EPTCS 411

Proceedings of the

Sixth International Workshop on

Formal Methods for Autonomous Systems

Manchester, UK, 11th and 12th of November 2024

Edited by: Matt Luckcuck and Mengwei Xu

Published: 21st November 2024
DOI: 10.4204/EPTCS.411
ISSN: 2075-2180
Open Publishing Association

Preface

This EPTCS volume contains the papers from the Sixth International Workshop on Formal Methods for Autonomous Systems (FMAS 2024), which was held between the 11th and 13th of November 2024. FMAS 2024 was co-located with 19th International Conference on integrated Formal Methods (iFM'24), hosted by the University of Manchester in the United Kingdom, in the University of Manchester's Core Technology Facility.

The goal of the FMAS workshop series is to bring together leading researchers who are using formal methods to tackle the unique challenges that autonomous systems present, so that they can publish and discuss their work with a growing community of researchers. Autonomous systems are highly complex and present unique challenges for the application of formal methods. Autonomous systems act without human intervention, and are often embedded in a robotic system, so that they can interact with the real world. As such, they exhibit the properties of safety-critical, cyber-physical, hybrid, and real-time systems. We are interested in work that uses formal methods to specify, model, or verify autonomous and/or robotic systems; in whole or in part. We are also interested in successful industrial applications and potential directions for this emerging application of formal methods.

As in previous years this year's FMAS was a hybrid event, ensuring that presenters and participants can be involved in FMAS remotely. During 2020 and 2021, FMAS was held as a fully online event, due to the restrictions needed to cope with the COVID-19 pandemic, and we have continued as a hybrid event since 2022. We feel that enabling online participation, while often challenging to organise, makes FMAS accessible to people not able to travel for the workshop. We hope that the extra effort taken to support online attendance was a useful option for those presenters and attendees who made use of it.

FMAS 2024 accepted papers in one of four categories: short papers could be either *vision papers* or *research previews*; and long papers could be either *experience reports* or *regular papers*. In total, FMAS 2024 received 19 submissions: 10 Regular papers, 4 Experience Reports, 4 Research Previews, and 1 Vision paper. We received submissions from researchers at institutions in: France, Germany, Ireland, Italy, the Netherlands, Norway, Portugal, Sweden, the United States of America, and the United Kingdom. Though we received fewer submissions than last year, we are encouraged to see the submissions being sent from a wide range of countries. Each paper was reviewed by three members of our Programme Committee; after the reviews we accepted a total of 14 papers: 8 Regular, 3 Experience Reports, 2 Previews, and 1 Vision paper.

FMAS 2024 hosted two invited speakers, sharing one with iFM in a joint session like last year. Dr Silvia Lizeth Tapia Tarifa, from the University of Oslo (Norway), gave a talk titled *"Self-Adaptation in Autonomous Systems"*; which presented techniques that are suitable for specifying the variability of Self-Adaptive Systems. Prof. Daniel Kröning from Amazon Web Services and Magdalen College, University of Oxford, gave the invited talk in our joint session with iFM. The talk, titled "Proof for Industrial Systems using Neural Certificates", which introduces a novel approach to model checking that combines machine learning and symbolic reasoning.

To celebrate the first five years of FMAS, we arranged a special issue with the Science of Computer Programming journal called "Advances in Formal Methods for Autonomous Systems"; inviting extensions of papers from the first five FMAS workshops, and novel work that had not been submitted to the workshop before. The special issue provides an opportunity for researchers and practitioners to present

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. i–iii, doi:10.4204/EPTCS.411.0

theory, techniques, and applications related to the use of formal methods in the engineering, design, and analysis of autonomous systems. The special issue's Guest Editors are Dr Matt Luckcuck, Dr Marie Farrell, Jun.-Prof. Dr Maike Schwammberger, and Dr Mario Gleirscher. We would like to thank the Editors-in-Chief, Prof. Andrea De Lucia and Prof. Mohammad Reza Mousavi, for their support with our first step into journal editing. The first papers from this special issue are appearing online as we write, with more to follow; an exciting new venue for the Formal Methods for Autonomous Systems community.

Looking forward, to the next five years of FMAS, we are taking steps to make the workshop even more useful for our community. This year, we had a fruitful discussion session about the direction of the topic area, to keep us informed about the community's needs. This year's Organising Committee includes four new members: Mengwei Xu (our Programme Committee chair), Diana Carolina Benjumea Hernandez, Akhila Bairy, and Simon Kolker. FMAS 2024 introduced two new features for our workshop: an invited tutorial from Dr Louise A. Dennis, who described the practical uses of the Model-Checking Agent Programming Languages (MCAPL) framework; and a *Best Paper* award. We plan to continue both of these new features in next year's workshop. Finally, selected papers from FMAS 2024 will be invited to a journal special issue that we are in the process of organising.

We would like to thank everyone who volunteered to be part of the FMAS 2024 Programme Committee, both returning reviewers and first-timers alike; their time and effort is what keeps the workshop running.
We also thank our invited speakers, Dr Silvia Lizeth Tapia Tarifa and Prof. Daniel Kröning (in the joint session with iFM); Louise A. Dennis for presenting FMAS's first invited tutorial; the authors who submitted papers; our EPTCS editor, Prof. Martin Wirsing, for his support during the preparation of these proceedings; the organisers of iFM – Marie Farrell, Mohammad Reza Mousavi, Laura Kovács, and Nikolai Kosmatov – for supporting our workshop; FME for its sponsorship of our student travel grants; and all of the attendees (both virtual and in-person) of FMAS 2024.

FMAS will return in 2025.

Matt Luckcuck and Mengwei Xu

November 2024

Program Committee

Oana Andrei	University of Glasgow (UK)
Marco Autili	Università dell'Aquila (Italy)
Akhila Bairy	Karlsruhe Institute of Technology (Germany)
Daniela Briola	Università degli Studi di Milano-Bicocca (Italy)
Rafael C. Cardoso	University of Aberdeen (UK)
Christian Colombo	University of Malta (Malta)
Louise A. Dennis	University of Manchester (UK)
Marie Farrell	University of Manchester (UK)
Angelo Ferrando	University of Modena and Reggio Emilia (Italy))
Michael Fisher	University of Manchester (UK)
Thomas Flinkow	Maynooth University (Ireland)
Mario Gleirscher	University of Bremen (Germany)
Mallory S. Graydon	NASA Langley Research Center (USA)
Jérémie Guiochet	University of Toulouse (France)
Taylor T. Johnson	Vanderbilt University (USA)
Verena Klös	Technical University of Dresden (Germany)
Livia Lestingi	Politecnico di Milano (Italy)
Sven Linker	Kernkonzept (Germany)
Matt Luckcuck	University of Nottingham (UK)
Anastasia Mavridou	NASA Ames Research Center (USA)
Alice Miller	University of Glasgow (UK)
Alvaro Miyazawa	University of York (UK)
Rosemary Monahan	Maynooth University (Ireland)
Yvonne Murray	University of Agder (Norway)
Dominique Méry	Université de Lorraine (France)
Natasha Neogi	NASA Langley Research Center (USA)
Colin Paterson	University of York (UK)
Baptiste Pelletier	ONERA – The French Aerospace Lab (France)
Andrea Pferscher	Graz University of Technology (Austria)
Juliane Päßler	University of Oslo (Norway)
Maike Schwammberger	Karlsruhe Institute of Technology (Germany)
Paulius Stankaitis	University of Stirling
James Stovold	Lancaster University Leipzig (Germany)
Silvia Lizeth Tapia Tarifa	University of Oslo (Norway)
Elena Troubitsyna	KTH Royal Institute of Technology (Sweden)
Gricel Vázquez	University of York (UK)
Hao Wu	Maynooth University (Ireland)
Mengwei Xu	University of Newcastle (UK)

Subreviewers

Samuel Sasaki	Vanderbilt University (USA)
Serena Serbinowska	Vanderbilt University (USA)

Invited Talk: Self-Adaptation in Autonomous Systems

Dr Silvia Lizeth Tapia Tarifa
University of Oslo, Norway

Self-adaptation is a crucial feature of autonomous systems that must cope with uncertainties in, e.g., their environment and their internal state. A self-adaptive system (SAS) can be realised as a multi-layered system, e.g., a two-layered systems that have a separation of concerns between the domain-specific functionalities of the system (the managed subsystem) and the adaptation logic (the managing subsystem), which introduces an external feedback loop for managing adaptation in the system; or as a three-layered system, where the third layer can implement a feedback loop for architectural self-adaptation, which is used to reconfigure the second layer (the adaptation logic of the managing subsystem).

In this talk I will present techniques that can capture the SAS's variability, concretely software product lines, where the managing subsystem of an SAS can be modelled as a control layer capable of dynamically switching between valid configurations of the managed subsystem; and declarative stages in a lifecycle of a system, where we do not require explicit modelling of the transitions between stages in the lifecycles of a system; however, characteristics on an stage can be observed, which after observation may trigger changes in the self-adaptation logic itself.

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. iv–iv, doi:10.4204/EPTCS.411.0.1

Invited Talk: Proof for Industrial Systems using Neural Certificates

Prof. Daniel Kröning
Amazon Web Services and Magdalen College, Oxford, UK

We introduce a novel approach to model checking software and hardware that combines machine learning and symbolic reasoning by using neural networks as formal proof certificates. We train our neural certificates from randomly generated executions of the system and we then symbolically check their validity which, upon the affirmative answer, establishes that the system provably satisfies the specification. We leverage the expressive power of neural networks to represent proof certificates and the fact that checking a certificate is much simpler than finding one. As a result, our machine learning procedure is entirely unsupervised, formally sound, and practically effective. We implemented a prototype and compared the performance of our method with the state-of-the-art academic and commercial model checkers on a set of Java programs and hardware designs written in SystemVerilog.

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. v–v, doi:10.4204/EPTCS.411.0.2

Table of Contents

Autonomous System Safety Properties with Multi-Machine Hybrid Event-B

Richard Banach
Department of Computer Science,
University of Manchester, Manchester, M13 9PL, UK
richard.banach@manchester.ac.uk

Event-B is a well known methodology for the verified design and development of systems that can be characterised as discrete transition systems. Hybrid Event-B is a conservative extension that interleaves the discrete transitions of Event-B (assumed to be temporally isolated) with episodes of continuously varying state change. While a single Hybrid Event-B machine is sufficient for applications with a single locus of control, it will not do for autonomous systems, which have several loci of control by default. Multi-machine Hybrid Event-B is designed to allow the specification of systems with several loci of control. The formalism is succinctly surveyed, pointing out the subtle semantic issues involved. The multi-machine formalism is then used to specify a relatively simple incident response system, involving a controller, two drones and three responders, working in a partly coordinated and partly independent fashion to manage a putative hazardous scenario.

1 Introduction

These days, there is an inexorable drive to automate as many of the tasks that humans engage in to progress their everyday lives as possible. The motives range from gaining efficiency, to enabling genuinely new activities which would otherwise be impossible. Autonomous systems can be deployed to achieve both aims. Properly designed, not only can they enable efficiencies in activities traditionally undertaken by humans (e.g. logistics), but they can also enable activities (e.g. reactor core investigation) that would be lethal for humans.

Proper design of autonomous systems implies the ability to place great trust in their operation, given that they act in a manner much less supervised than is the case for conventional systems. If a system is to be given responsibility for deciding its own course of action (to whatever degree is considered reasonable), it must be understood how the gamut of its decision making capabilities keep it aligned with the widest considerations that its behaviour may affect. In other words, the boundary between the autonomous system and the wider environment needs to be reliably appreciated, and the greater the creativity that the system is permitted to display in meeting its local challenges, the greater the burden of understanding that is imposed on system design to ensure that letting the system do its thing will not jeopardise either itself, or elements of the wider environment.

To try to ensure this, all available techniques for enhancing system assurance may be brought to bear. As well as the traditional approaches of careful design and testing, more formal approaches, that aim to strengthen the guarantees that can be given regarding system behaviour, may be used. This highlights safety properties, the focus of the B-Method.

The significant self-agency of autonomous systems means that they are often cyber-physical systems [9, 17, 11]. The consequent interaction between discrete and continuous dynamics raises a challenge for verification [10, 14]. The contribution of this paper is to show that the architecture and capabilities of multi-machine Hybrid Event-B are suitable for formalising the challenge just described.

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 1–19, doi:10.4204/EPTCS.411.1

© R. Banach
This work is licensed under the Creative Commons
Attribution-Noncommercial-No Derivative Works License.

Hybrid Event-B was envisaged as a conservative extension of conventional Event-B that was able to accommodate continuous and smooth behaviour in a rigorous way. The foundations of the formalism were investigated in three papers [6, 7, 4], referred to below as PaperI, PaperII, PaperIII. These may be consulted for full technical details; a less detailed overview appears in [5]. Of particular relevance to the present work is [7], in which the multi-machine version of the formalism was explored, arriving at a framework that copes equally comfortably with multi-machine continuous behaviour as it does with the more familiar discrete behviours more typically used in verification. Given that autonomous systems perforce exhibit a degree of disconnect between themselves and any environment, a modelling and verification framework that captures this structural aspect in separate constructs is intrinsically useful. The main thrust of the present paper is the assertion that these aspects of the multi-machine version of Hybrid Event-B are particularly expressive and helpful in organising the exploration and verification of the complex scenarios that arise when autonomous systems are investigated.

The rest of the paper is as follows. Section 2 introduces Hybrid Event-B as a natural extension of Event-B. Section 3 introduces Hybrid Event-B refinement. Section 4 presents the main features of multiple cooperating Hybrid Event-B machines. Section 5 applies the multi-machine formalism to present a simple incident response system, involving a controller, two drones and three responders, working in a partly coordinated and partly independent fashion to manage a putative hazardous scenario. Each of the agents mentioned is specified in its own machine, and the architecture of the multi-machine formalism conveniently captured the cooperation mechanisms needed. Section 6 briefly considers verification for Hybrid Event-B and of the case study, and Section 7 concludes.

2 From Event-B to Hybrid Event-B

The B-Method [1] and Event-B in particular [3] have, by now, a well established pedigree. By way of introduction, Fig. 1 shows a simple Event-B machine **Nodes**. A static node set *NSet* is defined in the context **NCtx**, and a dynamic set *nod* can add elements of *NSet* to itself via the guarded event *AddNode*. The dynamics of the machine consists of the state transitions that ensue as events are executed one by one. Each possible sequence of transitions can be collected into a system trace. It is assumed that, in the real world, some non-empty interval of real world time elapses between each event occurrence.

The restriction to discrete events makes the original Event-B poorly suited for investigating the hybrid and cyber-physical systems that are prevalent today. But the real-time gap assumed to be present between discrete event occurrence, makes a convenient opening for interleaving pieces of continuous behaviour into the time intervals between them, and this was how Hybrid Event-B was designed.

The intuition just described results in there being two kinds of event in Hybrid Event-B. There are **mode** events, which specify discrete changes of state —just as in Event-B— and there are **pliant** events, which specify episodes of continuous state update. Since we have continuous state update, a number

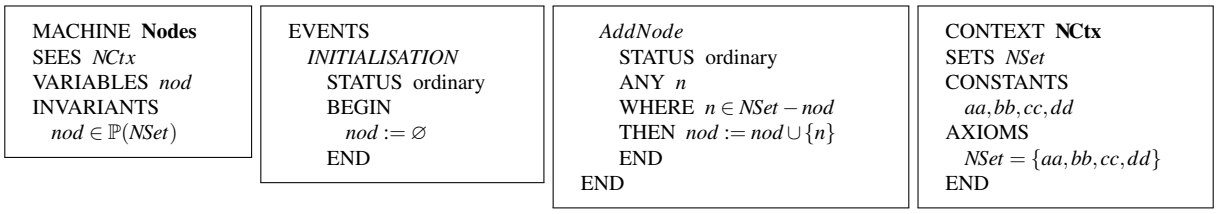


Figure 1: A simple Event-B machine, together with its context.

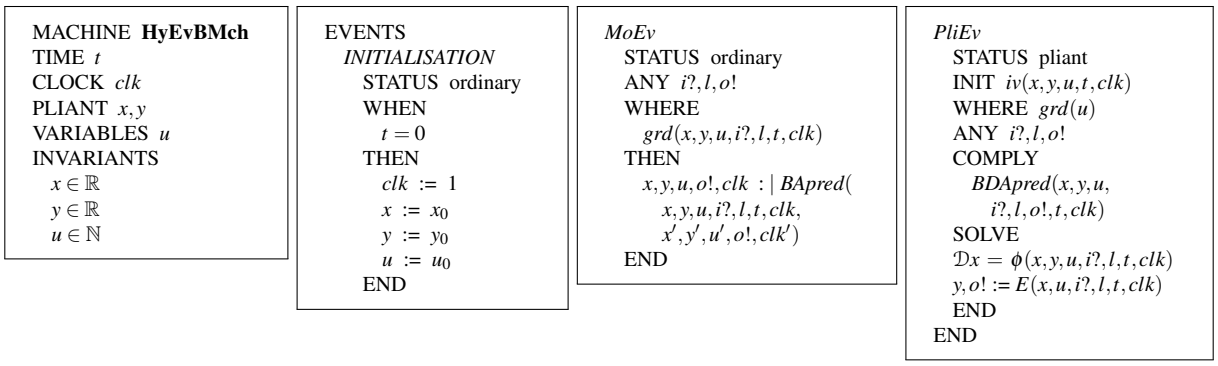


Figure 2: A schematic Hybrid Event-B machine.

of technical details have to be handled. Time becomes a crucial entity — all variables now become, semantically, functions of real time (and not, as in Event-B, functions of an index into a sequence of state values). We have to have a policy about how wild, or well behaved, the functions of time are permitted to be, and for that, we demand that these functions have well defined left and right limits everywhere, and are continuous from the right. We also want to use differential equations (ODEs) to specify behaviour, so we need to restrict to functions which are piecewise solutions to ODE systems.

Fig. 2 shows a simple Hybrid Event-B machine. After the machine name is the TIME declaration, which names the variable used to denote real time (if needed). This permits read-only access to time in the rest of the machine. Time is synchronised (via a WHEN clause) with the start of a run in the *INITIALISATION*. Next comes a CLOCK variable *clk*; these can be reset to some value (but are otherwise read-only) and allow time to be measured from convenient starting points. Then come the PLIANT and VARIABLES declarations. The former introduces the pliant variables, while the latter introduces the mode variables. Next come the INVARIANTS. These are required to hold *at all times*, so, for example the invariant '$x \in \mathbb{R}$' means that the function of time that is the semantic value of the variable x is a *real valued* function of time. A little thought shows that this is the same convention used in Event-B (aside from time being real rather than an index). Accepting that $x \in \mathbb{R}$ holds at all times, does not help in knowing how values of x at 'nearby' times are related. To keep things under control, we must insist that the semantic value of x is, again, at least piecewise, a solution to some ODE system.

The events are next. The mode events (STATUS ordinary) are *exactly* as in Event-B, and the syntax of *MoEv* in Fig. 2 shows which variables can be updated by a mode event, using the generic *BApred* syntax. The pliant events (STATUS pliant), such as *PliEv*, contain novel features and require more care. The WHERE guard is as in Event-B, imposing enabledness conditions that involve mode variables only. The INIT guard imposes enabledness conditions that involve the pliant variables (but which can also involve mode variables, if needed). The actual state update is specified in the COMPLY and SOLVE clauses. Since the purpose of a pliant event is to specify state update over an extended period of time, some tricky technical issues arise, which we mostly ignore in this overview. See PaperI for a full discussion.

The SOLVE clause is the easiest to describe. It can contain one or more ODEs (with syntax $\mathcal{D}x = \phi$) and one or more direct assignments (with syntax $y := E$). Provided the RHSs of these forms are as well behaved as stipulated above, no special problems arise. Thus, direct update to something which is already well behaved poses no problems, and integrating an ODE with well behaved RHS improves the already acceptable behaviour as regards smoothness. By contrast, the COMPLY clause imposes invariant-like constraints that must hold during the execution of the pliant event, and are indicated by *BDApred*, the

before-during-after clause (the analogue of Event-B's *BApred*). Essentially, the same caveats regarding time dependent behaviours that we mentioned in the context of invariants must hold for the properties that comprise *BDApred*.

As for almost all formal semantics of hybrid system notations, the semantics of Hybrid Event-B is operational. Starting with the *INITIALISATION* mode event, mode and pliant event executions alternate within a run. Each mode event execution must enable at least one pliant event and disable all mode events — after which a pliant event execution then takes over. Each pliant event execution runs until one of three things happens. (1) Some mode event becomes enabled. At that moment the pliant event execution is preempted, and then some enabled mode event execution takes over. (2) The running pliant event becomes infeasible and the run stops (finite termination). (3) Neither of the preceding options occurs and the pliant event runs forever (nontermination). Note the language in the immediately preceding remarks. They indicate that there are choices to be made at mode→pliant handover and pliant→mode handover. These choices are nondeterministic among all the events that are enabled at the requisite moment. They must all be of the right kind (either all mode or all pliant), or else the run aborts.

In more concrete terms, a run of a Hybrid Event-B machine looks like the following. Time $\mathscr{T}$ corresponds with a semi-infinite interval of the reals, e.g. $\mathscr{T} = \mathbb{R}_{\geq 0}$. The operational semantics partitions this into a sequence of non-empty, left-closed, right-open intervals $\mathscr{T} \equiv \langle [t_0 \ldots t_1), [t_1 \ldots t_2), \ldots \rangle$. Thus, if t_0 is the initial time point, the state of the machine's variables at t_0 is as specified in the *INITIALISATION* mode event. During $[t_0 \ldots t_1)$, i.e. during the set of times $t_0 \leq t < t_1$, the state evolves as specified in the pliant event chosen to execute immediately after *INITIALISATION*. This runs until it is preempted by some mode event which is enabled at time t_1. At that moment, the mode event executes, and the state changes discontinuously. The mode event's before-state is the left limit of the state evolution at t_1, and the mode event's after-state is the value at t_1 itself. This pattern repeats. So $[t_1 \ldots t_2)$ is the duration of the next pliant event execution, after which a mode event is executed at t_2, followed by the next pliant event execution. And so on.

To ensure that all executions of a Hybrid Event-B machine take place as described, a (considerable) number of proof obligations (POs) need to be shown to hold. Some of these, particularly concerning mode events, closely resemble their Event-B counterparts. Others, particularly concerning pliant events, contain novel elements. In all cases though, some potentially delicate technicalities connected with continuous behaviour intrude into the formulation of the POs. We do not have space to discuss these here, so we refer to PaperI and PaperIII for details.

3 Hybrid Event-B Refinement

As for all dialects of the B-Method, refinement is an important topic for Hybrid Event-B. Given that Hybrid Event-B pertains to a context in which real world time plays a significant role, there are a number of ways to formulate a refinement notion. Given further that in refinement there has to be an abstract machine and a concrete machine, the most urgent question concerns how the notions of 'real world time' are related to one another in the two machines. We could allow the two notions to be a bit 'elastic' with respect to one another, using something like a Skorokhod metric to measure how far apart the elasticity had stretched abstract and concrete time and behaviour. However, this was felt to be technically too obscure for a formalism like Hybrid Event-B, that was aimed at practical engineering purposes. So, for Hybrid Event-B, a more conservative approach was taken, and it became a matter of principle, that **Hybrid Event-B preserves the notion of real world time through refinement**. In fact, it can be said that, although it is never stated, this is an assumption that also typically applies to Event-B itself, insofar

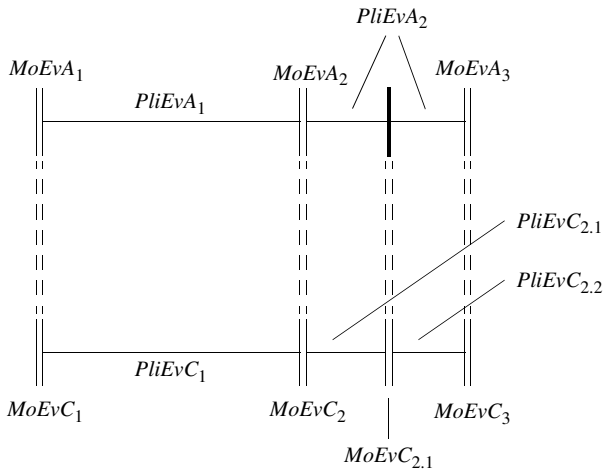


Figure 3: Illustrating Hybrid Event-B refinement.

as Event-B models are intended to reflect phenomena in the real world. If we add to this principle the desire that Hybrid Event-B refinement does not disturb the essential structural features of Event-B refinement, which amounts to saying that Hybrid Event-B refinement of mode events works as does refinement of events in Event-B, a quite strong set of constraints is generated.

Fig. 3 shows what happens. The closely spaced vertical bars represent the before- and after-states of mode event executions, while the horizontal lines between them represent the continuous state change of pliant event executions. For the moment let us forget the occurrences of $PliEv\_$ and pretend that the horizontal lines just represent the passage of time, i.e. that we are dealing with Event-B refinement in a situation in which real world time passes between event executions. The traditional Event-B refinement proof obligations tie together what happens at abstract and concrete levels quite closely. Thus for every abstract event execution, there is a corresponding concrete event execution of its refinement (by Event-B relative deadlock freedom). Conversely, for every event execution of the concrete refinement of an abstract event, there is a corresponding abstract event execution of its abstraction (by Event-B guard strengthening). Between execution occurrences of abstract event refinements, execution occurrences of 'new' events, freshly introduced at the concrete level, can appear. These must refine abstract skips (represented by a solid vertical bar in Fig. 3).

Transferring this verbatim to the Hybrid Event-B world says how refinement of mode events works. Fig. 3 thus shows abstract $MoEvA_1, MoEvA_2, MoEvA_3$, which are refined to $MoEvC_1, MoEvC_2, MoEvC_3$, at the same moments of time as their abstract counterparts. 'New' mode event $MoEvC_{2.1}$ executes at some point between $MoEvC_2$ and $MoEvC_3$.

Pliant events must fit round this. An immediate consequence is that the durations of abstract pliant event executions are equal to, or are partitioned by, suitable concrete pliant event execution durations — the intervals of abstract pliant event executions are related to the intervals of concrete pliant event executions via an inverse function. Beyond this, the natural extension of the principle that abstract mode events are refined by corresponding concrete mode events implies that abstract pliant events are refined by corresponding concrete pliant events too. In Fig. 3 this would imply that $PliEvA_1$ was refined by $PliEvC_1$ and that $PliEvA_2$ was refined by $PliEvC_{2.1}$.

Without 'new' concrete events, there would be little more to be said, and new mode events cause little trouble by themselves. It turns out that provided the concrete refinements of abstract events are not

constrained to have the same name, there is no possible distinction between old and new pliant events. Observe that the duration of $PliEvC_{2.1}$ in Fig. 3 is strictly shorter than the duration of $PliEvA_2$. To close the gap in time at the concrete level, some pliant event, $PliEvC_{2.2}$ must execute. There is no alternative to the fact that $PliEvC_{2.2}$ must refine $PliEvA_2$ — both events have an extended duration so there is no counterpart of the 'refining of skip' that can apply to mode events. The main attendant issue that arises is the fact that prior to the execution of $MoEvC_{2.1}$, $PliEvA_2$ was executing. And while it was not able to change any mode variables during this period, it certainly was able to change some pliant variables, this being its main purpose. So, by the time the guards of $PliEvC_{2.2}$ need to be checked, the abstract state is, in general, not the same as it was when the execution of $PliEvA_2$ was launched. This, despite the fact that the only event that the guards of $PliEvC_{2.2}$ might be checked against is still $PliEvA_2$. The relevant PO resolves this by making the checking of the *iv* guard of the corresponding abstract event optional. Checking of the abstract *grd* guard works, since *grd* only involves mode variables, which will not have changed during the time since $PliEvA_2$ was launched. Whether it makes sense or not to also check the abstract *iv* guard is highly dependent on the details of $PliEvA_2$ and of the rest of the abstract machine.

As for machines, a large number of proof obligations need to be discharged to ensure that a refinement behaves as just described. Again, we do not have space to discuss these here, so we refer to PaperI and PaperIII for details.

4 Multiple Hybrid Event-B Machines

In today's engineering landscape, having a design and development methodology that does not cater for separate development of components, is almost inconceivable. Therefore, since Hybrid Event-B is certainly intended for realistic formal system development, the implications of separate development, in the context of all the other constraints that the formalism imposes, must be confronted. Earlier work on combining and decomposing Event-B machines has yielded a number of schemes, based on shared variables, shared events and interfaces [2, 12, 13, 8, 15, 16]. The Hybrid Event-B scheme, investigated in detail in PaperII, is based on ideas from all of these, taking into account the special needs of both mode and pliant events.

When multiple syntactic constructs have to come together to make a bigger whole, a key issue is syntactic visibility: which elements of which entities are visible/readable/writable in which other entities, and what restrictions are in place to control this? This realisation was the key concept behind the introduction of the INTERFACE construct in Hybrid Event-B. An INTERFACE is a container that can hold a number of variables, the invariants that they must satisfy, and their initialisations. It is thus rather like a machine without events (or with one unstated default event simply demanding compliance with the invariants). A machine can access an interface via a CONNECTS clause or a READS clause. This allows the machine's events to inspect (READS) and update (CONNECTS) the interface's variables. The interface's invariants are aggregated with the machine's during verification. Since more than one machine can do this, we have a mechanism for sharing variables.

In multi-machine Hybrid Event-B, to prevent a verification free-for-all that would impede separate working, invariants are restricted to be of two kinds. There are type I invariants (tIi's) which only mention variables from the construct (MACHINE or INTERFACE) that declares them. By themselves these are too restrictive to allow sufficiently expressive multi-machine working, so there are also type II invariants (tIIi's) which are exclusive to interfaces, and are of the form $U(u) \Rightarrow V(v)$, where u are variables that belong to one interface (the tIIi's local variables and interface) and v are variables that belong to a different interface (the tIIi's remote variables and interface). Type II invariants are declared in the in-

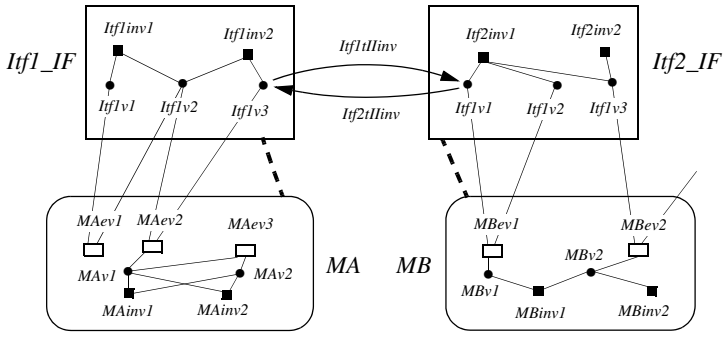


Figure 4: Multiple machines and their interfaces.

terface containing the local variables. The remote interface of a tIIi is mentioned in the local interface using a READS declaration while the local interface of a tIIi is mentioned in the remote interface using a REFERS declaration.

We give a small illustration of these principles in Fig. 4. Small black disks represent variables, while small black squares represent tIi's. Small rectangles represent events. Events and invariants are connected to the variables that mention them by thin lines. Interfaces are large rectangles containing the variables and invariants they encapsulate — there are two in Fig. 4, *Itf1_IF* and *Itf2_IF*. Machines are large rounded rectangles containing their local variables, tIi's and events — again there are two, *MA* and *MB* in the figure. The very short thin line with one end free from an event in *MB*, is an I/O variable connected to the environment. The CONNECTS relationship between a machine and an interface is depicted by a thick dashed line. Finally, tIIi's are represented by an arrow from a variable in the local interface to a variable in the remote interface (these containing the local and remote variable subsets of the tIIi respectively). The construct that aggregates all the machines and interfaces of a development is the PROJECT construct, which is not indicated in Fig. 4.

Refinement adds some potential complexity to the above. Additional constraints are needed to continue to prevent a verification free-for-all that would impede separate working. It is taken as unquestionable that machines and interfaces should both be capable of being refined. The restriction imposed on the refinement process is that the joint invariant that couples concrete variables to their abstract counterparts in a construct (be it a machine or an interface), should involve only the variables declared in the two constructs, and cannot involve variables declared in any other interface that either construct has access to. It is clear that such a restriction ensures that invariants proved at a high level continue to hold, suitably translated into concrete variables through the joint invariant, at lower levels of abstraction.

In a single Hybrid Event-B machine, mode events and pliant events alternate. Within a single machine context, once the nuances of continuous/discrete behaviour are appreciated, it is fairly easy to work out how POs should be designed that can adequately police the alternation. Things are more complicated in a multi-machine world. We design separate machines on the basis that their activities are expected to be largely (though obviously not totally) independent. What does the alternation between mode events and pliant events amount to in such an environment? The answer requires a design decision about the multi-machine world.

One guiding principle is that every machine, at all times, is executing some event (whether mode or pliant). Another is that in the purely discrete Event-B world, synchronised execution of events in multiple machines is a useful feature in some of the formulations cited above. Accordingly, the decision was taken to permit the synchronised execution of mode events via the introduction of a SYNCHronisation

construct that declares that families of mode events across different machines of the same project are to be scheduled together. Of course the conjunction of the relevant guards has to permit this to be the case.

Synchronised execution of events across multiple machines is of most interest in the context of *decomposition*, in which refinement of a single abstract event has made it too unwieldy for it to be considered as a single indivisible entity any more, and the execution of different parts of it in different machines makes more sense. Decomposition feels like the converse of composition (which is what the multi-machine mechanisms are addressing) but it is not — the event scheduling strategies of (Hybrid) Event-B get in the way. Thus suppose abstract machine *M* has two mode events *EvA* and *EvB*, both simultaneously enabled. Then, the scheduling strategy of *M* prevents the simultaneous execution of *EvA* and *EvB*. Now suppose *EvA* is decomposed into *EvAX* and *EvAY*, and *EvB* is decomposed into *EvBX* and *EvBY*, and the *X* parts execute in decomposed machine *MX*, and the *Y* parts execute in decomposed machine *MY*. It is not inconceivable that in a world where the two machines schedule their events independently, *EvAX* could be scheduled simultaneously with *EvBY*, giving the wrong semantics. It is for that reason that the SYNCH mechanism was introduced in Hybrid Event-B (following analogous mechanisms in Event-B). Decomposition of events requires further thought regarding inputs, outputs and local parameters. The design in PaperII requires that any input or output of a decomposed event is confined to one of the components (rather than being itself a synchronised activity). Local parameters are likewise assigned by one of the components and then shared with the rest, in a conceptually atomic single-writer multiple-readers rendezvous.

Note that we have said nothing of synchronising or decomposing pliant events. Pliant events raise their own issues in a multi-machine world. We recall that pliant events executing in a machine typically get preempted. If there are multiple machines, the preemptions in different machines should be independent, otherwise the different machines are behaving like a single machine. This presents little problem in the real world context, but creates technical complexity for a formal operational semantics which is concerned with constructing a consistent global system trace. At a preemption in machine *MA*, the non-preempted other machines must be 'paused' and their execution resumed after the scheduling choices in machine *MA* have been resolved. And because this is a strange thing to contemplate for the physical world, the physical world consequences of any such technicalities must be invisible. This affects potential decomposition strategies for pliant events. In the light of these considerations, the strategy adopted for decomposition of pliant events is that it has to be programmed explicitly by the user. If pliant events are required to execute simultaneously in two machines, their guards must be arranged to be the same (and the same as the guards of the parent event if they constitute a decomposition), and to prevent 'the wrong parts' of different decompositions from being scheduled simultaneously, no parent events in the same machine that are to be decomposed can ever be enabled simultaneously. Fortunately, it is easy enough to impose static restrictions on event guards that can ensure the conditions required. A full discussion can be found in PaperII.

5 A Small Incident Response Case Study

We illustrate the utility of the multi-machine Hybrid Event-B approach with a small case study. For lack of space we do not cover all aspects in equal depth, but focus on those structural elements which particularly contribute to convenient exploration of distributed and autonomous systems.

The case study concerns an incident response system. This consists of a number of largely independent agents: a controller, three responders that enter a potentially hazardous area to effect some appropriate measures, and two drones, that act as communication intermediaries between the controller

(which is assumed to be ground based) and the responders (which are assumed to be unable to communicate with the controller directly). Since the arena of the incident needing the response is assumed to be hazardous, two drones are provided, and they are expected to keep apart from each other, in case one of them suffers some disabling mishap. Described in such abstract terms, the case study is applicable to situations from natural and industrial disasters, through terrorist attacks, to battlefield warfighting.

Each of these agents is modelled using an individual Hybrid Event-B machine, and there are additionally CONTEXTs and INTERFACEs that organise the means by which the various agents can cooperate. The ability to do this successfully results from the ability to partition the functionality needed into constructs that capture convincingly self-contained subsets, and the adequate flexibility of the cooperation mechanisms, that enables their shared goals to be achieved. These mechanisms have to cope with six independent but cooperating agents in the complete system, each potentially involving independently determined smooth state change.

The essential lesson of the case study is that the design of multi-machine Hybrid Event-B permits the system to be described in such a way that experimentation with different scenarios can be easily achieved by changing just one machine and one context. These are **EnvironmentScenario_Mch** and **IncidentResponse_CTX**.

In the body of this paper we focus on the machines that capture the behaviours of the agents, relegating the contexts and interfaces that contain the static declarations that support these machines to the Appendix. Here and there we take some liberties with Event-B syntax for the sake of readability. Next, we see **EnvironmentScenario_Mch**.

MACHINE **EnvironmentScenario_Mch**
SEES *IncidentResponse_CTX*
CONNECTS *IncidentResponse_IF*
VARIABLES
  schedule
INVARIANTS
  $schedule : \text{seq}(\mathbb{R})$
  $\text{increasing}(schedule)$
EVENTS
  INITIALISATION
    STATUS ordinary
    BEGIN
      $schedule := INITSCHED$
    END
  PliTrue
    STATUS pliant
    COMPLY
      INVARIANTS
    END
... ...

... ...
  AddHazard
    STATUS ordinary
    ANY tg, xx, yy, sz, ht
    WHERE
      $\text{nonempty}(schedule) \land$
      $t = \text{head}(schedule) \land$
      $(tg \mapsto xx \mapsto yy \mapsto sz \mapsto ht) \notin hazards$
    BEGIN
      $hazards := hazards \cup \{tg \mapsto xx \mapsto yy \mapsto sz \mapsto ht\}$
      $schedule := \text{tail}(schedule)$
    END
  TakeHazard
    STATUS ordinary
    ANY tg, xx, yy, sz, ht
    WHERE
      $\text{nonempty}(schedule) \land$
      $t = \text{head}(schedule) \land$
      $(tg \mapsto xx \mapsto yy \mapsto sz \mapsto ht) \in hazards$
    BEGIN
      $hazards := hazards - \{tg \mapsto xx \mapsto yy \mapsto sz \mapsto ht\}$
      $schedule := \text{tail}(schedule)$
    END
END

The **EnvironmentScenario_Mch** machine above contains a *schedule* of interventions into the incident arena. These are restricted to be the introduction and removal of hazardous areas within the arena, and the *schedule* variable is a succession (i.e. sequence) of times at which the interventions take place. Each hazardous area has either a *SQ*uare footprint or a *CYL*indrical shape. So each is specified using a *SQ* or *CYL* tag tg, x and y coordinates, a size sz, and a height ht. The **EnvironmentScenario_Mch** machine uses the *AddHazard* event to introduce a fresh hazard when time reaches the first value in *schedule*, with the attributes of the hazard, written in B-Method notation as $(tg \mapsto xx \mapsto yy \mapsto sz \mapsto$

ht), being taken from the formal model's environment using the ANY clause. The *TakeHazard* event removes an existing hazard from the set of current hazards *hazards*. The *schedule* itself is statically defined in the **IncidentResponse_CTX** context. Changing the constants in the context, and the details of the **EnvironmentScenario_Mch** machine is all one needs to do to experiment with different incident management scenarios in this world.

Below, we see the project file, **IncidentResponse_Prj**. Its purpose is to orchestrate all the components of the system. Many specific details are curtailed to save space. The first declared item is the GLOBalINVariantS file **IncidentResponse_GI** which we discuss at the end. After that there is a list of contexts, interfaces and machines. The line 'CONTEXT **Drone1_CTX** IS' instantiates the context as a version of a generic library context for drones, **Drone_CTX**. After the WITH, there is a renaming mapping saying how identifiers in **Drone_CTX** should be replaced to get **Drone1_CTX**. This instantiation allows the two contexts **Drone1_CTX** and **Drone2_CTX** to have different constants $Vdr1$ and $Vdr2$ for the velocities of the drones **Drone1_Mch** and **Drone2_Mch** that SEE the respective contexts. After the interfaces there are the machines. **Controller_Mch** is discussed below, while **EnvironmentScenario_Mch** was discussed above. After these, there are the two drone instantiations **Drone1_Mch** and **Drone2_Mch**, being instantiations of a generic **Drone_Mch**. We have suppressed the details of the instantiations, which amount to adding '1' or '2' to identifiers in the generic machine.

Beyond instantiations, the project file contains SYNCH lines. These specify collections of mode events across multiple machines that must be scheduled simultaneously (i.e. only when all their guards are simultaneously true). Thus 'SYNCH **ActivateDrone1**' enforces the simultaneous execution of event

```
PROJECT IncidentResponse_Prj
GLOBINVS IncidentResponse_GI
CONTEXT IncidentResponse_CTX
CONTEXT Controller_CTX
CONTEXT Drone1_CTX IS
  Drone_CTX WITH
    Vdr → Vdr1
  END
CONTEXT Drone2_CTX IS
  Drone_CTX WITH
    Vdr → Vdr2
  END
CONTEXT Responder_CTX
INTERFACE IncidentResponse_IF
INTERFACE ControllerDrones_IF
INTERFACE ControllerResponder_IF
MACHINE Controller_Mch
MACHINE EnvironmentScenario_Mch
MACHINE Drone1_Mch IS
  Drone_Mch WITH
    •••
  END
MACHINE Drone2_Mch IS
  Drone_Mch WITH
    •••
  END
MACHINE Responder1_Mch IS
  Responder_Mch WITH
    •••
  END
MACHINE Responder2_Mch IS
  Responder_Mch WITH
    •••
  END
... ...
```

```
... ...
MACHINE Responder3_Mch IS
  Responder_Mch WITH
    •••
  END
SYNCH ActivateDrone1
    Controller_Mch.LaunchDrone1
    Drone1_Mch.Activate1
  END
SYNCH UpdateDrone1
    Controller_Mch.UpdateDrone1
    Drone1_Mch.Update1
  END
SYNCH DeActivateDrone1
    Controller_Mch.RecallDrone1
    Drone1_Mch.DeActivate1
  END
SYNCH ••• Drone2
SYNCH ActivateResponder1
    Controller_Mch.LaunchResponder1
    Responder1_Mch.Activate1
  END
SYNCH UpdateResponder1
    Controller_Mch.UpdateResponder1
    Responder1_Mch.Update1
  END
SYNCH DeActivateResponder1
    Controller_Mch.RecallResponder1
    Responder1_Mch.DeActivate1)
  END
SYNCH ••• Responder2
SYNCH ••• Responder3
END
```

LaunchDrone1 in the **Controller_Mch** machine and of **Activate1** in the **Drone1_Mch** machine. There are similar synchronisations for **Upate**ing and for **DeActivate**ing the **Drone1_Mch** machine. The analogous synchronisations for the **Drone2_Mch** machine are suppressed. A similar pattern applies to the responders. The details for the **Responder1_Mch** machine are given in full, while those for machines **Responder2_Mch** and **Responder3_Mch** are suppressed.

MACHINE **Controller_Mch**
SEES **IncidentResponse_CTX**
SEES **Controller_CTX**
CONNECTS **IncidentResponse_IF**
CONNECTS **ControllerDrone1_IF**
CONNECTS **ControllerDrone2_IF**
CONNECTS **ControllerResponder1_IF**
CONNECTS **ControllerResponder2_IF**
CONNECTS **ControllerResponder3_IF**
VARIABLES
mode
ctrhazards
cyclestart
drones2comd
responders2comd
INVARIANTS
$mode : CTRLSTATE$
$ctrhazards : \mathbb{P}(HAZTYPE \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R})$
$cyclestart : \mathbb{R}$
$drones2comd : \mathbb{P}(\{1,2\})$
$responders2comd : \mathbb{P}(\{1,2,3\})$
EVENTS
INITIALISATION
STATUS ordinary
BEGIN
$mode := OFF$
$ctrhazards := \varnothing$
$cyclestart := 0$
$drones2comd := \varnothing$
$resps2comd := \varnothing$
END
PliTrue
STATUS pliant
COMPLY
INVARIANTS
END
ActivateController
STATUS asynch
WHEN
$mode = OFF \wedge 0 < t < \delta$
THEN
$mode := DISPATCH$
$drones2comd := \{1,2\}$
$resps2comd := \{1,2,3\}$
END
LaunchDrone1
STATUS asynch
WHEN
$mode = DISPATCH \wedge 1 \in drones2comd \wedge t < \delta$
THEN
$drhazards := hazards$
$drones2comd := drones2comd - \{1\}$
END
LaunchDrone2
•••
END
... ...

