. Program. **148**, 88–106 (2017). Special issue on Automated Verification of Critical Systems (AVoCS 2015)
34. Lomuscio, A., Raimondi, F.: MCMAS: a model checker for multi-agent systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 450–454. Springer, Heidelberg (2006). https://doi.org/10.1007/11691372_31
35. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
36. van der Merwe, H., van der Merwe, B., Visser, W.: Verifying android applications using Java PathFinder. ACM SIGSOFT Softw. Eng. Notes **37**(6), 1–5 (2012)
37. Nguyen, L.V., Schilling, C., Bogomolov, S., Johnson, T.T.: Runtime verification for hybrid analysis tools. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 281–286. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_19
38. Penix, J., Visser, W., Engstrom, E., Larson, A., Weininger, N.: Verification of time partitioning in the DEOS scheduler kernel. In: Proceedings of the 22nd International Conference on Software Engineering, pp. 488–497 (2000)
39. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS 1977, pp. 46–57. IEEE Computer Society, Washington, DC (1977)
40. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: a BDI reasoning engine. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Platforms and Applications, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15, pp. 149–174. Springer, Boston (2005). https://doi.org/10.1007/0-387-26350-0_6
41. Raimondi, F., Lomuscio, A.: Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. J. Appl. Logic **5**(2), 235–251 (2007)

42. Rao, A.S., Georgeff, M.: BDI agents: from theory to practice. In: Proceedings of the 1st International Conference Multi-Agent Systems (ICMAS), San Francisco, USA, pp. 312–319, June 1995
43. Rao, A.S., Georgeff, M.P.: Modeling agents within a BDI-architecture. In: Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR&R), pp. 473–484 (1991)
44. Rao, A.: Agentspeak(L): BDI agents speak out in a logical computable language. In: Agents Breaking Away: Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW), pp. 42–55 (1996)
45. Sistla, A.P., Žefran, M., Feng, Y.: Runtime monitoring of stochastic cyber-physical systems with hybrid state. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 276–293. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_21
46. Tkachuk, O., Dwyer, M.B., Pasareanu, C.S.: Automated environment generation for software model checking. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), pp. 116–129 (2003)
47. Torroni, P., et al.: Modelling interactions via commitments and expectations. In: Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models. IGI Global (2009)