Recognising Assumption Violations in Autonomous Systems Verification

Extended Abstract

Angelo Ferrando
DIBRIS, Genova University, IT
Angelo.Ferrando@dibris.unige.it

Louise A. Dennis*
CS Dept, Liverpool University, UK
L.A.Dennis@liverpool.ac.uk

Davide Ancona
DIBRIS, Genova University, IT
Davide.Ancona@unige.it

Michael Fisher
CS Dept, Liverpool University, UK
MFisher@liverpool.ac.uk

Viviana Mascardi
DIBRIS, Genova University, IT
Viviana.Mascardi@unige.it

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 reports on a demonstration of the feasibility of this approach using 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

ACM Reference Format:
Angelo Ferrando, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi. 2018. Recognising Assumption Violations in Autonomous Systems Verification. In *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), Stockholm, Sweden, July 10–15, 2018,* IFAAMAS, 3 pages.

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 [6, 9], L. A. Dennis, M. Fisher et al. recommend 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 for example an intelligent cruise control agent in an autonomous vehicle, that can perceive the environmental predicates `safe`, meaning that it is safe for the vehicle 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`} to it: the generation of each random 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 we propose an approach for exploiting all 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

*Louise A. Dennis was supported by EPSRC grant EP/L024845/1, Verifiable Autonomy

Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. André, S. Koenig (eds.), July 10–15, 2018, Stockholm, Sweden.

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 [7] (which provides a model-checker, Agent Java Pathfinder (AJPF) for rational agents) using trace expressions [1–3] as the single formalism to generate both the environment model and the runtime monitor[1].

The choice of trace expressions instead of more widely used formalisms for model checking like Linear Temporal Logic (LTL) [10] is due to their expressive power: in [2], D. Ancona, A. Ferrando, and V. Mascardi demonstrated that trace expressions are able to express and verify sets of traces that are context-free – and even more.

2 RESULTS AND DISCUSSION

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 a simple rational agent for intelligent cruise control implemented in Gwendolen [5], that reacted to the perceptions `safe`, `at_speed_lim`, `driver_brakes`, and `driver_accelerates`. When model checked using a typical hand-constructed unstructured abstraction, verification takes 949 states and 18:55 minutes to verify that it is always the case that eventually the car believes is it safe or that it is in the process of braking:

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

The condition $B(car, safe) \rightarrow$ at the start of the formula considers the possibility that the car never believes it is safe and braking is only triggered when the *safe* belief is removed.

To test our approach, we first constructed a trace expression to represent an unstructured environment, similar to the one created by hand, i.e., one where the four percepts could all either be true or false at any moment. Verifying (P1) in an abstract model generated from this trace expression took 949 states and 20:04 minutes: the behaviour was exactly the same as that for the unstructured model that had been created manually. This validated that trace expressions without constraints create unstructured abstractions that behave the same way as hand crafted ones.

We then investigated the effect of structuring the model using a trace expression that assumed both that the driver only accelerated when it was safe to do so, and that the driver never accelerated and braked at the same time. With this abstraction (P1) takes 4:45 minutes to prove using 355 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. Our original agent violated this rule – it would only actually accelerate when the driver pressed the acceleration pedal 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. We ran this program in a simple simulator of a motorway. In the simulation it was indeed possible to cause a crash by accelerating in unsafe conditions. This is where the runtime monitor fits in. The monitor logs an exception at the moment when an unsafe acceleration takes place and generates an error message revealing that the constraint that the driver does not accelerate when conditions are unsafe has been violated.

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. We have demonstrated how trace expressions can be used as a unifying formalism to generate both a structured abstraction for model checking and a runtime monitor. Their expressive power paves the way to addressing challenging scenarios where:

(1) the behaviour of the system is modeled with a trace expression τ 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) τ is over-approximated by a Java model as shown in [8];
(3) the model checking stage is performed using the generated over-approximating Java model;
(4) the runtime verification stage uses τ, 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 lenght of the trace: this means that – even when the highest modeling expressivity of the formalism is exploited –, performances of RV remain acceptable.

In the future, we aim at providing arguments that the behaviour of the abstract environments generated by the system genuinely expresses the behaviour specified by the trace expressions. It would also be desirable 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) where at the moment we only model constraints between beliefs.

Finally, we 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 by the society and be trusted by its users, like in [4]: in a remote patient monitoring system where the program integrates sensory input, formal guarantees should be provided on the fact that the system respects some given medical guidelines (model checking stage), and a RV stage looking at sensors perceptions should monitor that those guidelines are continuously met.

[1]This implementation can be found in the MCAPL distribution, `mcapl.sourceforge.net`. In particular the version used to generate the results reported here can be found in the the `runtime_verification` branch of the distribution. The technical details are documented in the manual (also available from the distribution) and the experimental results can be found in the University of Liverpool Data Catalogue DOI: `10.17638/datacat.liverpool.ac.uk/438`

REFERENCES

[1] Davide Ancona, Matteo Barbieri, and Viviana Mascardi. 2013. Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In *SAC*. ACM, 1377–1379.

[2] 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 (Lecture Notes in Computer Science)*, Vol. 9660. Springer, 47–64.

[3] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2017. Parametric Runtime Verification of Multiagent Systems. In *AAMAS*. ACM, 1457–1459.

[4] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2018. Improving Flexibility and Dependability of Remote Patient Monitoring with Agent-Oriented Approaches. (2018). International Journal of Agent-Oriented Software Engineering. To appear.

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

[6] 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. https://doi.org/10.1007/s10515-014-0168-9

[7] Louise A. Dennis, Michael Fisher, Matthew P. Webster, and Rafael H. Bordini. 2012. Model checking agent programming languages. *Autom. Softw. Eng.* 19, 1 (2012), 5–63. https://doi.org/10.1007/s10515-011-0088-x

[8] Angelo Ferrando. 2016. 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.

[9] Michael Fisher, Louise A. Dennis, and Matthew P. Webster. 2013. Verifying autonomous systems. *Commun. ACM* 56, 9 (2013), 84–93.

[10] 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. https://doi.org/10.1109/SFCS.1977.32