... ...
LaunchResp1
STATUS asynch
WHEN
$mode = DISPATCH \wedge$
$1 \in resps2comd \wedge t < \delta$
THEN
$resp1hazards := hazards$
$resps2comd := resps2comd - \{1\}$
END
LaunchResp2
•••
END
LaunchResp3
•••
END
StartMonitoring
STATUS asynch
WHEN
$mode = DISPATCH \wedge t = \delta$
THEN
$mode := UPDATEHAZ$
$cyclestart := \Delta$
END
MonitorHazardsNull
STATUS asynch
WHEN
$mode = UPDATEHAZ \wedge 0 < t - cyclestart < \delta/2 \wedge$
$hazards = ctrhazards$
THEN
$cyclestart := cyclestart + \Delta$
END
MonitorHazardsUpdate
STATUS asynch
WHEN
$mode = UPDATEHAZ \wedge 0 < t - cyclestart < \delta/2 \wedge$
$hazards \neq ctrhazards$
THEN
$ctrhazards := hazards$
$drones2comd := \{1,2\}$
$resps2comd := \{1,2,3\}$
$cyclestart := cyclestart + \Delta$
END
UpdateDrone1
STATUS asynch
WHEN
$mode = UPDATEHAZ \wedge 1 \in drones2comd \wedge$
$cyclestart - \Delta + \delta/2 < t < cyclestart - \Delta + \delta$
THEN
$drhazards := hazards$
$drones2comd := drones2comd - \{1\}$
END
UpdateDrone2
•••
END
... ...

```
... ...
  UpdateResp1
    STATUS asynch
    WHEN
      mode = UPDATEHAZ ∧ 1 ∈ resps2comd ∧
        cyclestart − Δ + δ/2 < t < cyclestart − Δ + δ
    THEN
      resp1hazards := hazards
      resps2comd := resps2comd − {1}
    END
  UpdateResp2
      •••
    END
  UpdateResp3
      •••
    END
  EndMonitoring
    STATUS asynch
    WHEN
      mode = UPDATEHAZ ∧
        DURATION < t < DURATION + δ
    THEN
      mode := RECALL
      drones2comd := {1,2}
      resps2comd := {1,2,3}
    END
... ...
```

```
... ...
  RecallDrone1
    STATUS asynch
    WHEN
      mode = RECALL ∧ 1 ∈ drones2comd ∧
        DURATION < t < DURATION + δ
    THEN
      drones2comd := drones2comd − {1}
    END
  RecallDrone2
      •••
    END
  RecallResp1
    STATUS asynch
    WHEN
      mode = RECALL ∧ 1 ∈ resps2comd ∧
        DURATION < t < DURATION + δ
    THEN
      resps2comd := resps2comd − {1}
    END
  RecallResp2
      •••
    END
  RecallResp3
      •••
    END
  DeActivateController
    STATUS asynch
    WHEN
      mode = RECALL ∧
        DURATION + δ < t < DURATION + 2δ
    THEN
      mode := OFF
    END
END
```

Following the project file, there is the most complex machine of the project, the **Controller_Mch** machine, occupying the four panels above. Essentially, it is a finite state machine, perfectly expressible using Event-B alone if need be, except that the availability of real time for scheduling purposes simplifies the state space that would otherwise be needed.

The main job of the controller, once activated (event *ActivateController*) is: to send forth the drones and responders (events *LaunchDrone1*, *LaunchDrone2*, *LaunchResp1*, *LaunchResp2*, *LaunchResp3*), to poll the environment machine to discover changes to the collection of hazards that have been unearthed (events *StartMonitoring*, *MonitorHazardsNull*, *MonitorHazardsUpdate*) and to advise the drones and responders of the same (events *UpdateDrone1*, *UpdateDrone2*, *UpdateResp1*, *UpdateResp2*, *UpdateResp3*), and finally to recall the drones and responders when the work has been completed (events *RecallDrone1*, *RecallDrone2*, *RecallResp1*, *RecallResp2*, *RecallResp3*). Once this is done, the controller is deactivated (event *DeActivateController*). A default *PliTrue* pliant event covers the gaps between state change.

In the above, there are clearly groups of similar events that need to take place periodically (launching, monitoring, updating, recalling) for agents that are largely independent of each other most of the time. Achieving this using a pure finite state machine would entail either a state explosion to accommodate all possible interleavings of individual agent events, or an excessive use of sequentialisation to avoid it. The presence of time as an intrinsic feature in Hybrid Event-B (assumed synchronised across all machines) permits a more economical approach.

A 'large' time window Δ is defined along with a 'small' window δ. Launching is defined to take place within the first δ of elapsed time, via synchronised events, as described above. Since the main

synchronisation mechanism is the time interval of duration δ, arbitrary orderings of the launching events within that interval are permitted.

The same technique is used for all the other synchronised activities. Monitoring and update take place repeatedly after the elapse of each period of duration Δ. At $t = n\Delta$, the EnvironmentScenario machine potentially updates the *hazards*. Within the next $\delta/2$ period, the controller checks whether *hazards* has changed since the last check, and if so, schedules updates to the drones and responders to update their hazard data and to replan if necessary. To keep the model simple, this is accomplished by setting the *drones*2*comd* and *resps*2*comd* variables to the IDs of all the drones and responders. This triggers, in the next $\delta/2$ period, each of them to update it's local copy of the *hazards* variable via a synchronised event such as e.g., the combination of *UpdateDrone1* in the controller together with *Update* in first drone. Once that has happened, the drone or responder can recalculate its trajectory.

```
MACHINE Drone_Mch
SEES IncidentResponse_CTX
SEES Drone_CTX
CONNECTS ControllerDrones_IF
VARIABLES
  mode
  thex , they , thez
  trajectory
INVARIANTS
  mode : DRONESTATE
  thex , they , thez : ℝ , ℝ , ℝ
  trajectory : seq(ℝ × ℝ × ℝ)
EVENTS
  INITIALISATION
    STATUS ordinary
    BEGIN
      mode := OFF
      thex , they , thez := 0 , 0 , 0
      trajectory := ⟨⟩
    END
  PliTrue
    STATUS pliant
    WHEN
      mode = OFF
    COMPLY
      INVARIANTS
    END
  Activate
    STATUS ordinary
    WHEN
      mode = OFF
    BEGIN
      mode := SEEK
      thex , they , thez := drx , dry , drz
      trajectory := calcCentAvoidTraj(...)
    END
  Update
    STATUS ordinary
    WHEN
      mode ∈ {SEEK , RETURN}
    BEGIN
      thex , they , thez := drx , dry , drz
      trajectory := calcCentAvoidTraj(...)
    END
... ...
```

```
... ...
  Navigate
    STATUS pliant
    WHEN
      mode ∈ {SEEK , RETURN} ∧
      trajectory ≠ ⟨⟩
    SOLVE
      𝒟drx := Vdr × (first(trajectory)[1] − thex)
      𝒟dry := Vdr × (first(trajectory)[2] − they)
      𝒟drz := Vdr × (first(trajectory)[3] − thez)
    END
  Waypoint
    STATUS ordinary
    WHEN
      mode ∈ {SEEK , RETURN} ∧
      drx = first(trajectory)[1] ∧
      dry = first(trajectory)[2] ∧
      drz = first(trajectory)[3] ∧
      trajectory ≠ ⟨⟩
    BEGIN
      thex , they , thez := drx , dry , drz
      trajectory := rest(trajectory)
    END
  Hover
    STATUS pliant
    WHEN
      mode ∈ {SEEK , RETURN} ∧
      trajectory = ⟨⟩
    COMPLY
      INVARIANTS
    END
  DeActivate
    STATUS ordinary
    WHEN
      mode = SEEK
    BEGIN
      mode := RETURN
      thex , they , thez := drx , dry , drz
      trajectory := calcCentAvoidTraj(...)
    END
  SwitchOff
    STATUS ordinary
    WHEN
      mode = RETURN ∧
      drx = dry = drz = 0
    BEGIN
      mode := OFF
    END
END
```

The cycle of updating and recalculating continues until it is time to recall all the drones and responders. This is achieved by the same synchonisation mechanism. In other words, after *DURATION* has elapsed, similar windows of length δ are established and the drones and responders adopt first the *RETURN* mode, and after returning home, the *OFF* mode.

The generic drone machine **Drone_Mch** appears in the two panels on the previous page. Once a drone is activated through the synchronised *Activate* event, it ceases the default *PliTrue* behaviour and instead pursues the *Navigate* behaviour. The preceding mode event occurrence, whether an *Activate* or an *Update* event, caused it to remember its 3D position at that time in variables *thex*, *they*, *thez*, from

MACHINE **Responder_Mch**
SEES **Responder_CTX**
CONNECTS **ControllerResponder_IF**
VARIABLES
  $mode$
  $thex$, $they$
  $trajectory$
PLIANT
  $respx$, $respy$
INVARIANTS
  $mode : RESPSTATE$
  $thex$, $they : \mathbb{R}$, $\mathbb{R}$
  $respx$, $respy : \mathbb{R}$, $\mathbb{R}$
  $trajectory : \text{seq}(\mathbb{R} \times \mathbb{R})$
EVENTS
  INITIALISATION
    STATUS ordinary
    BEGIN
      $mode := OFF$
      $thex$, $they := 0$, 0
      $respx$, $respy := 0$, 0
      $trajectory := \langle\rangle$
    END
  PliTrue
    STATUS pliant
    WHEN
      $mode = OFF$
    COMPLY
      INVARIANTS
    END
  Activate
    STATUS ordinary
    WHEN
      $mode = OFF$
    BEGIN
      $mode := SEEK$
      $thex$, $they := respx$, $respy$
      $trajectory := calcTraj(\ldots)$
    END
  Update
    STATUS ordinary
    WHEN
      $mode \in \{SEEK , RETURN\}$
    BEGIN
      $thex$, $they := respx$, $respy$
      $trajectory := calcTraj(\ldots)$
    END
... ...

... ...
  Navigate
    STATUS pliant
    WHEN
      $mode \in \{SEEK , RETURN\} \wedge$
      $trajectory \neq \langle\rangle$
    SOLVE
      $\mathcal{D}respx := Vdr \times (\text{first}(trajectory)[1] - thex)$
      $\mathcal{D}respy := Vdr \times (\text{first}(trajectory)[2] - they)$
    END
  Waypoint
    STATUS ordinary
    WHEN
      $mode \in \{SEEK , RETURN\} \wedge$
      $respx = \text{first}(trajectory)[1] \wedge$
      $respy = \text{first}(trajectory)[2] \wedge$
      $trajectory \neq \langle\rangle$
    BEGIN
      $thex$, $they := respx$, $respy$
      $trajectory := \text{rest}(trajectory)$
    END
  Arrived
    STATUS asynch
    WHEN
      $mode = SEEK \wedge trajectory = \langle\rangle$
    BEGIN
      $mode := ARRIVED$
    END
  DoSomethingForAWhile
    STATUS pliant
    WHEN
      $mode = ARRIVED$
    COMPLY
      INVARIANTS
    END
  DeActivate
    STATUS ordinary
    WHEN
      $mode = ARRIVED$
    BEGIN
      $mode := RETURN$
      $thex$, $they := respx$, $respy$
      $trajectory := calcTraj(\ldots)$
    END
  SwitchOff
    STATUS ordinary
    WHEN
      $mode = RETURN \wedge$
      $respx = respy = respz = 0$
    BEGIN
      $mode := OFF$
    END
END

which it can calculate a trajectory towards its goal. The tactic taken is to navigate towards the centroid of the positions of the controller and responders. The two drones need to ensure that they avoid damage in a hazardous area, so if they need to fly close to one, they need to take heed of its geometry, including height. Also, to avoid simultaneous destruction of both the drones though mishap, they need to ensure that they stay far enough apart from each other. This is all done by the *calcCentAvoidTraj*$(\ldots)$ function that they use, except that, in this paper, we abstract away from the detailed calculations that would be needed to achieve this. Evidently, the drones need to share their positions to do all this, so they share a single interface which declares both, which they can use to exchange positions by communicating with each other. For reasons of brevity, in this paper we do not cover this aspect in detail either.

All of this missing detail could be handled via a suitable Hybrid Event-B refinement. For simplicity again, a trajectory calculated by the drone consists of a sequence of line segments between points in three dimensions, which the drones follow, in order. The traversal of each segment is accomplished within the *Navigate* event. This simply increments each of the spatial coordinates, using the drone velocity (which is statically defined in its context) in proportion to the coordinate difference to be traversed. Once a segment is completed, the first segment is deleted from the trajectory in the *Waypoint* event, and the drone follows the next segment, until there is no trajectory left, at which point the drone stays where it is, maintaining distance from its partner.

The last big machine of the development is the generic responder machine **Responder_Mch** which appears on the previous page. This is similar to the drone machine in most respects, notably the technique used for synchronisation with the controller, but with a few essential differences. For example, the responders stay on the ground, so do not need a z coordinate. They are also assumed to not need to communicate with each other regarding position, so in our model, each responder can have its own interface. They do not need to fly above hazards, so their trajectory calculations will be different, expressed in the function *calcTraj*$(\ldots)$. And once arrived, they do some work until they, along with the drones, are recalled by the controller, which is achieved using the customary mechanisms.

6 Verification of Hybrid Event-B Models

The main purpose of the Hybrid Event-B approach is the verification of models expressed within the formalism, in order to derive increased confidence about their dependability. As for its predecessors in the classical B-Method and Event-B, the emphasis is on safety properties, expressed within the invariants declared in the model's syntactic constructs. In the case of Hybrid Event-B, these can be found inside machines and interfaces. At the time of writing, there is no tool support for Hybrid Event-B, so a comprehensive mechanically supported verification exercise cannot be reported on. Nevertheless, the POs that define correctness in Hybrid Event-B have been investigated in detail in the papers cited earlier, so an outline of what would be involved can be given.

Two things deserve to be highlighted. The first is that, as indicated earlier, the design of multi-machine Hybrid Event-B is such that the syntactic scope of all the POs is precisely defined by the rules of construction that models have to conform to. The second is that, the present model has deliberately been designed to be so simple, that the overwhelming majority of POs become rather trivial.

One set of POs concerns feasibility: i.e., is there an after-state for every enabled mode event? The simple assignments to constants in their bodies says yes; likewise for pliant events, which have either trivial COMPLY *INVARIANTS* bodies, or merely specify the following of a linear trajectory. Another major set of POs concerns the preservation of the invariants. Since, by design, these are almost exclusively trivial typing declarations, the answer will again be yes. Such POs can be dealt with by inspection.

Slightly more complicated are POs that deal with correct handover: mode/pliant and pliant/mode. This involves calculating disjunctions of guards, to ensure that when an event completes, an event of the right kind can succeede it. Again, extreme simplicity makes this a task that can be done by hand. So we can be confident our model is as it should be, even if mechanical corroboration would be even better.

The preceding remarks prepare the ground for discussing the last construct of the project, the global invariants **IncidentResponse_GI**. The global invariants offer the possibility of explicitly stating invariants of the system that are properties of the system as a whole, and not merely of some part of it. Clearly such a capability is important when safety properties depend on the safe cooperation of all parts of the system. Global invariants are intended to be derivable from the remaining invariants of a correct model, but this is not a hard constraint. The intention is that global invariants would be checked in the latter stages of a development, once the correctness of the constituent parts of the project is established.

GLOBINVS **IncidentResponse_GI**
SEES **IncidentResponse_CTX**
CONNECTS **IncidentResponse_IF**
INVARIANTS
$t \notin \bigcup(ii \bullet ii \in \mathrm{dom}(INITSCHED) \mid [INITSCHED(ii) \ldots INITSCHED(ii) + \delta]) \Rightarrow$
$(\ (hazards = ctrhazards) \wedge (hazards = drhazards) \wedge$
$(hazards = resp1hazards)\ \wedge (hazards = resp2hazards) \wedge (hazards = resp3hazards)\ )$
END

The **IncidentResponse_GI** construct above contains the following rather simple property of this kind. It states that outside the periods when the individual machines' perceptions of the hazard configurations are being updated (which are the time intervals of length δ following those integral multiples of Δ defined in the *INITSCHED* constant of the IncidentResponse context), they all agree on the hazards.

The conclusion of the invariant is a conjunction of statements each of which is a property of a part of the system. So it cannot be proved before all the pieces have been successfully constructed. A technical detail is that, for simplicity, the individual conjuncts (appropriately guarded) were not included as invariants of the relevant machines above. This itself does not make the global invariant unprovable of course. Global invariants, in particular, underline the connection between the Hybrid Event-B approach to system correctness and the reference to safety properties in this paper's title.

7 Conclusions

In the early part of the paper, we surveyed Hybrid Event-B and how it evolved from Event-B. The inclusion of real time and smooth state change entails considerable additional technical complexity in the semantics, and we discussed the details of this to the extent that space permitted. Multi-machine working brings in a whole raft of additional technical details to worry about, principally concerned with synchronisations and with the interplay of structure and verification needs. We surveyed this as far as possible. The multi-machine Hybrid Event-B approach is particularly suited to the formalisation and verification of autonomous systems, as these systems are seldom isolated, must fend for themselves for long periods of time while nevertheless communicating, but only intermittently, and are often cyber-physical. So modelling them naturally partitions into a constellation of cooperating but largely self-contained Hybrid Event-B machines, supported by suitable syntactic constructs.

The point of all this was to lay the groundwork for a small multi-machine example, to support the case just made. A simple system of intermittently communicating drones, responders and a controller was modelled. It is reasonable to say that this illustrated convincingly the capabilities of multi-machine Hybrid Event-B to fluently capture the kinds of behaviour pattern needed in such systems.

References

[1] J.-R. Abrial (1996): *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, doi:10.1017/CBO9780511624162.

[2] J.-R. Abrial (2005): *Event-B: Structure and Laws*. In: *Rodin Project Deliverable D7: Event-B Language*. http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf.

[3] J.-R. Abrial (2010): *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, doi:10.1017/CBO9781139195881.

[4] R. Banach (2024): *Core Hybrid Event-B III: Fundamentals of a Reasoning Framework*. *Sci. Comp. Prog.* 231, p. 103002, doi:10.1016/j.scico.2023.103002. 54pp.

[5] R. Banach (2024): *Hybrid Event-B*. In Mery & Singh, editors: *Modelling Software-based Systems, 2 Vols*, ISTE Press. To appear.

[6] R. Banach, M. Butler, S. Qin, N. Verma & H. Zhu (2015): *Core Hybrid Event-B I: Single Hybrid Event-B Machines*. *Sci. Comp. Prog.* 105, pp. 92–123, doi:10.1016/j.scico.2015.02.003.

[7] R. Banach, M. Butler, S. Qin & H. Zhu (2017): *Core Hybrid Event-B II: Multiple Cooperating Hybrid Event-B Machines*. *Sci. Comp. Prog.* 139, pp. 1–35, doi:10.1016/j.scico.2016.12.003.

[8] M. Butler (2009): *Decomposition Strategies for Event-B*. In Leuschel, Wehrheim, editor: *Proc. IFM-09*, 5423, Springer, LNCS, pp. 20–38.

[9] L. Carloni, R. Passerone, A. Pinto & A. Sangiovanni-Vincentelli (2006): *Languages and Tools for Hybrid Systems Design*. *Foundations and Trends in Electronic Design Automation* 1, pp. 1–193, doi:10.1561/1000000001.

[10] P.-L. Garoche (2019): *Formal Verification of Control System Software*. Princeton University Press, doi:10.23943/princeton/9780691181301.001.0001.

[11] E. Geisberger & M. Broy (eds.) (2015): *Living in a Networked World. Integrated Research Agenda Cyber-Physical Systems (agendaCPS)*. http://www.acatech.de/ fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Publikationen/Projektberichte/acaetch_STUDIE_agendaCPS_eng_WEB.pdf.

[12] S. Hallerstede & J.-R. Abrial (2010): *Event-B Decomposition for Parallel Programs*. In Frappier, Glässer, Khurshid, Laleau, Reeves, editor: *Proc. ABZ-10*, 5977, Springer, LNCS, pp. 319–333.

[13] S. Hallerstede & T. Hoang (2012): *Refinement by Interface Instantiation*. In Derrick, Fitzgerald, Gnesi, Khurshid, Leuschel, Reeves, Riccobene, editor: *Proc. ABZ-12*, 7316, Springer, LNCS, pp. 223–237.

[14] R. Sanfelice (2021): *Hybrid Feedback Control*. Princeton University Press, doi:10.2307/j.ctv131btfx.

[15] R. Silva & M. Butler (2009): *Supporting Reuse of Event-B Developments through Generic Instantiation*. In Breitman, Cavalcanti, editor: *Proc. ICFEM-09*, 5885, Springer, LNCS, pp. 466–484.

[16] R. Silva, C. Pascal, T. Hoang & M. Butler (2011): *Decomposition Tool for Event-B*. *Soft. Prac. Exp.* 41, pp. 199–208, doi:10.1002/spe.1002.

[17] P. Tabuada (2009): *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, doi:10.1007/978-1-4419-0224-5.

A Appendix: Contexts, Interfaces and Instantiated Machine Outlines

CONTEXT **IncidentResponse_CTX**
CONSTANTS
SQ , CYL
$INITSCHED$
δ , Δ
$DURATION$
$RESP1dest$, $RESP2dest$, $RESP3dest$
SETS
$HAZTYPE$
AXIOMS
$partition(HAZTYPE, \{SQ\}, \{CYL\})$
$\delta : \mathbb{R}$, $\delta = 0.1$
$\Delta : \mathbb{R}$, $\Delta = 1$
... ...

... ...
$INITSCHED : \text{seq}(\mathbb{R})$
$INITSCHED = \langle 12, 30, 55 \rangle$
$DURATION : \mathbb{R}$
$DURATION = 79.7$
$RESP1dest : \mathbb{R} \times \mathbb{R}$
$RESP1dest = (12.3 \mapsto 15.0)$
$RESP2dest : \mathbb{R} \times \mathbb{R}$
$RESP2dest = (-11.2 \mapsto 14.0)$
$RESP3dest : \mathbb{R} \times \mathbb{R}$
$RESP3dest = (2.1 \mapsto 29.0)$
THEOREMS
$DURATION > \text{last}(INITSCHED)$
END

INTERFACE **IncidentResponse_IF**
SEES $IncidentResponse\_CTX$
TIME t
VARIABLES
$hazards$
... ...

... ...
INVARIANTS
$hazards : \mathbb{P}(HAZTYPE \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R})$
:— sq/cyl, (x, y) coords, size from (x, y), height
INITIALISATION
$t := 0$
$hazards := \varnothing$
END

CONTEXT **Controller_CTX**
CONSTANTS
OFF , $DISPATCH$, $RECALL$, $UPDATEHAZ$
SETS
$CTRLSTATE$
... ...

... ...
AXIOMS
$partition(CTRLSTATE , \{OFF\} , \{DISPATCH\} , \{UPDATEHAZ\} , \{RECALL\})$
END

CONTEXT **Drone_CTX**
CONSTANTS
OFF , $SEEK$, $RETURN$
Vdr
SETS
$DRONESTATE$
... ...

... ...
AXIOMS
$partition(DRONESTATE , \{OFF\} , \{SEEK\} , \{RETURN\})$
$Vdr : \mathbb{R}$
END

INTERFACE **ControllerDrones_IF**
TIME t
VARIABLES
$drhazards$
PLIANT
$dr1x$, $dr1y$, $dr1z$
$dr2x$, $dr2y$, $dr2z$
... ...

... ...
INVARIANTS
$drhazards : \mathbb{P}(HAZTYPE \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R})$
$dr1x$, $dr1y$, $dr1z : \mathbb{R}$, $\mathbb{R}$, $\mathbb{R}$
$dr2x$, $dr2y$, $dr2z : \mathbb{R}$, $\mathbb{R}$, $\mathbb{R}$
INITIALISATION
$t := 0$
$drhazards := \varnothing$
$dr1x$, $dr1y$, $dr1z := 0 , 0 , 0$
$dr2x$, $dr2y$, $dr2z := 0 , 0 , 0$
END

CONTEXT **Responder_CTX**
CONSTANTS
  OFF , $SEEK$, $ARRIVED$, $RETURN$
SETS
  $RESPSTATE$
AXIOMS
  $partition(RESPSTATE , \{OFF\} ,$
    $\{SEEK\} , \{ARRIVED\} , \{RETURN\})$
END

INTERFACE **ControllerResponder_IF**
TIME t
VARIABLES
  $resphazards$
INVARIANTS
  $resphazards : \mathbb{P}(HAZTYPE \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R})$
INITIALISATION
  $t := 0$
  $resphazards := \varnothing$
END

MACHINE **Drone1_Mch**
  $\bullet\bullet\bullet$
END
MACHINE **Drone2_Mch**
  $\bullet\bullet\bullet$
END
$\ldots$ $\ldots$

$\ldots$ $\ldots$
MACHINE **Responder1_Mch**
  $\bullet\bullet\bullet$
END
MACHINE **Responder2_Mch**
  $\bullet\bullet\bullet$
END
MACHINE **Responder3_Mch**
  $\bullet\bullet\bullet$
END

Formal Simulation and Visualisation of Hybrid Programs

An Extension of a Proof-of-Concept Tool

Pedro Mendes
University of Minho, Portugal
pg50685@alunos.uminho.pt

Ricardo Correia
University of Minho, Portugal
pg47607@alunos.uminho.pt

Renato Neves
INESC-TEC & University of Minho, Portugal
nevrenato@di.uminho.pt

José Proença
CISTER, Faculty of Sciences of the University of Porto, Portugal
jose.proenca@fc.up.pt

The design and analysis of systems that combine computational behaviour with physical processes' continuous dynamics – such as movement, velocity, and voltage – is a famous, challenging task. Several theoretical results from programming theory emerged in the last decades to tackle the issue; some of which are the basis of a *proof-of-concept* tool, called Lince, that aids in the analysis of such systems, by presenting simulations of their respective behaviours.

However being a proof-of-concept, the tool is quite limited with respect to usability, and when attempting to apply it to a set of common, concrete problems, involving autonomous driving and others, it either simply cannot simulate them or fails to provide a satisfactory user-experience.

The current work complements the aforementioned theoretical approaches with a more practical perspective, by improving Lince along several dimensions: to name a few, richer syntactic constructs, more operations, more informative plotting systems and errors messages, and a better performance overall. We illustrate our improvements via a variety of examples that involve both autonomous driving and electrical systems.

1 Introduction

Motivation and context. This paper concerns the design and analysis of hybrid systems (*i.e.* those that combine discrete with continuous behaviour) from a programming-oriented perspective. Such a view emerged recently in a series of works [24, 21, 11, 15], and revolves around the idea of importing principles and techniques from programming theory to better handle the behaviour of hybrid systems. In this context programs combine standard program constructs, such as conditionals and while-loops, with certain kinds of differential statement meant to express the dynamics of physical processes, such as time, energy, and motion. Consider the following example of such a program:

$$p' = v, v' = 2 \texttt{ for } 1 \;;\; p' = v, v' = -2 \texttt{ for } 1 \tag{1}$$

In a nutshell, it is a sequential composition $(;)$ of two programs where each expresses how the position (p) and velocity (v) of a vehicle evolve over time. The program on the left ($p' = v, v' = 2$ `for` 1) is a differential statement that reads "the vehicle accelerates at the rate of $2\,{}^{m}/_{s^2}$ for 1 second". The other program corresponds to a deceleration. Both position and velocity over time are presented in Fig. 1, where we see that the vehicle travels 2 meters and then stops.

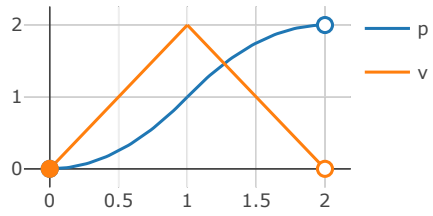


Figure 1: Simulation of (1).

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 20–37, doi:10.4204/EPTCS.411.2

© R. Correia, P. Mendes, R. Neves, J. Proença
This work is licensed under the
Creative Commons Attribution License.

Actually there has been a rapid proliferation of such systems, not only in the domain of autonomous driving but also in the medical industry and industrial infrastructures, among others [24, 12, 19, 21]. This spurred extensive research on languages, semantics, and tools for their design and analysis. An example is our work [10, 11] on the semantics of hybrid programs – *i.e.* those that combine program constructs with differential statements, such as in (1) – from which arises a mathematical basis for reasoning about their behaviour, both operationally and denotationally. A proof-of-concept tool, called Lince, also emerged from this: its engine is a previously developed operational semantics [11] that yields trajectories of hybrid programs, just as we saw in Fig. 1. However because our focus was rather theoretical, the tool was not developed with usability in mind, and thus lacks basic features for tackling a broad range of important scenarios. Let us illustrate this problem with a very simple example.

Problem scenario. Suppose that we wish to move a stationary object a distance of *dist* meters – a basic task in autonomous driving. For simplicity assume that we have access only to the acceleration rates $a\,{}^{m}/_{s^2}$ and $-a\,{}^{m}/_{s^2}$, where $a > 0$. Our mission can be accomplished by taking the following variation of Eq. (1),

$$p' = v, v' = a \texttt{ for } t \texttt{ ; } p' = v, v' = -a \texttt{ for } t \tag{2}$$

for a suitable duration t. Then in order to calculate t (*i.e.* the prescribed duration of each differential statement) we simply observe that,

$$dist = \int_0^t v_a(x)\,dx \;+\; \int_0^t v_{-a}(x)\,dx$$

where $v_a(x) = a \cdot x$ and $v_{-a}(x) = v_a(t) + -a \cdot x$ are the velocity functions with respect to the time intervals $[0,t]$ and $[t, 2 \cdot t]$ associated with the program's execution. We now observe, by recalling Fig. 1, that the value *dist* corresponds to the area of a triangle with basis $2 \cdot t$ and height $v_a(t)$. This geometric shape yields the equations,

$$\begin{cases} dist & = {}^{1}/_{2} \cdot (2 \cdot t) \cdot v_a(t) \;\; \textbf{(area)} \\ v_a(t) & = a \cdot t \;\; \textbf{(height)} \end{cases} \qquad \Longrightarrow \qquad t = \sqrt{\frac{dist}{a}}$$

Finally observe that if $dist = 3$ and $a = 1$ then $t = \sqrt{3}$. Unfortunately the previous version of Lince does not support square root operations which renders our mission impossible to accomplish.

Contributions and outline. As already alluded to, this paper complements our previous theoretical work on the semantics of hybrid programming [10, 11]. Specifically it improves our proof-of-concept tool Lince so that it can handle a myriad of important scenarios, whilst maintaining both its simplicity and theoretical underpinnings. The improvements were made along different dimensions, and we highlight the most relevant ones next[1].

Extension of basic operations. As illustrated before, the previous version of Lince lacked essential arithmetic operations for handling most basic tasks. Thus as the first main contribution we added standard arithmetic operations, including divisions, trigonometric functions, and square root extractions. Notably the fact that many of these operations are partial required us to extend the operational semantics developed in [11] (the main engine of Lince) with the possibility of failure. The extended semantics is detailed in Section 2 and it is of course the basis of the new engine behind improved Lince.

[1] The improved version can be checked online at http://arcatools.org/lince.

Extension of numerical methods. Again because our focus in previous work was rather theoretical the previous version of Lince was unable to simulate standard scenarios in hybrid programming. A main reason for this was our method of obtaining solutions of systems of ordinary differential equations (ODEs), which although *exact* lacked in scalability. Precisely for this reason we now integrate a complementary, numerical solver in Lince with the obvious compromise that the solutions obtained for such systems are no longer exact.

The benefits of the extended language (and respective semantics), the numerical solver, and a number of quality-of-life features, are summarised in Section 3 and illustrated with a standard, running example concerning the famous concept of harmonic oscillation.

Extension of visualisation mechanisms. Lince is constituted by two core components: the simulator which, by recurring to the aforementioned operational semantics, parses a received program and presents its output with respect to a *single* time instant. And the visualiser which presents (a sample of) the trajectory over time respective to the program at hand, by querying the operational semantics for a certain sequence of time instants. After trying to properly visualise the behaviour of several types of hybrid program with Lince we identified two major limitations with respect to this architecture. First many real-world problems involve multiple spatial dimensions and thus the described view of trajectories over time is often not the best representation of a hybrid program's behaviour. Second the user is often interested in observing the overall program behaviour for varying initial conditions, concerning for example position and velocity. We therefore present in Section 4 an improved visualiser for Lince that precisely addresses these two issues. We illustrate it via another classical scenario in autonomous driving, *viz.* manoeuvring around an obstacle.

In Section 5 we illustrate that, whilst keeping its simplicity, Lince can now handle complex central problems in the theory of hybrid systems; we focus specifically on the task of one player pursuing another, *e.g.* a vehicle, a drone, or simply a projectile. Such pursuit games were discussed for example in [20, 2, 6, 18], from an (hybrid-)automata, state-chart, and duration calculus perspective. Here we present a programming-oriented approach. Finally in Section 6 we discuss future work and conclude.

Related work. Several tools for the design and analysis of hybrid systems were already proposed, *e.g.* in the areas of deductive verification [24], model checking [3, 8, 4], simulation [16, 9, 15, 11], and program semantics [24, 15, 11]. But only a few are committed to a programming-oriented approach, rooted on formal semantics, and with effective simulation capabilities. The only ones we are aware of are [15] and our own tool Lince [11]. Interestingly both cases adopt complementary approaches as well: the former harbours a very powerful concurrent language, particularly well-suited for large-scale distributed systems. The latter, harbouring a sequential while-language, aims at being minimalistic whilst still capturing a broad range of interesting problems on which to study different aspects of (pure) hybrid computation at a suitable abstraction level.

Aside from the obvious pedagogical benefit, our minimalistic approach also allows to capitalise on different programming theories more easily. For example already in [11] we connected our tool to a compositional, denotational semantics – particularly well-suited to study hybrid program equivalence and combinations with other paradigms. An analogous concurrent semantics for [15] would be notoriously more difficult to achieve (*cf.* [26, 28]). Similarly our language is amenable to algebraic reasoning in the style of (weak) Kleene algebras [17, 14] whilst the connection between the latter and concurrent object-oriented programming (as adopted in [15]) is less clear.

2 Lince's Foundations Extended with the Possibility of Failure

We now extend part of Lince's foundations with the possibility of failure. Specifically we present an extension of the language in [11] with partial operations, such as division and square root extraction, and introduce a corresponding operational semantics. As explained in the introduction, such is necessary for extending Lince to 'real-world problems' whilst preserving its merit of having a firm, mathematical basis.

Language. First we postulate a finite set $X = \{x_1, \ldots, x_n\}$ of variables and a stock of partial functions $f : \mathbb{R}^n \rightharpoonup \mathbb{R}$ that contains the usual arithmetic operations. Then we define expressions and boolean conditions via the following BNF grammars,

$$e ::= x \mid f(e, \ldots, e) \qquad\qquad b ::= e \leq e \mid b \wedge b \mid b \vee b \mid \neg b \mid \texttt{tt} \mid \texttt{ff}$$

We omit the explanation of these grammars as they are widely used (see *e.g.* [28, 26]). Next, we qualify as 'linear' those expressions e which aside from the use of variables involve only the operations $+$ and $r \cdot (-)$ for some $r \in \mathbb{R}$. For example the expression $2 \cdot x$ is linear but the expression $x \cdot x$ is not. The concept of linearity is key in the grammar of hybrid programs which we present next.

Programs are built according to the following BNF grammars,

$$\begin{aligned} \mathtt{a} &::= x'_1 = \ell_1, \ldots, x'_n = \ell_n \ \texttt{for}\ e \mid x := e \\ \mathtt{p} &::= \mathtt{a} \mid \mathtt{p};\mathtt{p} \mid \texttt{if}\ b\ \texttt{then}\ \mathtt{p}\ \texttt{else}\ \mathtt{p} \mid \texttt{while}\ b\ \texttt{do}\ \{\ \mathtt{p}\ \} \end{aligned}$$

where the terms ℓ_i $(1 \leq i \leq n)$ are linear expressions. We qualify as 'atomic' those hybrid programs that are built according to the first grammar. They can be either classical assignments or *differential* statements as described in the introduction. The linearity constraint is here imposed merely to ensure that the latter kind of statement will always have unique solutions, which renders our semantics more lightweight whilst still being able to treat a broad range of problems (see more details in [11]).

The language of hybrid programs p itself is simply the usual while-language [28, 26] extended with the aforementioned differential statements. It is easy to check that our grammar indeed extends that in the previous version of Lince [11] where *all* expressions involved in the assignments and the durations of differential statements had to be linear. This has of course significant implications in the operational semantics introduced in [11].

Operational semantics. We need a series of preliminaries. First for simplicity we abbreviate differential statements $x'_1 = \ell_1, \ldots, x'_n = \ell_n$ `for` e simply to $\vec{x}' = \vec{\ell}$ `for` e, where $\vec{x}'$ and $\vec{\ell}$ abbreviate the corresponding vectors of variables $x'_1 \ldots x'_n$ and linear expressions $\ell_1 \ldots \ell_n$. We call functions of the type $\sigma : X \to \mathbb{R}$ *environments*; they map variables to the respective valuations. We use the notation $\sigma[\vec{x} \mapsto \vec{v}]$ to denote the environment that maps each x_i in $\vec{x}$ to v_i in $\vec{v}$ and the remaining variables as in σ. Finally we denote by $\phi_\sigma^{\vec{x}'=\vec{\ell}} : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$ the (unique) solution of a system of differential equations $\vec{x}' = \vec{\ell}$ with σ as the initial condition (recall our previous constraint about linearity which ensures that such solutions indeed exist). When clear from context, we omit both the superscript and subscript in $\phi_\sigma^{\vec{x}'=\vec{\ell}}$. Next, for an expression e the notation $[\![e]\!](\sigma)$ denotes the standard (partial) interpretation of expressions [28, 26] according to σ, and analogously for $[\![b]\!](\sigma)$ where b is a boolean expression. For example $[\![x+1]\!](\sigma) = \sigma(x) + 1$ and $[\![1/x]\!](\sigma)$ is undefined if $\sigma(x) = 0$.

We now present an operational semantics for the language. Following traditions in programming theory [22, 28, 26], we present it from two different, complementary perspectives, which gives a much more complete understanding of the language's features. Specifically we present the semantics in two

different styles: one formalises the idea of a machine "running" a hybrid program and describes its *step-by-step evolution*. The other abstracts away from all *intermediate steps* of this machine and is therefore generally more suitable to reason about "input-output behaviours" (although we do not explore such a feature here). Whilst the former style is the basis of Lince's new version, the latter style is conceptually more intuitive and therefore we present it first. The current section concludes with a proof that both semantics are in fact equivalent. The curious reader can consult for example [28, 26] for a thorough account on the key differences between the small-step and big-step styles in general program semantics.

Our 'big-step' operational semantics is given by an 'input-output' relation $\Downarrow$ which relates programs $\mathtt{p}$, environments σ, and time instants t to outputs v. The expression $\mathtt{p}, \sigma, t \Downarrow v$ can be read as "at time instant t the program $\mathtt{p}$ starting from state σ outputs v". The relation $\Downarrow$ is built inductively according to the rules in Fig. 2. Specifically the first three rules describe how differential statements are evaluated: first one computes the duration $[\![e]\!](\sigma)$ of the differential statement at hand and an error is raised if $[\![e]\!](\sigma)$ is undefined; otherwise the output v is the respective modified state (as dictated by the differential statement) paired with one of the flags *stop* or *skip*. Intuitively the flag *stop* indicates that we 'reached' the time instant at which the program needs to be evaluated and therefore the evaluation can stop moving forward in time, which fact is reflected in rule (**seq-stop**). The flag *skip* is simply the negation of *stop*. The remaining rules follow analogous principles and therefore we refrain from detailing them – instead we will briefly show how the semantics works via instructive, concrete examples.

Example 2.1. Let us consider the following very simple program,

$$x' = -1 \ \texttt{for}\ 1 \ \texttt{;}\ x := {}^{1}\!/_{x}$$

which continuously decreases the value of variable x during 1 second and then applies the (discrete) operation $x := {}^{1}\!/_{x}$. Suppose as well that our initial state is the environment σ defined by $x \mapsto 1$. Then by an application of rule **(diff-stop)** one deduces that this program outputs the environment $x \mapsto 1 - t$ at every time instant $t < 1$. On the other hand, by an application of rules **(diff-skip)**, **(asg-err)**, and **(seq-skip)** one easily deduces that the evaluation of the program fails at every time instant $t \geq 1$, due to a division by 0.

Notably the fact that failure occurs only at the time instants $t \geq 1$ is a fundamental difference with respect to the famous hybrid programming language detailed in [24]. In the *op. cit.* the language was designed in the spirit of Kleene algebra, which in particular forces the previous program to be *indistinguishable* from *e.g.* the program $x := {}^{x}\!/_{0}$. Whilst such a feature could be desirable in some verification scenarios it is clearly unnatural in a simulation-based environment such as ours.

Let us continue unravelling prominent features of our semantics with another example. Consider the following hybrid program,

$$\texttt{while}\ x \neq 0\ \texttt{do}\ \{\ x' = -1\ \texttt{for}\ {}^{x}\!/_{2}\ \}\ \texttt{;}\ x := {}^{1}\!/_{x}$$

paired with the environment $x \mapsto 1$. This program is an instance of a so-called Zeno loop: *viz.* the loop involved unfolds *infinitely* many times with the duration of each iteration becoming shorter and shorter (see details *e.g.* in [11]). In this particular case it is straightforward to check that the duration of the i-th iteration is given by ${}^{1}\!/_{2^i}$, and thus that the total duration $\sum_{i=1}^{\infty} {}^{1}\!/_{2^i}$ of the loop will be 1. Now, by applying the operational rules in Fig. 2 one can successfully evaluate the program at every time instant $t < 1$ (intuitively because every such t is reached in a *finite* number of iterations). The same is *false* for time instant $t = 1$ since such requires a complete unfolding of the loop which is of course computationally unfeasible. Thus operationally the potential point of failure $x := {}^{1}\!/_{x}$ in the program above never occurs, as the Zeno loop makes it impossible to actually reach this command in the evaluation. These infinite

$$\textbf{(diff-skip)}\quad \frac{[\![e]\!](\sigma) = t}{\vec{x}' = \vec{\ell}\ \texttt{for}\ e,\sigma,t \Downarrow skip,\sigma[\vec{x} \mapsto \phi(t)]}$$

$$\textbf{(diff-stop)}\quad \frac{[\![e]\!](\sigma) > t}{\vec{x}' = \vec{\ell}\ \texttt{for}\ e,\sigma,t \Downarrow stop,\sigma[\vec{x} \mapsto \phi(t)]} \qquad \textbf{(diff-err)}\quad \frac{[\![e]\!](\sigma)\ \text{undefined}}{\vec{x}' = \vec{\ell}\ \texttt{for}\ e,\sigma,t \Downarrow err}$$

$$\textbf{(asg-skip)}\quad \frac{[\![e]\!](\sigma)\ \text{defined}}{x := e,\sigma,0 \Downarrow skip,\sigma[x \mapsto [\![e]\!](\sigma)]} \qquad \textbf{(asg-err)}\quad \frac{[\![e]\!](\sigma)\ \text{undefined}}{x := e,\sigma,t \Downarrow err}$$

$$\textbf{(seq-skip)}\quad \frac{\mathtt{p},\sigma,t \Downarrow skip,\tau \qquad \mathtt{q},\tau,u \Downarrow v}{\mathtt{p;q},\sigma,t+u \Downarrow v}$$

$$\textbf{(seq-stop)}\quad \frac{\mathtt{p},\sigma,t \Downarrow stop,\tau}{\mathtt{p;q},\sigma,t \Downarrow stop,\tau} \qquad \textbf{(seq-err)}\quad \frac{\mathtt{p},\sigma,t \Downarrow err}{\mathtt{p;q},\sigma,t \Downarrow err}$$

$$\textbf{(if-true)}\quad \frac{[\![b]\!](\sigma) = \mathtt{tt} \qquad \mathtt{p},\sigma,t \Downarrow v}{\texttt{if}\ b\ \texttt{then}\ \mathtt{p}\ \texttt{else}\ \mathtt{q},\sigma,t \Downarrow v}$$

$$\textbf{(if-false)}\quad \frac{[\![b]\!](\sigma) = \mathtt{ff} \qquad \mathtt{q},\sigma,t \Downarrow v}{\texttt{if}\ b\ \texttt{then}\ \mathtt{p}\ \texttt{else}\ \mathtt{q},\sigma,t \Downarrow v} \qquad \textbf{(if-err)}\quad \frac{[\![b]\!](\sigma)\ \text{undefined}}{\texttt{if}\ b\ \texttt{then}\ \mathtt{p}\ \texttt{else}\ \mathtt{q},\sigma,t \Downarrow err}$$

$$\textbf{(wh-true)}\quad \frac{[\![b]\!](\sigma) = \mathtt{tt} \qquad \mathtt{p;}\texttt{while}\ b\ \texttt{do}\ \{\ p\ \},\sigma,t \Downarrow v}{\texttt{while}\ b\ \texttt{do}\ \{\ \mathtt{p}\ \},\sigma,t \Downarrow v}$$

$$\textbf{(wh-false)}\quad \frac{[\![b]\!](\sigma) = \mathtt{ff}}{\texttt{while}\ b\ \texttt{do}\ \{\ \mathtt{p}\ \},\sigma,0 \Downarrow skip,\sigma} \qquad \textbf{(wh-err)}\quad \frac{[\![b]\!](\sigma)\ \text{undefined}}{\texttt{while}\ b\ \texttt{do}\ \{\ \mathtt{p}\ \},\sigma,t \Downarrow err}$$

Figure 2: Extension of the big-step operational semantics in [11] with the possibility of failure.

behaviours are bounded in Lince by manually setting limits on the total time and on the number of unfoldings of while-loops, adjustable for each program.

Next, the semantics in the aforementioned 'small-step' style is given in the form of a relation $\rightarrow$ that is defined inductively according to the rules in Fig. 3. These rules follow an analogous reasoning to the ones in Fig. 2 so we refrain from repeating explanations.

As detailed in Corollary 1 our small-step semantics is deterministic. This is of course a key property in what concerns its implementation and subsequent use in Lince for simulating hybrid programs. The corollary is based on the following theorem.

Theorem 2.1. For every program $\mathtt{p}$, environment σ, and time instant t there is *at most one* applicable reduction rule.

Let $\rightarrow^\star$ be the transitive closure of the small-step relation $\rightarrow$ that was previously presented. Intuitively $\rightarrow^\star$ represents an evaluation of one or more steps according to the small-step semantics. If $\mathtt{p},\sigma,t \rightarrow^\star v$ we call v 'non-terminal' whenever it is of the form $\mathtt{p}',\sigma',t'$ for some hybrid program $\mathtt{p}'$, environment σ', and time instant t'; we call v 'terminal' otherwise.

$$\textbf{(asg}^{\rightarrow}\textbf{)} \quad x := e, \sigma, t \ \rightarrow\ skip, \sigma[x \mapsto [\![e]\!](\sigma)], t \qquad (\textit{if } [\![e]\!](\sigma) \text{ defined})$$

$$\textbf{(asg-err}^{\rightarrow}\textbf{)} \quad x := e, \sigma, t \ \rightarrow\ err \qquad (\textit{if } [\![e]\!](\sigma) \text{ undefined})$$

$$\textbf{(diff-stop}^{\rightarrow}\textbf{)} \quad \vec{x}' = \vec{\ell} \ \texttt{for}\ e, \sigma, t \ \rightarrow\ stop, \sigma[\vec{x} \mapsto \phi(t)], 0 \qquad (\textit{if } [\![e]\!](\sigma) > t)$$

$$\textbf{(diff-skip}^{\rightarrow}\textbf{)} \quad \vec{x}' = \vec{\ell} \ \texttt{for}\ e, \sigma, t \ \rightarrow\ skip, \sigma[\vec{x} \mapsto \sigma(t)], t - [\![e]\!](\sigma) \qquad (\textit{if } [\![e]\!](\sigma) \leq t)$$

$$\textbf{(diff-err}^{\rightarrow}\textbf{)} \quad \vec{x}' = \vec{\ell} \ \texttt{for}\ e, \sigma, t \ \rightarrow\ err \qquad (\textit{if } [\![e]\!](\sigma) \text{ undefined})$$

$$\textbf{(if-true}^{\rightarrow}\textbf{)} \quad \texttt{if}\ b\ \texttt{then p else q}, \sigma, t \ \rightarrow\ \texttt{p}, \sigma, t \qquad (\textit{if } [\![b]\!](\sigma) = \texttt{tt})$$

$$\textbf{(if-false}^{\rightarrow}\textbf{)} \quad \texttt{if}\ b\ \texttt{then p else q}, \sigma, t \ \rightarrow\ \texttt{q}, \sigma, t \qquad (\textit{if } [\![b]\!](\sigma) = \texttt{ff})$$

$$\textbf{(if-err}^{\rightarrow}\textbf{)} \quad \texttt{if}\ b\ \texttt{then p else q}, \sigma, t \ \rightarrow\ err \qquad (\textit{if } [\![b]\!](\sigma) \text{ undefined})$$

$$\textbf{(wh-true}^{\rightarrow}\textbf{)} \quad \texttt{while}\ b\ \texttt{do}\ \{\ \texttt{p}\ \}, \sigma, t \ \rightarrow\ \texttt{p;while}\ b\ \texttt{do}\ \{\ \texttt{p}\ \}, \sigma, t \qquad (\textit{if } [\![b]\!](\sigma) = \texttt{tt})$$

$$\textbf{(wh-false}^{\rightarrow}\textbf{)} \quad \texttt{while}\ b\ \texttt{do}\ \{\ \texttt{p}\ \}, \sigma, t \ \rightarrow\ skip, \sigma, t \qquad (\textit{if } [\![b]\!](\sigma) = \texttt{ff})$$

$$\textbf{(wh-err}^{\rightarrow}\textbf{)} \quad \texttt{while}\ b\ \texttt{do}\ \{\ \texttt{p}\ \}, \sigma, t \ \rightarrow\ err \qquad (\textit{if } [\![b]\!](\sigma) \text{ undefined})$$

$$\textbf{(seq-stop}^{\rightarrow}\textbf{)} \ \frac{\texttt{p}, \sigma, t \ \rightarrow\ stop, \sigma', t'}{\texttt{p;q}, \sigma, t \ \rightarrow\ stop, \sigma', t'} \qquad \textbf{(seq-skip}^{\rightarrow}\textbf{)} \ \frac{\texttt{p}, \sigma, t \ \rightarrow\ skip, \sigma', t'}{\texttt{p;q}, \sigma, t \ \rightarrow\ \texttt{q}, \sigma', t'}$$

$$\textbf{(seq-err}^{\rightarrow}\textbf{)} \ \frac{\texttt{p}, \sigma, t \ \rightarrow\ err}{\texttt{p;q}, \sigma, t \ \rightarrow\ err} \qquad \textbf{(seq}^{\rightarrow}\textbf{)} \ \frac{\texttt{p}, \sigma, t \ \rightarrow\ \texttt{p}', \sigma', t'}{\texttt{p;q}, \sigma, t \ \rightarrow\ \texttt{p}'\texttt{;q}, \sigma', t'} \quad (\textit{if } \texttt{p}' \neq stop \textit{ and } \texttt{p}' \neq skip)$$

Figure 3: Extension of the small-step operational semantics in [11] with the possibility of failure.

Corollary 1 (Determinism)**.** Consider a program p, an environment σ, and a time instant t. If $\texttt{p}, \sigma, t \ \rightarrow^{\star}\ v$ and $\texttt{p}, \sigma, t \ \rightarrow^{\star}\ u$ with both v and u terminal then we have $v = u$.

Proof. Follows by induction on the number of reduction steps and Theorem 2.1. □

Next we will show that the small-step semantics and its big-step counterpart are indeed equivalent. We will use the two following results for this effect.

Lemma 2.1. Given a program p, an environment σ and a time instant t

1. if $\texttt{p}, \sigma, t \ \rightarrow\ \texttt{p}', \sigma', t'$ and $\texttt{p}', \sigma', t' \ \Downarrow\ skip, \sigma''$ then $\texttt{p}, \sigma, t \ \Downarrow\ skip, \sigma''$;
2. if $\texttt{p}, \sigma, t \ \rightarrow\ \texttt{p}', \sigma', t'$ and $\texttt{p}', \sigma', t' \ \Downarrow\ stop, \sigma''$ then $\texttt{p}, \sigma, t \ \Downarrow\ stop, \sigma''$;
3. if $\texttt{p}, \sigma, t \ \rightarrow\ \texttt{p}', \sigma', t'$ and $\texttt{p}', \sigma', t' \ \Downarrow\ err$ then $\texttt{p}, \sigma, t \ \Downarrow\ err$;

Proof. Follows by induction over the rules concerning the small-step relation. □

Proposition 1. For all program p and q, environments σ and σ', and time instants t, t' and s, if $\mathtt{p},\sigma,t \rightarrow \mathtt{q},\sigma',t'$ then $\mathtt{p},\sigma,t+s \rightarrow \mathtt{q},\sigma',t'+s$; and if $\mathtt{p},\sigma,t \rightarrow \mathit{skip},\sigma',t'$ then $\mathtt{p},\sigma,t+s \rightarrow \mathit{skip},\sigma',t'+s$. If $\mathtt{p},\sigma,t \rightarrow \mathit{err}$ then $\mathtt{p},\sigma,t+s \rightarrow \mathit{err}$

Proof. Follows straightforwardly by induction over the rules concerning the small-step relation and the algebraic properties of addition. □

Theorem 2.2 (Equivalence)**.** The small-step semantics and the big-step semantics are related in the following manner. Given a program p, an environment σ and a time instant t

1. $\mathtt{p},\sigma,t \Downarrow \mathit{skip},\sigma'$ iff $\mathtt{p},\sigma,t \rightarrow^{\star} \mathit{skip},\sigma',0$;
2. $\mathtt{p},\sigma,t \Downarrow \mathit{stop},\sigma'$ iff $\mathtt{p},\sigma,t \rightarrow^{\star} \mathit{stop},\sigma',0$;
3. $\mathtt{p},\sigma,t \Downarrow \mathit{err}$ iff $\mathtt{p},\sigma,t \rightarrow^{\star} \mathit{err}$.

Proof. The right-to-left direction is obtained by induction over the length of the small-step reduction sequence using Lemma 2.1. The left-to-right direction follows by induction over the big-step derivations together with Proposition 1. □

3 An Improved Simulator for Hybrid Programs

This section summarises several improvements made to Lince's simulator of hybrid programs since its original publication [11]. These include (1) more expressive assignments and durations in differential statements (by virtue of the results in the preceding section); (2) a more user-friendly program syntax (by means of syntactic sugar); (3) more informative error messages; and (4) a numerical solver of systems of ordinary differential equations. In order to render our summary more lively we complement it with a running example involving an RLC circuit in series with an On-Off source. It is designed to stabilise voltage across the capacitor in the circuit at a specific value.

Running example: RLC circuits and harmonic oscillation. We present in Fig. 4 the simulation of an *RLC circuit in series* (RLCS). This simulation models an electric system composed of a resistor, a capacitor, an inductor, and a power source connected in series. The power source strategically switches on and off, as a way to stabilise voltage across the capacitor at a target value (say, $10V$). Such systems are known to yield interesting results that are practically relevant for energy storage voltage control systems, which help to mitigate voltage imbalances that could otherwise damage electronic equipment. More details about such circuits and associated differential equations are available for example in [29, 13]. We present in Fig. 4 two variations of an RLCS circuit: one in which the capacitor voltage is in an underdamped regime – with a resistance `rU` of 0.5Ω, a capacitance `c` of $0.047F$, and an inductance `l` of $0.047H$ – and one in which the capacitor voltage is in an overdamped regime – with a resistance `r0` of 4Ω and the same values as before for the capacitance and inductance. The general idea of our program is that the associated controller will read the voltage across the capacitor (variable `under` for the underdamped case, `over` for the overdamped one) every 0.01 seconds, and set the voltage at the source either to 0 (off) or $18V$ (on) depending on the value read.

Improvement's summary. The program just described is highly problematic for the original version of Lince. This is due to two fundamental reasons related to the ODEs involved: specifically (1) the equations used in the ODEs violate the linearity condition presented in Section 2 (they include variable multiplications); and (2) the original solver of ODEs, mentioned in the introduction, fails to produce

```
under:=0; dU:=0; vU:=0; rU:=0.5;
over:=0;  dO:=0; vO:=0; rO:=4;
c:=0.047; l:=0.047;

while true do {
  if (under<10) then vU:=18;
                else vU:=0;
  if (over<10)  then vO:=18;
                else vO:=0;
  under'=dU, over'=dO,
  dU'=-(dU*rU/l)
      -under/(l*c)+vU/(l*c),
  dO'=-(dO*rO/l)
      -over/(l*c)+vO/(l*c)
  for 0.01;
}
```

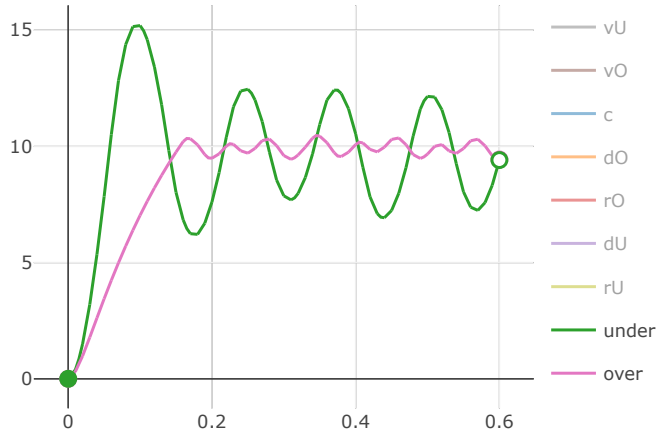


Figure 4: Hybrid program (left) and its plot (right) of two variations of an RLC circuit that tries to maintain the voltage in the capacitor at $10V$.

solutions after few iterations, due to the sheer, exponential growth of the involved expressions' size. We detail these issues and others next.

Richer expressions. As illustrated in the introduction and in the previous RLCS example, there are several essential, non-linear operations that are necessary to accomodate if one wishes to employ Lince in the analysis of diverse, common hybrid scenarios. We therefore now permit non-linear expressions outside of ODEs, essentially by using as basis the grammar of hybrid programs that was described in Section 2. Thus expressions outside the ODEs can now include for example the operations: division and multiplication of variables, more complex mathematical functions (such as square root extraction, exponentials, logarithms, minimum/maximum, and (co)sine), and mathematical constants (namely pi and Euler's constant).

As for expressions inside ODEs, the linearity constraint is kept but the associated parser is much less rigid. A core feature is that it now tries to convert non-linear expressions into equivalent linear ones via algebraic laws. For example, it converts the expression $x \cdot 5$, which syntactically is not a linear expression, into the linear one $5 \cdot x$ since multiplication is commutative. Most notably, it converts non-linear expressions $x \cdot y$ into scalar multiplications $s \cdot x$ or $s \cdot y$ if it can infer that either x or y is a constant with value s. Such a feature is critical in our RLCS example, where we multiply variables in the respective ODEs.

More informative error messages. Several errors were undetected at an early stage of the simulation process, which resulted in unintelligible error messages in many situations. We thus added and improved the detection and notification of several key errors occurring in typical usages of Lince, including when: (1) a partial function fails (such as in division by 0); (2) a variable is not properly initialised; (3) the number of arguments of a function is incorrect; (4) the solver fails to solve a system of ODEs; and (5) ODEs contain non-linear expressions after de-sugaring. For example, in our RLCS simulation when defining `c` to be 0 we now obtain the error "*Error: the divisor of the division 'rU/(c)' is zero.*". In our experience, this more precise detection and notification of errors drastically improved user experience.

Numerical solver. As already mentioned, several hybrid programs such as our RLCS example cannot be properly handled by the (exact) solver of ODEs (*viz.* SageMath [27]) used by Lince. We have therefore implemented an alternative, numerical solver based on the popular fourth-order Runge-Kutta method. At the theoretical level, this only required a small adaptation of the operational semantics presented

in Section 2. Specifically we no longer assume that the solution $\phi_{\sigma}^{\vec{x}'=\vec{\ell}}$ associated to a system of ODEs $\vec{x}' = \vec{\ell}$ and an initial condition σ is exact. At the practical level, this allowed us to keep the size of expressions involved in computations highly manageable thus allowing Lince to cover a broader range of examples such as the RLCS.

4 An Improved Visualiser for Hybrid Programs

Many hybrid programs cannot be easily understood by simply plotting values of variables over time. For example, in some cases one may wish to analyse the movement of a vehicle in a 2D plane, or to analyse how its behaviour varies due to changes in its initial position and velocity. This section presents an extension of Lince's visualisation capabilities in these two directions. In the same spirit of the preceding section, we complement our description with a running example.

Running example: avoiding and manoeuvring around obstacles. The *Automatic Emergency Braking* (AEB) system is an autonomous driving device that after reading its distance to an obstacle and its current velocity, decides whether to decelerate until stopping [1]. Here we present a more advanced version of the AEB that after stopping also manoeuvres around the obstacle – clearly a process involving two or even three spatial dimensions. Such a system is called *Automatic Emergency Braking with an Overtaking Manoeuvre* (AEBOM).

The continuous dynamics of the AEBOM (*i.e.* the differential equations involved) is typically given by Dubins dynamics which essentially describe the object's orientation over time (an angle) and its effect on the object's velocity along the different spatial dimensions [25]. We adopt this approach as well. For simplicity we additionally assume that our object is a robot that is able to rotate around itself. The overall process of our AEBOM is thus as follows: move forward until detecting the obstacle and in which case decelerate until stopping; then rotate to the left and move forward a prescribed number of meters (that depends on the obstacle's size); then rotate right and move forward again a prescribed number of meters; and finally repeat the last step.

Figure 5 depicts the original visualisation of the AEBOM simulation on the left, and a customised 2D visualisation that uses our extension on the right. The respective implementation of the AEBOM, included in Lince online, is not relevant to show at this stage, because our focus is at the moment on describing new visualisation mechanisms and not features concerning code. Observe as well that the plot on the right provides novel insights with respect to the one on the left: whilst in the right it is clear that the robot does not collide with the obstacle and performs the overtaking manoeuvre safely, in the left it is much harder to see that the same occurs. We provide more details about our improved plotting system next.

Higher-dimensional trajectories and beyond. Our new visualisation framework in Lince uses the Plotly JavaScript library to display plots[2]. Among other things, we now support 2D and 3D scatter plots, and include dedicated markers such as the large circles indicating the start and end points of trajectories. When hovering over these markers, extra information is displayed, *e.g.* the respective values, relevant information about the conditionals involved, and potential warnings. We also exploit the animation functionality of Plotly in plots that do not include the time component, by moving a highlighting circle through the trajectories capturing how values vary throughout time. This feature is active by default. To take all these possiblities into account, Lince allows the user to adjust different settings of the plot under

[2] http://plotly.com/

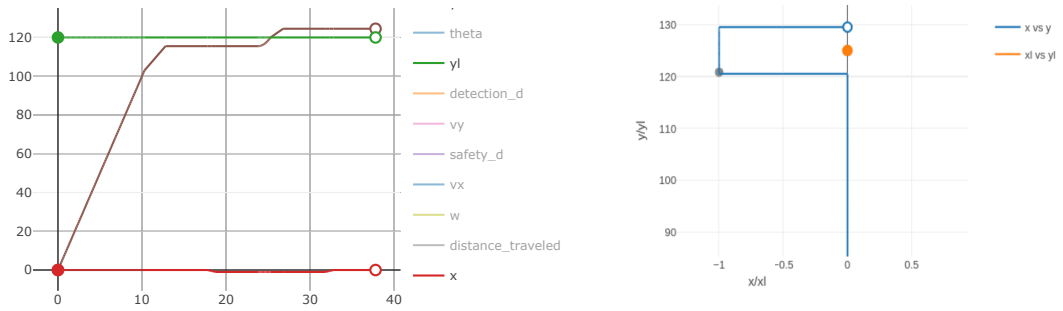


Figure 5: Plot of AEBOM using the traditional plotting system in Lince (left), and a new customised 2D plot (right) relating `x` with `y` (the robot's coordinates) and `xl` with `yl` (the obstacle's coordinates).

analysis so that she can obtain the best possible configuration for her needs. We very briefly detail such settings next:

- *Axis*: Allows defining the relationships between variables which will automatically be presented in the respective plots. For example, by setting `[x, y, v]`, if the graph type is scatter, three separate graphs will be generated where the vertical axis represents each of the variables `x`, `y`, and `v`, while the horizontal axis represents time. Choosing which variables to map to the axes is crucial for proper data analysis, allowing direct visual comparisons between different variables over time or with each other.
- *Max Time*: Refers to the duration of the simulation.
- *Max Iterations*: Specifies the maximum number of iterations (in while-loops) that the simulation can perform.
- *Graph Type*: Defines the type of graph to be used for visualising the simulation data, by selecting from the available types ('scatter' or 'scatter3d'). In a nutshell, a scatter plot is a 2D graph used to display the relationship between two variables, with data points plotted in the two-dimensional plane. Scatter3D serves the same purpose but involves three variables, with data points plotted in the three-dimensional space.

The summarised settings are presented in Fig. 6, where the values there listed are the ones used to obtain the plot in Fig. 5 on the right.

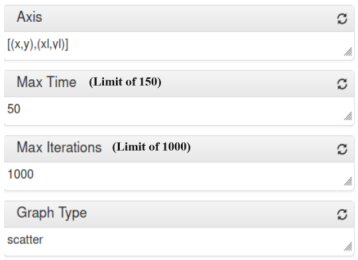


Figure 6: Input boxes that allow for the configuration of the visualisation.

Variability of initial conditions. As mentioned before, it is highly relevant take into account how the behaviour of a hybrid program varies due to changes in its initial conditions. In the AEBOM previously described in particular, it is of fundamental importance to understand how the robot manoeuvres around

an obstacle with respect to different initial positions and velocities – for it is unrealistic to expect that it moves with well-known, exact conditions. A similar, more general discussion can be consulted in [25].

In order to address this aspect we extended Lince in two steps: first its syntax now allows the listing of different initial conditions at the same time. Such is illustrated in Fig. 7 on the left, with a snippet of code used to specify initial values with respect to our robot in the AEBOM example. The latter's initial position (x,y) for example, can now be either $(0,0)$, $(2,0)$, or $(4,0)$; and similarly we have different initial velocities (vy) towards the obstacle, 4, 8, and $12\,{}^{m}/_{s}$. Second Lince now pre-processes such listings in the code and derives all possible combinations of initial conditions, which of course yields several hybrid programs at once (in the standard syntax). These data is then fed into Lince's visualiser which presents multiple simulations overlapped in the same plot. Such is seen in Fig. 7 on the right, again with our AEBOM example, where we see that our robot behaves in the same way under different initial conditions.

```
//Initial Conditions
x  := [0,2,4];
y  :=  0;
vx :=  0;
vy := [4,8,12];
xl := [0,2,4];
yl := [120,135,150];

...
```

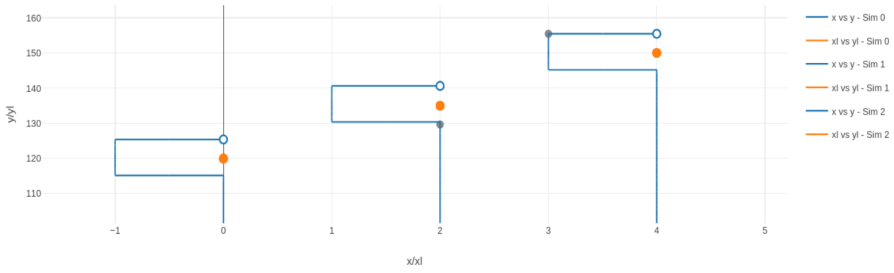


Figure 7: Visualisation of multiple simulations overlapped concerning the AEBOM.

5 Lince at Work: a Showcase of the Overall List of Improvements

This section illustrates the overall list of improvements made to Lince (as described in the preceding sections) working together in the design and analysis of a complex hybrid scenario – specifically we focus on a multi-dimensional pursuit game between two players (for example two drones) [20, 2, 6, 18]. Our illustration focuses mainly on two aspects: (1) Lince's capability tosimulate such scenarios, with optimally configured 3D plotting systems; and (2) the time that Lince takes to simulate increasingly larger systems, to provide insights over limitations of the current implementation.

Pursuit Games. Pursuit games are a captivating class of problems involving multiple agents, where at least one them (the pursuer) aims to capture or reach another (the evader) [20, 2, 6, 18, 25]. Such games are extensively studied across various disciplines, including mathematics, game theory, robotics, and computer science, due to their practical and theoretical significance. Indeed they model a wide range of real-world situations, from military and security operations to animal behaviour and industrial applications.

In this section we explore a specific 3D pursuit game, where we perceive the pursuer as a drone that attempts to capture another one in the three-dimensional space. This scenario is particularly challenging, due to the additional complexity introduced by the third dimension which requires a higher level of planning and coordination between the drones' movements. In order to model this problem we base our game's continuous dynamics on Dubins dynamics [25], *i.e.* as in Section 4 but now in three dimensions.

Our overarching strategy for the pursuer is to simply point its orientation to the evader's position at every iteration in a certain while-loop. Of course there are other options, such as that of (variations of)

Dubins paths [25, 5], but our version already suffices to properly illustrate Lince at work. Technically our approach utilises the angular velocity tensor to perform 3D infinitesimal rotations [7]. Additionally we use the cross product between the projection of the relative velocity vector and the relative position vector in each plane to determine the orientation of rotation among the three axes. We do not show here the coding details of all these processes, since this is unnecessary for our illustration. However the interested reader can consult details about these in [5, 7], and the complete code of our program is included in the examples available in Lince online.

We now show the simulation of our game in Lince across different scenarios. In the first case, the pursuer starts from the position `(300,300,600)` with a velocity of `(-20,-10,0)`$^{m}/_{s}$, while the evader begins at the position `(600,600,500)` with a velocity of `(10,0,10)`$^{m}/_{s}$. The pursuer's angular velocity along each axis is `(1/20)*2*pi()`$^{rad}/_{s}$ (20 seconds to complete a full rotation); and for the evader `(1/40)*2*pi()`$^{rad}/_{s}$ (40 seconds to complete a full rotation). The pursuer is allowed to actuate every 0.1s, and it wins the game if it reaches a distance of less than one meter with respect to the evader. Finally, for simplicity we assume a pre-defined set of movements for the latter player. Using these parameters, we simulated the corresponding program in Lince and generated a 3D scatter plot of the positional variables for both the pursuer and the evader, resulting in the graphical representation shown in the Fig. 8 after 73 seconds.

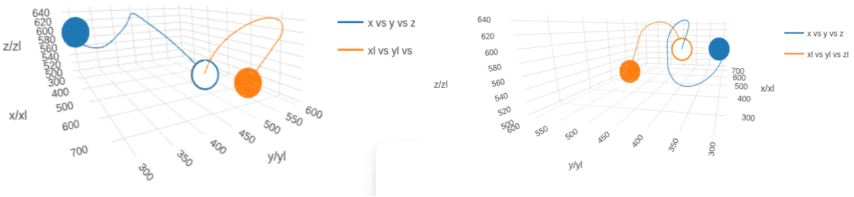


Figure 8: Two views of the same plot, where a pursuer (blue) captures an evader (orange).

We can see that the decision strategy for the pursuer adopted in this hybrid program successfully guided it to the evader, resulting in a capture at the position `(691.26,441.92,561.12)` after 27.7 seconds. However if we change the initial velocity of the evader to a higher value, such as `(20,0,9)`$^{m}/_{s}$, we no longer can visualise the capture of the evader within the limits used for this simulation (Fig. 9). Indeed, Lince supports the customisation of bounds both on the maximal time and on the number of times loops are unfolded, to avoid infinite computations. In this case, using larger bounds would allow the pursuer to capture the evader in the plot.

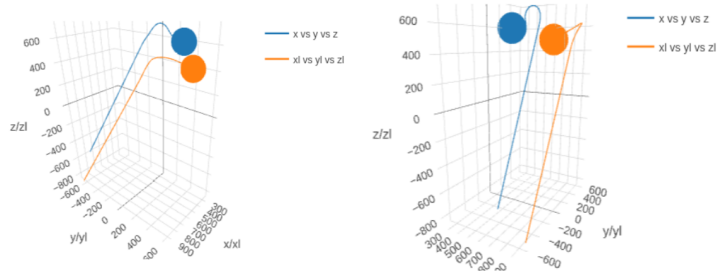


Figure 9: Similar plot to the one in Fig. 8, but using different initial velocities while keeping the same bounds on the size of the plot; this leaves out the point of the capture.

Finally by taking advantage of the variability results presented in Section 4 we very briefly study the

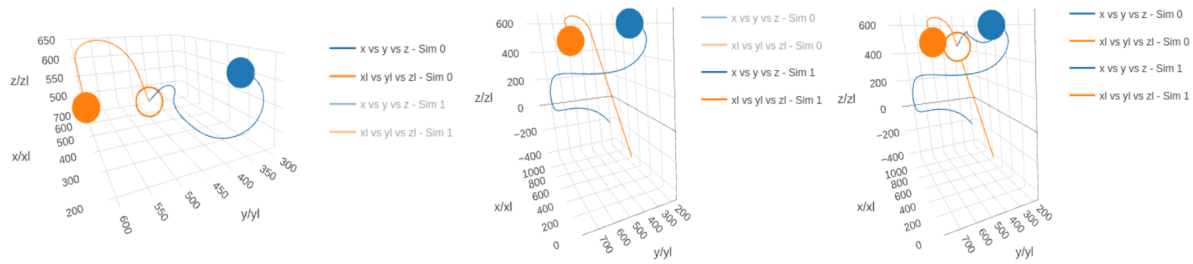


Figure 10: Two simulations (left and middle) of a pursuit game using different initial velocities (`(1/40)*2*pi()`$^{rad}/_{s}$ and `(1/100)*2*pi()`, respectively); the right plot depicts both simulations overlaid.

effects of using different velocities in this pursuit game. Specifically we adjust the angular velocity of the pursuer along each axis to be either `(1/40)*2*pi()`$^{rad}/_{s}$ or `(1/100)*2*pi()`$^{rad}/_{s}$, whilst keeping all other aspects. The resulting graphical representation (after 220 seconds) is shown in Fig. 10. From the plots we observe that the pursuer successfully captures the evader when the angular velocity is `(1/40)*2*pi()`$^{rad}/_{s}$ at the position `(692.07,415.62,464.63)` in 34.8 seconds (left plot). However with an angular velocity of `(1/100)*2*pi()`$^{rad}/_{s}$, the pursuer does not capture the evader in this time frame (middle plot). These simulations showcase Lince's ability to model and simulate complex scenarios, thus providing valuable insights into a system's behavior.

A brief overview of Lince's time performance. As shown in the previous example, Lince still has a few limitations concerning performance. In order to give the reader a more concrete idea of them we provide next an overview of how Lince fares perfomance-wise against the examples presented in this paper. First we need to give further context on how Lince operates.

The first main observation is that now that Lince is equipped with an effective numerical solver (recall Section 3) it can operate in two starkly different ways: one analytical with exact methods that rely on SageMath's framework [27], the other numerical, based on progressively closer approximations as described in Section 3. Both operation modes have significant differences performance-wise: most notably the former is obviously slower and gives timeouts much more frequently than the latter (recall our RLCS example in Section 3). Interestingly the bottleneck hinges not only on the employment of a precise solver, but also on the fact that:

1. this solver is external to Lince, specifically our tool needs to interact with a server, with all the usual delays that this implies;
2. along the evaluation of a hybrid program, Lince needs to simplify resulting expressions over and over to make them tractable (due to them being symbolic and not numerical).

We saw first-hand in Section 3 how all these extra tasks running behind the curtains inhibit Lince to simulate programs such as the RCLS circuit. The numerical solver, on the other hand, avoid these problems, but at the cost of less precision which may have deep implications if one wishes to have full guarantees that a simulation is correct, particularly if the system at hand is chaotic [23]. Needless to say, to find methods that have the virtues of both approaches is a very interesting challenge.

Table 1 lists several execution times of Lince against different variations of the examples presented in the paper. More specifically, each row represents one of our three examples with varying sampling times and total number of iterations. The example AEB is a variation of AEBOM, where the vehicle stops instead of performing an overtaking manoeuvre. All these examples are fully available in our improved Lince online.

Table 1: An overview of Lince's time performance with respect to the examples discussed in this paper. We consider different sampling times, number of iterations, and both exact and approximate methods.

	Sampling Time	**Nº of Iterations**	**Time Symb-Server**	**Time Symb-Total**	**Time Numerical-Total**
RLCS	0.01s	1000	-	-	11.46s
	0.1s	1000	-	-	10.98s
	1s	150	-	-	1.14s
AEB	0.01s	184	23.56s	23.70s	0.41s
	0.1s	19	13.04s	13.08s	0.18s
	1s	2	11.90s	11.97s	0.14s
AEBOM	0.01s	1000	-	-	8.85s
	0.1s	128	-	-	0.62s
	1s	21	-	-	0.35s
Pursuit Games	0.01s	1000	-	-	66.60s
	0.1s	322	-	-	18.26s
	1s	150	-	-	7.85s

We used a Linux laptop with a Intel quad-core i5 processor and 16GB RAM running both the server and the client. The columns ***Sampling time*** and ***Nº of Iterations*** refer respectively to the rate at which computational tasks need to be performed and the total number of times the while-loop in the program involved is unfolded. The column ***Time Symb-Total*** presents the time since a new program is loaded, before parsing, until the plot is displayed in the browser. The column ***Time Symb-Server*** measures only the time taken since the launch of a dedicated process running SageMath until it is terminated at the end of a trajectory. The column ***Time Numerical-Total*** measures the time taken since a program is loaded until its plot is displayed, computed using numerical approximations. Some observations over the values on Table 1 follow below.

- Most examples, except for AEB, reach a timeout (set in our server) when using the symbolic analysis, marked in the table with "-". The feasibility of AEB is mainly due to the smaller number of required calls to the symbolic engine.
- In the AEB example we observe that, when using exact methods, around 99
- The numerical mechanisms in the AEB example yield simulations significantly faster than in the exact counterpart.
- The total time taken to numerically simulate the RLCS and AEBOM examples are shorter than in the Pursuit Games example. This is because these two examples involve fewer computations and the Pursuit Games use a 3D scatter plot, which is more computationally intensive than the 2D scatter plot.
- Larger sampling times imply reduced times in generating both the exact and numerical plots, due to the decreased number of computational operations. Consequently, it takes longer to simulate controllers with higher precision that actuate on physical processes such as movement, velocity, and time. However, many critical systems, e.g., in the context of autonomous driving and other embedded systems, may require such a high precision.

6 Conclusion and Future Work

We presented an improved version of Lince, which can now handle a broader class of hybrid programs and aims overall at improving user experience. As previously discussed, this required an extension with the possibility of failure of the operational semantics introduced in [11], the implementation of an efficient numerical solver, and more informative error messages, among other things.

We believe that our work opens up several research paths that we would like to explore next. For example, thanks to the numerical solver it is now straightforward to extend our language with non-linear differential equations, which widens even more the range of programs that Lince can currently tackle. Another interesting research path is the addition of probabilistic constructs to Lince, such as measure sampling. We conjecture that this could be handled easily in Lince via a random-number generator and part of the implemented variability mechanisms that were presented in Section 4.

Yet another interesting research line is to connect Lince to the theorem prover for hybrid programs KeYmaera X [25] – specifically the connection would consist of a suitable encoding from programs written in Lince to programs written in KeYmaera X. Such would establish a workflow in which the engineer first *analyses* a given hybrid program via simulation mechanisms (provided by Lince) and subsequently *proves* properties about this program (*e.g.* correctness) via KeYmaera X.

Acknowledgments. This work is financed by National Funds through FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project IBEX, with reference 10.54499/PTDC/CCI-COM/4280/2021. This work is also partially supported by National Funds through FCT/MCTES, within the CISTER Unit (UIDP/UIDB/04234/2020); and by the EU-/Next Generation, within the Recovery and Resilience Plan, within project Route 25 (TRB/2022/00061 – C645463824-00000063).

References

[1] Proctor Acura: *Technology Guide: What is an Automatic Braking System?* https://www.proctoracura.com/automatic-braking-system-guide.

[2] T. Anderson, R. de Lemos, J. S. Fitzgerald & A. Saeed (1993): *On formal support for industrial-scale requirements analysis*. In Robert L. Grossman, Anil Nerode, Anders P. Ravn & Hans Rischel, editors: *Hybrid Systems*, Springer Berlin Heidelberg, pp. 426–451, doi:10.1007/3-540-57318-6_39.

[3] Paolo Ballarini, Hilal Djafri, Marie Duflot, Serge Haddad & Nihal Pekergin (2011): *COSMOS: A Statistical Model Checker for the Hybrid Automata Stochastic Logic*. In: *Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011*, IEEE Computer Society, pp. 143–144, doi:10.1109/QEST.2011.24.

[4] Davide Bresolin, Luca Geretti, Tiziano Villa & Pieter Collins (2015): *An Introduction to the Verification of Hybrid Systems Using Ariadne*, pp. 339–346. Lecture Notes in Control and Information Sciences, Springer, doi:10.1007/978-3-319-10407-2_39.

[5] Xuan-Nam Bui, J.-D. Boissonnat, P. Soueres & J.-P. Laumond (1994): *Shortest path synthesis for Dubins non-holonomic robot*. In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pp. 2–7 vol.1, doi:10.1109/ROBOT.1994.351019.

[6] Zhou Chaochen, Anders P. Ravn & Michael R. Hansen (1992): *An Extended Duration Calculus for Hybrid Real-Time Systems*. In Robert L. Grossman, Anil Nerode, Anders P. Ravn & Hans Rischel, editors: *Hybrid Systems*, *Lecture Notes in Computer Science* 736, Springer, pp. 36–59, doi:10.1007/3-540-57318-6_23.

[7] Garanin Dmitry (2008): *Rotational motion of rigid bodies*. https://www.lehman.edu/faculty/dgaranin/Mechanics/Mechanis_of_rigid_bodies.pdf.

[8] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang & Oded Maler (2011): *SpaceEx: Scalable Verification of Hybrid Systems*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, pp. 379–395, doi:10.1007/978-3-642-22110-1_30.

[9] Peter Fritzson (2014): *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach*. John Wiley & Sons, doi:10.1002/9781118989166.

[10] Sergey Goncharov & Renato Neves (2019): *An Adequate While-Language for Hybrid Computation*. In Ekaterina Komendantskaya, editor: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019*, ACM, pp. 11:1–11:15, doi:10.1145/3354166.3354176.

[11] Sergey Goncharov, Renato Neves & José Proença (2020): *Implementing Hybrid Semantics: From Functional to Imperative*. In Violet Ka I Pun, Volker Stolz & Adenilso Simão, editors: *Theoretical Aspects of Computing - ICTAC 2020 - 17th International Colloquium, Macau, China, November 30 - December 4, 2020, Proceedings*, *Lecture Notes in Computer Science* 12545, Springer, pp. 262–282, doi:10.1007/978-3-030-64276-1_14.

[12] Volkan Gunes, Steffen Peter, Tony Givargis & Frank Vahid (2014): *A Survey on Concepts, Applications, and Challenges in Cyber-Physical Systems*. *Transactions on Internet and Information Systems* 8(12), pp. 4242–4268, doi:10.3837/TIIS.2014.12.001.

[13] Ahammodullah Hasan, Md Abdul Halim & MA Meia (2019): *Application of linear differential equation in an analysis transient and steady response for second order RLC closed series circuit*. *American Journal of Circuits, Systems and Signal Processing* 5(1), pp. 1–8.

[14] Peter Höfner (2009): *Algebraic calculi for hybrid systems*. Ph.D. thesis, University of Augsburg. Available at http://opus.bibliothek.uni-augsburg.de/volltexte/2010/1481/.

[15] Eduard Kamburjan, Stefan Mitsch & Reiner Hähnle (2022): *A Hybrid Programming Language for Formal Modeling and Verification of Hybrid Systems*. *Leibniz Transactions on Embedded Systems* 8(2), pp. 04:1–04:34, doi:10.4230/LITES.8.2.4.

[16] Harold Klee (2007): *Simulation of Dynamic Systems with MATLAB and Simulink*. CRC Press, Inc., USA.

[17] Dexter Kozen (1997): *Kleene algebra with tests* 19(3), p. 427–443. doi:10.1145/256167.256195.

[18] Tomas Krilavicius (2006): *Hybrid Techniques for Hybrid Systems*. Ph.D. thesis, University of Twente, Enschede, Netherlands. Available at http://eprints.eemcs.utwente.nl/9609/.

[19] Edward A. Lee & Sanjit A. Seshia (2016): *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press.

[20] Zohar Manna & Amir Pnueli (1992): *Verifying Hybrid Systems*. In Robert L. Grossman, Anil Nerode, Anders P. Ravn & Hans Rischel, editors: *Hybrid Systems*, *Lecture Notes in Computer Science* 736, Springer, pp. 4–35, doi:10.1007/3-540-57318-6_22.

[21] Renato Neves (2018): *Hybrid programs*. Ph.D. thesis, University of Minho. Available at https://repositorium.sdum.uminho.pt/handle/1822/56808.

[22] E-R Olderog (1992): *Nets, terms and formulas: three views of concurrent processes and their relationship*. Cambridge University Press.

[23] Lawrence Perko (2013): *Differential equations and dynamical systems*. 7, Springer Science & Business Media, doi:10.1007/978-1-4613-0003-8.

[24] André Platzer (2010): *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*. Springer, doi:10.1007/978-3-642-14509-4.

[25] André Platzer (2018): *Logical Foundations of Cyber-Physical Systems*. Springer, doi:10.1007/978-3-319-63588-0.

[26] John C Reynolds (1998): *Theories of programming languages*. Cambridge University Press, doi:10.1017/CBO9780511626364.

[27] W. A. Stein et al. (2015): *Sage Mathematics Software (Version 6.4.1)*. The Sage Development Team. http://www.sagemath.org.

[28] Glynn Winskel (1993): *The formal semantics of programming languages - an introduction*. Foundation of computing series, MIT Press, doi:10.7551/mitpress/3054.001.0001.

[29] Yue Zhang & Anurag Srivastava (2021): *Voltage Control Strategy for Energy Storage System in Sustainable Distribution System Operation*. Energies 14(4), doi:10.3390/en14040832.

ROSMonitoring 2.0: Extending ROS Runtime Verification to Services and Ordered Topics

Maryam Ghaffari Saadat
University of Manchester
Manchester, United Kingdom
maryam.ghaffarisaadat@manchester.ac.uk

Angelo Ferrando
University of Modena and Reggio Emilia
Modena, Italy
angelo.ferrando@unimore.it

Louise A. Dennis
University of Manchester
Manchester, United Kingdom
louise.dennis@manchester.ac.uk

Michael Fisher
University of Manchester
Manchester, United Kingdom
michael.fisher@manchester.ac.uk

Formal verification of robotic applications presents challenges due to their hybrid nature and distributed architecture. This paper introduces ROSMonitoring 2.0, an extension of ROSMonitoring designed to facilitate the monitoring of both topics and services while considering the order in which messages are published and received. The framework has been enhanced to support these novel features for ROS1 – and partially ROS2 environments – offering improved real-time support, security, scalability, and interoperability. We discuss the modifications made to accommodate these advancements and present results obtained from a case study involving the runtime monitoring of specific components of a fire-fighting Uncrewed Aerial Vehicle (UAV).

1 Introduction

The formal verification of robotic applications is a challenging task. Due to their heterogeneous and component-based nature, establishing the correctness of robotic systems can be particularly difficult. Various approaches exist to tackle this problem, ranging from testing methods [8, 21, 9] to static [16, 14] or dynamic [19, 15] formal verification. In this work, we focus on the latter approach to verification, specifically the extension of ROSMonitoring [15], a Runtime Verification (RV) framework developed for monitoring robotic systems deployed in the Robot Operating System (ROS) [2]. ROS is widely used, providing a *de facto* standard for robotic components. ROS encourages component-based development of robotic systems where individual components run in parallel, may be distributed across several processors, and communicate via messages. We tackle ROSMonitoring because it is a novel, formalism-agnostic, and widely used framework for the runtime monitoring of ROS applications. ROSMonitoring allows the specification of formal properties externally to ROS, without imposing any constraints on the formalism to be used. The properties that can typically be monitored in ROSMonitoring concern messages exchanged between different ROS components, called nodes. Such message communication is achieved through a publish-subscribe mechanism, where some nodes (referred to as *publishers*) publish messages on a topic and other nodes (referred to as *subscribers*) subscribe to these topics to listen for the published messages. Through ROSMonitoring, it is possible to specify the communication flow on such topics. For instance, one can determine which messages are allowed in the current state of the system, the correct order amongst them, and other relevant criteria.

Unfortunately, not all aspects of verifying ROS applications are based solely on message communication. In fact, when developing ROS systems, other communication mechanisms can also be utilised,

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 38–55, doi:10.4204/EPTCS.411.3

© M. G. Saadat, A. Ferrando, L. A. Dennis, & M. Fisher
This work is licensed under the
Creative Commons Attribution License.

such as *services*. Unlike the publish-subscribe mechanism used with topics, services provide a way for nodes in ROS to directly offer functionalities to each other. While topics are typically used to transmit data from sensors, services serve as an interface that enables nodes to offer specific functionalities to others within the ROS system. Unlike topics, services are commonly synchronous, meaning that when a node calls a service, it waits for a response from the receiving node. This is in contrast to topics, where subscription is non-blocking, and the subscriber node is simply notified whenever a new message is published on the topic, without any waiting involved. Services are not supported in ROSMonitoring, restricting the framework's functionality to solely monitoring messages.

Another current limitation of ROSMonitoring pertains to the handling of message order. The framework orders messages based on the chronological order in which they are received by subscriber nodes. However, this approach only considers the viewpoint of subscribers, which may not always be suitable. In some scenarios, it may be necessary to consider the order of messages based on when they were sent. For instance, if one message was sent before another, the former should be analysed before the latter by the monitor, appearing earlier in the resulting trace of events. Unfortunately, ROSMonitoring does not currently provide a representation of the order in which messages are published and received. Generally, messages on a single topic are received by subscribers in the order they were published. However, if a property needs to monitor several topics, then it is unusual for the messages from more than one topic to be received in the order they were published. Reordering messages according to publication time is necessary if checking conditional actions that respond to specific event patterns.

In this paper, we introduce ROSMonitoring 2.0, an extension of ROSMonitoring designed to facilitate the monitoring of both topics and services while also considering the order in which messages are published and received. The framework has been enhanced to support these novel features for ROS. Some of these features, *i.e.*, service monitoring, have also been ported to ROS2[1] environment as well. We discuss the modifications made to accommodate these advancements and present results obtained from a case study involving the runtime monitoring of specific components of a fire-fighting Uncrewed Aerial Vehicle (UAV).

2 Preliminaries

In this section, we briefly introduce Runtime Verification and the ROSMonitoring framework. We emphasise the primary distinction between RV and static verification techniques. Additionally, we provide an overview of the ROSMonitoring framework and briefly outline its main features.

2.1 Runtime Verification

Runtime Verification (RV) is a lightweight formal verification technique that checks the behaviour of a system while it is running [23]. Unlike model checking, RV does not suffer from the state space explosion problem typical in static verification methods and is therefore much more scalable [11]. RV is particularly suitable for robotic applications due to resource limitations and system complexity that make full verification at design-time challenging. While static verification techniques focus on abstracting system components, RV checks system behaviour directly. RV addresses the word inclusion problem [5], determining if a given event trace belongs to the set of traces denoted by a formal property (referred to as the property's language). This verification process is polynomial in time relative to the trace length. In

[1] ROS2 is the upgraded version of ROS1, providing improved real-time support, security, scalability, and enhanced interoperability with multiple communication middleware options.

contrast, model checking exhaustively verifies if a system satisfies or violates a property by analysing all possible system executions, tackling the language inclusion problem, and is typically PSPACE-complete for non-deterministic finite automata [31]. RV commonly employs runtime monitors, automatically synthesised from formal properties, often expressed using Linear-time Temporal Logic (LTL) [28]. These monitors gather information from system execution traces and conclude whether the system satisfies or violates the property. A monitor returns $\top$ if the trace satisfies the property, $\bot$ if it violates it, and ? if there is insufficient information. Depending on the property's formalism, ? may further split into $?_{\top}$ or $?_{\bot}$ indicating partial satisfaction or partial violation, respectively.

2.2 ROSMonitoring

ROSMonitoring [15] is a framework for performing RV on ROS applications. ROSMonitoring allows the user to add monitors to ROS applications, which intercept the messages exchanged between components, called "ROS nodes"[2], and check whether the relevant messages conform to a given formal property. In the following we describe these three different aspects in more detail.

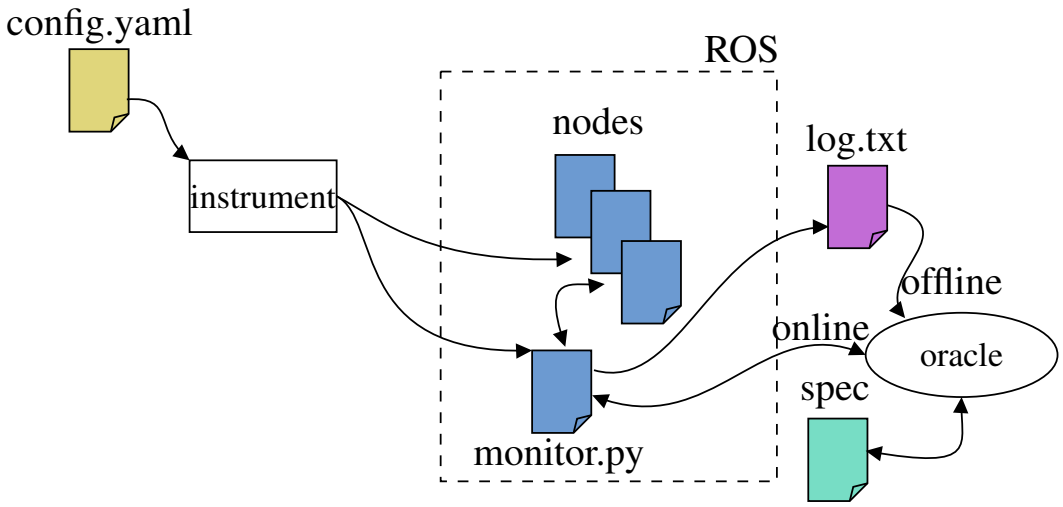


Figure 1: High-level overview of ROSMonitoring [15].

2.2.1 Instrumentation

ROSMonitoring starts with a YAML configuration file to guide the instrumentation process required to generate the monitors. Within this file, the user can specify the communication channels, called "ROS topics", to be intercepted by each monitor. In particular, the user indicates the name of the topic, the ROS message type expected in that topic, and the type of action that the monitor should perform. After preferences have been configured in *config.yaml*, the last step is to run the generator script to automatically generate the monitors and instrument the required ROS launch files.

[2]ROS is node-based, each robot can be composed of multiple nodes.

2.2.2 Oracle

ROSMonitoring decouples the message interception (monitor) and the formal verification aspects (oracle) and so is highly customizable. Different formalisms can be used to represent the properties to be verified, including Past MTL, Past STL, and Past LTL (MTL [22], STL [24], and LTL [28] with past-time operators, respectively). Using the formalism of choice, an external entity can be created to handle the trace of events reported by the monitors in ROS (generated through instrumentation). ROSMonitoring requires very few constraints for adding a new oracle. It uses JSON[3] (JavaScript Object Notation) as a data-interchange format for serialising the messages that are observed by the ROS monitor. JSON is commonly used for transmitting data between a server and a web application. In JSON, data is represented as key-value pairs enclosed in curly braces, making it a popular choice for APIs and data storage. An oracle will parse the JSON messages, check whether they satisfy or violate the formal property, and report back to the ROS monitor.

2.2.3 ROS monitor

The instrumentation process generates monitors to intercept the messages of interest. Each monitor is automatically generated as a ROS node in Python, which is a native language supported in ROS. ROSMonitoring provides two types of monitors: 1) offline monitors which simply log the intercepted events in a specified file to be parsed by the Oracle later to determine whether they satisfy a given set of properties, and 2) online monitors which query the Oracle in real time about whether the intercepted messages satisfy the given properties. While offline monitors only log the observed messages, online monitors could either log messages along with the Oracle verdict updated after each message or filter messages that the Oracle deems have violated the given properties. To clarify the difference, in the case of logging without filtering, if the online monitor finds a violation of the property under analysis, it publishes a warning message containing as much information as possible about the violated property. This warning message can be used by the system to handle the violation and to react appropriately. However, the monitor does not stop the message from propagating further in the system. In contrast, if filtering is enabled, since monitors can be placed between the communication of different nodes, ROSMonitoring monitors enforces the property under analysis by not propagating messages that represent a property violation. This is achieved by directing communication on the monitored topics to pass through the monitors.

3 Motivating example

In this section, we explain the rationale behind extending the ROSMonitoring framework. We use, as an example, a Battery Supervisor system [4] designed for a UAV (Uncrewed Aerial Vehicle) with three essential components depicted in Figure 2: the Battery, the Battery Supervisor, and the LED Panel. The Battery periodically reports the remaining battery percentage. The Battery Supervisor is responsible for checking the battery level and reporting its status. It subscribes to the battery percentage updates and analyses them. If the battery percentage is above 40%, it signals a ‘healthy’ status. If it is between 30% and 40%, it flags a ‘warning’ status. And if it falls below 30%, it indicates a ‘critical’ status. The LED Panel reflects the battery status through coloured LED lights. The Battery Supervisor is connected to the LED Panel and whenever it detects a change in the battery status, it sends a signal to the LED Panel to

[3] `https://www.json.org/`

[4] Full code for this example is available in the Git Repository for ROSMonitoring 2.0 Case Study.

adjust the lights accordingly. For instance, if the battery is in a critical state, the red light might flash to indicate urgency[5].

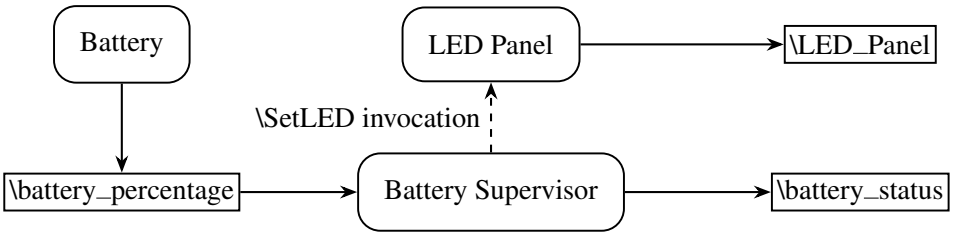


Figure 2: Motivating example with three components: Battery, Battery Supervisor, and LED Panel. Battery publishes on topic */battery_percentage*; Battery Supervisor subscribes to */battery_percentage*, publishes on topic */battery_status*, and invokes service */SetLED*; LED Panel publishes on topic */LED_Panel* and responds to */SetLED* service requests.

In this example, we are interested in ensuring that the messages exchanged between different components correspond correctly. For instance, that every status update provided by the Battery Supervisor accurately reflects the current battery percentage received from the Battery. However, while messages published on a single topic generally arrive at the subscribers according to their publication order, messages on different topics can arrive out of order. As a result, a battery status message could be observed before its corresponding battery percentage. Therefore, we need additional mechanisms to account for this before sending the messages to the Oracle for verification. This motivates our extension to reorder the messages according to their publication time.

Beyond message correspondence, we are also interested in verifying the interaction between the Battery Supervisor and the LED Panel service. We would like to confirm that every time the LED lights are adjusted based on the battery status, it is triggered by a legitimate status update. Conversely, we would like to check that every change in battery status is promptly followed by a request to adjust the LED lights, maintaining synchronisation between the visual feedback and the actual battery condition. However, services are not supported by the ROSMonitoring framework. This motivates our extension to support services. Since some of the properties we are interested in monitoring include both topics and services, we have also developed support for reordering service requests and responses according to publication time as well.

4 ROSMonitoring 2.0

ROSMonitoring 2.0 is fully available[6] for ROS1, while only partially available[7] for ROS2 (service monitoring has been added but not message reordering). In this section, we present two novel aspects of ROSMonitoring 2.0. Firstly, in Section 4.1, we detail the mechanism enabling monitoring of ROS services. Secondly, in Section 4.2, we introduce an algorithm for reordering messages based on their publication time, demonstrating its correctness under the assumption that messages on each topic arrive sequentially. Notably, such reordering mechanism is extended to support services in addition to topics.

[5]This case study was inspired by the example for RS services in the Robotics Back-End Tutorial as well as a solution to Challenge 3 of the MBZIRC Challenge competition 2020 [1].

[6]`https://github.com/autonomy-and-verification-uol/ROSMonitoring/tree/master`

[7]`https://github.com/autonomy-and-verification-uol/ROSMonitoring/tree/ros2`

4.1 Service extension

While ROS topics excel in broadcasting data streams or events asynchronously to multiple nodes, ROS services are designed for synchronous, point-to-point communication to request specific actions or services from other nodes in the system. Consequently, when monitoring services, our monitor node must directly intervene in the communication between the server and client. The monitor node then assumes the role of a server for the client, and conversely acts as a client for the server. The sequence diagram in Figure 3 illustrates the service verification process in ROSMonitoring 2.0 where message filtering is enabled (which subsumes the non-filtering scenario). The scenario begins with the Client sending a service request $callService(req, res)$ to the Monitor. Subsequently, the Monitor forwards the request to the Oracle for verification via a callback mechanism specific to the service. If the Oracle identifies the request as inconsistent with the defined property, it responds with a negative verdict (*i.e.* either $?_{\bot}$ or $\bot$)). In response, the Monitor publishes an error message and notifies the client of the discrepancy, bypassing the service invocation. Conversely, if the Oracle confirms the consistency of the request with the property, it returns a positive verdict (*i.e.*, either $?_{\top}$ or $\top$). The Monitor proceeds to invoke the service and awaits a response from the server. Upon receiving the response, the Monitor relays it back to the Oracle for evaluation. Should the Oracle determine the response to be erroneous (*i.e.*, the returned verdict is either $?_{\bot}$ or $\bot$), the Monitor again publishes an error message and notifies the client accordingly. Otherwise, it delivers the response to the client as expected.

In contrast to the standard ROSMonitoring behaviour, handling services necessitates additional verification steps. The Monitor must check both the service request and its corresponding response with the Oracle. Verifying the request is crucial to prevent invoking the service in case of a violation. Moreover, the Monitor must act as an intermediary between the client and server. This mechanism mirrors ROSMonitoring's behaviour when topic filtering is enabled, albeit with an extension in the case of services to invoke the actual service upon successful request verification.

4.2 Ordered topics extension

To recover the publication order of messages in real time, ROSMonitoring 2.0 adds timestamps to each message. These timestamps are then utilised in Algorithm 1 to propagate messages to the Oracle in their original publication order. Moreover, this approach is based on the following assumption.

Assumption 1. *Messages on each single topic arrive at subscribers in the order of publication.*

In order to use the reordering feature in the ROS monitor, in the callback function for every topic, instead of propagating the message (*msg*) directly to the Oracle, Algorithm 1 calls $addToBuffer(msg, t)$. Such a procedure accumulates messages from each topic into their respective buffers. Message release is withheld until all buffers contain at least one message, at which point the message with the earliest publication timestamp is released and sent to the Oracle. To prevent more than one thread to change the buffers simultaneously, we use locks to block write-access for a single thread.

Lemma 1. *In Algorithm 1, messages in each buffer maintain the order of publication timestamps.*

Proof. Suppose there is a topic t such that its corresponding list in dictionary *buffers* is out of order. Without loss of generality, assume there are two messages m_1 and m_2 on topic t with m_1 published before m_2 but stored in $buffers[t]$ in reverse order, *i.e.* $[..,m_2,m_1,..]$. Due to Assumption 1, since both

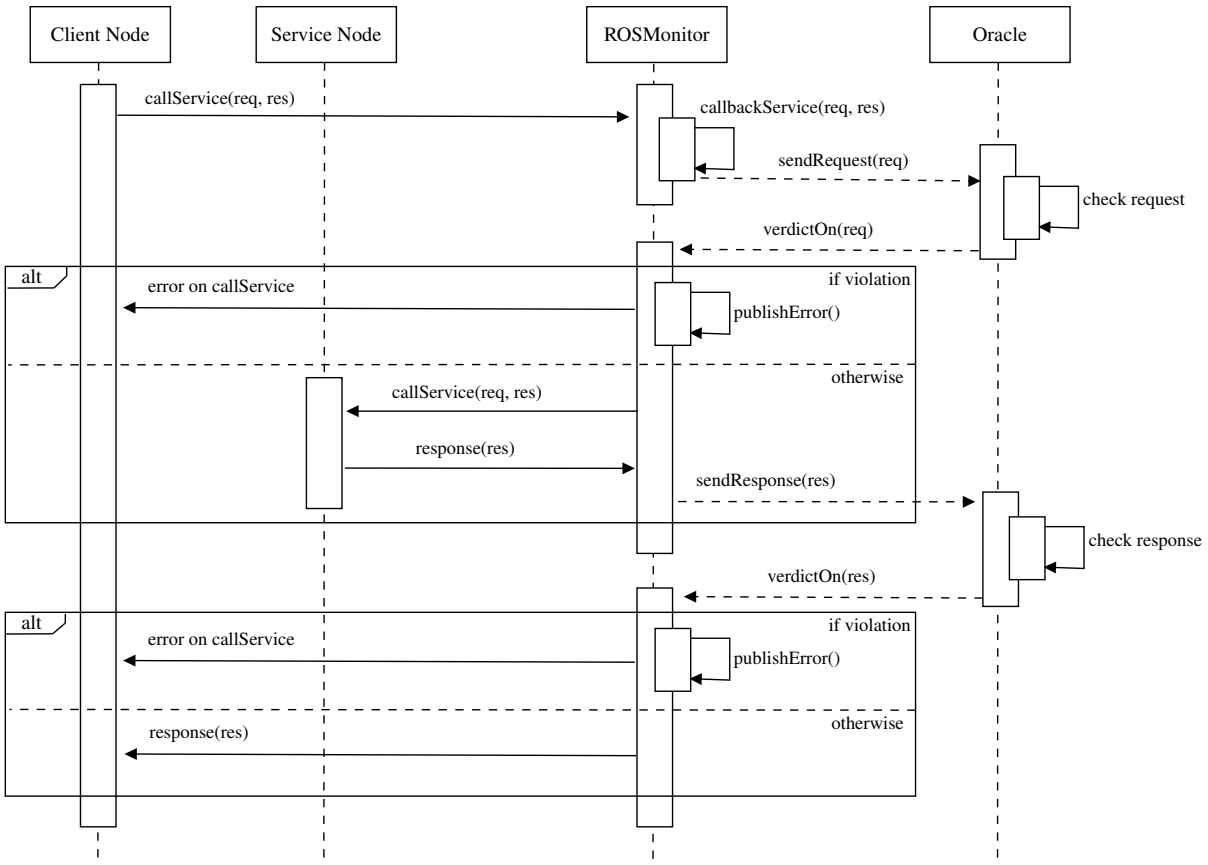


Figure 3: Service verification in ROSMonitoring 2.0 when filtering is enabled.

m_1 and m_2 are on the same topic, they are received by the ROS monitor in the correct order. Therefore, $addToBuffer(m_1, t)$ is called before $addToBuffer(m_2, t)$. Consequently, m_1 is appended to the list $buffers[t]$ before m_2. This contradicts our assumption that m_2 is stored before m_1 in $buffers[t]$. Thus, it follows, by contradiction, that Lemma 1 holds. □

Theorem 1. *Algorithm 1 propagates messages to the Oracle in the order of their publication.*

Proof. In order to prove that Algorithm 1 is correct, we assume the opposite, namely that two messages, m_1 and m_2, were propagated to the Oracle in reverse order of their publication timestamps. Without loss of generality suppose m_1 was published earlier than m_2 but was propagated to the Oracle after m_2.

If the messages are on the same topic then, due to Assumption 1, we reach a contradiction which means our assumption is incorrect and the proof is complete. Otherwise, the messages are on distinct topics. Hence, by construction, they are stored in separate lists in *buffers*. Furthermore, the algorithm only sends messages to the Oracle if the buffers for all topics are non-empty. Therefore, both m_1 and m_2 must be present in their corresponding buffers at the time m_2 is propagated to the Oracle. But the algorithm, by construction, always chooses the message with the smallest timestamp to propagate to the Oracle next. This contradicts our assumption that m_2 is propagated before m_1 despite having a larger timestamp. Consequently, we can conclude, by contradiction, that Algorithm 1 is correct. □

A successful application of the ordering mechanism necessitates careful consideration to mitigate the

Algorithm 1: Algorithm for propagating messages to the Oracle in the order they were published

Input :

msg:	a ROS message received by the monitor on a topic *t*
ws:	a global websocket
buffer:	a global dictionary mapping each topic to a list of timestamps of unprocessed messages published on that topic
messages:	a global dictionary mapping publication timestamp to corresponding message

Output: Verdict published by Oracle based on messages in the order of publication time

```
1  Function sendEarliestMessageToOracle():
2      using buffer and messages
3      min_time_stamp = minimum timestamp in messages dictionary
       message = messages[min_time_stamp]
       send message to Oracle
4      verdict = Oracle's response
5      remove min_time_stamp from buffer
6      remove message from messages
7      Publish verdict
8  Function addToBuffer(msg, t):
9      using ws, buffer, and messages
10     time_stamp_of_msg = getTime(msg)
11     add time_stamp_of_msg to buffer[t]
12     messages[time_stamp_of_msg] = msg
13     lock websocket ws
14     while no topic has an empty buffer do
15         sendEarliestMessageToOracle ()
16     end
17     unlock websocket ws
```

risk of deadlocks. Specifically, when a topic t undergoes filtering by the monitor, and another topic or service x relies on it, both t and x should not be concurrently included in the ordering process. Otherwise, the buffering of a message m_1 on topic t can lead to a deadlock scenario, as it cannot be released until an x message is buffered. Conversely, an x message cannot be generated until a t message is published, which, in turn, cannot occur until m_1 is released from the buffer. Furthermore, it is essential to carefully evaluate dependencies between topics and services when determining which should be ordered based on their publication times.

5 Experimental evaluation

In our case study, we illustrate the practical implementation of ROSMonitoring 2.0 through a scenario involving a Battery Supervisor system for a UAV. We developed an online ROS monitor which runs alongside the system and checks its behaviour against a set of properties in real time. The experiments were conducted using ROS1 Noetic distribution. As shown in Figure 2, our case study comprises three interconnected nodes: the Battery, responsible for publishing the remaining battery percentage; the Battery Supervisor, which subscribes to the battery percentage topic and publishes status updates based on predefined thresholds; and the LED Panel, which reflects the battery status through LED lights.

The Battery node periodically broadcasts the battery percentage on the */battery_percentage* topic. Meanwhile, the Battery Supervisor node, operating at a slower rate than the Battery, subscribes to this topic and publishes status updates on the */battery_status* topic. These status updates indicate the battery status as follows: status 1 for a percentage higher than 40%, status 2 for a percentage between 30% and 40%, and status 3 for a percentage between 0% and 30%. Since the Battery Supervisor has a slower publication rate, it may not report the status for every percentage published by the Battery. This is intentional to ensure that while the battery status is reported regularly, energy usage and communications are optimised.

As shown in Figure 4, to facilitate monitoring system behaviour, supplementary topics and a service are added to the original example in Figure 2. For instance, to ensure synchronisation between the Battery and the Battery Supervisor, an additional topic */input_accepted* is introduced. This topic tracks which battery percentage messages have been processed by the Battery Supervisor. Furthermore, the Battery Supervisor publishes a message on topic */status_change* if the battery status changes. The Battery Supervisor node subscribes to the */status_change* topic itself to separate the processing of */battery_percentage* messages from the invocation of service call to */SetLED* . As explained further below, this is a workaround to prevent deadlocks when using the ordering mechanism. Upon detecting a change in status by comparing the current status with the previous one, the Battery Supervisor publishes a */status_change* message. Once the Battery Supervisor receives a */status_change* message, it calls the */SetLED_mon* service to update the LED lights on the LED Panel accordingly. After receiving a */SetLED_mon* service request, the ROS monitor checks that the request is valid and it calls the */SetLED* service. Upon receiving a service call, the LED Panel publishes a message on the */status_accepted* topic to record which status update was acknowledged, followed by a message on the */LED_panel* topic reporting the current state of the LED lights (green, yellow, and red). The LED Panel also sends a response to the ROS monitor which is relayed back to the Battery Supervisor.

Subscribing to the */status_change* topic may appear peculiar for the Battery Supervisor, which publishes it. While it might seem more straightforward to invoke the */SetLED* service where the status calculation occurs based on received */battery_percentage* messages, such an approach risks deadlock. The reason is that the buffer is unable to release a service request message until accepting the next percentage message. But no further percentage messages can be accepted until a service response is received and that can only happen if the service request is released. To resolve this, we separated the publication of topics from the function which initiates service requests so that the service does not block the receipt of messages needed for producing a response.

The properties we selected to verify are as follows with formal definitions in Table 1:

1. **Topic only:** Correspondence of */battery_status* with */battery_percentage* and */input_accepted* :
   (a) Every */battery_status* message corresponds to a */input_accepted* message and correctly reports the status based on its corresponding

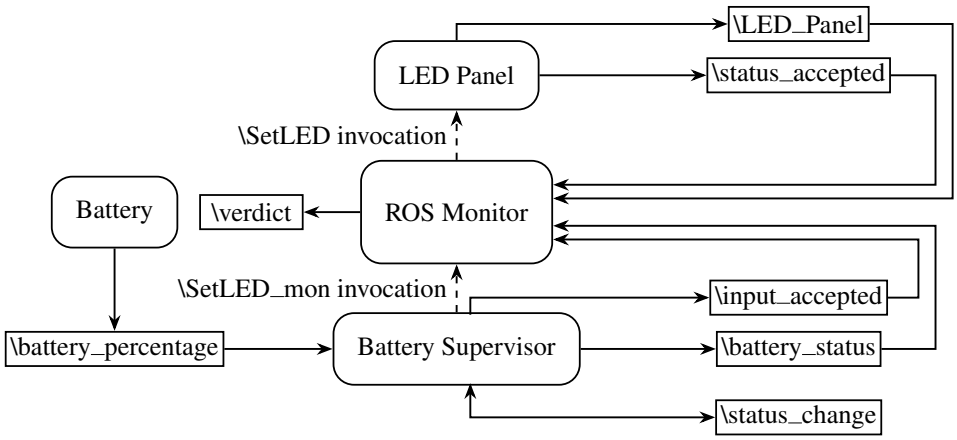


Figure 4: Case study with ROS Monitor and additional topics */input_accepted*, */status_accepted*, */status_change*, */verdict* and service */SetLED_mon*.

/battery_percentage message.

(b) Every */input_accepted* message is followed by a */battery_status* message within 100 time steps.

2. **Topic and Service:** Correspondence of */SetLED* service request with */battery_status* :
   (a) Every */SetLED* service request corresponds to a */battery_status* message and a change in battery status.
   (b) Every change of status reported via */battery_status* messages is followed by a */SetLED* service request within 100 time steps.
3. **Service only:** Correspondence of */SetLED* service request and response:
   (a) Every */SetLED* service response corresponds to a */SetLED* service request.
   (b) Every */SetLED* service request is followed by a */SetLED* service request within 100 time steps.

Formalisation of these properties requires definition of predicates, summarised in Table 2, based on the JSON messages sent to the Oracle. For topics, generic predicates *topic* and *id* are defined. Additionally, for the */battery_percentage* topic, predicate *percentage* is defined, taking values 1 for percentages between 40% and 100%, 2 for percentages between 30% and 40%, 3 for percentages between 0% and 30%, and '*INVALID*' for other values. Consistently, the *percentage* and *status* predicates share values to enable referencing, as seen in Property 1a. For the */battery_status* topic, predicate *status* holds the String version of message status, with '*INVALID*' assigned if the status is not 1, 2, or 3. Additionally, predicate *status_change* is defined to be 'True' if the corresponding field in the message is 'true'. Note that the monitor is not subscribed to the */status_change* topic which is used to trigger an LED Panel response. The reason for this redundancy is deadlock prevention. If the */status_change* topic was ordered, then its release would be contingent on a service request joining the buffers which cannot happen unless the */status_change* message is released. Our solution to this potential deadlock was to keep the */status_change* topic unordered and add a field *status_change* to the *battery_status* topic for the Oracle to determine if the LED panel is responding correctly. Such redundancies could be considered as a general technique to prevent deadlocks. For the */SetLED* service, predicates *request* and *response*

Property ID	Property formal specification
1a	forall[i]. (forall[s]. {topic: "/*battery_status*", id: *i, status: *s} → once({topic: "/*input_accepted*", id: *i}) and once({topic: "/*battery_percentage*", id: *i, percentage: *s}))
1b	forall[i]. not ({topic: "/*battery_status*", id: *i}) → once[1:]{topic: "/*battery_status*", id: *i} or not (once[100:]({topic: "/*input_accepted*", id: *i}))
2a	forall[i]. (forall[s]. {service: "/*SetLED*", req_id: *i, req_status: *s} → once({topic: "/*battery_status*", id: *i, status: *s, status_change: True})
2b	forall[i]. not ({service: "/*SetLED*", req_id: *i, req_status: *s}) → once[1:] {service: "/*SetLED*", req_id: *i, req_status: *s} or not (once[100:]({topic: "/*battery_status*", id: *i, status: *s, status_change: True})
3a	forall[i]. {service: "/*SetLED*", response: True, res_id: *i} → once({service: "/*SetLED*", request: True, req_id: *i})
3b	forall[i]. not ({service: "/*SetLED*", response: True, res_id: *i}) → once[1:] {service: "/*SetLED*", response: True, res_id: *i} or not (once[100:]({service: "/*SetLED*", request: True, req_id: *i}))

Table 1: Properties in Past Metric Temporal Logic (Past MTL) according to the Reelay Expression Format (`https://doganulus.github.io/reelay/rye/`).

Message Type	Predicate	Description
/*battery_percentage*	*topic*	Topic of the message, i.e. /*battery_percentage*
	id	Unique sequentially assigned ID for the message
	percentage	'1' if percentage $>$ 40 and percentage $\leq$ 100 '2' if percentage $>$ 30 and percentage $\leq$ 40 '3' if percentage $\leq$ 30 and percentage $\geq$ 0 'INVALID' if percentage $<$ 0 or percentage $>$ 100
/*battery_status*	*topic*	Topic of the message, i.e. /*battery_status*
	id	The ID of the corresponding percentage message
	status	'0' if status is 0 '1' if status is 1 '2' if status is 2 '3' if status is 3 'INVALID' if status is not 0, 1, 2, or 3
	status_change	'True' if and only if the corresponding field is 'true'
/*SetLED*	*request*	'True' if and only if the message is a service request for /*SetLED*
	req_id	ID of the corresponding /*battery_status* message
	req_status	Status of the corresponding /*battery_status* message
	response	'True' if and only if the message is a service response for /*SetLED*

Table 2: Predicates construction based upon JSON messages sent to Oracle.

indicate message type. For /*SetLED* service request messages, predicates *req_id* and *req_status* store additional information used in Properties 2a and 2b to verify legitimate requests triggered by corresponding /*battery_status* messages.

A significant concern associated with the implementation of runtime verification is its potential adverse effect on overall system performance. To gauge the extent of overhead induced by monitoring services, we modified the client node to measure the time elapsed between dispatching a /*SetLED* service request and receiving the corresponding response. All experiments shown in Figures 5, 6, and 7 were conducted over 10 runs, with the results averaged. Each run was terminated once the battery percentage reached zero. Frequencies for the Battery, Battery Supervisor, and LED Panel were set to 25, 10, and 35 Hertz respectively. The mean and standard deviation are reported for each experiment. As illustrated in Figure 6, the overhead incurred by monitoring /*SetLED* without ordering appears negligible, but the introduction of ordering substantially delays the process, particularly noticeable during the last status change. To approximate the overhead attributed to the ordering mechanism, we adapted the

monitor code to record the time difference between message buffering and transmission to the Oracle (reported in Figure 7). The results depicted in Figure 5 exhibit a similar trend, wherein the release time deviates from the buffering time until the second service request, after which the waiting time stabilises at a minimised level. This is because after the initial service request and response at the beginning of the execution, messages keep accumulating in the buffers since the */SetLED* service buffer remains empty until the second service request is triggered by battery percentage reaching 40%. This threshold is reached when the message with ID 60 is sent. With a service request in the corresponding buffer, the algorithm proceeds to release messages from buffers in the order of publication. Since the next service request is triggered when the battery percentage reaches 30%, i.e. message ID 70, the service buffer will not be empty for long. This explains the minimal waiting time between buffering and transmission after the 60th message. Once the battery percentage reaches 0, the execution is interrupted. This allows the remaining messages in the buffers to be released in the order of publication without requiring all buffers to be nonempty. Moreover, as shown in Figure 7, the waiting time for */SetLED* service requests escalates over time, whereas response messages appear to be promptly released from the buffer. This is because at the time that a service request is buffered, a number of messages have accumulated in the other buffers which need to be released before the service request. In contrast, since the service response is buffered shortly after the corresponding service request is released, there are only a few messages buffered in between which need to be released before the service response.

With regards to verification accuracy, monitoring with ordering consistently yielded accurate verdicts without any incorrect assessments. Conversely, in cases where monitoring excluded ordering, */battery_status* messages often reached the monitor prior to their corresponding */battery_percentage* or */input_accepted* signals. Similarly, service request and response messages consistently preceded the corresponding */status_change* signals. These out-of-order message arrivals led to frequent false negative verdicts in each run. Hence, in weighing the trade-off between performance and accuracy, it becomes evident that monitoring with ordering is most suitable for safety-critical systems where time sensitivity is not paramount. On the other hand, monitoring without ordering may offer enhanced performance at the expense of accuracy, making it more suitable for scenarios where real-time constraints are less stringent.

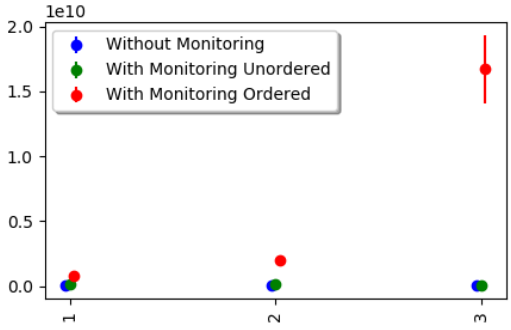


Figure 6: Mean and standard deviation of time (nanoseconds) taken for */SetLED* service request to receive a response (reported on three service calls). Data shown for 10 runs without monitoring, with monitoring excluding ordering, and with monitoring including ordering.

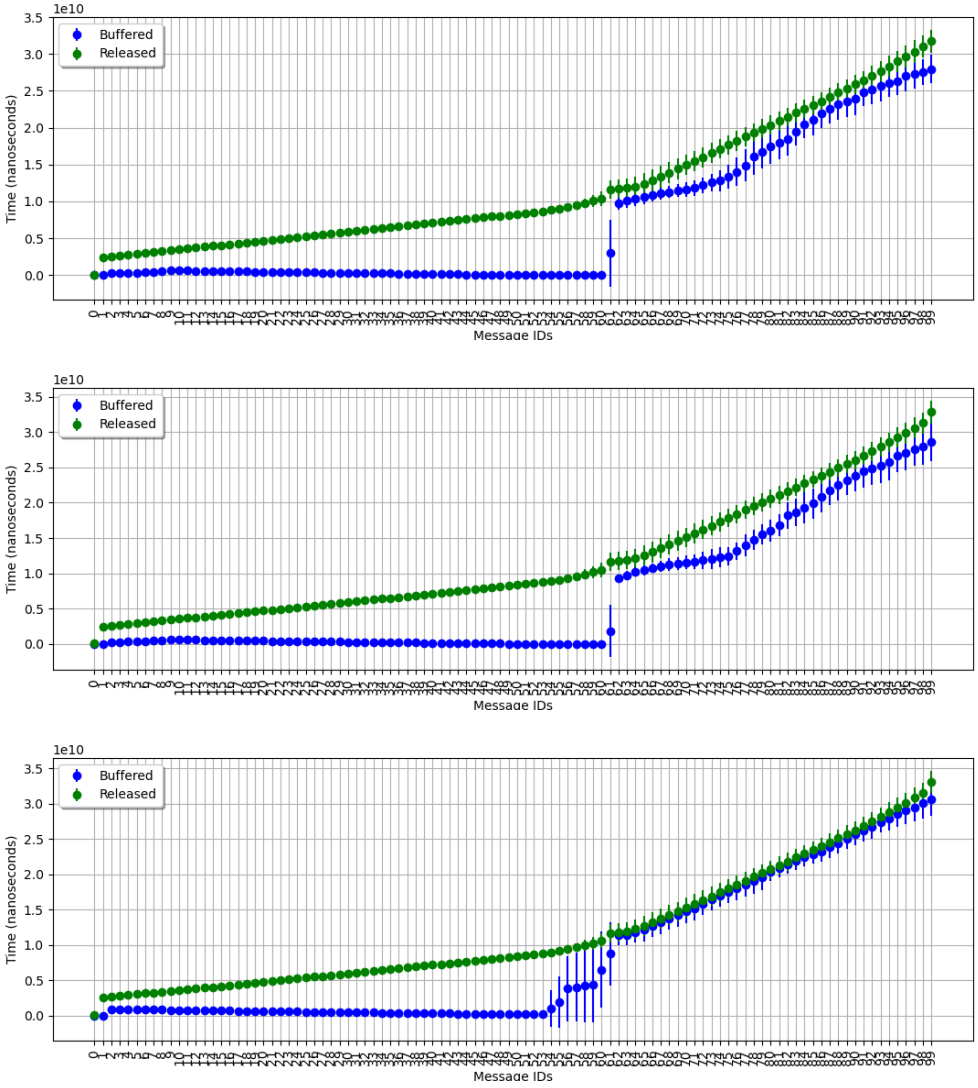


Figure 5: Mean and standard deviation of time (nanoseconds) taken for monitor with ordering to buffer and release messages on */battery_percentage* (top), */input_accepted* (middle), ans */battery_status* (bottom) topics since publication. Data shown for 10 runs.

6 Related Work

In this section, we discuss the most recent approaches to RV of ROS and position them in relation to ROSMonitoring 2.0.

ROSRV [19] shares similarities with our framework in achieving automatic RV of applications in ROS. Both tools utilise monitors not only to passively observe but also to intercept and handle incorrect behaviours in message exchanges among nodes. The main difference lies in how they integrate the monitor into the system. ROSRV replaces the ROS Master node with RVMaster, directing all node communication through it and establishing peer-to-peer communication with the monitor as the intermediary. In contrast, ROSMonitoring adds the monitor through node instrumentation without altering

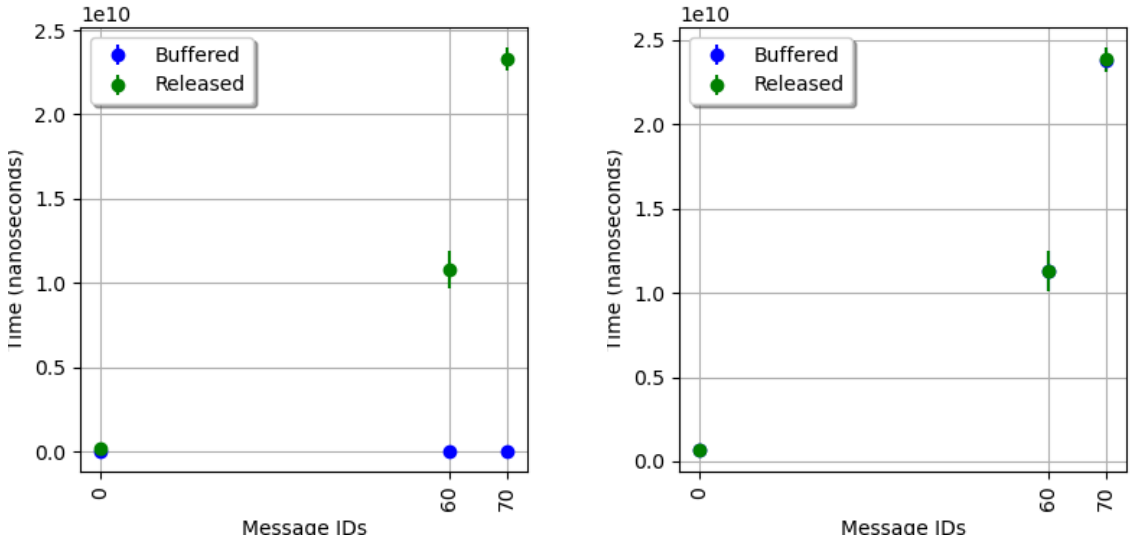


Figure 7: Mean and standard deviation of time (nanoseconds) taken for monitor with ordering to buffer and release */SetLED* request (left) and response (right) messages since generation. Time for buffering and releasing overlap for response messages in the right plot. Data shown for 10 runs.

the ROS Master node. Additionally, the new and extended version ROSMonitoring 2.0 presented in this paper further differentiates the two frameworks, as ROSRV does not support the verification of services or the customisation of the order of topics. HAROS [30] is a framework dedicated to ensuring the quality of ROS systems. Although HAROS primarily focuses on static analysis, it possesses the capability to generate runtime monitors and conduct property-based testing. Differently from ROSMonitoring 2.0, HAROS does not support ROS2 (not even partially). Furthermore, one notable distinction between ROSMonitoring 2.0 and HAROS is that our specifications do not incorporate ROS-specific details, and the process for generating monitors does not rely on understanding the topology of the ROS graph. DeROS [4] is a domain-specific language and monitoring system tailored for ROS. Although DeRoS's language incorporates explicit topic notions, it lacks native support for reordering or service handling. Moreover, it is exclusively compatible with ROS. An extension of Ogma supports the runtime monitoring of ROS2 applications [27]. It outlines a formal approach to generate runtime monitors for autonomous robots from structured natural language requirements, expressed in FRET [17]. This extension integrates FRET and Copilot [26] via Ogma to translate requirements into temporal logic formulas and generate monitor specifications. Unlike ROSMonitoring 2.0, which focuses on monitoring and potentially filtering topics and services, this extension is limited to detecting and reporting violations only. Nonetheless, [17] provides a lightweight verification solution for complex ROS2 applications, ensuring safe operation. MARVer [13] is an integrated runtime verification system designed to ensure the safety and security of industrial robotic systems. It offers a lightweight yet effective approach to monitoring the behaviour of robotic systems in real-time, enabling the detection of security attacks and potential safety hazards. By being based on ROSMonitoring, MARVer can leverage the new features introduced in this work.

The work in [7] introduces TeSSLa-ROS-Bridge, a RV system designed for robotic systems built in ROS. Unlike other RV approaches, TeSSLa-ROS-Bridge utilises Stream-based Runtime Verification

(SRV), which specifies stream transformations to detect errors and control system behaviour (currently supported in ROSMonitoring as well). The system allows TeSSLa monitors to run alongside ROS-based robotic systems, enabling real-time monitoring. Compared to ROSMonitoring, which focuses on monitoring and filtering topics and services, TeSSLa-ROS-Bridge offers a different approach by leveraging stream-based runtime verification to monitor and control robotic systems. RTAMT is an online monitoring library for Signal Temporal Logic (STL), supporting both discrete and dense-time interpretations. In [25], RTAMT4ROS is introduced, integrating RTAMT with ROS. This integration enables specification-based RV methods in robotic applications, enhancing safety assurance in complex autonomous systems. However, similar to other RV frameworks, RTAMT4ROS solely supports topics monitoring and relies exclusively on ROS. Alternative runtime monitoring systems such as Lola [12], Java-MOP [10], detectEr [3], Hydra [29], DejaVu [18], LamaConv [20], and TraceContract [6] could potentially be applied to robotics applications. However, these systems are not explicitly designed for ROS, and integrating them into ROS would require additional development effort and potentially incur runtime costs.

7 Conclusions and Future Work

This paper introduces ROSMonitoring 2.0, an extension of the ROSMonitoring framework designed to enable the Runtime Verification of robotic applications developed in ROS. ROSMonitoring 2.0 expands upon its predecessor by facilitating the verification of services, in addition to topics, and by accommodating ordered topics, rather than solely unordered ones. Notably, the new features of ROSMonitoring 2.0 do not necessitate changes to the compositional and formalism-agnostic aspects of ROSMonitoring; only the synthesis of ROS monitors is adjusted. This approach not only leverages all existing features in ROSMonitoring but also ensures full backward compatibility with existing ROS applications based on ROSMonitoring. Furthermore, the proposed ordering algorithm and service interception process hold applicability beyond the scope of ROSMonitoring 2.0, potentially benefiting other systems as well.

It is also worth noting that the introduction of ordering messages according to the order they are published does not mutually exclude the standard ROSMonitoring topic checking, based on the order the messages are received. In this sense, in ROSMonitoring 2.0 it is also possible to combine both ordering features to monitor both the publish and receive order of messages. This becomes relevant in scenarios where it is necessary to identify which exact node is the faulty one, rather than being only interested in checking the presence of a property violation (which could be relevant in other scenarios instead).

As a future direction, we aim to formally verify that our case study is deadlock-free and establish design principles for ensuring deadlock-freeness. Additionally, threading will be explored as an alternative solution to address potential deadlock issues. We also plan to extend our research to additional case studies in the robotics domain, focusing on complex systems involving multiple services with strong interdependencies across services, topics, and interfaces.

Moreover, we intend to expand the framework to support ROS actions. ROS actions allow robots to execute complex, asynchronous tasks by setting goals, providing feedback, and retrieving results, thus facilitating modular and scalable behaviours for navigation, manipulation, and planning. Although actions are asynchronous and non-blocking, which reduces the monitoring burden compared to services, they introduce challenges in tracking progress against runtime goals.

In parallel, we plan to enhance our message ordering algorithm by introducing timeouts, preventing messages from waiting indefinitely, particularly in unreliable communication scenarios. However, careful consideration is required, as timeouts may disrupt the message order when delays occur, rather than

message loss. This could be especially important for scenarios with strict timing requirements, where a balance must be struck between message order and timely delivery.

Furthermore, a comprehensive performance evaluation of ROSMonitoring 2.0 will be a critical focus. We aim to assess key metrics such as execution time, resource usage, and system overhead, benchmarking our approach against existing alternatives. Such an evaluation will provide deeper insights into the framework's efficiency and scalability and guide further optimisations.

Lastly, our goal is to port all the features presented in this paper to ROS2, which currently only supports service monitoring and lacks message reordering functionality. This migration will proceed once additional evaluations and testing have been completed on the ROS1 version of ROSMonitoring 2.0.

References

[1] *MBZIRC 2020: The Mohamed Bin Zayed International Robotics Challenge*. `https://rsl.ethz.ch/research/challenges-competitions/mbzirc2020.html`. Accessed on April 8th, 2024.

[2] *ROS: Robot Operating System*. `https://www.ros.org/`. Accessed on April 8th, 2024.

[3] Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza & Anna Ingólfsdóttir (2024): *A Monitoring Tool for Linear-Time μHML*. *Sci. Comput. Program.* 232, p. 103031. Available at `https://doi.org/10.1016/j.scico.2023.103031`.

[4] Sorin Adam, Morten Larsen, Kjeld Jensen & Ulrik Pagh Schultz (2014): *Towards Rule-Based Dynamic Safety Monitoring for Mobile Robots*. In Davide Brugali, Jan F. Broenink, Torsten Kroeger & Bruce A. MacDonald, editors: *Proc. 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, *Lecture Notes in Computer Science* 8810, Springer, pp. 207–218. Available at `https://doi.org/10.1007/978-3-319-11900-7_18`.

[5] Davide Ancona, Luca Franceschini, Angelo Ferrando & Viviana Mascardi (2021): *RML: Theory and practice of a domain specific language for runtime verification*. *Sci. Comput. Program.* 205, p. 102610, doi:10.1016/J.SCICO.2021.102610.

[6] Howard Barringer & Klaus Havelund (2011): *TraceContract: A Scala DSL for Trace Analysis*. In Michael J. Butler & Wolfram Schulte, editors: *Proc. 17th International Symposium on Formal Methods (FM)*, *Lecture Notes in Computer Science* 6664, Springer, pp. 57–72. Available at `https://doi.org/10.1007/978-3-642-21437-0_7`.

[7] Marian Johannes Begemann, Hannes Kallwies, Martin Leucker & Malte Schmitz (2023): *TeSSLa-ROS-Bridge - Runtime Verification of Robotic Systems*. In Erika Ábrahám, Clemens Dubslaff & Silvia Lizeth Tapia Tarifa, editors: *Proc. 20th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, *Lecture Notes in Computer Science* 14446, Springer, pp. 388–398. Available at `https://doi.org/10.1007/978-3-031-47963-2_23`.

[8] Guido Breitenhuber (2020): *Towards Application Level Testing of ROS Networks*. In: *Proc. Fourth IEEE International Conference on Robotic Computing (IRC)*, IEEE, pp. 436–442. Available at `https://doi.org/10.1109/IRC.2020.00081`.

[9] Maria A. S. Brito, Simone R. S. Souza & Paulo S. L. Souza (2022): *Integration testing for robotic systems*. *Softw. Qual. J.* 30(1), pp. 3–35. Available at `https://doi.org/10.1007/s11219-020-09535-w`.

[10] Feng Chen & Grigore Rosu (2005): *Java-MOP: A Monitoring Oriented Programming Environment for Java*. In Nicolas Halbwachs & Lenore D. Zuck, editors: *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, *Lecture Notes in Computer Science* 3440, Springer, pp. 546–550. Available at `https://doi.org/10.1007/978-3-540-31980-1_36`.

[11] Edmund M. Clarke, Orna Grumberg & Doron A. Peled (2001): *Model checking*. MIT Press, doi:10.1016/B978-044450813-3/50026-6. Available at `http://books.google.de/books?id=Nmc4wEaLXFEC`.

[12] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra & Zohar Manna (2005): *LOLA: Runtime Monitoring of Synchronous Systems*. In: *Proc. 12th International Symposium on Temporal Representation and Reasoning (TIME)*, IEEE Computer Society, pp. 166–174. Available at `https://doi.org/10.1109/TIME.2005.26`.

[13] Elif Degirmenci, Yunus Sabri Kirca, Özlem Örnek, Mert Bulut, Serhat Kahraman, Metin Ozkan & Ahmet Yazici (2023): *Developing an Integrated Runtime Verification for Safety and Security of Industrial Robot Inspection System*. In Fumiya Iida, Perla Maiolino, Arsen Abdulali & Mingfeng Wang, editors: *Proc. 24th Annual Conference on Towards Autonomous Robotic Systems (TAROS)*, *Lecture Notes in Computer Science* 14136, Springer, pp. 126–137. Available at `https://doi.org/10.1007/978-3-031-43360-3_11`.

[14] Ankush Desai, Tommaso Dreossi & Sanjit A. Seshia (2017): *Combining Model Checking and Runtime Verification for Safe Robotics*. In Shuvendu K. Lahiri & Giles Reger, editors: *Proc. 17th International Conference on Runtime Verification (RV)*, *Lecture Notes in Computer Science* 10548, Springer, pp. 172–189. Available at `https://doi.org/10.1007/978-3-319-67531-2_11`.

[15] Angelo Ferrando, Rafael C. Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini & Viviana Mascardi (2020): *ROSMonitoring: A Runtime Verification Framework for ROS*. In Abdelkhalick Mohammad, Xin Dong & Matteo Russo, editors: *Proc. 21st Annual Conference on Towards Autonomous Robotic Systems (TAROS)*, *Lecture Notes in Computer Science* 12228, Springer, pp. 387–399. Available at `https://doi.org/10.1007/978-3-030-63486-5_40`.

[16] Mohammed Foughali, Bernard Berthomieu, Silvano Dal-Zilio, Félix Ingrand & Anthony Mallet (2016): *Model Checking Real-Time Properties on the Functional Layer of Autonomous Robots*. In Kazuhiro Ogata, Mark Lawford & Shaoying Liu, editors: *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings*, *Lecture Notes in Computer Science* 10009, pp. 383–399. Available at `https://doi.org/10.1007/978-3-319-47846-3_24`.

[17] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, Julian Rhein, Johann Schumann & Nija Shi (2020): *Formal Requirements Elicitation with FRET*. In Mehrdad Sabetzadeh, Andreas Vogelsang, Sallam Abualhaija, Markus Borg, Fabiano Dalpiaz, Maya Daneva, Nelly Condori-Fernández, Xavier Franch, Davide Fucci, Vincenzo Gervasi, Eduard C. Groen, Renata S. S. Guizzardi, Andrea Herrmann, Jennifer Horkoff, Luisa Mich, Anna Perini & Angelo Susi, editors: *Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020)*, *CEUR Workshop Proceedings* 2584, CEUR-WS.org. Available at `https://ceur-ws.org/Vol-2584/PT-paper4.pdf`.

[18] Klaus Havelund, Doron Peled & Dogan Ulus (2018): *DejaVu: A Monitoring Tool for First-Order Temporal Logic*. In: *Proc. 3rd Workshop on Monitoring and Testing of Cyber-Physical Systems, MT@CPSWeek 2018*, IEEE, pp. 12–13. Available at `https://doi.org/10.1109/MT-CPS.2018.00013`.

[19] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon M. Moore, Qingzhou Luo, Aravind Sundaresan & Grigore Rosu (2014): *ROSRV: Runtime Verification for Robots*. In Borzoo Bonakdarpour & Scott A. Smolka, editors: *Proc. 5th International Conference on Runtime Verification (RV)*, *Lecture Notes in Computer Science* 8734, Springer, pp. 247–254, doi:10.1007/978-3-319-11164-3_20.

[20] Institute for Software Engineering and Programming Languages: *LamaConv - Logics and Automata Converter Library*. `www.isp.uni-luebeck.de/lamaconv`.

[21] Gert Kanter & Jüri Vain (2020): *Model-based testing of autonomous robots using TestIt*. *J. Reliab. Intell. Environ.* 6(1), pp. 15–30. Available at `https://doi.org/10.1007/s40860-019-00095-w`.

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

[23] Martin Leucker & Christian Schallhart (2009): *A Brief Account of Runtime Verification*. *J. Log. Algebraic Methods Program.* 78(5), pp. 293–303. Available at https://doi.org/10.1016/j.jlap.2008.08.004.

[24] Oded Maler & Dejan Nickovic (2004): *Monitoring Temporal Properties of Continuous Signals*. In Yassine Lakhnech & Sergio Yovine, editors: *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* 3253, Springer, pp. 152–166, doi:10.1007/978-3-540-30206-3_12.

[25] Dejan Nickovic & Tomoya Yamaguchi (2020): *RTAMT: Online Robustness Monitors from STL*. In Dang Van Hung & Oleg Sokolsky, editors: *Proc. 18th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, *Lecture Notes in Computer Science* 12302, Springer, pp. 564–571. Available at https://doi.org/10.1007/978-3-030-59152-6_34.

[26] Ivan Perez & Alwyn Goodloe (2020): *Copilot 3*. Technical Report, doi:10.13140/RG.2.2.35163.80163.

[27] Ivan Perez, Anastasia Mavridou, Thomas Pressburger, Alexander Will & Patrick J. Martin (2022): *Monitoring ROS2: from Requirements to Autonomous Robots*. In Matt Luckcuck & Marie Farrell, editors: *Proceedings Fourth International Workshop on Formal Methods for Autonomous Systems (FMAS) and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE), FMAS/ASYDE@SEFM 2022*, *EPTCS* 371, pp. 208–216, doi:10.4204/EPTCS.371.15.

[28] Amir Pnueli (1977): *The Temporal Logic of Programs*. In: *Proc. 18th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, pp. 46–57. Available at https://doi.org/10.1109/SFCS.1977.32.

[29] Martin Raszyk, David A. Basin & Dmitriy Traytel (2020): *Multi-head Monitoring of Metric Dynamic Logic*. In Dang Van Hung & Oleg Sokolsky, editors: *Proc. 18th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, *Lecture Notes in Computer Science* 12302, Springer, pp. 233–250. Available at https://doi.org/10.1007/978-3-030-59152-6_13.

[30] André Santos, Alcino Cunha, Nuno Macedo & Cláudio Lourenço (2016): *A framework for quality assessment of ROS repositories*. In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp. 4491–4496. Available at https://doi.org/10.1109/IROS.2016.7759661.

[31] A. Prasad Sistla & Edmund M. Clarke (1985): *The Complexity of Propositional Linear Temporal Logics*. *J. ACM* 32(3), pp. 733–749. Available at https://doi.org/10.1145/3828.3837.

Verification of Behavior Trees with Contingency Monitors

Serena S. Serbinowska
0000-0002-9259-1586
Vanderbilt University
Nashville TN, USA
serena.serbinowska@vanderbilt.edu

Nicholas Potteiger
0009-0005-0406-0355
Vanderbilt University
Nashville TN, USA
nicholas.potteiger@vanderbilt.edu

Anne M. Tumlin
0009-0000-1635-8793
Vanderbilt University
Nashville TN, USA
anne.m.tumlin@vanderbilt.edu

Taylor T. Johnson
0000-0001-8021-9923
Vanderbilt University
Nashville TN, USA
taylor.johnson@vanderbilt.edu

Behavior Trees (*BT*s) are high level controllers that have found use in a wide range of robotics tasks. As they grow in popularity and usage, it is crucial to ensure that the appropriate tools and methods are available for ensuring they work as intended. To that end, we created a new methodology by which to create Runtime Monitors for *BT*s. These monitors can be used by the *BT* to correct when undesirable behavior is detected and are capable of handling LTL specifications. We demonstrate that in terms of runtime, the generated monitors are on par with monitors generated by existing tools and highlight certain features that make our method more desirable in various situations. We note that our method allows for our monitors to be swapped out with alternate monitors with fairly minimal user effort. Finally, our method ties in with our existing tool, BehaVerify, allowing for the verification of *BT*s with monitors.

1 Introduction

A Behavior Tree (*BT*) is a high-level tree-structured controller with leaf nodes that interact with the environment and interior nodes that control which branches of the tree are executed. The tree-structure means that *BT*s are often more intuitive than equivalent finite state machines, but are also powerful tools capable of being used in many environments. Furthermore, the inherently recursive nature of tree structures allows for adaptability, modularity, and reuse.

*BT*s originated in video games and were used for Non Playable Characters (NPCs). NPCs are, in essence, virtual agents in a digital environment. As time progressed, NPCs needed to respond to more complex environments. The video game industry responded to this by creating *BT*s: designer friendly controllers for complex systems. In light of this, it is unsurprising that the controllers subsequently made the jump to areas such as robotics and drone control. Bipedal locomotion for robots [18], vision measurement systems of road users [26], and swarms of agents have all utilized *BT*s. A recent survey [21] provides even more examples of *BT*s in action.

It is clear that *BT*s are continuing to grow in popularity and usage. As they expand into new domains, especially real-world safety-critical domains, it is imperative to be able to provide guarantees about their correctness. Two methods for providing such guarantees are runtime monitoring and design time verification. Runtime monitoring can be used to alert the *BT* if there is danger of a violation occurring, allowing the tree to self-correct, while design time verification can be used to ensure the model is correct.

At present, tools already exist for the creation of runtime monitors, such as NASA's Copilot [24] and NuRV [7], though they are not designed for *BT*s specifically. However, it is important that the tools not only exist, but be compatible with the *BT*, and that the *BT* reacts correctly to these tools. After all, if the

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 56–72, doi:10.4204/EPTCS.411.4

monitor correctly indicates a dangerous situation is occurring but the *BT* ignores this warning, then the danger has not been averted.

The primary contributions of this work are the following:

1. We provide a formal definition for *BT*s with Monitors (*BTM*).
2. We expand the Domain Specific Language (DSL) of BehaVerify [27], allowing it to describe *BTM*s. BehaVerify was originally created for design time verification on *BT*s.
3. We present software for converting Linear Temporal Logic (LTL) specifications written in the DSL into input for the existing tool LTL2BA [16].
   (a) We then translate the output of LTL2BA back into the DSL. This enables BehaVerify to generate nuXmv [6] models, allowing us to use Design Time Verification to confirm that the *BTM* works as intended.
   (b) Additionally, we can translate the output of LTL2BA directly into a C or Python monitor, for use with code generated by BehaVerify.
   (c) Furthermore, we compare the generated C monitors to monitors generated by Copilot [24] and demonstrate our monitors are on par in terms of performance and offer certain improvements in terms of correctness.

2 Related Work

There is a broad body of work utilizing behavior trees for planning purposes in robotic systems [5, 9–11, 22, 31, 35, 36], illustrating their broad usage in safety-critical systems. There are several practical implementations of *BT*s (such as PyTrees [32] and its Robotic Operating System (ROS) extension PyTreesRos, BehaviorTree.cpp [1], and Unreal Engine [13]). Each of these feature a Blackboard (shared memory between nodes). For a variety of practical reasons, our tool targets the implementation of *BT*s presented in PyTrees, though we hope to target BehaviorTree.cpp in the future as well.

There are several existing works that develop and apply formal verification for *BT*s. There are several tools for model verification of *BT*s: [4], BehaVerify [27], BTCompiler, ArcadeBT [19], and MoVe4BT [23]. Prior to this work, however, none of these tools supported Runtime Verification of *BT*s. [8] does runtime verification for a fragment of Timed Propositional Temporal Logic (TPTL) for *BT*s, but failed to configure our examples to work with it.

The authors in [25] utilized the runtime verification framework NASA Copilot [24] to ensure that a flying aircraft maintains an airspeed above a threshold using natural language and Past Linear Temporal Logic (PLTL). Similarly, stream runtime verification (SRV) monitors were generated using HLola [17] in [34] to seamlessly integrate with a UAV hybrid navigation controller for post decision making and online remediation actions. Furthermore, the authors in [33] develop an architecture that allows for construction of runtime monitors that can be integrated into an Urban Air Mobility (UAM) System. They demonstrate that runtime monitors are built using NASA OGMA [25] and NASA FRET [25] for battery monitoring of a UAV simulated in Microsoft AirSim [30]. Runtime monitoring instrumentation frameworks for ROS specifically also have been developed [14]. Our approach generally differs from these works because we are interested in creating monitors for *BT*s specifically. However, our approach also clearly has overlap with several of these methods; we too seek to enable remediation actions for the models being monitored. Furthermore, while we create our own monitors, our general framework is compatible with alternative monitors. In light of this, we compare our monitors to those created by Copilot in Section 5.

Another related aspect is that of the Simplex architecture [2, 28, 29]. The Simplex architecture describes control switching logic swaps between multiple controllers depending on the current state of the system. Our work differs from this approach in two main methods: first our approach is focused explicitly on *BT*s, and second our approach does not utilize multiple controllers. Instead, we utilize the structure of *BT*s to integrate the 'switch' into the *BT* itself.

The work that is most closely related to ours is [8]. This work is about *BT*s equipped with runtime monitors based on Timed Propositional Temporal Logic (TPTL) and provides a formal definition of the setup. However, unlike our work, the monitors are not meant to be interacted with; the *BT* has no way of reacting to a violation. In terms of generating monitors by transforming temporal logic specifications to automata, our approach is similar to that of ltl2mon and LamaConv [3, 12, 15], but differs in that we consider *BT*s. We further differentiate our work through the design time verification aspect. Our method allows us to prove that a *BT* equipped with a monitor and its contingency response for detected violations is guaranteed to satisfy a specification.

3 Preliminaries

This section provides a formal definition for Linear Temporal Logic and Buchi Automata. This is followed by an intuitive overview of *BT*s and a formal definition.

3.1 Linear Temporal Logic

A Linear Temporal Logic (LTL) formula is evaluated on a trace. A trace is a sequence of states $Tr \triangleq [n_0,n_1,...]$. Here n_0 is the state at time 0, n_1 is the state at time 1, etc. The grammar of an LTL formula is presented in Grammar 1.

$\langle \text{LTL} \rangle$	::=	$\langle \text{a} \rangle$	#First Order Logic Formula
	|	$\neg\langle \text{LTL} \rangle \mid \langle \text{LTL} \rangle \vee \langle \text{LTL} \rangle$	#Minimal Boolean operators
	|	$\bigcirc(\langle \text{LTL} \rangle) \mid (\langle \text{LTL} \rangle) \mathscr{U} (\langle \text{LTL} \rangle)$	#Temporal operators next and until

Grammar 1: Minimal LTL Grammar.

We assume the reader is familiar with Boolean logic, so we will not describe them here. $\bigcirc(\phi)$ is true at time t if ϕ is true at time $t+1$. $\phi_1 \mathscr{U} \phi_2$ is true at time t if $\exists t''$ such that $t \leq t''$ and ϕ_2 is true at t'' and $\forall t'$ such that $t \leq t' < t''$, ϕ_1 is true at t'.

In addition to the grammar presented in Grammar 1, we also utilize $\square(\phi)$ (globally) and $\Diamond(\phi)$ (finally). These do not increase the expressiveness of LTL, but make writing formulas easier. $\square(\phi)$ is true at time t if $\forall t'$ such that $t \leq t'$, ϕ is true at time t'. $\Diamond(\phi)$ is true at time t if $\exists t'$ such that $t \leq t'$, ϕ is true at time t'.

Finally, we say that ϕ is true for the entire trace if it is true at time 0. Notationally, we will write $Tr \vDash \phi$ to mean that ϕ is true for the entire trace Tr, $Tr_{[i,j]} \vDash \phi$ if we are looking at the segment of the trace $[n_i,n_{i+1},...n_j]$, and utilize $\nvDash$ in the same way but to mean not true.

3.2 Buchi Automata

A Buchi Automaton (*BA*) is a tuple (Q,Σ,Δ,q_0,F).

1. Q is a finite set representing the states *BA* can be in.
2. Σ is a finite set representing the possible inputs.

3. Δ is a function from $Q \times \Sigma \mapsto 2^Q$. Here 2^Q is the power set of Q. This function describes the nondeterministic transitions available from a given state-input combination.
4. q_0 is an element of Q. This is the initial state.
5. F is a finite set such that $F \subseteq Q$. This is the set of accepting states.

Then, *BA* accepts a given sequence of inputs $[a_0,a_1,...]$ if and only if there exists a sequence $[q_0,q_1,...]$ such that

1. $\forall j \in \mathbb{Z}$ s.t. $j \geq 0, q_{j+1} \in \Delta(q_j,a_j)$
2. $\forall j \in \mathbb{Z}, \exists k \in \mathbb{Z}$ s.t. $j < k \wedge a_k \in F$

Thus we accept if we begin in the initial state, take valid transitions, and enter accepting states an infinite number of times. We accept if any such trace is possible; thus a single trace *can* be used to prove that the input is accepted but it *cannot* be used to prove that the input is *not* accepted.

3.3 Behavior Tree Overview

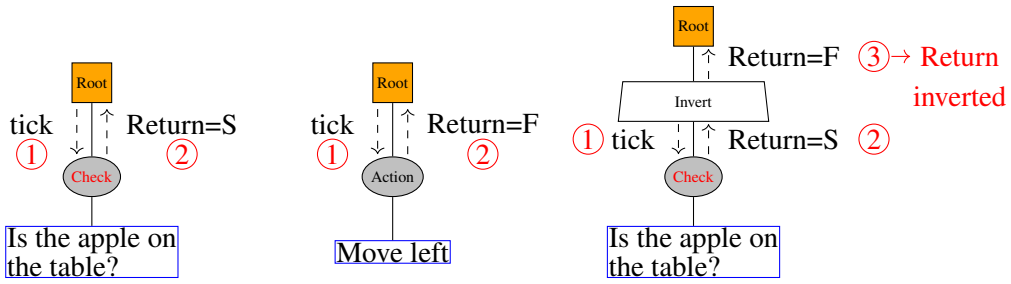
*BT*s are rooted trees with parent-child relationships. Each node has one parent, except the root which has no parent. When executing, a *BT* starts from the root and follows a Depth First Traversal. Nodes can change this traversal order and leave certain branches unexplored based on what their children return. This process is started by an external activation signal called a *tick*. In practice, trees are often structured recursively and parents propagate this signal to their children, but for this paper a *tick* will only be used to refer to the root receiving the external signal. Note that *BT*s are inactive until they receive a tick. In the interest of conciseness, we will omit these periods of inactivity from various diagrams. We assume that a tick will only arrive while the tree in inactive.

Each node is always in one of four states: Success (*S*), Failure (*F*), Running (*R*), or Invalid (*I*). We will use *active* and *executing* to describe where we are in the execution of the tree. When a new tick arrives, each node is set to *I* and the root becomes both active and executing. Until the root finishes executing, exactly one node will be active at all times but more than one node can be executing. A node that is executing is similar to a function that has been called but has not yet turned. A node that is active is similar to a function that is currently being stepped through. When a node finishes executing it returns *S*, *R*, or *F*.

In addition to ticks, we use *timestep* or *t* to track each time the active node changes. Both tick and timestep will be enumerated sequentially starting from 1. Refer to Figure 2 for an example.

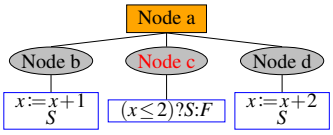
Nodes can be grouped into three categories: leaf, decorator, and composite.

Leaf Nodes Leaf nodes do not have children. It is common to categorize leaf nodes as checks/guards (*e.g.* at boundary?) and actions (*e.g.* go forward). Checks evaluate a boolean condition and return *S* if true and *F* otherwise. Consider Subfigure (a) of Figure 1; if there is an apple on the table, the check will return *S*. If there is not, *F* will be returned. Either way, the status will be returned to the root which will proceed accordingly. It is important to note that checks *only* check a condition and return the appropriate status; they do not set variables or take any sort of action.. By contrast, actions can execute actions, for lack of a better word. For instance, in Subfigure (b) of Figure 1, the action executes the action of moving left. It is important to note that actions are also not restricted in what status they return. While Subfigure (b) of Figure 1 shows *F* being returned, it would be valid to create a version of this action that always return *S*, or it could return *R* because it hasn't finished, or it could return *F* based on some sort of conditional logic.



(a) Check Node (**Leaf**) (b) Action Node (**Leaf**) (c) Inverter Node (**Decorator**)

Figure 1: Example Leaf and Decorator Nodes.



Tick	1	1	1	1	1	1	1	2	2	2	2	2
t	1	2	3	4	5	6	7	8	9	10	11	12
Active	a	b	a	c	a	d	a	a	b	a	c	a
Returns	-	*S*	-	*S*	-	*S*	*S*	-	*S*	-	*F*	*F*
x	0	1	1	1	1	3	3	3	4	4	4	4

Figure 2: A *BT* consisting of a sequence node (a) with two actions (b, d) and a check (c). We use the ternary operator $i?j:k$ to mean if i then j else k. Tick indicates the number of times the tree has been ticked. At each timestep t, the variable x is updated based on the active node. If a node is finished, then it returns one of S, F, or R.

Decorator Nodes Decorator nodes 'decorate' their children, allowing for easy adjustments to be made. A decorator node will always have exactly one child, but that child can have children of its own. An inverter (decorator that swaps S and F) is in Subfigure (c) of Figure 1.

Composite Nodes Composite nodes control the traversal of the tree. Changing a composite node will change the conditions under which branches of the tree are activated. The three primary types of composite nodes are selector, sequence, and parallel nodes. The children of composite nodes are ordered and are activated according to the order, which we will treat as being left-to-right, both visually and in our language.

1. Recall our notation of Success (S), Running (R), and Failure (F).
2. **Selector Nodes:** Selector/Fallback nodes activate their children from left-to-right until one of them returns S or R, at which point the selector itself returns S or R. If a child node returns F, the selector activates the next child. Selectors can be used to prioritize behaviors, ordering them from most preferable at the left to least preferable at the right. Another way to think of these nodes as providing a series of fallbacks in case the intended action fails.
3. **Sequence Nodes:** Sequence nodes activate their children in a left-to-right order, requiring each child to return S before moving on to the next. The sequence returns S only if all its children return S. It returns F or R as soon as one of its children does. This is ideal for defining a series of actions that must be performed in a specific order to achieve a goal. Figure 2 shows an example sequence node with two actions and one check. The figure clearly demonstrates the order in which the children become active. Additionally, it showcases how composite nodes control the flow of

logic in a BT; during tick 1, all three children are active at some point. However, during tick 2 node (d) never becomes active because node (c) returned F.

4. **Parallel Nodes:** For the purpose of this paper, we do not consider truly parallel nodes (i.e., nodes that activate all their children simultaneously). Instead, our parallel nodes activate nodes one at a time in a left-to-right order. However, unlike selector and sequence nodes, a parallel node will *always* tick all of its children. Once the last child returns, the parallel node uses a policy that considers what all of the children returned. The two most common policies are Success on All, which requires that all children return S for S to be returned, and Success on One, which requires only one child to return S for S to be returned. This is consistent with [32]. It is possible to define more complex policies, but this is beyond the scope of this paper.

3.4 Formal Definition of Behavior Trees

A Behavior Tree (BT) is a tuple $(S,V,\Sigma_T,\Delta_T,s_0,v_0)$.

1. S is a finite set that describes the state of the tree itself (which node is active, what nodes have returned so far, etc).
2. V is a finite set that describes the state of the Blackboard variables (the persistent memory of the tree).
3. Σ_T is a finite set of all possible 'inputs' to the tree (e.g., environmental factors).
4. Δ_T is a function $S\times V\times\Sigma_T\mapsto S\times V$. This function takes the current state of the tree, the variables, and environmental factors and produces a new tree state and new variable values.
5. s_0 is an element of S that describes the initial state of the tree.
6. v_0 is an element of V that describes the initial state of the Blackboard variables.

This definition does not allow for nondeterminism, though it could be simulated through a 'seed' variable. Furthermore, this definition is slightly too permissive; additional restrictions would need to be placed on Δ_T to enforce the tree structure. We omit these details as we believe that they would complicate the issue without providing additional insight.

For a given sequence of inputs $[a_0,a_1,...]$, the corresponding BT trace is a sequence $[(s_0,v_0),(s_1,v_1),...]$ such that

$$\forall j\in\mathbb{Z}\text{ s.t. }j\geq 0,\Delta_T(s_j,v_j,a_j)=(s_{j+1},v_{j+1})$$

4 Problem Statement and Methodology

While we have defined BTs, we have not defined Behavior Trees with Monitors (BTMs). In this section we will first define BTM, then formally state the problem we are addressing, and finally present our method for addressing the issue.

4.1 Formal Definition of Behavior Trees with Monitors

A BTM is a tuple $(S,V,M,\Sigma_T,\Delta_M,s_0,v_0,m_0)$.

1. S, V, Σ_T, s_0, and v_0 are unchanged from BT.
2. M is a finite set describing the state of the monitor.

3. Δ_M is a function $S \times V \times \Sigma_T \times M \mapsto S \times V \times M$. This function takes the current state of the tree, the variables, environmental factors, and the state of the monitor and produces a new tree state, new variables state, and a new monitor state.

4. m_0 is an element of M and describes the initial state of the monitor.

For a given sequence of inputs $[a_0, a_1, ...]$, the corresponding BTM trace is a sequence $[(s_0, v_0, m_0), (s_1, v_1, m_1), ...]$ such that

$$\forall j \in \mathbb{Z} \text{ s.t. } j \geq 0, \Delta_T(s_j, v_j, a_j, m_j) = (s_{j+1}, v_{j+1}, m_{j+1})$$

4.2 Problem Statement

Our problem statement follows. **Given a BT and a ϕ, create a BTM that monitors ϕ such that BTM is capable of reacting to a violation of ϕ.** To demonstrate the utiltity of this process, we will also consider ϕ_1, a second specification that holds for BTM but not for BT.

To accomplish this goal we modified the Domain Specific Language (DSL) of BehaVerify (outlined below) to allow the use of monitors within BTs. We utilize the tool LTL2BA [16] and specifically the implementation at [1] to translate an LTL formula into a monitor in the form of a BA. Then we create an implementation of that BA for use with the BT. Finally, to verify that BTM satisfies the property, we also convert the output of LTL2BA into an implementation of the monitor within the DSL and utilize BehaVerify to create a nuXmv model that proves ϕ_1 is true for BTM but false for BT.

4.3 BehaVerify

BehaVerify uses a Domain Specific Language (DSL) that allows the user to specify a BT. As the DSL itself [2] is complex, we will provide an overview here.

The user defines a finite set of typed variables that will be used by the BT. The user also defines a finite set of leaf nodes. Each leaf node is a finite sequence of statements, exactly one of which is a return statement while the rest are variable statements. A variable statement consists of the variable whose value is being updated and a sequence of 'if statements' that determines the new value. Nondeterminism is allowed in variable statements. The return statement is similar, but is used to determined what status the node will return. Note that the node does *not* stop execution when the return statement is executed; it is only used to determine the return status, not to 'return' from the node. Finally, the user creates a finite tree consisting of composite (selector, sequence, and parallel), decorator (inverter and X_is_Y), and user-defined leaf nodes.

Monitors We add special syntax to our DSL for the creation and use of monitors. The user provides an LTL specification that is to be monitored. Furthermore, the user specifies where in the tree the monitor should be used, and how the tree should react to the possible outputs of the monitor. We describe the details of transforming the monitor in Subsection 4.4.

We present two pipelines for making monitors. The first is for generating a Python implementation of a BT and either Python or C code for the monitor(s). The second pipeline is for generating a nuXmv model of a BT and its monitor(s).

[1] https://github.com/utwente-fmt/ltl2ba

[2] https://github.com/verivital/behaverify/blob/main/metamodel/behaverify.tx

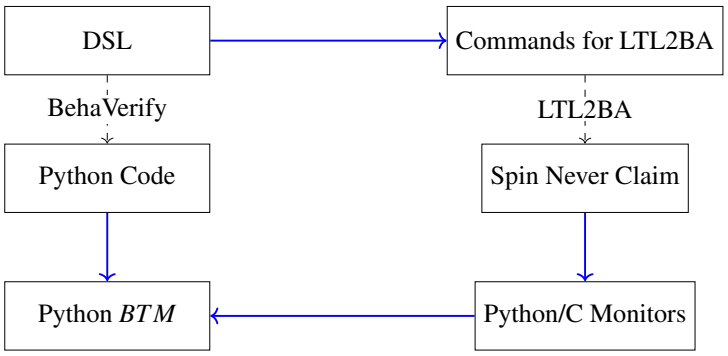


Figure 3: Diagram of how BehaVerify generates Python code for *BT*s with monitors. LTL2BA is a tool for converting an LTL specification to a *BA*. Spin is a model checker, and a Never Claim can be checked using Spin. Solid blue arrows mark new contributions.

4.4 Generating Implementations

We utilize the following process to generate a *BTM* using BehaVerify.

1. The user creates a DSL file specifying the *BT* and any monitors it uses.
2. BehaVerify creates a Python implementation for *BT*, ignoring the monitors.
3. For each monitor in the DSL file, BehaVerify creates a command for use with LTL2BA.
4. For each command, LTL2BA creates a Buchi Automaton (*BA*). The output is in the form of a 'never claim' for use with the Spin [20] model checker.
5. For each *BA*, BehaVerify creates a corresponding Python implementation.
6. The monitors are combined with the generated Python code.

This process can be seen in Figure 3. Below we provide some additional details.

Command Creation LTL2BA commands can contain temporal operators (e.g. 'globally'), boolean operators (e.g. 'and'), boolean constants ('true' and 'false'), and lowercase alphanumeric strings representing boolean variables. Our conversion process prioritizes making the resulting LTL formula as small as possible. Thus the formula $\Box(a \vee b)$ (here $\Box$ means globally) would be converted to $\Box p0$, where $p0$ is boolean predicate representing $a \vee b$.

Monitor We provide a quick refresher on *BA*. For details, see Subsection 3.2. A *BA* is a nondeterministic automaton with transition guards. Thus from a given state, *BA* can transition to any other state provided a transition to that state exists and the associated guard condition is true. Within the formal definition, these guards are encoded into the transition function. Some of the states in the *BA* are 'accepting' states. The *BA* accepts a trace if there exists a sequence where the *BA* is infinitely often in an accepting state.

To mimic this behavior, the monitor takes as input a set of states that the *BA* could be in along with the current model state. The current model state provides all the necessary information to determine if a transition guard is true. This, combined with the possible states, is used to create a new set of possible states. If there are no possible states, then there is no longer any way for the specification to be true, meaning it must be false. If there is a possible state that is an accepting state with a transition to itself and the guard is always true, then the specification is guaranteed to be true. Otherwise, the specification could still prove to be true or false (unknown). The monitor returns both the new set of possible states and the verdict to the user. If a violation occurs, the monitor can be 'reset'. This is important as we want our monitor to be repeatably usable; without a reset it would have to continuously report that a violation occurred.

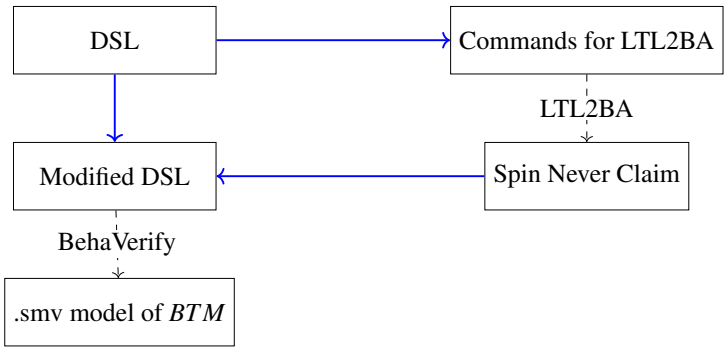


Figure 4: Diagram of how BehaVerify generates .smv files for *BT*s with monitors that can be verified using nuXmv. LTL2BA is a tool for converting an LTL specification to a *BA*. Spin is a model checker, and a Never Claim can be checked using Spin. Solid blue arrows mark new contributions.

4.5 nuXmv Model Generation

The nuXmv pipeline is very similar to the Python pipeline, and as such we will avoid going into the inner workings of this pipeline. The main difference is that we create a monitor using the DSL of BehaVerify allowing BehaVerify to create a .smv model for use with nuXmv.

5 Monitor Comparison

We created two scaling scenarios and ran timing comparisons for the generated monitors created by BehaVerify and Copilot. We planned to compare two monitors generated by NuRV, but did not ultimately do so (see Subsection 5.5 for details). Additionally, we utilized the generated Python code with monitors to generate example traces. While these example traces are not conclusive proof, they were sufficient to demonstrate some differences in the generated monitors. The results demonstrate that the monitors generated by BehaVerify are not outclassed by existing tools and in some cases are preferable from a correctness standpoint. Furthermore, this demonstrates the versatility of our setup; it is fairly painless to bring in outside monitors should the need arise.

The rest of this section will describe the scaling scenarios, present the results, and then reason about the results.

5.1 Scenarios

For our scenarios, a drone navigates a grid and tries to reach a destination. Once the destination is reached, a new destination is randomly generated. We generated grids from size 10 by 10 to 50 by 50 in two styles: dense fixed and sparse random. The dense fixed grids start with a 5 by 5 grid and copies this layout to fill the entire grid. The sparse random grids are randomly generated. See Figure 5 for details.

At each time step, the drone attempts to move towards the target according to the control logic. We want the drone to satisfy the following specifications

$$\phi_S = \square(\forall(x_o,y_o) \in Obs, (x_d,y_d) \neq (x_o,y_o))$$
$$\phi_L = \square(\lozenge((x_d,y_d) = (x_g,y_g) \vee \exists(x_o,y_o) \in Obs \text{ s.t. } (x_g,y_g) = (x_o,y_o)))$$

Here *Obs* is a predetermined finite set of obstacles. ϕ_S is a safety specification that states that the drone's position (x_d,y_d) is never equal to the position of an obstacle (x_o,y_o). ϕ_L is a liveness specification that states that the drone's position is eventually equal to the destination (x_g,y_g), or the destination is an obstacle. We utilize the quantifiers $\forall,\exists$ (for all and exists, respectively) for convenience here; in practice we write out each individual obstacle.

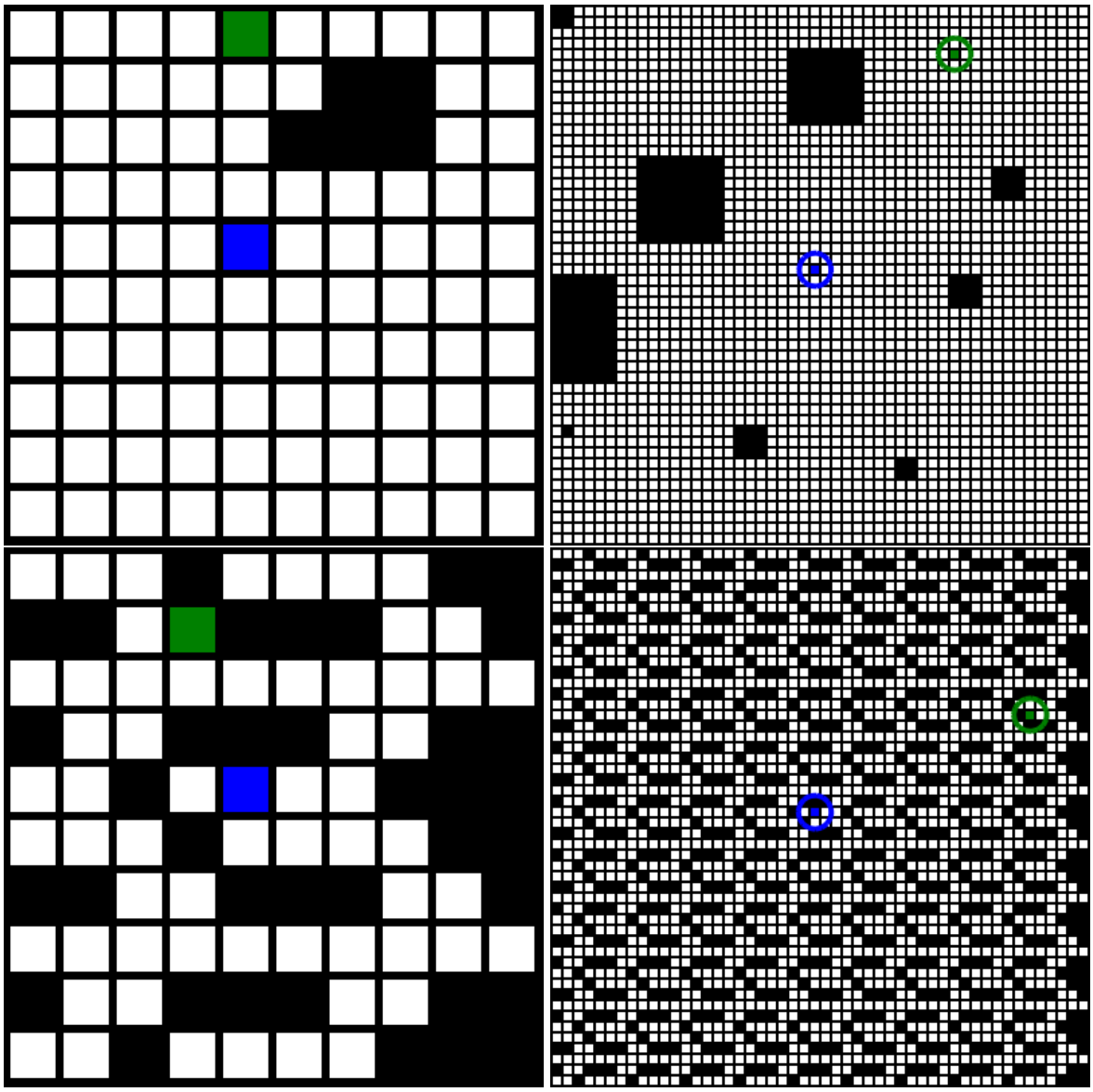


Figure 5: Images representing some of the grids used for the scaling experiments. The upper grids are sparse and were randomly generated. The lower grids are dense and were created by copying a 5 by 5 grid repeatedly. The left grids are 10 by 10 and the right grids are 50 by 50. Black squares are obstacles, the blue square is the drone, and the green square is the destination.

Then the specifications we monitor are

$$\phi_{S1} = \Box(\forall(x_o,y_o) \in Obs, (x_d + (x_\Delta * s), y_d + (y_\Delta * s)) \neq (x_o,y_o))$$

$$\phi_{L1} = \Box \begin{pmatrix} (x_\Delta,y_\Delta) = (1,0) \implies \bigcirc((x_\Delta,y_\Delta) \neq (-1,0)) \wedge \\ (x_\Delta,y_\Delta) = (-1,0) \implies \bigcirc((x_\Delta,y_\Delta) \neq (1,0)) \wedge \\ (x_\Delta,y_\Delta) = (0,1) \implies \bigcirc((x_\Delta,y_\Delta) \neq (0,-1)) \wedge \\ (x_\Delta,y_\Delta) = (0,-1) \implies \bigcirc((x_\Delta,y_\Delta) \neq (0,1)) \end{pmatrix}$$

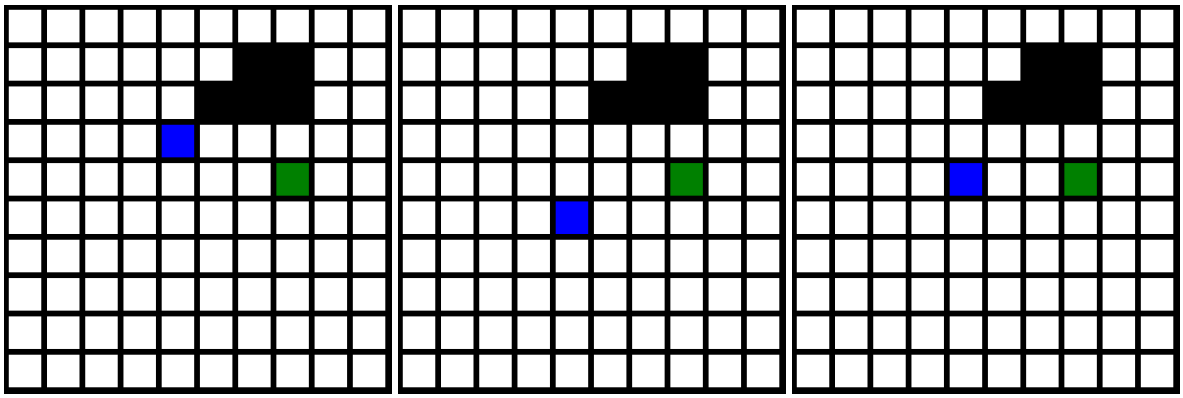


Figure 6: Pictures are ordered left to right. The drone (blue) is trying to reach the destination (green) while avoiding obstacles (black). In the left image, the control logic tells the drone to go down. Because neither monitor reported a violation, the drone moved 2 squares and is now in the situation shown in the middle image. The control logic now tells the drone to go up. The liveness monitor reports a violation; if we go up two squares, the drone will be stuck in a loop. Therefore, the drone only goes up one square and is now in the situation shown in the right image. If the drone has not been equipped with the liveness monitor, it would have gone back to the state shown in the left image, then middle, then left, etc.

(x_d,y_d) is the current location of the drone, (x_Δ,y_Δ) is one of $(1,0)$, $(-1,0)$, $(0,1)$, $(0,-1)$, $(0,0)$, describing the possible directions the drone moves in, and s is the speed of the drone (either 1 or 2). Thus the safety specification monitor is violated if we are on a collision course and the liveness specification monitor is violated if we do a 180 turn. We note that both specifications being monitored are safety specifications; however, we will refer to the second as a liveness specification as it is being used to ensure the liveness specification is not violated. Please note the following: we are *not* claiming that by monitoring these conditions any *BT* will satisfy the desired specifications; rather, the purpose is to demonstrate that the monitors can be used to correct specific flaws in a *BT*. Furthermore, it is possible to design and insert these monitors into the *BT* without the use of a special tool; however, such a task may prove more complex then writing an LTL specification and utilizing our tool.

By default, the drone will try to move 2 squares; if either the safety monitor or liveness monitor are triggered, it will only move one square. The safety monitor ensures that the drone does not move into obstacles. The liveness monitor ensures that the drone escapes potential loops, as seen in Figure 6. Thus both monitors are necessary for the drone to function as intended. Note that this example is not meant to illustrate good programming practice for drone controllers; there are no doubt better methods by which to control a drone. Rather, the purpose of this example is to demonstrate how one can use the monitors to ensure a system functions correctly.

5.2 Motivation

We were originally creating a controller for a drone in virtual neighborhood simulated using AirSim (see Figure 7). The drone would fly at a fixed height and knew where obstacles were before hand. We created a grid-world abstraction of the problem and created a controller that we were able to verify navigated correctly under certain assumptions. One of the assumptions was that the drone would always move at most one tile at a time. While it is simple to ensure that this is the case, it requires flying the drone slowly at all times, which is not desirable. Flying the drone at faster speeds, however, created both safety

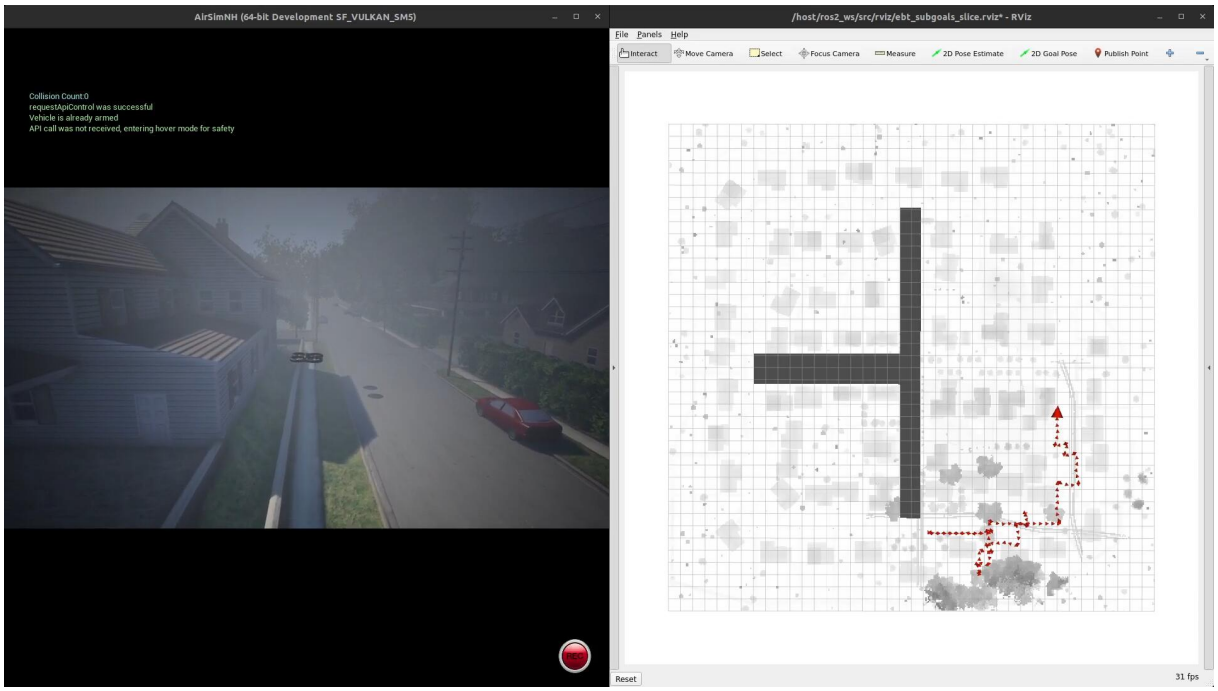


Figure 7: A screenshot of the drone flying in AirSim and a grid visualization.

and liveness violations. Thus by equipping our *BT* with the described monitors, we were able to safely increase speed without compromising safety. Finally, we were able to utilize nuXmv to verify that the *BTM* is safe (see Section 6 for details).

5.3 Results

All results (Figure 8) were generated on a computer with a 24 core 13th Gen Intel(R) Core(TM) i7-13700K and 64GB of DDR5 RAM. Results were run in a quiet environment when no other user run processes were active. The code for the experiments is available at [3].

5.4 Analysis

Timing We ran 10000 simulation runs with 1000 iterations (number of times the drone tried to move) and took the median for both Copilot and BehaVerify. To ensure that we are comparing only the monitors, we also ran a version of the code with no monitor with the same settings. This monitorless value was then subtracted out from the timing results for both BehaVerify and Copilot. We believe that the timing results demonstrate that the monitors generated by BehaVerify and Copilot are reasonably close. As expected, the dense fixed grid pattern produces a far more linear timing relation than the sparse random grid. This is because the number of obstacles in the dense fixed grid is always greatly increasing, while the number of obstacles in the sparse random setup is random, and it is the number of obstacles that is largely responsible for the complexity of monitoring the specifications.

[3] https://github.com/verivital/behaverify/tree/main/REPRODUCIBILITY/2024_FMAS_BTM

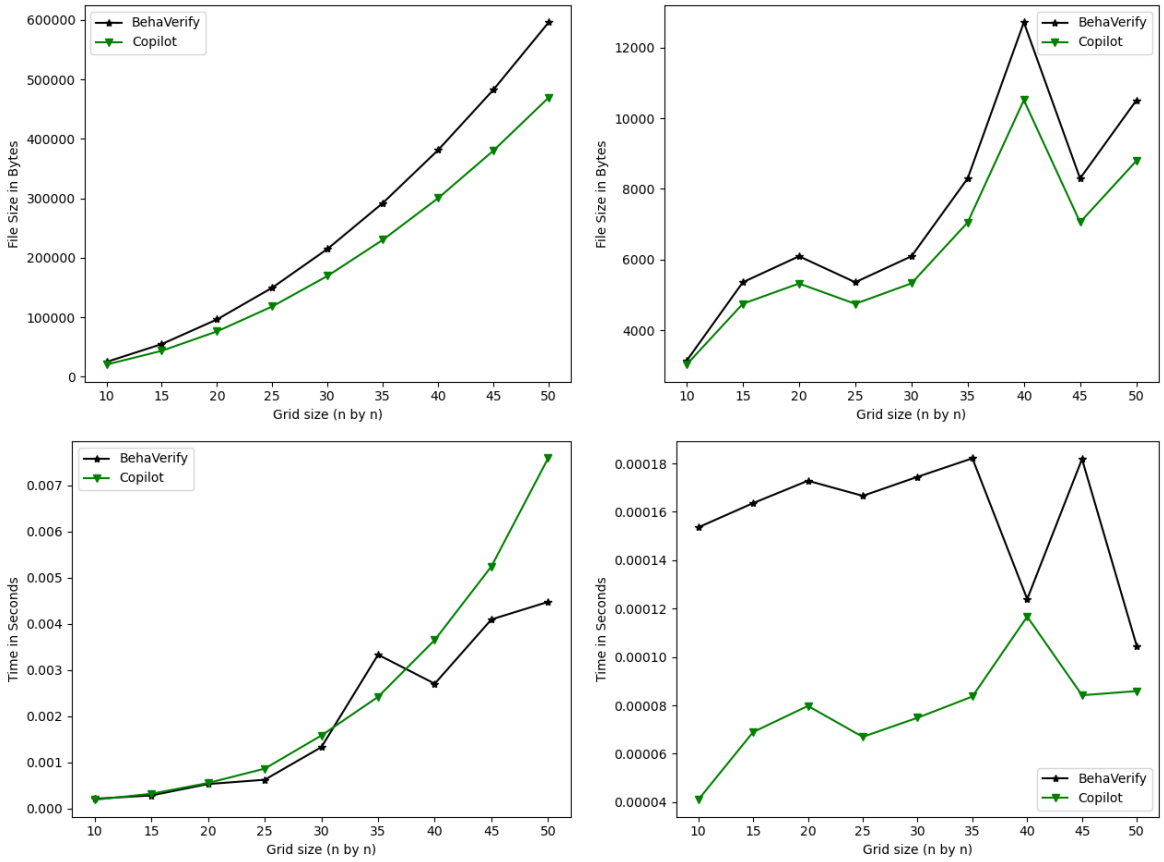


Figure 8: Top left: file sizes of safety monitors for dense fixed. Top right: file sizes of safety monitors for sparse random. Bottom left: median time (in seconds) for safety and liveness monitors on dense fixed. Bottom right: median time (in seconds) for safety and liveness monitors on sparse random. We did not include the size of liveness monitors as it did not change with the size of the grid. The timing results are the median of 10000 runs. Each run had 1000 iterations (number of times the drone tried to move). We subtracted how long it took to run the code without a monitor.

File Size We measured the file size as an indicator of the complexity of the monitoring algorithm. While BehaVerify clearly generates larger files than Copilot, we do not believe that the difference is sufficient to prefer Copilot's Monitors. This is in contrast to NuRV, which is discussed below in Subsection 5.5.

Correctness The safety monitors that Copilot generated worked as intended, but the liveness monitor had an issue: it would report a violation with a one-step delay. Specifically, if a violation occurred in the i^{th} state, then Copilot would report it in the $i+1^{th}$ step. This behavior is documented in the Copilot tutorial in example 7 [4]. By contrast, both the safety and liveness monitors generated by BehaVerify worked as intended, ensuring the drone functioned as intended.

This brings us to an important note about how our process is laid out. While BehaVerify is capable of generating monitors, the generated Python code can utilize *any* provided monitor. Indeed, we confirmed that it is possible to utilize Copilot for the safety monitor and BehaVerify for the liveness monitor and that

[4] https://copilot-language.github.io/downloads/copilot_tutorial.pdf

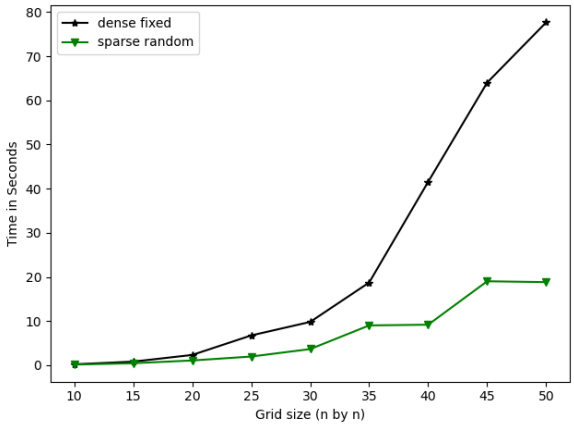


Figure 9: The graph shows the time (in seconds) to verify that the BTM is safe (does not crash into obstacles). The verification was done with nuXmv. Liveness specifications are considerably harder to verify, with the 9 by 9 sparse grid taking about 25 minutes to verify. As this is the easiest grid we have for this task, we did not complete liveness verification for any other grids.

this works as intended.

5.5 NuRV

Because BehaVerify creates .smv files, we originally tried using the output of BehaVerify as input to NuRV. Unfortunately, a variety of issues prevented this from being feasible. For instance, NuRV monitors are aware of the transition system. This enable NuRV monitors to potentially detect violations well in advance or to verify liveness conditions, but it also means the files are much larger. To combat this, we created simplified .smv models with much simpler transition systems. This proved ineffective; the smallest file generated by NuRV was 26.04 MB, while the largest file generated by BehaVerify was 582 KB.

6 Design Time Verification

BehaVerify was originally created for Design Time Verification. As such, when approaching the topic of Runtime Verification, we were interested if we could use Design Time Verification for the Runtime Monitors. As such, we translated the monitors that were created by BehaVerify back into the DSL for BehaVerify and then utilized nuXmv to verify that the BT with monitors satisfied both the safety and liveness specification. While we created such translations for each combination of grid type and grid size presented in Section 5, some of the resulting models proved to complex for liveness analysis in nuXmv. The results for safety verification can be seen in Figure 9. As you can see, it is entirely feasible to use design time verification to confirm that the safety monitors are correct and ensure the system works as intended. If the safety monitor is removed, nuXmv will demonstrate that the system is not safe by providing a counter example trace resulting in a crash. If the liveness monitor is removed, nuXmv will demonstrate that the system can get stuck in a loop by providing a counter example trace.

The liveness situation is somewhat trickier, as the specifications are substantially harder to verify. However, considering the fact that the same liveness monitor is used for all grids and that we verified it for one grid, even this limited verification process can provide some evidence to indicate that the monitor is correct. As with the safety monitor, the removal of the liveness monitor results in a specification violation that nuXmv detects. In this case, nuXmv returns a counterexample where the drone becomes stuck in a loop, going back and forth between two points without reaching the destination.

7 Conclusions and Future Work

We presented a formal problem statement for incorporating contingency monitors within *BT*s, thus creating *BTM*s. On the implementation side, we expanded the DSL of BehaVerify to incorporate these monitors, and demonstrated that our code is capable of generating implementations of the monitors that are on par with existing tools. However, our overall approach also brings the advantage of Design Time Verification for the entire *BTM*. We subsequently hope to expand the target range of BehaVerify, specifically to create .cpp implementations that make use of BehaviorTrees.cpp.

Acknowledgements

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) through grant numbers 2220426 and 2220401, the Defense Advanced Research Projects Agency (DARPA) under contract number FA8750-23-C-0518, and the Air Force Office of Scientific Research (AFOSR) under contract numbers FA9550-22-1-0019 and FA9550-23-1-0135. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of AFOSR, DARPA, or NSF.

References

[1] Auryn Robotics: *Tutorial 02: Blackboard and Ports*. Available at https://www.behaviortree.dev/docs/tutorial-basics/tutorial_02_basic_ports.

[2] Stanley Bak, Deepti K. Chivukula, Olugbemiga Adekunle, Mu Sun, Marco Caccamo & Lui Sha (2009): *The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety*. In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 99–107, doi:10.1109/RTAS.2009.20.

[3] Andreas Bauer, Martin Leucker & Christian Schallhart (2011): *Runtime Verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol.* 20(4), doi:10.1145/2000799.2000800.

[4] Oliver Biggar & Mohammad Zamani (2020): *A Framework for Formal Verification of Behavior Trees With Linear Temporal Logic*. *IEEE Robotics and Automation Letters* 5(2), pp. 2341–2348, doi:10.1109/LRA.2020.2970634.

[5] Oliver Biggar, Mohammad Zamani & Iman Shames (2021): *An Expressiveness Hierarchy of Behavior Trees and Related Architectures*. *IEEE Robotics and Automation Letters* 6(3), pp. 5397–5404, doi:10.1109/lra.2021.3074337.

[6] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri & Stefano Tonetta (2014): *The nuXmv Symbolic Model Checker*. In: *CAV*, pp. 334–342. Available at http://dx.doi.org/10.1007/978-3-319-08867-9_22.

[7] Alessandro Cimatti, Chun Tian & Stefano Tonetta (2019): *NuRV: A nuXmv Extension for Runtime Verification*. In Bernd Finkbeiner & Leonardo Mariani, editors: *Runtime Verification*, Springer International Publishing, Cham, pp. 382–392, doi:10.1007/978-3-030-32079-9_23.

[8] Michele Colledanchise, Giuseppe Cicala, Daniele E. Domenichelli, Lorenzo Natale & Armando Tacchella (2021): *Formalizing the Execution Context of Behavior Trees for Runtime Verification of Deliberative Policies*. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE Press, pp. 9841–9848, doi:10.1109/IROS51168.2021.9636129.

[9] Michele Colledanchise & Petter Ögren (2014): *How Behavior Trees modularize robustness and safety in hybrid systems*. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1482–1488, doi:10.1109/IROS.2014.6942752.

[10] Michele Colledanchise & Petter Ögren (2016): *How Behavior Trees generalize the Teleo-Reactive paradigm and And-Or-Trees*. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 424–429, doi:10.1109/IROS.2016.7759089.

[11] Michele Colledanchise & Petter Ögren (2017): *How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees*. *IEEE Transactions on Robotics* 33(2), pp. 372–389, doi:10.1109/TRO.2016.2633567.

[12] Antoine El-Hokayem & Yliès Falcone (2018): *Bringing Runtime Verification Home*. In Christian Colombo & Martin Leucker, editors: *Runtime Verification*, Springer International Publishing, Cham, pp. 222–240, doi:10.1007/978-3-030-03769-7_13.

[13] EpicGames (2021): *Behavior tree overview*. Available at https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/.

[14] Angelo Ferrando, Rafael C. Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini & Viviana Mascardi (2020): *ROSMonitoring: A Runtime Verification Framework for ROS*. In Abdelkhalick Mohammad, Xin Dong & Matteo Russo, editors: *Towards Autonomous Robotic Systems*, Springer International Publishing, Cham, pp. 387–399, doi:10.1007/978-3-030-63486-5_40.

[15] Angelo Ferrando & Vadim Malvone (2022): *Towards the Combination of Model Checking and Runtime Verification on Multi-agent Systems*. In Frank Dignum, Philippe Mathieu, Juan Manuel Corchado & Fernando De La Prieta, editors: *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complex Systems Simulation. The PAAMS Collection*, Springer International Publishing, Cham, pp. 140–152, doi:10.1007/978-3-031-18192-4_12.

[16] Paul Gastin & Denis Oddoux (2001): *Fast LTL to Büchi Automata Translation*. In Gérard Berry, Hubert Comon & Alain Finkel, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 53–65, doi:10.1007/3-540-44585-4_6.

[17] Felipe Gorostiaga & César Sánchez (2021): *HLola: a Very Functional Tool for Extensible Stream Runtime Verification*. In Jan Friso Groote & Kim Guldstrand Larsen, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 349–356, doi:10.1007/978-3-030-72013-1_18.

[18] Zhaoyuan Gu, Nathan Boyd & Ye Zhao (2022): *Reactive Locomotion Decision-Making and Robust Motion Planning for Real-Time Perturbation Recovery*. In: *2022 International Conference on Robotics and Automation (ICRA)*, pp. 1896–1902, doi:10.1109/ICRA46639.2022.9812068.

[19] Thomas Henn, Marcus Völker, Stefan Kowalewski, Minh Trinh, Oliver Petrovic & Christian Brecher (2022): *Verification of Behavior Trees using Linear Constrained Horn Clauses*. In Jan Friso Groote & Marieke Huisman, editors: *Formal Methods for Industrial Critical Systems*, Springer International Publishing, Cham, pp. 211–225, doi:10.1007/978-3-031-15008-1_14.

[20] G.J. Holzmann (1997): *The model checker SPIN*. *IEEE Transactions on Software Engineering* 23(5), pp. 279–295, doi:10.1109/32.588521.

[21] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren & Christian Smith (2022): *A survey of Behavior Trees in robotics and AI*. *Robotics and Autonomous Systems* 154, p. 104096, doi:10.1016/j.robot.2022.104096.

[22] Alejandro Marzinotto, Michele Colledanchise, Christian Smith & Petter Ögren (2014): *Towards a unified behavior trees framework for robot control*. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5420–5427, doi:10.1109/ICRA.2014.6907656.

[23] Huang Peishan, Hong Weijiang, Chen Zhenbang & Wang Ji: *MoVe4BT: Modeling & Verification For BT*. Available at https://move4bt.github.io/. Accessed: 2023-12-14.

[24] Ivan Perez, Frank Dedden & Alwyn Goodloe (2020): *Copilot 3*. Technical Report Technical Report NASA/TM-2020-220587, NASA.

[25] Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alwyn Goodloe & Dimitra Giannakopoulou (2022): *Automated Translation of Natural Language Requirements to Runtime Monitors*. In Dana Fisman & Grigore

Rosu, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 387–395, doi:10.1007/978-3-030-99524-9_21.

[26] Fangbo Qin, De Xu, Blake Hannaford & Tiantian Hao (2023): *Object-Agnostic Vision Measurement Framework Based on One-Shot Learning and Behavior Tree*. *IEEE Transactions on Cybernetics* 53(8), pp. 5202–5215, doi:10.1109/TCYB.2022.3181054.

[27] Serena S. Serbinowska & Taylor T. Johnson (2022): *BehaVerify: Verifying Temporal Logic Specifications For Behavior Trees*. In: *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, Springer-Verlag, Berlin, Heidelberg, pp. 307–323, doi:10.1007/978-3-031-17108-6_19.

[28] D. Seto, B. Krogh, L. Sha & A. Chutinan (1998): *The Simplex architecture for safe online control system upgrades*. In: *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*, 6, pp. 3504–3508 vol.6, doi:10.1109/ACC.1998.703255.

[29] D. Seto & L. Sha (1999): *A Case Study on Analytical Analysis of the Inverted Pendulum Real-Time Control System*. Technical Report, DTIC and NTIS. 10.21236/ADA373286.

[30] Shital Shah, Debadeepta Dey, Chris Lovett & Ashish Kapoor (2018): *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles*. In Marco Hutter & Roland Siegwart, editors: *Field and Service Robotics*, Springer International Publishing, Cham, pp. 621–635, doi:10.1007/978-3-319-67361-5_40.

[31] Christopher I. Sprague & Petter Ögren (2022): *Continuous-Time Behavior Trees as Discontinuous Dynamical Systems*. *IEEE Control Systems Letters* 6, pp. 1891–1896, doi:10.1109/LCSYS.2021.3134453.

[32] Daniel Stonier: *PyTrees Module API*. Available at https://py-trees.readthedocs.io/en/devel/modules.html. Accessed: 2023-12-14.

[33] Alexander Will, Aidan Collins, Robert Grizzard, Smitha Gautham, Patrick Martin, Evan Dill & Carl Elks (2023): *An Integrated Runtime Verification and Simulation Testbed for UAM Hazard Assessment*. In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, pp. 42–48, doi:10.1109/DSN-S58398.2023.00023.

[34] Sebastián Zudaire, Felipe Gorostiaga, César Sánchez, Gerardo Schneider & Sebastián Uchitel (2021): *Assumption Monitoring Using Runtime Verification for UAV Temporal Task Plan Executions*. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6824–6830, doi:10.1109/ICRA48506.2021.9561671.

[35] Petter Ögren (2020): *Convergence Analysis of Hybrid Control Systems in the Form of Backward Chained Behavior Trees*. *IEEE Robotics and Automation Letters* 5(4), pp. 6073–6080, doi:10.1109/LRA.2020.3010747.

[36] Petter Ögren & Christopher I. Sprague (2022): *Behavior Trees in Robot Control Systems*. *Annual Review of Control, Robotics, and Autonomous Systems* 5(Volume 5, 2022), pp. 81–107, doi:10.1146/annurev-control-042920-095314.

RV4Chatbot: Are Chatbots Allowed to Dream of Electric Sheep?

Andrea Gatti Viviana Mascardi
Department of Informatics, Bioengineering, Robotics and Systems Engineering
University of Genoa
Genoa, Italy
forename.surname@unige.it

Angelo Ferrando
Department of Physics, Informatics and Mathematics
University of Modena and Reggio Emilia
Modena, Italy
angelo.ferrando@unimore.it

Chatbots have become integral to various application domains, including those with safety-critical considerations. As a result, there is a pressing need for methods that ensure chatbots consistently adhere to expected, safe behaviours. In this paper, we introduce RV4Chatbot, a Runtime Verification framework designed to monitor deviations in chatbot behaviour. We formalise expected behaviours as interaction protocols between the user and the chatbot. We present the RV4Chatbot design and describe two implementations that instantiate it: RV4Rasa, for monitoring chatbots created with the Rasa framework, and RV4Dialogflow, for monitoring Dialogflow chatbots. Additionally, we detail experiments conducted in a factory automation scenario using both RV4Rasa and RV4Dialogflow.

1 Introduction

On November 30th, 2022, ChatGPT was unveiled [40] deeply shaking the industry and academic worlds. The impression, at that time, was that ChatGPT and chatbots based on Large Language Models (LLM) would have irreversibly changed the way chatbots were designed and built, wiping away any pre-existing technology.

After almost two years, a more balanced view on the future is emerging, with the shared feeling that there is still room for chatbots that do not rely on generative AI techniques.

There are many ways to classify chatbots based for example on the knowledge domain, the service provided, the goals, the response generation method [2], the locus of control (chatbot- or user-driven) and duration of the interaction (short or long) [23], or their affordances and disaffordances [32, 35]. For the purposes of this paper, the simplest and most suitable classification of text-based chatbots divides them into conversational AI and generative AI ones [18].

DialogFlow [28, 42], Rasa [11, 41], Wit.ai [38, 39], just to name a few, are text-based conversational AI chatbots, also referred to as intent-based chatbots. They can understand the users questions, no matter how they are phrased, thanks to Natural Language Understanding (NLU) capabilities that allow them to detect the user's intent and further contextual information. The NLU component exploits machine learning techniques for the intent classification and performs well even with a small amount of training sentences. The answer to be provided to the user is not autonomously generated by the chatbot, but is designed by the chatbot's developer. Conversational AI chatbots can remember conversations with users and incorporate contextual information into their interactions.

ChatGPT, Gemini [29], Jasper Chat [31] are examples of generative AI chatbots. They go far beyond conversational AI chatbots thanks to their capability of generating new content as their answer in form of high-quality text, images and sound based on LLMs they are trained on. This impressive power,

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 73–90, doi:10.4204/EPTCS.411.5

© A. Gatti, V. Mascardi & A. Ferrando
This work is licensed under the Creative Commons Attribution License.

however, does not come without pitfalls. Besides religious bias [1], gender bias and stereotypes [33], and hallucinations [48], major privacy concerns are associated with LLMs.

In March 2023, Italy's data regulator imposed a temporary ban on ChatGPT due to concerns related to data security. During its development, in November 2023, an open letter was signed by nine Italian scientific associations including the Italian Association for AI and the Italian Association for Computer Vision, Pattern Recognition and Machine Learning, and by around 500 scientists, asking the Italian government to guarantee that strict rules for the use of generative AI were included in the European AI Act [21].

Scientific studies on LLM privacy leakage are so recent to be still unpublished at the time of writing, but many pre-prints by academic scholars show that the problem is real [17,46,47]. Personal Identifiable Information (PII) protection can only be complied with by organisations able to have a private installation of a LLM within a private cloud or on premise [30]. The resources needed to implement this solution make it not affordable for most companies and universities.

The global LLM market size (that includes the generative AI chatbot market plus a wide range of other applications) is projected to reach 259,886 Million USD revenue by 2029 [26], while the conversational AI market is expected to reach 29,800 Million USD by 2028 [37]: the market forecasts and the privacy, ethical, and economical issues of LLM suggest that traditional conversational AI chatbots will still be needed and used by many players in the next few years.

Although more controllable than their generative evolution, the behaviour of conversational AI chatbots can also be unsafe. In a factory automation scenario, where an intent-based chatbot provides a natural language interface between the user and a virtual representation of a factory, a conversation becomes unsafe if the user requests to position an object where another object has already been placed, or if the distance between objects is insufficient. Similarly, a conversation is unsafe if the chatbot provides the coordinates of an object that the user never inserted.

To cope with safety issues in conversational AI chatbots, we present an approach to verify at runtime the conversation between the user and the chatbot. Runtime Verification (RV) [8] is a formal verification technique used to analyse the runtime behaviour of software and hardware systems concerning specific formal properties. A RV monitor emits boolean verdicts that state whether the property is satisfied or not by the currently observed events. The default functioning is to state that something went wrong when it just went wrong, and trigger *recovery* actions. In some cases, the monitor may intervene before the wrong event is generated or the unsafe action is done, hence allowing for *prevention*. With respect to other formal verification techniques, such as Model Checking [19] and Theorem Provers [36], RV is more dynamic and lightweight and shares some similarities with software testing, being focused on checking how the system behaves while it is running.

To perform RV of chatbots, we have designed a general and formalism-agnostic framework named RV4Chatbot. We show RV4Chatbot versatility by instantiating it for RV of chatbots created with Rasa, widely used to develop chatbots in local environments, and Dialogflow, used in cloud-based applications. We demonstrate how our engineering decisions render RV4Chatbot a highly practical methodology and how it can seamlessly integrate with existing chatbot frameworks. It is essential to note that our utilisation of Rasa and DialogFlow serves only to illustrate potential applications of our approach. Our ultimate aim is to encompass any chatbot development framework.

The paper is structured as follows. After overviewing the related work in Section 2, Section 3 introduces one example that motivates the need of RV4Chatbot. After that, Section 4 describes the architecture and the data and control flow of RV4Chatbot. Sections 5 and 6 describe, respectively, RV4Rasa and RV4Dialogflow, the two concrete instantiations of the RV4Chatbot logical architecture. Section 7 discusses the formalisation of some relevant safety properties in the motivating scenario, using a highly

expressive RV language, and the experiments we carried out to verify those properties with RV4Rasa and RV4Dialogflow. Section 8 concludes the paper and highlights the possible future directions.

2 Related Work

RV of interaction protocols attracted the attention of researchers starting from the beginning of the millennium. The first interaction protocols to be verified at runtime involved web services [34], cloud applications [44], cryptography [9]. RV of interactions among autonomous software agents followed soon [3, 7][1].

Despite the large interest in RV of interactions and the pressing need to monitor what chatbots say and do, to the best of our knowledge no studies on RV of human-chatbot interactions exist, if we exclude the very recent works where we were involved.

Apart from [22] which serves as the foundation for this paper but is tailored for a specific chatbot framework, the only other work in the literature that deals with the formal verification of chatbot systems is [20], introducing a framework known as RV4JaCa. In that work we integrated RV within the multiagent system (MAS) domain and demonstrated how to monitor agent interaction protocols within that context. The focus there was not on the chatbot itself, but on the software agents interacting with it. The main contribution was hence in the MAS domain, although applied in a scenario where messages for agents are generated by a chatbot.

Expanding the boundaries of our investigation, we can mention a recent proposal that approaches formal verification of chatbots from a static perspective, instead of at runtime as we do. In [25] the authors introduce a strategy for verifying chatbot conversational flows during the design phase using the UPPAAL tool [10], a well-known model checker. The approach is tested by designing a hotel booking chatbot and receiving feedback from developers. The strategy is found to have an acceptable learning curve and potential for improving chatbot development. In contrast to our approach, the work presented in [25] focuses on abstracting the chatbot using a model and subsequently verifying it through model checking. Due to the distinct inherent natures of these two verification approaches, we envision the possibility of integrating them to harness their respective strengths. Specifically, our technique could enhance the visibility of [25] by providing information that is only available at runtime. Conversely, the exhaustiveness of [25] could be leveraged by our approach to simplify the properties for monitoring, thanks to prior knowledge of the chatbot's behavioural model.

If we further expand our search and give up formality, hence resorting to software testing of chatbots, some works from J. Bozic's research group can be mentioned. The paper [14] introduces a planning-based testing approach for chatbots, focusing on functional testing, specifically in the context of tourism chatbots for hotel reservations. Planning is used to generate test scenarios, and a testing framework automates the execution of test cases. The results show success in testing chatbots, but some issues, such as intent recognition errors, need further attention. Metamorphic testing is illustrated in [15], where metamorphic relations are used instead of traditional test oracles due to the unpredictable nature of AI systems. On a similar line of research, the work [13] introduces an approach that leverages ontologies to generate test cases and addresses the absence of a test oracle by using a metamorphic testing approach. The method is demonstrated on a real tourism chatbot.

A methodology that automates the generation of coherence, sturdiness, and precision tests for chatbots and leverages the test results to enhance their precision is presented in [16]. The methodology is

[1] Starting from 2012, a large share of the scientific production of the authors dealt with RV of agent interaction protocols, see `https://rmlatdibris.github.io/biblio.html`. We limit ourselves to cite the most relevant works among these.

implemented in a tool called Charm, which uses Botium [12] for automated test execution. The paper also presents experiments conducted to improve third-party-built DialogFlow chatbots.

While these works share similarities with ours in that they focus on the actual, runtime execution of the chatbot rather than its abstraction, none is based on a rigorous and formal specification that guides the correctness checks.

To conclude, we emphasise that our goal is to assess the overall correctness of a conversation between a user and a chatbot as a coherent whole, rather than focusing on how individual message utterances are generated by the chatbot itself. This distinction is pivotal and sets our work apart from existing literature on the formal verification of Machine Learning models, as surveyed in [43]. In such literature, the emphasis is generally on verifying the Machine Learning model. In contrast, in RV4Chatbot, our focus is not on whether the model correctly produces or classifies individual messages, but rather on the consistency of these messages within a conversation. In this sense, RV4Chatbot delves deeper into the conversational semantics of the messages exchanged with or generated by the chatbot, rather than attempting to dissect the chatbot to understand its internal behaviour, which is often treated as a black-box.

3 Motivating Example

Chatbots can be exploited for achieving three main goals: providing specific information stored in a fixed source (information chatbots); holding a natural conversation with the user (chat-based chatbots); and understanding the tasks that the user wants to perform, hence executing functions to perform them (task-based chatbots) [2].

Usually, information chatbots provide an answer to one questions and go back to a state – the only state they can be – where they are ready to answer a new question. There is no need for them to keep memory of what the user already asked or said, and to carry out a coherent and fluent conversation. The correctness of the chatbot is related with the correctness of the search engine in its backend. Given that we aim at verifying the conversation flow rather than the quality of the retrieved information, RV of information chatbots following our approach is out of our scope.

Chat-based and task-based chatbots, on the other hand, engage into conversations that should evolve in different ways depending on what the user utters. For example, a chat-based chatbot may show different reactions to the very same request from the user, depending on how much the user insists upon it. In a task-based chatbot, the possibility for the user to ask for some task to be performed may depend on the fact that some prerequisite task had been asked, and hence performed, before.

Without loss of generality, the following motivating example focuses on a task-based chatbot. The application of RV4Chatbot to chat-based chatbots is left for future work, as it would mainly require adapting the types of properties to be verified, rather than altering the verification methodology. In essence, RV4Chatbot is not restricted to task-based chatbots; its architecture is sufficiently flexible to support the verification of any intent-based chatbot.

A Task-based Chatbot in the Factory Automation Domain. This example, presented in the VORTEX 2023 workshop [22] and briefly summarised here, is set in the field of robotics and involves the development of a task-based chatbot assisting in the creation of a simulated factory work floor. The chatbot's role is to guide users through this process, taking into account both the users' requirements and the factory regulations[2] concerning what can or cannot be added or removed from the factory work

[2]See ISO 10218-1:2011 standard on Robots and robotic devices - Safety requirements for industrial robots [45].

floor for safety reasons. The user interacts with the chatbot by requesting to add a robot to a specific position on the factory work floor, removing a robot, or relocating a previously added robot to a different position. For example, properties verified at runtime might include ensuring that objects are not added to an already occupied position on the factory floor, confirming that each removal request corresponds to an object that actually exists in the current state, and enforcing spacing rules between objects as defined by safety regulations. The validity of these actions depends on the state of the simulated work floor, which evolves as the user-chatbot conversation progresses. RV4Chatbot checks these properties dynamically to detect and prevent any violations that could compromise the safety or coherence of the conversation.

4 RV4Chatbot: The Foundation

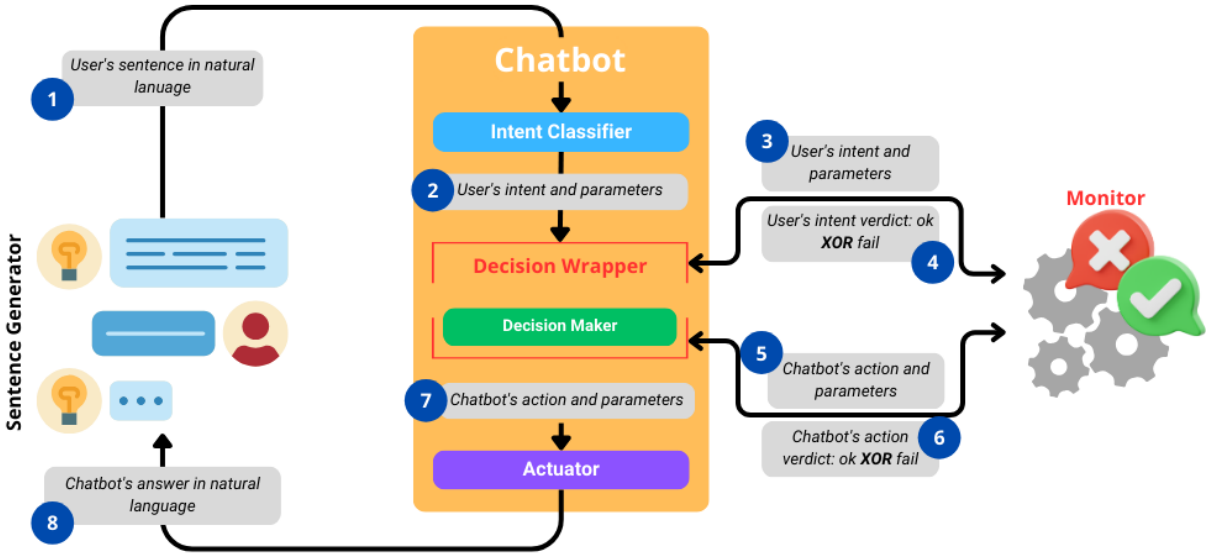


Figure 1: RV4Chatbot architecture.

Figure 1 illustrates the operation of an intent-based chatbot. RV4Chatbot specifically focuses on verifying intent-based chatbots, leaving the verification of non-intent-based chatbots for future work, as mentioned in both the introduction and conclusion sections.

A human user, or more generally, a sentence generator, produces a sentence in natural language (1). This sentence is categorised by a classifier based on its intent, and its parameters are extracted. The intent classifier is responsible for Natural Language Understanding (NLU). The recognised intent, along with its parameters (2), is then passed to a decision maker that determines the chatbot's response (7) to the input sentence. This decision-making process is typically hard-coded and integrated directly into the chatbot framework. Finally, the chatbot generates the response to be delivered to the user (8). The module responsible for this generation process is termed the 'actuator'.

RV4Chatbot introduces a decision wrapper, depicted in red with right angles inside the chatbot architecture, to manage actions numbered (3) to (6) on the right side of the figure. The decision wrapper extends the decision maker to allow the data characterising a chatbot's lifecycle—user's intents and chatbot's actions, both with optional parameters—to be sent to an external monitor where Runtime Verification (RV) occurs. Depending on the chatbot's framework and its modularity, implementing the decision

wrapper may vary in complexity and intrusiveness. The decision wrapper instruments the 'System Under Scrutiny' (the chatbot in this application), using standard RV terminology. In RV4Chatbot, instrumentation is confined solely to this module.

The decision wrapper sends the recognised intents and parameters (3) to the monitor. Regardless of its implementation and the language used for modeling properties to verify, the monitor observes one event at a time and emits a boolean verdict indicating whether the event complies with the property (4). If the verdict is true (or inconclusive[3]), control returns to the decision maker, which decides the subsequent action. Before executing the action, the decision wrapper sends it to the monitor (5), which again verifies its compliance with the property and emits a verdict (6).

A true (or inconclusive) verdict from the monitor does not alter the chatbot's standard execution flow. A false verdict—whether originating from an unexpected user intent or a disallowed action by the chatbot—returns the chatbot to a listening state, displaying a message explaining the failure to the user. In both cases, no unsafe actions are performed.

Numerous intent-based chatbot frameworks are documented in the literature. Although their implementations may vary significantly, their main components and functionalities are accurately represented in Figure 1. Similarly, many RV monitors exist. Regardless of the monitor used, it must at least be able to observe events from the System Under Scrutiny and output a verdict that is either true, false, or inconclusive. This is the only assumption we make regarding the RV monitor's function, and it is satisfied by the definition of a monitor. Thus, the RV4Chatbot logical architecture is parametric in both the chatbot framework and the monitor.

To automate the experiments presented in Section 7.3, we developed a piece of software capable of reading natural language sentences from a file and sending them to the chatbot using the APIs provided by the chatbot frameworks considered in this paper. Although our primary interest lies in RV, we soon realised that the files of simulated user sentences could be seen as test cases, and that the software component named 'sentence generator' in Figure 1 (left side) could be used to run batches of tests. We re-engineered this component and elevated it to the status of one of the RV4Chatbot components. This approach allows testing the chatbot during its development by exploiting the monitor as an offline test engine. The advantage of this method is that once the chatbot has been tested offline and then deployed, the monitor can continue to function at runtime, in line with its primary objective. No code changes are required in the monitor or the instrumented chatbot when switching from offline testing to RV; only the source of sentences changes, becoming a human user in the latter case.

Now that we have completed the introduction of RV4Chatbot, we can focus on its two instantiations for the RV of Rasa and Dialogflow chatbots.

5 RV4Rasa

5.1 Rasa

"*Rasa Open Source is an open source conversational AI platform allowing developers to understand and hold conversations, and connect to messaging channels and third party systems through a set of APIs.*"[4]

Rasa [11] is composed by two different tools: Rasa NLU and Rasa Core. When a message is received from the user, Rasa NLU extracts the intent and the entities (namely, structured pieces of information

[3] We remind the reader that in RV, it is common to have at least a third outcome indicating that the monitor does not yet have sufficient information to determine whether the property under analysis is satisfied or violated.

[4] `https://rasa.com/docs/rasa/`

inside a user message) from it. The structured information is then passed to the Tracker object. This object is used to store the dialogue state. The tracker object is then passed to the policies. Each policy has a ranking and can return a list containing one score for every possible action to perform next. Rasa Core will perform the action with the best score provided by the highest ranked policy. The action server executes the action, the tracker object is updated and then passed again to the policies. When no more actions to be performed are available, the policies return the 'listen' action, waiting for a user input.

Rasa NLU and policies use three files for training:

- `nlu.yml`, containing all the example sentences uses to train the intent classification and the entities extraction;
- `stories.yml`, containing all the paths that a conversation can follow;
- `rules.yml`, containing stricter conversation patterns and actions that must take place if triggered.

Rasa Core employs two main files for the flow control and configuration:

- `domain.yml`, containing all the information on what Rasa NLU can extract (intents and entities), definition of slots (stored values), available actions and responses;
- `config.yml`, containing the pipeline for the Rasa NLU training and the policies definition.

Rasa actions can be defined either as strings to answer or as complete Python classes to be executed when called.

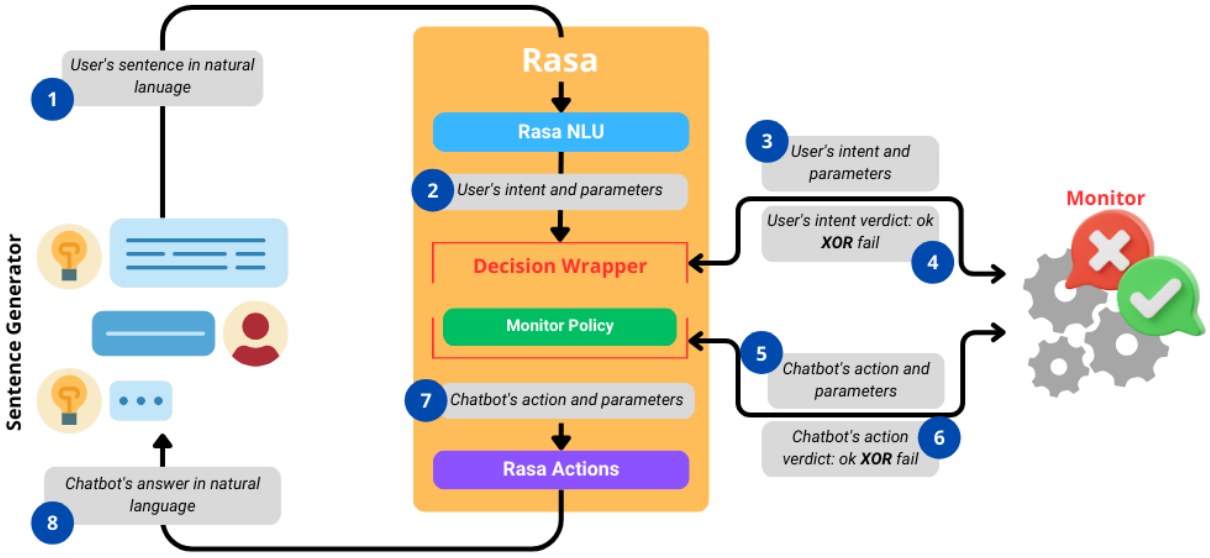


Figure 2: The RV4Rasa instantiation of RV4Chatbot.

5.2 The RV4Rasa instantiation of RV4Chatbot

The Rasa architecture aligns closely with that of RV4Chatbot, as shown in Figure 2. Each element of the RV4Chatbot architecture maps directly to a specific component in the RV4Rasa instantiation, without requiring any modifications to Rasa's original design.

The Sentence Generator can be either the human user using the shell provided by Rasa or a script that sends messages as POST requests to the Rasa server provided by Rasa. The Intent Classifier is the

Rasa NLU that extracts intent and entities. The Decision Wrapper in Rasa is managed by the policies. In particular, for this module it is necessary to add a policy (`monitorPolicy`) that sends an event to the monitor for each action executed. The Actuator overlaps with the Rasa Actions that can perform any piece of provided Python code.

Notice that the policies predict the next action based on the previous ones so the `monitorPolicy` can only stop the chatbot immediately after the wrong action has been executed. This paves the way to *recovery* from wrong actions, but not to *prevention*. However, by exploiting Rasa policies the developer only needs to add the `monitorPolicy` to them, without any other change to the chatbot; RV will be performed automatically thanks to the `monitorPolicy`. The simplicity and the minimal invasiveness of 'injecting' RV capabilities into Rasa this way, motivates our decision to give up prevention, and accept that the monitor realizes that something went wrong, after this already happened. Actually, ex-post notification is a standard operating way in RV.

5.3 Challenges in the RV4Rasa design and development

The main effort required by the RV4Rasa development was understanding how policies work, and implementing the `monitorPolicy`. In fact, whereas Rasa's documentation is extensive and well-assorted for a basic usage, it is almost completely absent when policies come into play. The policy is added in the config file as follows:

```
policies:
...
  - name: policies.monitorPolicy.MonitorPolicy
    priority: 6
    error_action: "utter_error_message"
```

In its implementation the main class, `monitorPolicy`, inherits Rasa's Policy class; in particular, it inherits and redefines:

- `__init__`, the initialisation method, here the error action provided by the user is saved or set to a default value if needed;
- `predict_action_probabilities`, called every time the policy runs and returning the list of probabilities. This method may also return no value at all, and this is exactly the way we use it, to keep the conversation flowing as if no RV were performed, if no errors occur.

Note that, the `priority` assigned to the policy is user-defined and ensures that Rasa gives precedence to the `monitorPolicy` over other custom policies. This higher priority is crucial since the `monitorPolicy` addresses safety aspects, which must take precedence in the chatbot's decision-making process.

5.4 Source Code

To instantiate RV4Rasa there are only two additions to be made in the chatbot:

1. `monitor_policy.py`: 150 lines of code;
2. `config.yml`: 3 further lines should be added to the configuration file, to turn Rasa into RV4Rasa.

The code of RV4Rasa is available at `https://github.com/driacats/RV4Chat/tree/main/Rasa`.

6 RV4Dialogflow

6.1 Dialogflow

Dialogflow [27] is a lifelike conversational AI platform developed by Google that enables users to create virtual agents equipped with intents, entities (similar to those in Rasa), and fulfillment. Fulfillment refers to the capability of these agents to interact with external systems or APIs to retrieve dynamic responses, process data, or execute specific actions based on the user's input, going beyond pre-defined static responses.

Dialogflow performs NLU using *intents*, defined via a name and a set of training example sentences. Dialogflow trains a model able to identify, for each user message sent on the chat, the *nearest* intent and the confidence score. Training sentences may also contain *entities*, namely pieces of information that may be significant for the conversation and that should be extracted from the text. For each intent, a bunch of possible answers may be displayed. However, some messages cannot be answered from inside Dialogflow, as they require to process data or execute operations. In this case, users can use the *fulfillment* for sending a message to an external server that will execute the correct actions, and provide an answer back.

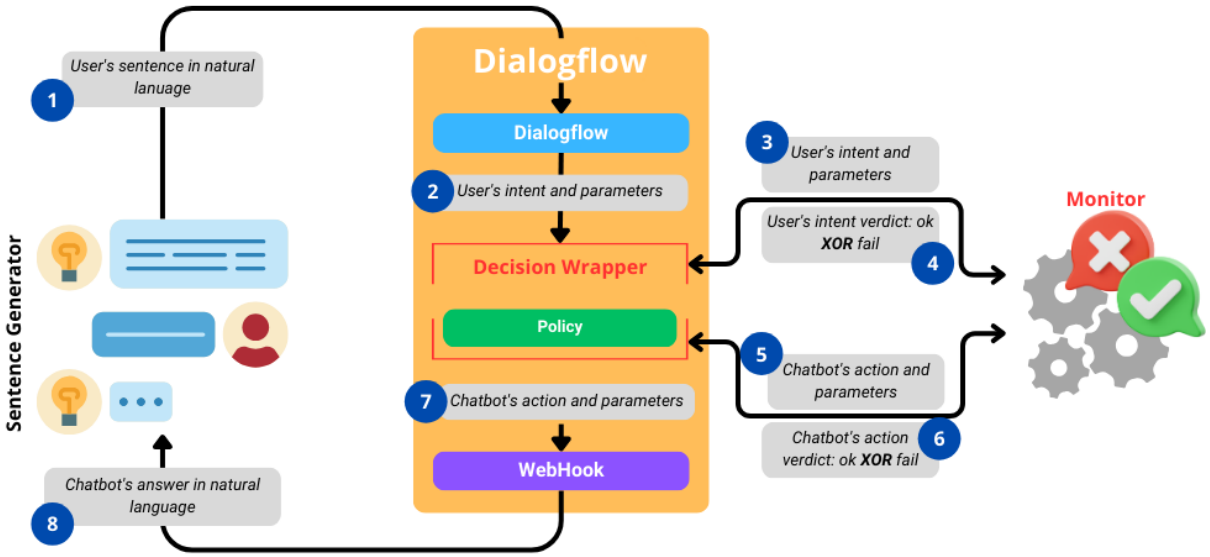


Figure 3: The RV4Dialogflow instantiation of RV4Chatbot.

6.2 The RV4Dialogflow instantiation of RV4Chatbot

RV4Dialogflow is structured as shown in Figure 3. To instantiate RV4Chatbot in Dialogflow, we had to add a brand new component to the Dialogflow architecture. This made the design and implementation of RV4Dialogflow much more complex than the RV4Rasa one.

This additional component can be generated directly from an exported Dialogflow agent using an instrumentation script that we developed. We call this brand new component *policy*, for analogy with RV4Rasa.

The instrumentation script has two outputs: a .zip file containing the new Dialogflow agent, and the policy python script. In the new Dialogflow agent, every message is forwarded to the policy that controls

the flow. The policy maintains the original agent's flow, forwarding only the necessary messages back to DialogFlow through a webhook[5]. Additionally, for each message and action performed, the policy sends a message to the monitor.

6.3 Challenges in the RV4Dialogflow design and development

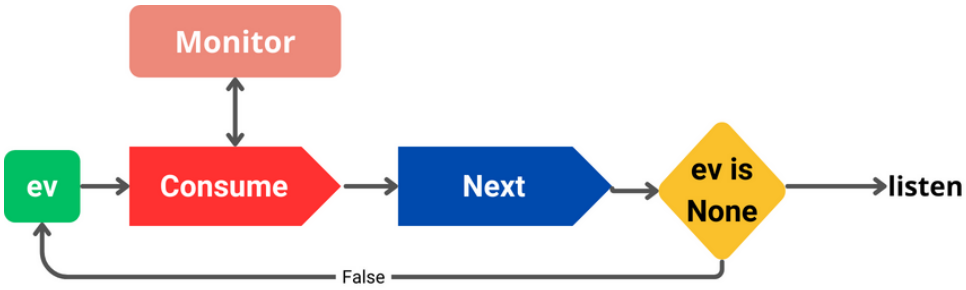


Figure 4: RV4Dialogflow policy flow.

The RV4Dialogflow policy works with *events* as shown in Figure 4 (reported as ev). There are two main types of events: user and bot events.

The policy can receive messages both from the user or the bot. When it receives a message it creates an event. An event in this domain can be of five main types: (1) a user message with all its features; (2) a bot message with all its features; (3) a plain answer to be sent as answer; (4) an action to be performed by the webhook; (5) the error action.

The event is then consumed: it is sent to the monitor and then if it is a plain answer it is sent on the chat, if it is an action it is performed. Obviously, if the monitor claims an error the action is set immediately to the error one.

When the event is consumed the policy computes the next one. For examples, if the event is a user message the next event is the answer. The policy consumes and computes next event until the next event is None, in this case it listens for new inputs.

6.4 Source Code

To instantiate RV4Dialogflow the needed files are:

1. `instrumenter.py`: 190 lines of code;
2. `policy.py`: (generated by the instrumenter) from 150 lines of code;

The code of RV4Dialogflow is available at `http://github.com/driacats/RV4Chat/tree/main/Dialogflow`.

7 Experiments

The formalism we use to model properties to be verified at runtime is named Runtime Monitoring Language (RML) and has been selected for its high expressive power that goes beyond Linear Temporal

[5]Which is the mechanism used in DialogFlow to communicate to DialogFlow from an external service, that in RV4Dialogflow is the policy.

Logic (LTL) [5] and the familiarity of the authors. As explained in Section 4, the RV4Chatbot framework is meant to be instantiated with any conversational chatbot framework and any RV language and tool. RML is one among the many existing RV languages and its adoption is only functional to run experiments with RV4Rasa and RV4Dialogflow.

In this section, we briefly introduce RML and illustrate the RML properties that capture safety requirements in the factory automation case study, providing the RML encoding of one of the properties verified in that scenario (the complete encoding can be found in [22]). All these properties are correctly verified by the monitor, so we do not allocate space to the qualitative experiments we conducted, as they can be summarised by stating that "the monitor always works as expected". Instead, we present performance experiments, demonstrating that the addition of the monitor to the chatbot introduces negligible overhead.

7.1 Runtime Monitoring Language

The Runtime Monitoring Language (RML [4, 6]) is a Domain-Specific Language (DSL) for specifying highly expressive properties in RV such as non context-free ones. We chose to use RML in this work because of its support of parametric specifications and its native use for defining interaction protocols.

Since RML is just a means for our purposes, we only provide a condensed view of its syntax and denotational semantics in terms of the represented traces of events. A detailed explanation of some of its operators is provided in Section 7.2 where RML specifications are provided. The complete presentation can be found in [6].

In RML, a property is expressed as a tuple $\langle t, ETs \rangle$, with t a term and $ETs = \{ ET_1, \ldots, ET_n \}$ a set of event types. An event type ET is represented as a set of pairs $\{ k_1 : v_1, \ldots, k_n : v_n \}$, where each pair identifies a specific piece of information (k_i) and its value (v_i). An event Ev is denoted as a set of pairs $\{ k'_1 : v'_1, \ldots, k'_m : v'_m \}$. Given an event type ET, an event Ev matches ET if $ET \subseteq Ev$, which means $\forall (k_i : v_i) \in ET \cdot \exists (k_j : v_j) \in Ev \cdot k_i = k_j \wedge v_i = v_j$. In other words, an event type ET specifies the requirements that an event Ev has to satisfy to be considered valid.

An RML term t, with t_1, t_2 and t' as other RML terms, can be:

- ET, denoting a set of singleton traces containing the events Ev s.t. $ET \subseteq Ev$;
- $t_1 \; t_2$, denoting the sequential composition of two sets of traces;
- $t_1 \mid t_2$, denoting the unordered composition of two sets of traces (also called shuffle or interleaving);
- $t_1 \wedge t_2$, denoting the intersection of two sets of traces;
- $t_1 \vee t_2$, denoting the union of two sets of traces;
- $\{ \; let \; x; \; t' \; \}$, denoting the set of traces t' where the variable x can be used (i.e., the variable x can appear in event types in t', and can be unified with values);
- $t'*$, denoting the set of chains of concatenations of traces in t'.

Event types can contain variables (we use the terms *argument* and *variable* interchangeably). In RML, recursion is modelled by syntactic equations involving RML terms, such as $t = ET_1 \; t \vee ET_2$, modeling a finite (possibly empty) sequence of events matching the event type ET_1 ended by one event matching ET_2, or the infinite trace including only events matching ET_1.

7.2 Factory Automation Domain properties

The three properties to be verified in the factory automation domain have been presented in the VORTEX 2023 paper [22]. We report one of them to better clarify the use of RML and the kind of protocols we are interested in verifying at runtime.

The first property aims at ensuring that the user does not add an object in an already taken position. The corresponding RML specification is reported in the following (as in [22]).

$$
\begin{aligned}
AddObject = \{\ & let\ x, y; \\
& (msg\_user\_to\_bot \wedge add\_object(x,y)) \\
& (msg\_bot\_to\_user \wedge object\_added) \\
& (not\_add\_object(x,y)* \wedge AddObject)\ \} \\
ETs = \{\ & msg\_user\_to\_bot, \\
& msg\_bot\_to\_user, add\_object(x,y), \\
& object\_added\ \} \\
msg\_user\_to\_bot = \{\ & sender: \text{“}user\text{”},\ receiver: \text{“}bot\text{”}\ \} \\
msg\_bot\_to\_user = \{\ & sender: \text{“}bot\text{”},\ receiver: \text{“}user\text{”}\ \} \\
add\_object(x,y) = \{\ & intent: \{\ name: \text{“}add\_object\text{”}\ \}, \\
& slots: \{\ horizontal: x, vertical: y\ \}\ \} \\
object\_added = \{\ & last\_action: \text{“}utter\_add\_object\text{”}\ \}
\end{aligned}
$$

As an example, the user's request *"Add a robot in position (3, 5)"* is safe only if the position *(3, 5)* is empty. But, the position is empty if the user did not already ask to put objects there. Hence, the history of the previous interactions must be taken into account, to verify the feasibility of a new object addition. The property is parametric w.r.t. coordinates and is defined recursively; it involves the definition of four event types and exploits the *let*, $\wedge$, and $*$ RML operators.

Figure 5 presents screenshots of the simulated environment in which the Rasa (and Dialogflow) chatbots operate. These screenshots specifically demonstrate how, by interacting with the chatbot, the user can add new objects to the simulated factory floor. This interaction occurs under the scrutiny of the RML monitor, which checks, among other things, the previously mentioned property.

The second property, whose RML encoding is more complex than the previous one, deals with the addition of an object in a position which is relative to another object in the simulation. The user's request may be *"Add a robot on the left of Robot3"*. In order to be a safe request, *Robot3* should have been previously positioned somewhere, hence previous messages involving added and removed objects, their name, and their position in the simulation must be taken into account. The property is parametric w.r.t. coordinates and objects names, and defined recursively; it involves eight event types and exploits the $|$ and $\vee$ RML operators, besides *let*, $\wedge$, and $*$. The $|$ operator is used, for example, to cope with the interleaving of future additions relative to the currently added object and additions of objects that are not relative to the newly added one. Disjunction is used to discriminate between the situation where one added object is then removed, and hence no further references to it are allowed, and the situation where no removal takes place, and references are safe.

The third property exploits the RML feature of constraining values in event types. It checks that any observed message has the value associated with its *confidence* field – as returned by the NLU component of the chatbot – greater than 60%.

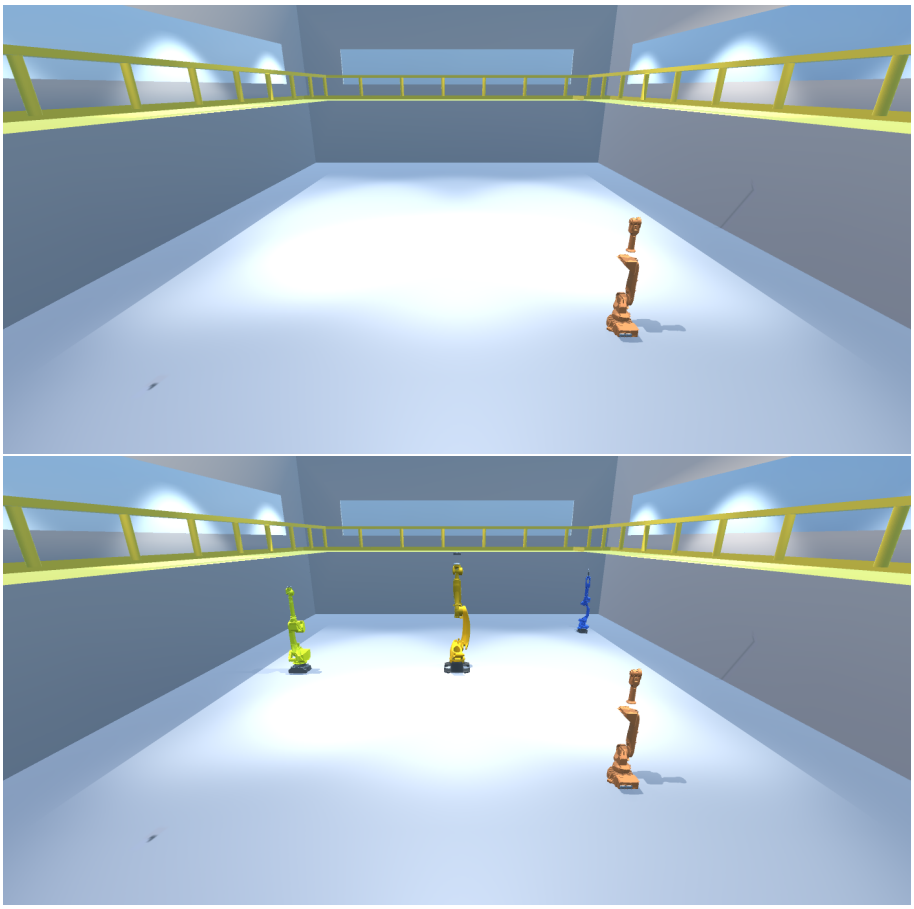


Figure 5: Initial scenario of the simulated factory floor (above) and the result after further iterations of adding new objects in the scene (below) taken from [24].

7.3 Performance Evaluation

All the experiments can be tested using the code provided here `https://github.com/driacats/RV4Chat/tree/main/Examples`. In particular there are three important scripts for each experiment:

- `start_service.py`: this script allows the user to select the platform (Rasa, Dialogflow), the monitor (no monitor, dummy monitor, real monitor), and the scenario (factory automation), and starts the service;
- `run_test.py`: this script launches the test conversation. The input messages are stored in the `test_input.txt` file, and the conversation is iterated a certain number of times for each combination of platform and monitor. For each iteration, a file is created to store the response times for each message sent;
- `chat.py`: this script provides a chat interface to test the chatbot. It takes as argument the platform and manages the connection automatically. To test the program it is sufficient to start the service and then launch the chat with the same platform.

The tests have been performed using RV4Rasa and RV4Dialogflow with three different monitoring levels:

1. without a monitor;

2. with a dummy monitor that replies always `True`;
3. with a real monitor that checks the properties discussed in the previous sections.

For this experiment the chatbot can identify three intents and five entities. The three intents are (1) add an object (2) add a object with a reference to another object (3) remove an object, while the entities are (1) object to add or remove (2) vertical position (3) horizontal position (4) relative position (5) reference object.

The Dialogflow WebHook in this case is more complex and manages the addition and the removal of objects inside a real virtual environment. The implementation of this experiment in Rasa, with a real Virtual Environment in the backend and a Multi-Agent System in the middle, has been presented in [22]. For the tests presented here, the API calls that in [22] accessed the virtual environment are instead sent to a dummy script that provides a terminal based representation of a virtual space and simulates the execution. No virtual environment implementation is involved in this experiment, which is aimed at testing the performance of the RV mechanism.

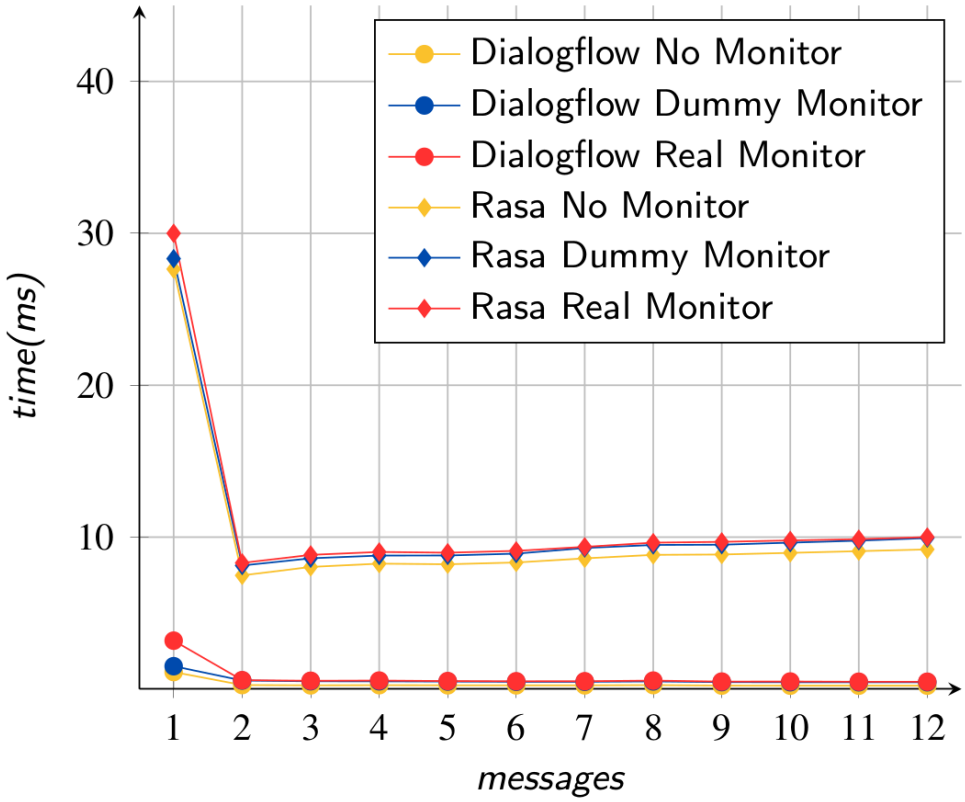


Figure 6: Factory Automation Domain times on a test conversation of 12 messages. Messages for the test are: (1) *Add a table* (2) *Add a box right of table1* (3) *Add a robot in front on the left* (4) *Add a robot in front on the right* (5) *Remove box0* (6) *Remove robot1* (7) *Add a table behind on the left* (8) *Add a robot behind on the right* (9) *Remove table1* (10) *Remove table2* (11) *Remove robot2* (12) *Remove robot3*.

As shown in Figure 6, the monitor does not affect the execution time of the chatbot. The first message exhibits a significant time delay compared to the subsequent messages when using Rasa. This behaviour is due to Rasa itself: the Rasa Tracker object and all necessary instances for the conversation are ini-

tialised with the first message rather than at the server launch, resulting in a significantly higher time required to process the first message compared to the others.

8 Conclusions and Future Work

This paper introduces RV4Chatbot, a framework for verifying the behaviour of conversational AI chatbots. RV4Chatbot achieves this in a versatile manner, imposing minimal constraints on both the chatbot creation framework and the monitors deployed at runtime for formal verification. To demonstrate its efficacy, this paper presents two implementations of RV4Chatbot: RV4Rasa and RV4Dialogflow. The engineering and experimental outcomes of these implementations are detailed, particularly when applied to safety-critical case studies in domains such as factory automation.

The experimental findings underscore RV4Chatbot's generality, efficiency, and lightweight nature in terms of the overhead introduced by its monitoring components.

Looking ahead, our plans involve further exploration and experimentation with RV4Chatbot, including its application to more complex case studies that can better challenge the framework's robustness and performance. This will allow us to assess how the performance overhead of RV4Chatbot is impacted when applied to larger, real-world conversational systems with increased message volumes, more complex dialogue flows, and higher interaction frequencies. Additionally, while our current focus is on conversational AI chatbots, we plan to evaluate the scalability of RV4Chatbot to understand how it performs as the number of monitored properties, intents, and concurrent conversations grows. Preliminary intuition suggests that the framework's modularity may support scaling to moderately large applications, but this hypothesis needs to be tested empirically.

Furthermore, the insights and experiences gained from this work may facilitate future developments for handling generative chatbots. In such scenarios, where intents may be unavailable and decision-making is based on machine learning techniques, we aim to refine and expand RV4Chatbot to integrate more dynamic monitoring approaches that can accommodate the unpredictability and complexity of generative AI.

References

[1] Abubakar Abid, Maheen Farooqi & James Zou (2021): *Persistent Anti-Muslim Bias in Large Language Models*. In: *AIES*, ACM, pp. 298–306, doi:10.1145/3461702.3462624.

[2] Eleni Adamopoulou & Lefteris Moussiades (2020): *Chatbots: History, technology, and applications*. *Machine Learning with Applications* 2, p. 100006, doi:10.1016/j.mlwa.2020.100006.

[3] Hind Alotaibi & Hussein Zedan (2010): *Runtime verification of safety properties in multi-agents systems*. In: *10th International Conference on Intelligent Systems Design and Applications, ISDA 2010, November 29 - December 1, 2010, Cairo, Egypt*, IEEE, pp. 356–362, doi:10.1109/ISDA.2010.5687238.

[4] Davide Ancona, Angelo Ferrando, Luca Franceschini & Viviana Mascardi: *RML web site*. Available at `https://rmlatdibris.github.io/`. Accessed on November 16, 2024.

[5] Davide Ancona, Angelo Ferrando & Viviana Mascardi (2016): *Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification*. In: *Theory and Practice of Formal Methods*, *LNCS* 9660, Springer, pp. 47–64, doi:10.1007/978-3-319-30734-3_6.

[6] Davide Ancona, Luca Franceschini, Angelo Ferrando & Viviana Mascardi (2021): *RML: Theory and practice of a domain specific language for runtime verification*. *Sci. Comput. Program.* 205, p. 102610, doi:10.1016/j.scico.2021.102610.

[7] Najwa Abu Bakar & Ali Selamat (2013): *Runtime Verification of Multi-agent Systems Interaction Quality*. In: *Intelligent Information and Database Systems - 5th Asian Conf., ACIIDS 2013*, *LNCS* 7802, Springer, Berlin, Heidelberg, pp. 435–444, doi:10.1007/978-3-642-36546-1_45.

[8] Ezio Bartocci, Yliès Falcone, Adrian Francalanza & Giles Reger (2018): *Introduction to Runtime Verification*. In: *Lectures on Runtime Verification - Introductory and Advanced Topics*, *LNCS* 10457, Springer, pp. 1–33, doi:10.1007/978-3-319-75632-5_1.

[9] Andreas Bauer & Jan Jürjens (2010): *Runtime verification of cryptographic protocols*. *computers & security* 29(3), pp. 315–330, doi:10.1016/j.cose.2009.09.003.

[10] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson & Wang Yi (1995): *UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems*. In: *DIMACS/SYCON WS on Verification and Control of Hybrid Systems*, *LNCS* 1066, Springer, pp. 232–243, doi:10.1007/BFB0020949.

[11] Tom Bocklisch, Joey Faulkner, Nick Pawlowski & Alan Nichol (2017): *Rasa: Open Source Language Understanding and Dialogue Management*. *CoRR* abs/1712.05181, doi:10.48550/arXiv.1712.05181. arXiv:1712.05181.

[12] Botium: *Bots Testing Bots*. Available at `https://botium-docs.readthedocs.io/en/latest/`. Accessed on November 16, 2024.

[13] Josip Bozic (2022): *Ontology-based metamorphic testing for chatbots*. *Softw. Qual. J.* 30(1), pp. 227–251, doi:10.1007/s11219-020-09544-9.

[14] Josip Bozic, Oliver A. Tazl & Franz Wotawa (2019): *Chatbot Testing Using AI Planning*. In: *IEEE Int. Conf. On Artificial Intelligence Testing, AITest 2019*, IEEE, pp. 37–44, doi:10.1109/AITest.2019.00-10.

[15] Josip Bozic & Franz Wotawa (2019): *Testing Chatbots Using Metamorphic Relations*. In: *Testing Software and Systems - 31st IFIP WG 6.1 Int. Conf., ICTSS 2019*, *LNCS* 11812, Springer, pp. 41–55, doi:10.1007/978-3-030-31280-0_3.

[16] Sergio Bravo-Santos, Esther Guerra & Juan de Lara (2020): *Testing Chatbots with Charm*. In: *Quality of Information and Communications Technology - 13th Int. Conf., QUATIC 2020*, *CCIS* 1266, Springer, pp. 426–438, doi:10.1007/978-3-030-58793-2_34.

[17] Xiaoyi Chen, Siyuan Tang, Rui Zhu, Shijun Yan, Lei Jin, Zihao Wang, Liya Su, XiaoFeng Wang & Haixu Tang (2023): *The Janus Interface: How Fine-Tuning in Large Language Models Amplifies the Privacy Risks*. *CoRR* abs/2310.15469, doi:10.48550/ARXIV.2310.15469. arXiv:2310.15469.

[18] Bella Church (2023): *5 types of chatbot and how to choose the right one for your business*. Available at `https://www.ibm.com/blog/chatbot-types/`. Accessed on November 16, 2024.

[19] Edmund M Clarke (1997): *Model checking*. In: *Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, Springer, pp. 54–56, doi:10.1007/BFb0058022.

[20] Debora C Engelmann, Angelo Ferrando, Alison R Panisson, Davide Ancona, Rafael H Bordini & Viviana Mascardi (2023): *RV4JaCa — Towards Runtime Verification of Multi-Agent Systems and Robotic Applications*. *Robotics* 12(2), p. 49, doi:10.3390/robotics12020049.

[21] European Parliament (2023): *Artificial Intelligence Act*. Available at `https://www.europarl.europa.eu/news/en/press-room/20231206IPR15699/artificial-intelligence-act-deal-on-comprehensive-rules-for-trustworthy-ai`. Accessed on November 16, 2024.

[22] Angelo Ferrando, Andrea Gatti & Viviana Mascardi (2023): *RV4Rasa: A Formalism-Agnostic Runtime Verification Framework for Verifying ChatBots in Rasa*. In: *6th Int. WS on Verification and Monitoring at Runtime Execution, VORTEX 2023*, ACM, pp. 1–8, doi:10.1145/3605159.3605855.

[23] Asbjørn Følstad, Marita Skjuve & Petter Bae Brandtzæg (2018): *Different Chatbots for Different Purposes: Towards a Typology of Chatbots to Understand Interaction Design*. In Svetlana S. Bodrunova, Olessia Koltsova, Asbjørn Følstad, Harry Halpin, Polina Kolozaridi, Leonid Yuldashev, Anna S. Smoliarova & Heiko

Niedermayer, editors: *Internet Science - INSCI 2018 International Workshops, St. Petersburg, Russia, October 24-26, 2018, Revised Selected Papers*, *Lecture Notes in Computer Science* 11551, Springer, pp. 145–156, doi:10.1007/978-3-030-17705-8_13.

[24] Andrea Gatti & Viviana Mascardi (2023): *VEsNA, a Framework for Virtual Environments via Natural Language Agents and Its Application to Factory Automation*. *Robotics* 12(2), p. 46, doi:10.3390/ROBOTICS12020046.

[25] Sousa S. Geovana Ramos, Nunes R. Genaína & Dias C. Edna (2023): *A Modeling Strategy for the Verification of Context-Oriented Chatbot Conversational Flows via Model Checking*. *Journal of Universal Computer Science* 29(7), pp. 805–835, doi:10.3897/jucs.91311.

[26] Global Information, Inc. – GII (2024): *Global Large Language Model (LLM) Market Research Report*. Available at `https://www.giiresearch.com/report/qyr1384359-global-large-language-model-llm-market-research.html`. Accessed on November 16, 2024.

[27] Google: *DialogFlow: Online Resource, `https://cloud.google.com/dialogflow/`*. Available at `https://cloud.google.com/dialogflow/`.

[28] Google: *Dialogflow web site*. Available at `https://cloud.google.com/dialogflow`. Accessed on November 16, 2024.

[29] Google: *Gemini web site*. Available at `https://gemini.google.com/`. Accessed on November 16, 2024.

[30] Cobus Greyling (2023): *Conversational UIs & LLMs*. Available at `https://cobusgreyling.medium.com/large-language-model-llm-disruption-of-chatbots-8115fffadc22`. Accessed on November 16, 2024.

[31] Jasper AI: *Jasper web site*. Available at `https://www.jasper.ai/chat`. Accessed on November 16, 2024.

[32] Jaeho Jeon, Seongyong Lee & Hohsung Choe (2023): *Beyond ChatGPT: A conceptual framework and systematic review of speech-recognition chatbots for language learning*. *Comput. Educ.* 206, p. 104898, doi:10.1016/j.compedu.2023.104898.

[33] Hadas Kotek, Rikker Dockum & David Q. Sun (2023): *Gender bias and stereotypes in Large Language Models*. In: *The ACM Collective Intelligence Conf., CI 2023*, ACM, pp. 12–24, doi:10.1145/3582269.3615599.

[34] Zheng Li, Yan Jin & Jun Han (2006): *A runtime monitoring and validation framework for web service interactions*. In: *Australian Software Engineering Conf. (ASWEC'06)*, IEEE, pp. 10–pp, doi:10.1109/ASWEC.2006.6.

[35] Xiaolin Lin, Bin Shao & Xuequn Wang (2022): *Employees' perceptions of chatbots in B2B marketing: Affordances vs. disaffordances*. *Industrial Marketing Management* 101, pp. 45–56, doi:10.1016/j.indmarman.2021.11.016. Available at `https://www.sciencedirect.com/science/article/pii/S001985012100242X`.

[36] Donald W. Loveland (1978): *Automated theorem proving: a logical basis*. *Fundamental studies in computer science* 6, North-Holland.

[37] MarketsandMarkets (2023): *Conversational AI Market*. Available at `https://www.marketsandmarkets.com/Market-Reports/conversational-ai-market-49043506.html`. Accessed on November 16, 2024.

[38] Meta: *Wit.ai web site*. Available at `https://wit.ai/`. Accessed on November 16, 2024.

[39] Martin Mitrevski (2018): *Getting started with wit.ai*, doi:10.1007/978-1-4842-3396-2_5.

[40] Open AI (2022): *Introducing ChatGPT*. Available at `https://openai.com/blog/chatgpt`. Accessed on November 16, 2024.

[41] Rasa technologies: *Rasa web site*. Available at `https://rasa.com/`. Accessed on November 16, 2024.

[42] Navin Sabharwal, Amit Agrawal, Navin Sabharwal & Amit Agrawal (2020): *Introduction to Google Dialogflow*.

[43] Sanjit A. Seshia, Ankush Desai, Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte & Xiangyu Yue (2018): *Formal Specification for Deep Neural Networks*. In: *Automated Technology for Verification and Analysis - 16th Int. Symposium, ATVA 2018*, *LNCS* 11138, Springer, pp. 20–34, doi:10.1007/978-3-030-01090-4_2.

[44] Jin Shao, Hao Wei, Qianxiang Wang & Hong Mei (2010): *A Runtime Model Based Monitoring Approach for Cloud*. In: *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*, IEEE Computer Society, pp. 313–320, doi:10.1109/CLOUD.2010.31.

[45] Technical Committee:ISO/TC 299 Robotics (2011): *Robots and robotic devices – Safety requirements for industrial robots*. Standard.

[46] Xianjun Yang, Xiao Wang, Qi Zhang, Linda R. Petzold, William Yang Wang, Xun Zhao & Dahua Lin (2023): *Shadow Alignment: The Ease of Subverting Safely-Aligned Language Models*. *CoRR* abs/2310.02949, doi:10.48550/ARXIV.2310.02949. arXiv:2310.02949.

[47] Zheng Xin Yong, Cristina Menghini & Stephen H. Bach (2023): *Low-Resource Languages Jailbreak GPT-4*. *CoRR* abs/2310.02446, doi:10.48550/ARXIV.2310.02446. arXiv:2310.02446.

[48] Yue Zhang, Yafu Li, Leyang Cui & et al. (2023): *Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models*. arXiv:2309.01219.

Model Checking and Verification of Synchronisation Properties of Cobot Welding

Yvonne Murray
Pioneer Robotics AS
Dept. of Mechatronics, University of Agder
Norway
ym@pioneer-robotics.no

Henrik Nordlie
Robotics Group, Faculty of Science & Technology
Norwegian University of Life Sciences (NMBU)
Norway

David A. Anisi
Dept. of Mechatronics, University of Agder
Robotics Group, Faculty of Science & Technology
Norwegian University of Life Sciences (NMBU)
Norway

Pedro Ribeiro
Dept. of Computer Science
University of York
UK

Ana Cavalcanti
Dept. of Computer Science
University of York
UK

This paper describes use of model checking to verify synchronisation properties of an industrial welding system consisting of a cobot arm and an external turntable. The robots must move synchronously, but sometimes get out of synchronisation, giving rise to unsatisfactory weld qualities in problem areas, such as around corners. These mistakes are costly, since time is lost both in the robotic welding and in manual repairs needed to improve the weld. Verification of the synchronisation properties has shown that they are fulfilled as long as assumptions of correctness made about parts outside the scope of the model hold, indicating limitations in the hardware. These results have indicated the source of the problem, and motivated a re-calibration of the real-life system. This has drastically improved the welding results, and is a demonstration of how formal methods can be useful in an industrial setting.

1 Introduction

Robotic welding is commonly used in industrial workshops to increase efficiency and repeatability, and reduce dangerous and ergonomically straining work for human welders [12]. To address the needs of small and medium-sized enterprises (SMEs), which produce a large variety of products in small quantities, the welding system must be easy and fast to re-program, and highly flexible. To this end, Pioneer Robotics have developed the IntelliWelder M06 [17], a flexible and light-weight welding system consisting of a Universal Robots (UR) UR10e cobot [8] equipped with a welding torch and a Carpano FIVE MOT turntable serving as an external axis (EXAX). Fig. 1 shows the components of the IntelliWelder.

The main challenge in the operation of the IntelliWelder M06 has been to get a high quality, continuous weld in difficult areas, such as around corners. To get the best results, it is important that the welding robot and the turntable move continuously in a *synchronous* fashion. By using synchronous welding, it is possible to achieve a continuous weld of high quality, ensuring the weld is not jagged and interrupted. When the synchronisation does not work properly, the welding gun does not move forward in an even motion, and can move too fast or too slow. Moving too fast does not give the metal and filler enough time to heat up and weld together, while moving too slow results in build-up of filler material. Both of these problems can be seen in the weld depicted in Fig. 2.

This experience report describes how we have addressed some challenges faced when the UR robot and the EXAX move synchronously while welding. Relevant parts of the system have been modelled in

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 91–108, doi:10.4204/EPTCS.411.6

© Y. Murray, H. Nordlie, D.A. Anisi, P. Ribeiro & A. Cavalcanti
This work is licensed under the
Creative Commons Attribution License.

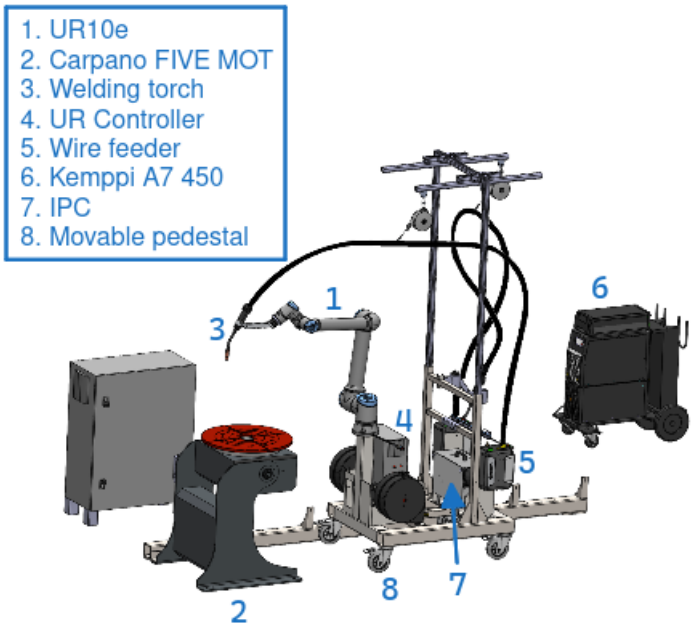


Figure 1: The IntelliWelder system with the different components marked by number [16].

Figure 2: Typical welding issue where there is buildup of filler material (A) and coverage is not sufficient (B), creating an irregular and weakened weld.

RoboChart [14], a domain-specific language for modelling and verification of robotic systems. Using our RoboChart model, key synchronisation properties have been verified using the refinement model checker FDR [25]. Details omitted here are in [16]; the work demonstrates how formal verification can be used in industry, and how the results can be used to localise the error source, leading to system improvements.

Previous research on multi-robot welding include [20], which focuses on nominal trajectory planning and self-coordination, and [27], which studies trajectory smoothing in a dual-robot collaborative welding system. Neither of these lines of work use formal methods or model checking. Closer to our research, the work in [19] combines graphic and formal methods to analyse collaborative behaviour such as deadlock and equivalence properties. None of these works, however, consider the issue of correct time synchronisation during multi-robot execution like we do in our case study.

The rest of this paper is organised as follows. In Section 2, we motivate our use of formal methods and model checking. In Section 3, we detail the system architecture and requirements, before the model is presented in Section 4. In Section 5, we present the verification results and their practical implications. Finally, in Section 6, we conclude, describe ongoing work, and suggest further work.

2 Formal Verification and Model Checking

To find and mitigate faults and undesired behaviour in robotic systems, they are traditionally subject to testing, including simulation before deployment. For real-world, complex robotic systems, however, it is impossible to test every possible scenario and input sequence. Moreover, even if a fault is discovered, error source localisation remains a challenge. In this setting, formal verification methods are a useful supplement. Model checking [1, 7] is a formal method to verify that given properties are fulfilled, regardless of inputs. If a property does not hold, model checking provides a counterexample that can pinpoint the cause of error. Adopting such methods is valuable in the design of real, industrial systems.

RoboTool [14] is a suite of plugins for the Eclipse IDE supporting use of the RoboStar framework [5]. Our previous work on verification of an industrial control system [15] using RoboStar has shown its proficiency and strengths. In RoboStar, a key artefact is a RoboChart [13] model that reflects the real system design. Once this model is created, assertions for the selected properties can be written and verified using the CSP process algebra and its model checker, FDR [10].

As our use case considers an already existing IntelliWelder system, we need to alter the idealised workflow of RoboStar [5] by effectively "reverse engineering" the RoboChart model from the existing system.

3 IntelliWelder and Synchronous Welding

In this section, we describe our case study: its architecture (Section 3.1), software (Section 3.2), and requirements (Section 3.3), identifying the problem we are addressing with model checking.

3.1 System Architecture

An illustration of the IntelliWelder's architecture can be seen in Fig. 3. To realise synchronous welding, Delfoi offline robot programming software from Visual Components [26] is used for creating waypoints and welds in a 3D layout consisting of the UR10e [22], the Carpano FIVE turntable, and the workpiece to be welded. Welds are created simply by selecting an edge on the workpiece CAD model. Delfoi then creates waypoints for both the UR robot and the Carpano turntable, so that each waypoint for the robot has a corresponding waypoint for the turntable, creating nominally synchronised movements.

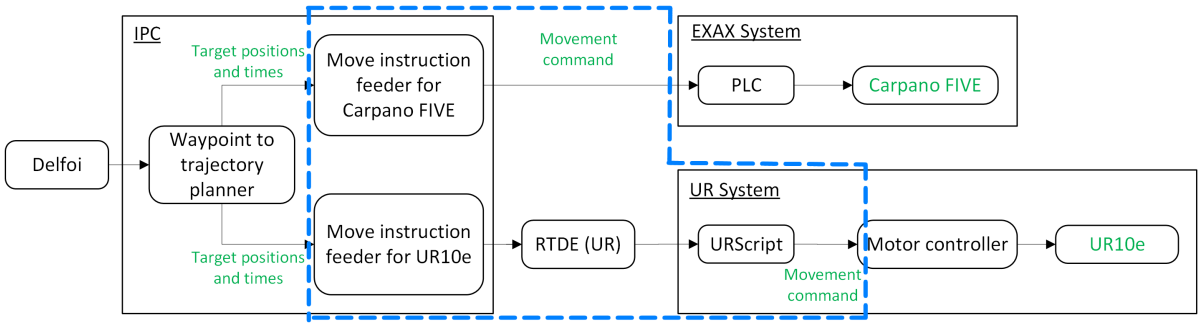


Figure 3: System architecture of the IntelliWelder. The blue dotted line indicates the scope of the RoboChart model: part of the Industrial PC (IPC), the Real-Time Data Exchange (RTDE) for the UR robot, and the URScript.

The waypoint paths generated in Delfoi are transferred to the Industrial PC (IPC). The waypoints are then processed to convert them into trajectories based on the desired forward welding speed and other

welding parameters. The waypoints for the external axis remain unchanged and are converted into a trajectory, while the waypoints for the UR robot are sampled at a higher resolution before being turned into its trajectory. Consequently, the UR robot has more waypoints to process than the Carpano FIVE. Once the trajectories are generated, the IPC sends them as movement requests for the system to execute.

The Carpano FIVE is controlled by a Programmable Logic Controller (PLC) that receives movement commands from the IPC. These commands can be based on position and velocity, or just velocity. The PLC then regulates movement using a PID. The Real-Time Data Exchange (RTDE) synchronises external applications with the UR robot [23]. It relays messages from the IPC to the UR robot via a TCP/IP connection. The UR robot controller executes URScript applications and manages movement via a PID.

3.2 Existing URScript code

In the current implementation, every time a movement request for the UR robot arrives, the URScript code runs on the UR controller. Required variables are read from the registers, updated by the RTDE, and the code checks if the target time for the next waypoint has already passed (that is, the robot is behind schedule). If so, the code logs a warning and continues to the next target.

Next, the URScript decides which movement type is preferable to reach the next waypoint, based on variables like blend radius, offset, joint velocities, and whether the next movement involves a sharp turn. The script selects between MoveJ, MoveL and MoveP, which are standard UR robot movements described in the user manual [24]. However, if none of the standard movement types are suitable, a custom-made function called MoveL_with_t is used for the movement. The custom URScript function MoveL_with_t uses the standard MoveL command with the next target pose and next target time as arguments. In that way, the UR robot can calculate the necessary velocity to reach the next target within the target time, using maximum acceleration. The reason for this being a fallback solution, only used when the standard movements are not feasible, is that it does not include a blend radius.

With a blend radius, we ensure that when the UR robot is within a given distance of the waypoint, it starts moving towards the next waypoint instead of completely finishing the move to the current waypoint. So instead of coming to a brief stop at the waypoint, it keeps moving towards the next, giving a smoother transition. Lack of a blend radius results in a jagged movement that is not ideal for welding.

The next section describes the requirements that the design just presented is expected to satisfy.

3.3 Requirements

We present here both system requirements (in Section 3.3.1), and requirements specifically for the components that we model as described in Figure 3 (in Section 3.3.2).

3.3.1 System-Wide Requirements

There are several requirements for the IntelliWelder system as a whole, discussed in detail in [16]. The most important of these requirements are the following two:

1. The welding torch must always stay in an area defined by a maximum deviation from the weld frame. This includes both position and orientation.

2. The welding torch must always move forward in the weld frame with a speed that is within a given maximum deviation of the desired forward weld speed.

These requirements need to be refined into specific requirements for Delfoi, the IPC planner, the calculation of arguments for the robot commands, and the execution of movements. Additionally, they expand to include requirements related to information communication and code execution time.

3.3.2 Model Requirements

With the system wide requirements in mind, the following requirements for the modelled component (see Figure 3) can be obtained, as described in detail in [16]:

R1 The component should detect events that imply that the system is out of sync.

R2 For each movement request received from the IPC, the corresponding robot should receive a movement command unless the system is out of sync.

The requirements R1 and R2 above are the properties we verify using model checking. If some of the assertions fail, it can help to pinpoint existing mistakes in the software. If all assertions pass, it indicates that some of the assumptions made on the component's context and the hardware are invalid.

To check the hardware, two different cases are evaluated: one where it is impossible for the robot to receive a waypoint that is already in the past (nominal case), and one where that is possible (realistic case). If the model checking results vary between the two, for example, if the assertions pass in the nominal case (which assumes the hardware is able to keep to the planned trajectory) but fail in the more realistic case, it is an indication that assumptions about velocities, accelerations, and perfect move execution, made on the real-life system, are inaccurate. We recall that the system does present a problem. So, if the problem is not present when the hardware executes the planned trajectory, then we can conclude that our assumptions about the hardware are not satisfied, and so, inaccurate.

In the next section, we present the RoboChart model we use to carry out our verification.

4 Modelling in RoboChart

Our RoboChart model reflects the system architecture already described, and the existing code and specifications. Any possible communication delays are assumed to be handled separately and are hence negligible for our purposes here. The components modelled receive movement requests as inputs. Thus, trajectory planning is outside of the model's scope and the feasibility of planned trajectories is assumed.

As previously noted, the number of waypoints differs for the UR robot and the Carpano FIVE, so both are expected to receive and execute commands concurrently and independently. In terms of control flow, the model's scope extends until the point where these movement commands are initiated for the Carpano FIVE and the UR robot. From that point, they handle the execution of movements. We expect and assume that the actual execution of movement commands by the UR robot and Carpano FIVE is correct and, therefore, that is also beyond the model's scope.

The definition of the model's scope reflects the fact that our goal is to check that our use of the Carpano FIVE and UR robot commands is appropriate. Therefore, in the RoboChart model, these commands are captured as services of the robotic platform, which we do not further specify.

The component modelled is responsible for selecting the most appropriate movement type for each request. It also detects if the system is out of sync, meaning the target time for a movement request has already passed, resulting in a negative time budget. In the model, this is indicated by the occurrence of an out-of-sync event, and is considered a critical failure.

Next, we justify the abstractions and simplifications made in the RoboChart model (Section 4.1), and then present the RoboChart model itself (Section 4.2).

4.1 Abstractions and Simplifications

It is well-known that model checking eventually encounters state-explosion problems. To keep the complexity and verification time at bay, the following abstractions and simplifications have been made.

Reduction of number of joints: The Carpano FIVE turntable has two joints, one for tilting and one for turning. The IntelliWelder, however, only uses the turning axis during the synchronous welding. Thus, this is the only axis that is considered in the model. The UR robot has six rotational joints, but it is modelled with only two. Although this selection can be made arbitrarily, selecting one from the first three joints (to represent position) and the other from the last three joints (to represent orientation) is advocated. By modelling two axes, the model still captures potential problems related to multiple joint values. Extending the model to include six axes affects the computational complexity of the model, as it would lead to additional parameters of operations (explained in the next section). These operations, however, are not further specified as they represent services that are out of the scope of our verification. So, additional parameters are not relevant for the verification, but just the fact that they are available.

Limited value ranges: To decrease the computation time and complexity, the value ranges of the variables of type real, recording distance, are limited. To cover both negative, positive, and zero-values, the integer range [-1..1] has been chosen. Similarly, the int variables recording discrete time are limited to the two ranges [0..2] and [-1..1]. With the first range, with only positive values, the assumption is that the UR robot and EXAX are never so late that the next waypoint is already in the past. When a negative value -1 is included, we can check whether the goal time for the next waypoint has passed. Lastly, the variables recording the current waypoint for the UR robot and the turntable, of datatype nat, are limited to the ranges [0..3] and [0..1], respectively. So, the maximum number of waypoints for the UR robot are 4 and for the turntable 2, capturing that the number of waypoints can vary in the real system.

Omitted variables: Some variables defined in the URScript are not used in the model to minimize the number of variables. For instance, current and target positions are not included if they are only used to calculate a distance. Instead, the distance is input directly. Adopting a similar approach, other variables are omitted or given a boolean rather than a numerical type to reduce the state space.

4.2 RoboChart Model

In writing a RoboChart model, a key decision is the definition of events and operations that capture services of the robotic platform. The previous section describes our assumptions, some of which are reflected in these definitions. These services are not further specified and establish the interface of the model. Properties are described in terms of interaction with the modelled component via these services.

Fig. 4 shows the robotic platform of our model (on the right), with three input events (start_system, next_UR_move and next_EXAX_move) and the five operations that can be called (four operations in ur_ops and one in exax_ops). The events, declared in the events interface, are used to initiate the system and send new move requests as previous moves are completed. The operations are defined in two separate interfaces: ur_ops and exax_ops, corresponding to the move commands that are executed by the UR robot and the Carpano FIVE. Although the model declares one robotic platform, it captures services of both the robot and the turntable used by the software.

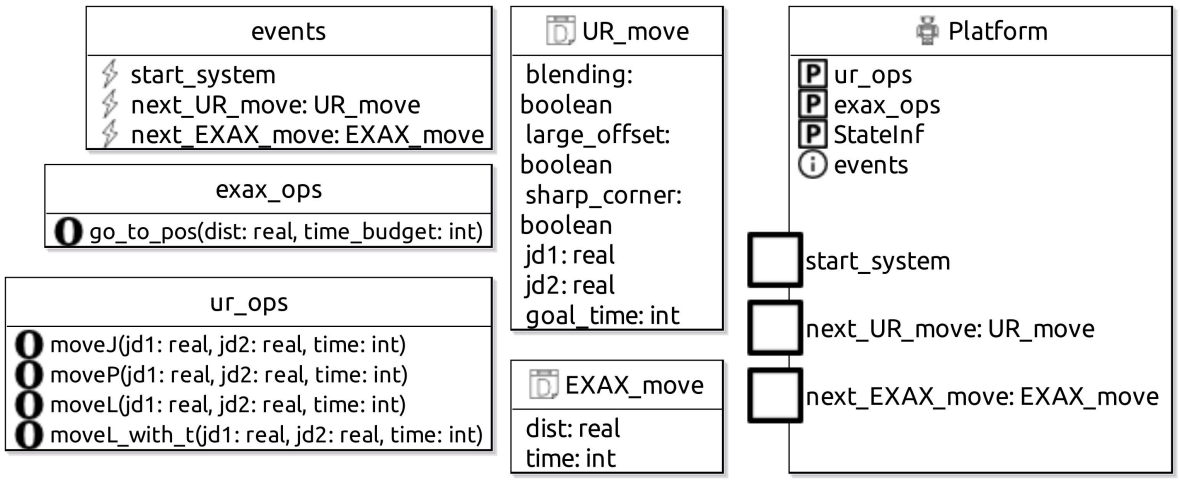


Figure 4: Robotic platform with its defined events, provided operations and custom record types to represent move commands.

Module A RoboChart model is defined by a module block, including the robotic platform and, in our case, one controller block, as shown in Fig. 5. Our module is called main, and the behaviour of our controller is defined by five parallel state machines, acting over the events and operations of the robotic platform as declared in three required interfaces and reflected in connections between the robotic platform and the controller (arrows between the blocks annotated with async).

A RoboChart controller defines how the events of the robotic platform connect to its state machines. In our example, start_system is used by the machine called System. The other two events are used by the machine state_check. A controller also defines how its state machines are connected to each other, via their events, to exchange information and synchronise their behaviour. In Fig. 5, the Controller block includes five blocks, each a reference to one of its state machines, as indicated by the keyword ref. In what follows, we present the definition of these machines.

The System **state machine** Its definition is shown in Fig. 6. In each state of System, a shared variable sys_state is updated to record the current state of the system. This variable is used in the state_check machine presented later to decide whether or not a movement request should be forwarded to the EXAX or to the UR state machine, that is to the turntable or to the UR robot.

The initial junction, a black circle with an i, indicates wait_for_start as the initial state of System, where it waits for the event start_system. When start_system happens, System moves to the working state, where it stays until either the UR robot or the EXAX finishes all of their waypoints, as indicated

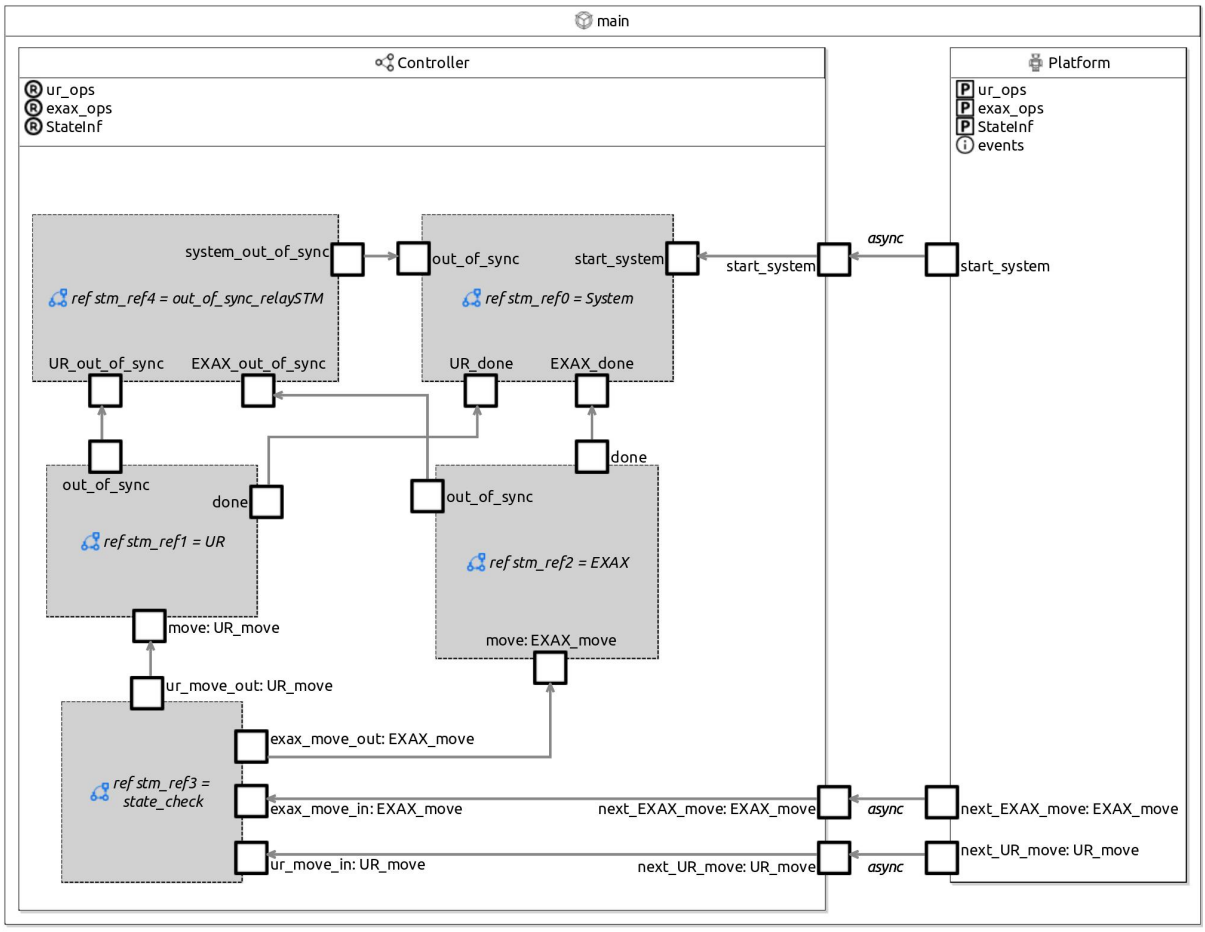


Figure 5: Main module including the Controller with all state machines and the robotic platform.

by events UR_done and EXAX_done, or an out_of_sync event occurs. An out_of_sync event, from any of the states working, UR_finished or EXAX_finished, results in a transition to the final state: white circle with an F. This means that the System state machine cannot progress further.

If both UR_done and EXAX_done occur, regardless of in which order, System goes through either the state UR_finished (if the UR robot finishes first) or EXAX_finished (if the EXAX finishes first), before going back to wait_for_start. This is the end of a welding operation.

The EXAX state machine Its definition, shown in Fig. 7, captures the behaviour of the turntable. EXAX starts in the wait_for_move state, waiting for a move command. The variable curr_waypoint is initialised to 0, and with the constant n_waypoints defined as 1, as in Fig. 7, the turntable goes through two waypoints. When EXAX receives a move event, it stores the requested distance and time to move in a variable exax_move. In the junction (dark circle), it is checked if the time of the movement request (exax_move.time) is strictly negative. If it is, an out_of_sync event is triggered and EXAX terminates. Otherwise, EXAX moves to a state by_position, where the operation go_to_pos is called using as arguments the values in exax_move. If curr_waypoint is greater or equal to n_waypoints, curr_waypoint is reset and the done event is triggered. This is then relayed, by the controller, to the

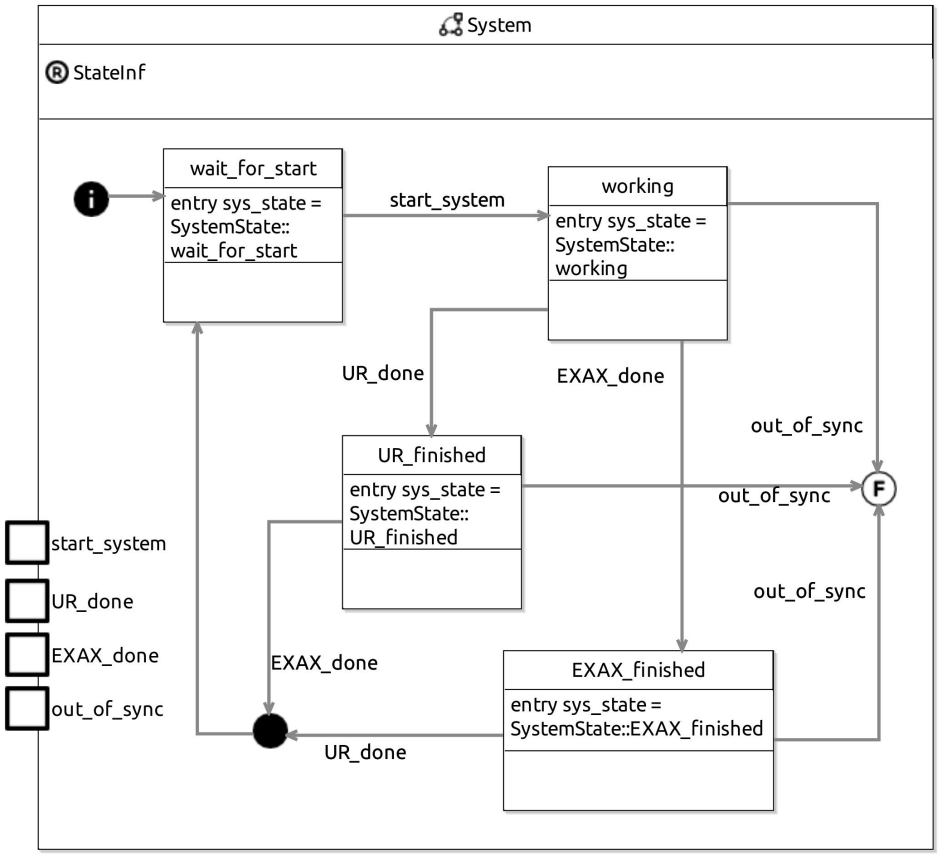


Figure 6: System state machine.

System state machine via its event EXAX_done. (System then transitions to its state EXAX_finished).

The UR state machine It is defined in Fig. 8 to model the behaviour of the UR robot, and is similar to EXAX (Fig. 7). The same method of counting and incrementing waypoints is used, and the done and out_of_sync events are used in the same way. The UR robot, however, chooses the most suitable movement type, so choose_cmd is more complex than the by_position state of EXAX.

Upon entering the choose_cmd state, the boolean variable choosing is set to true. This ensures that a move command must be chosen before leaving the state, since the only transition out of the state choose_cmd has a guard that requires the value of choosing to be false.

Which move command is chosen depends on whether or not the move request includes blending, a large offset from the ideal path (set to 0.8mm in our use case), or a sharp corner. The first junction checks whether the move request includes a blend radius or not, that is, whether ur_move.blending is true or false, where the variable ur_move records the data associated with the move request as defined in the transition out of wait_for_move. If it does include a blend radius, the next junction checks whether the move request includes a large offset (ur_move.large_offset).

If the offset is smaller than the threshold, moveJ is chosen: UR transitions to the moveJ state, where the operation of the same name is called and the variable choosing is set to false in the exit action. With

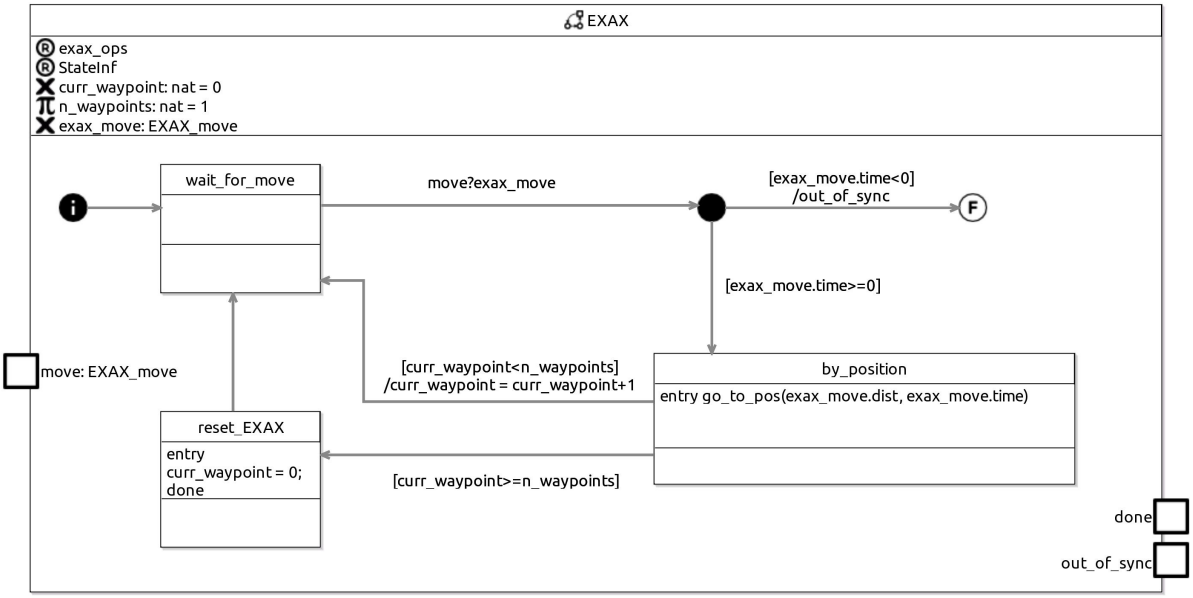


Figure 7: EXAX state machine

that choose_cmd is exited. If the offset is large, the next junction checks whether the move request contains a sharp corner or not (ur_move.sharp_corner). If it does not, moveP is suitable, but if it does, it is necessary to use moveL_with_t. In each case, like for moveJ, the entry action of a state calls the right operation and the exit action updates choosing. If the move command does not include blending, the system enters the big_dist_check state after the very first junction. In the entry action of that state, the check_big_dist function checks if the absolute value of either of the joint distances is larger than a given value (in the example, 1), since the distance determines if moveL is sufficient. If the distance is short, a moveL_with_t command is issued in the state of the same name.

The out_of_sync Relay **state machine** It is simple and omitted here; it relays the out_of_sync event from the EXAX and UR to System. This is necessary just because RoboChart prohibits connecting two different events to the same input of another machine. Full details can be found in [16].

The state_check **state machine** It is defined in Fig. 9 and has a single state checker with two self-transitions. They are triggered by events that accept and record a move command in local variables ur_move or exax_move depending on whether the UR or the EXAX received a move request (whether an input event ur_move_in or exax_move_in happens).

The guards of the transitions ensure that these inputs are accepted only if the system is in a state where the move command should be forwarded to the UR or EXAX machines. There are only two states where they should move: states working or EXAX_finished, for the event ur_move_in, and working or UR_finished, for exax_move_in. In the actions of the transitions, if a move request for the UR robot arrives, it is forwarded to the UR state machine. Similarly, a move request for EXAX is forwarded to the EXAX state machine. With the guards in the transitions, the state_check machine ensures that no move operations can be executed before the system has started, and that once a robot has reached all its waypoints, no further move operations can be executed until the system is reset and restarted.

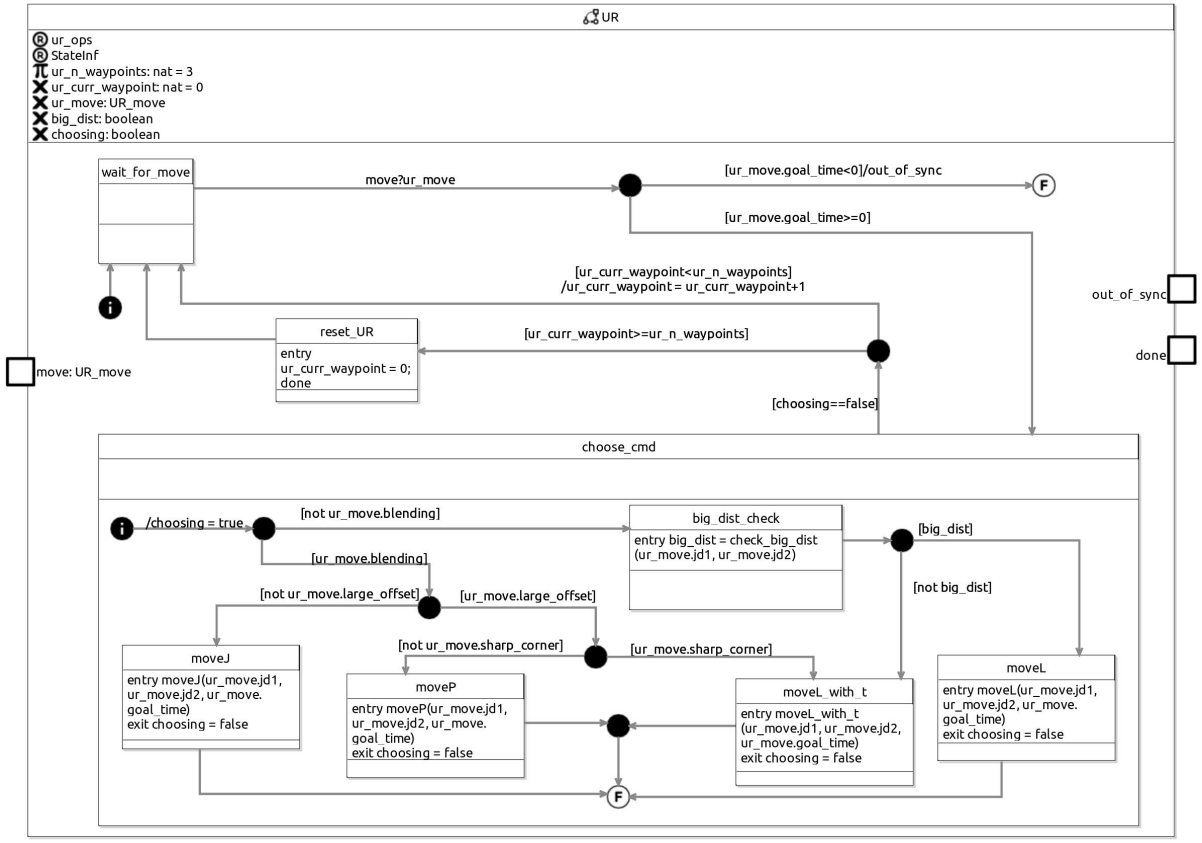


Figure 8: UR state machine.

5 Model Checking

In this section, we describe the model checking and its results: the defined properties and assertions (Section 5.1), the results from FDR (Section 5.2), and their implications for the real-life system (Section 5.3).

5.1 Verification of Selected Properties

Based on the requirements from Section 3.3, assertions can be formulated in natural language, and later defined in *tock*-CSP, which is a dialect of the process algebra CSP where the event *tock* marks the passage of discrete time. To this end, the following properties are to be validated through model checking [16]:

- Every time an EXAX_move or UR_move input event is triggered by the robotic platform, the corresponding movement operation for EXAX or UR, respectively, is called. This is captured in **assertion A1** and **A2** for EXAX , and in **assertion A3** and **A4** for UR.
- The EXAX state machine and the UR state machine do not terminate. This is captured in **assertion A5** and in **assertion A6**, respectively.
- If no out_of_sync event occurs in the System state machine, the state machine does not terminate. This is captured in **assertion A7**.

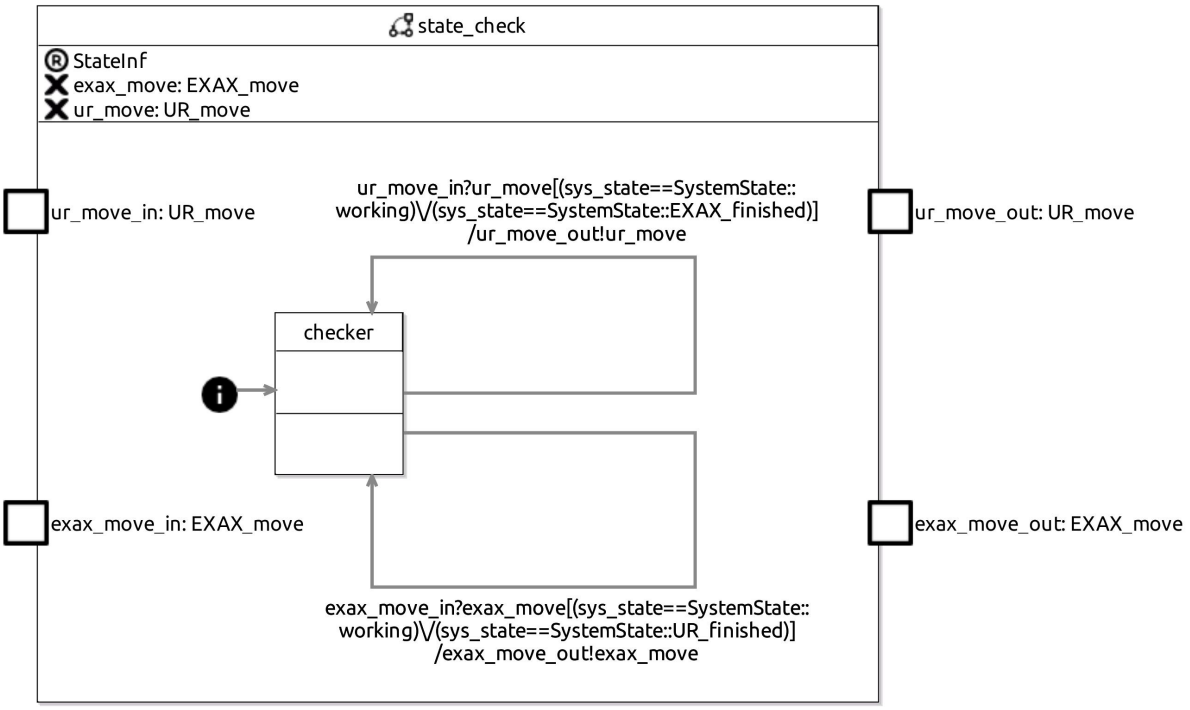


Figure 9: State machine responsible for relaying the move commands of the UR and EXAX only if System is in a state where those state machines should receive commands.

All the assertions described above are detailed next.

Assertion A1 We present in Listing 1 the RoboTool script defining `assertion A1`. With that, RoboTool can use FDR to check whether the assertion holds or not. FDR uses *tock*-CSP processes that define the semantics of the state machines presented in the previous section. These *tock*-CSP processes are automatically calculated by RoboTool. The scripts are written in a mixture of natural language and CSP.

As defined in Listing 1, `assertion A1` requires that `EXAX refines SpecA1 in the traces model`, on line 8. This means that `assertion A1` requires the traces of the process `EXAX` for the machine of the same name to be also traces of the process `SpecA1` defined in lines 1-7.

```
1 timed csp  SpecA1 csp-begin
2 Timed (OneStep) {
3  SpecA1 = let
4         Def = (CHAOS(Events) [| {|EXAX::move.in|} |>
5         ADeadline({|EXAX::go_to_posCall|}, 0)); Def
6  within timed_priority(Def) }
7 csp-end
8 timed assertion A1: EXAX refines SpecA1 in the traces model.
```

Listing 1: Definition of `SpecA1` and `assertion A1`.

`SpecA1` is defined directly in CSP, as indicated in lines 1 and 7. Moreover, it is defined within a `Timed` section (line 2). So, it is a *tock*-CSP process. It is given by the equation in line 3.

The definition of **SpecA1** uses a **let-within** construct. In the **let** clause, a process **Def** (lines 4-5) is defined. In the **within** clause, it is used to define **SpecA1** using a **timed_priority** function. This is just a technicality of FDR: **timed_priority** enforces the understanding of *tock* as a special event that marks the passage of time. So, the behaviour of **SpecA1** is really that of **Def**.

The behaviour of **Def** is initially that of **CHAOS(Events)**, a process that allows any event to occur. It can, however, be interrupted (operator **[| ... |>]**) by the CSP event **EXAX::move.in**, which represents the input move. Upon interruption, the behaviour of **Def** is give by **ADeadline**. This is a parameterised CSP process defined in the RoboTool *tock*-CSP mechanisation [2] that takes a set of events and a deadline, given as a number of *tock* events, as arguments. It requires that one of the events in the provided set, in this case only **EXAX::go_to_posCall**, the CSP process representing a a call to `go_to_pos`, occurs within the deadline, which here is set to 0. Thus, the call to the `go_to_pos` operation is required to happen immediately when the **EXAX::move_in** event occurs.

Assertion A2 For **assertion A1** to be meaningful, it is necessary to ensure that the EXAX state machine is timelock-free, due to the trivial case where the process refuses the event *tock*. This is checked with **assertion A2**. Listing 2 shows the definition of **A2**, where **EXAX2** on line 3 is defined as a version of **EXAX** in whose traces the events **EXAX::go_to_posCall** are ignored (using the hidden operator: ****). This is done because the machine can timelock in the call to that operation, that is, refuse the *tock* event. This is because that call, being in an `entry` action, is urgent, and deadlines create potential timelocks. In *tock*-CSP, when a deadline is reached, *tock* is refused. For the **EXAX::D__** process (line 3), two arguments (0, 1) are needed due to technicalities of the CSP model of RoboChart. The first argument is an ID-value, and the second is the value of `n_waypoints`, which is not fixed in the model of a machine.

```
1 timed csp  EXAX2 csp-begin
2 Timed(OneStep) {
3   EXAX2 = EXAX::D__(0, 1) \ {| EXAX::go_to_posCall |}
4 }
5 csp-end
6 assertion A2: EXAX2 is timelock-free.
```

Listing 2: Definition of **EXAX2** and **assertion A2**.

Assertion A3 The definition of **assertion A3** and **SpecA3** can be seen in Listing 3. This is the UR equivalent to **assertion A1** and **SpecA1**. This assertion ensures that for each move event, one of the four move operation calls must be made before any time is allowed to pass.

```
1 timed csp  SpecA3 csp-begin
2 Timed(OneStep) {
3   SpecA3 = let
4     Def = (CHAOS(Events) [| {|UR::move.in|} |> ADeadline(
5       {|UR::moveJCall,UR::movePCall,UR::moveLCall, UR::moveL_with_tCall|},0));
6     Def
7 within timed_priority(Def) }
8 csp-end
9 timed assertion A3: UR refines SpecA3 in the traces model.
```

Listing 3: Definition of **SpecA3** and **assertion A3**

Assertion A4 Also for the UR STM it is important to ensure timelock-freedom, and this is done in `assertion A4`, which is the UR equivalent to `assertion A2` and omitted here.

Assertions A5 and A6 They are defined in Listing 4; they require that the EXAX and UR state machines, respectively, do not terminate. These assertions are expected to pass given that no out_of_sync occurs.

```
1 assertion A5: EXAX does not terminate.
2 assertion A6: UR does not terminate.
```

Listing 4: Definition of `assertion A5` and `assertion A6`.

Assertion A7 Listing 5 shows the definition of `SysTerminates`, as well as the definition of a process `Stop`, and `assertion A7`. The process `SysTerminates` (line 3) is based on another process (`SysConstrained` from line 2) which is a version of the system where out_of_sync events are skipped. So, the `SysTerminates` process on line 3 only takes into account the termination event of the System state machine. This is done by hiding all events except `System::terminate` using the `|\` operator. If System terminates despite the out_of_sync event being ignored, the assertion should fail, and this is captured by comparing `SysTerminates` to the process `Stop` (line 6-8). `Stop` is equivalent to the CSP process `STOP`, which is a deadlock. This means that the `Stop` process can never perform any events before terminating, and so by demanding that `SysTerminates refines Stop in the traces model`, it can be ensured that this process never performs `System::terminate`, and thus never terminates. The assertion is expected to always pass since the out_of_sync event is being ignored. Still, it shows that in the cases where it does not occur, the System state machine does not terminate.

```
1  timed csp  Systerminates associated to System csp-begin
2   SysConstrained = (System::D__(0) [| {| System::out_of_sync |} |] SKIP)
3   SysTerminates = (SysConstrained ; System::terminate -> SKIP) |\ {| System::
       terminate |}
4  csp-end
5
6  csp Stop csp-begin
7      Stop = STOP
8  csp-end
9
10 assertion A7: SysTerminates refines Stop in the traces model.
```

Listing 5: Definition of `SysTerminates`, `Stop` and `assertion A7`.

5.2 Results from Checking the Assertions

As previously mentioned, the assertions have been run with two different value ranges for the time variable, core_int. The range [0..2] implies that it is not possible to receive negative time budgets for the movements, meaning that the movements are always performed in accordance with nominal plans. The range [-1..1] implies that negative time budgets can occur, signifying either an infeasible plan from Delfoi or incorrect execution of movements by either the UR robot or the turntable. The assertions have been checked on a computer with an AMD Dual EPYC 7501 (2*32 cores) processor and 2TiB of RAM.

Assertion	Result	Elapsed Time			Complexity	
		Compilation	Verification	Total	States	Transitions
A1	✓	10.79s	0.34s	11.13s	228	713
A2	✓	11.10s	0.32s	11.42s	228	713
A3	✓	14.42s	0.51s	14.93s	5,060	17,001
A4	✓	14.15s	0.57s	15.72s	5,060	17,001
A5	✓	0.12s	0.39s	0.51s	228	713
A6	✓	0.12s	0.60s	0.72s	5,060	17,001
A7	✓	0.64s	0.55s	1.19s	32	197

Table 1: Results of model-checking all assertions in FDR with core_int = [0..2].

Assertion	Result	Elapsed Time			Complexity	
		Compilation	Verification	Total	States	Transitions
A1	X	11.16s	0.26s	11.42s	11	87
A2	X	10.94s	0.28s	11.22s	99	388
A3	X	15.23s	0.26s	15.49s	75	1,235
A4	X	14.19s	0.35s	14.54s	931	4,412
A5	X	0.10s	0.40s	0.50s	153	554
A6	X	0.12s	0.48s	0.60s	1,497	6,329
A7	✓	0.70s	0.58s	1.28s	32	197

Table 2: Results of model-checking all assertions in FDR with core_int = [-1..1].

Nominal case with only positive time budgets, [0..2] As summarized in Table 1, all assertions pass. This outcome is desirable for verifying the synchronisation properties. Since A1 and A3 pass, it can be concluded that a movement operation is always called for each movement request received. Relating back to the requirements in Section 3.3, it indicates that **R2** is satisfied.

Realistic case with possibility of negative time budget, [-1..1] As summarised in Table 2, the only assertion that passes is A7. Fig. 10 shows an example of a trace related to the failed assertion A5. It shows an out_of_sync event, which should lead to termination due to EXAX_move having a negative value for the time variable. The counterexample, given at the bottom in Fig. 10, shows that the system does not terminate. Since out_of_sync events are possible, it shows that **R1** from Section 3.3 is satisfied.

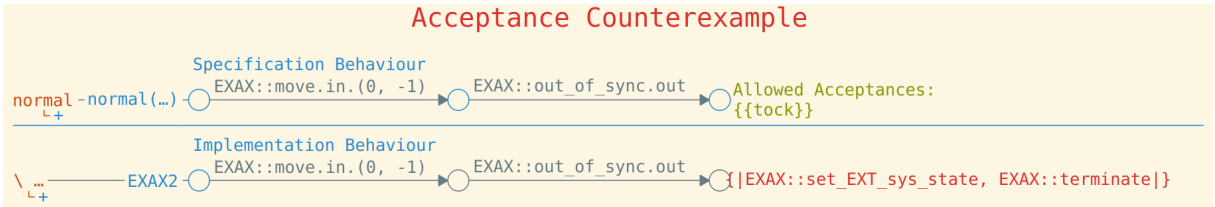


Figure 10: Trace showing a counterexample to assertion A5 - **EXAX does not terminate**.

5.3 Implications and Real-Life System Improvements

The fact that all assertions in the nominal case ($core\_int = [0..2]$) pass, and all assertions apart from one in the realistic case ($core\_int = [-1..1]$) fail, indicates that the robots are unable to follow the nominal plans. This could be, for instance, due to hardware limitations, like insufficient maximum speed or acceleration, or inaccuracy in their trajectory following. To minimise errors, a full re-calibration of the real-life system has been done. Since the programming is offline in Delfoi, it is crucial that the physical system is calibrated precisely so the digital CAD model is correctly positioned and orientated. The resulting weld after the calibration, as seen in Fig. 11, shows a significant improvement in quality.

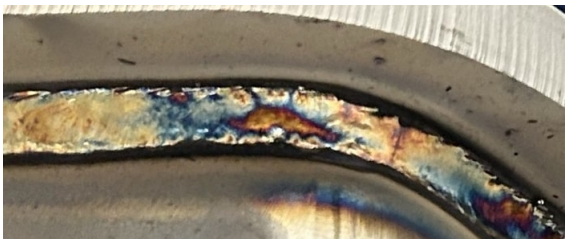


Figure 11: A corner of the workpiece showing significantly improved welding quality after system re-calibration.

6 Conclusion and Further Work

Applying model checking to an already existing industrial robotic system with known weaknesses has proved to be both challenging and useful. The main challenge lies in ensuring that the model catches the essential characteristics of the real-life system. Abstractions and assumptions need to be made to keep the computational complexity at a reasonable level. However, it is crucial that they are not so limiting that the model fails to capture the behaviour and possible errors. Keeping an eye on this so-called "reality gap" between the model and the real system is vital.

Even though improvements have been made on the real-life system based on the findings from the model checking, there is still room for improvement. An interesting path for further work is to verify the assumptions made on the hardware, to achieve a co-verification similar to that in [15]. It is also beneficial to verify the offline programming in Delfoi, and set requirements for the generation of waypoints. This will ensure the feasibility of the planned trajectories.

Further work will use the model further along the RoboStar workflow in [5]. The next step is to automatically generate a simulation model RoboSim [3]. The RoboStar team is also working on model-based testing, so we can generate forbidden traces from RoboChart models. They are specifications of tests that can then be run against the real-life software. This is an interesting way to address the reality gap.

Acknowledgements

David Anisi has received partial funding from the Norwegian Research Council (RCN) RoboFarmer, project number 336712. Ana Cavalcanti and Pedro Ribeiro are funded by the Royal Academy of Engineering (Grant No CiET1718/45), and the UKRI (UK Research and Innovation Council), Grants No EP/R025479/1 and EP/V026801/1.

References

[1] Christel Baier & Joost-Pieter Katoen (2008): *Principles of model checking*. MIT Press.

[2] J. Baxter, P. Ribeiro & A. L. C. Cavalcanti (2022): *Sound reasoning in tock-CSP*. Acta Informatica 59, pp. 125–162, doi:10.1007/s00236-020-00394-3.

[3] A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li & J. Timmis (2019): *Verified simulation for robotics*. Science of Computer Programming 174, pp. 1–37, doi:10.1016/j.scico.2019.01.004. Available at papers/CSMRCD19.pdf.

[4] Ana Cavalcanti, Will Barnett, James Baxter, Gustavo Carvalho, Madiel Conserva Filho, Alvaro Miyazawa, Pedro Ribeiro & Augusto Sampaio (2021): *RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing*. Springer International Publishing, doi:10.1007/978-3-030-66494-7_9.

[5] Ana Cavalcanti, Will Barnett, James Baxter, Gustavo Carvalho, Madiel Conserva Filho, Alvaro Miyazawa, Pedro Ribeiro & Augusto Sampaio (2021): *RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing*. In: Software Engineering for Robotics, Springer, doi:10.1007/978-3-030-66494-7_9. Available at https://link.springer.com/10.1007/978-3-030-66494-7_9.

[6] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve et al. (2021): *On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward*. Proceedings of the National Academy of Sciences 118, doi:10.1073/pnas.1907856118.

[7] Edmund M. Clarke (1997): *Model checking*. In: Foundations of Software Technology and Theoretical Computer Science, 1346, Springer Berlin Heidelberg, doi:10.1007/BFb0058022. Available at http://link.springer.com/10.1007/BFb0058022.

[8] J Edward Colgate, Witaya Wannasuphoprasit & Michael A Peshkin (1996): *Cobots: Robots for collaboration with human operators*. In: ASME international mechanical engineering congress and exposition, 15281, American Society of Mechanical Engineers, doi:10.1115/IMECE1996-0367.

[9] Eclipse Foundation (visited August 5, 2024): *Eclipse website*. Available at http://www.eclipse.org/.

[10] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov & Andrew W. Roscoe (2014): *FDR3 — A Modern Refinement Checker for CSP*. In: Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, doi:10.1007/978-3-642-54862-8_13.

[11] C. A. R. Hoare (1978): *Communicating sequential processes*. Communications of the ACM 21, doi:10.1145/359576.359585. Available at https://dl.acm.org/doi/10.1145/359576.359585.

[12] P Kah, M Shrestha, E Hiltunen & J Martikainen (2015): *Robotic arc welding sensors and programming in industrial applications*. International journal of mechanical and materials engineering 10, doi:10.1186/s40712-015-0042-y.

[13] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis & J. C. P. Woodcock (2019): *RoboChart: modelling and verification of the functional behaviour of robotic applications*. Software & Systems Modeling 18(5), pp. 3097–3149, doi:10.1007/s10270-018-00710-z.

[14] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis & Jim Woodcock (2019): *RoboChart: modelling and verification of the functional behaviour of robotic applications*. Software & Systems Modeling 18, doi:10.1007/s10270-018-00710-z. Available at http://link.springer.com/10.1007/s10270-018-00710-z.

[15] Yvonne Murray, Martin Sirevåg, Pedro Ribeiro, David A. Anisi & Morten Mossige (2022): *Safety assurance of an industrial robotic control system using hardware/software co-verification*. Science of Computer Programming 216, doi:10.1016/j.scico.2021.102766. Available at https://linkinghub.elsevier.com/retrieve/pii/S0167642321001593.

[16] Henrik Nordlie (2024): *Formal verification of synchronization properties of a multi-robot welding system*. Master's thesis, Norwegian University of Life Sciences, Ås, Norway.

[17] Pioneer Robotics AS (visited August 15, 2024): *IntelliWelder - UR+ certified product*. Available at https://www.pioneer-robotics.no/cobot/intelliwelder/.

[18] J Norberto Pires, Altino Loureiro & Gunnar Bölmsjo (2006): *Welding robots: technology, system issues and application*. Springer Science & Business Media, doi:10.1007/1-84628-191-1.

[19] Gang Ren, Qingsong Hua, Pan Deng, Chao Yang & Jianwei Zhang (2017): *A Multi-Perspective Method for Analysis of Cooperative Behaviors Among Industrial Devices of Smart Factory*. IEEE Access 5, doi:10.1109/ACCESS.2017.2708127.

[20] Günther Starke, Daniel Hahn, Diana G. Pedroza Yanez & Luz M. Ugalde Leal (2016): *Self-organization and self-coordination in welding automation with collaborating teams of industrial robots*. Machines (Basel) 4, doi:10.3390/machines4040023.

[21] THG Automation (visited August 19, 2024): *In Sync: The Benefits of Coordinated Motion*. Available at https://thgautomation.com/2024/06/27/in-sync-the-benefits-of-coordinated-motion/.

[22] Universal Robots (visited August 5, 2024): *Universal Robots - UR10e Website*. Available at https://www.universal-robots.com/products/ur10-robot/.

[23] Universal Robots (visited August 8, 2024): *Real-Time Data Exchange Guide*. Available at https://www.universal-robots.com/articles/ur/interface-communication/real-time-data-exchange-rtde-guide/.

[24] Universal Robots (visited July 23, 2024): *Universal Robots e-Series User Manual*. Available at https://www.universal-robots.com/download/manuals-e-seriesur20ur30/user/ur10e/59/user-manual-ur10e-e-series-sw-59-english-international-en/.

[25] University of Oxford (visited August 15, 2024): *FDR4 - The CSP Refinement Checker*. https://cocotec.io/fdr/. Available at https://cocotec.io/fdr/.

[26] Visual Components (visited August 15, 2024): *Robot Offline Programming*. Available at https://www.visualcomponents.com/products/robot-offline-programming/.

[27] Jiahao Xiong, Zhongtao Fu, Miao Li, Zhicheng Gao, Xiaozhi Zhang & Xubing Chen (2021): *Trajectory-Smooth Optimization and Simulation of Dual-Robot Collaborative Welding*. In: Intelligent Robotics and Applications, 13014, Springer, doi:10.1007/978-3-030-89098-8_66.

Synthesising Robust Controllers for Robot Collectives with Recurrent Tasks: A Case Study

Till Schnittka and Mario Gleirscher
University of Bremen, Germany
{schnitti,gleirsch}@uni-bremen.de

When designing correct-by-construction controllers for autonomous collectives, three key challenges are the task specification, the modelling, and its use at practical scale. In this paper, we focus on a simple yet useful abstraction for high-level controller synthesis for robot collectives with optimisation goals (e.g., maximum cleanliness, minimum energy consumption) and recurrence (e.g., re-establish contamination and charge thresholds) and safety (e.g., avoid full discharge, mutually exclusive room occupation) constraints. Due to technical limitations (related to scalability and using constraints in the synthesis), we simplify our graph-based setting from a stochastic two-player game into a single-player game on a partially observable Markov decision process (POMDP). Robustness against environmental uncertainty is encoded via partial observability. Linear-time correctness properties are verified separately after synthesising the POMDP strategy. We contribute at-scale guidance on POMDP modelling and controller synthesis for tasked robot collectives exemplified by the scenario of battery-driven robots responsible for cleaning public buildings with utilisation constraints.

1 Introduction

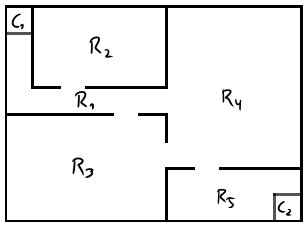
Hygiene in public buildings has been hotly debated ever since the increased safety requirements during the coronavirus pandemic. Autonomous robot collectives can help to relieve cleaning staff and keep highly frequented buildings (e.g., hospitals, schools) clean. However, commercially available solutions have their limitations, for example, a need for manual task programming or a lack of online adaptability. In contrast to domestic homes, there are strict regulations (e.g., [8] for schools) that stipulate which areas must be cleaned and at what intervals. Changing operational conditions (e.g., room occupation, equipment reconfiguration, cleaning profile, regulations) create the need for an automatic generation of cleaning schedules for robot collectives and for proving their compliance with hygiene requirements. This scenario is an instance of a multi-faceted *recurrent scheduling* problem discussed below.

Cleaning Buildings as a Running Example. When planning the cleaning of a building (e.g., a school), we can use its layout in the form of a room plan (Figure 1a). Apart from a set of m rooms $\mathscr{R} = \{\mathscr{R}_1, \ldots, \mathscr{R}_m\}$ with an assigned area, such plans include the connections between rooms and the locations of n charging stations $C = \{C_1, \ldots, C_n\}$. A collective of $k \leq n$ robots $B = \{B_1, \ldots, B_k\}$ is responsible for cleaning $\mathscr{R}$. B is tasked to keep $\mathscr{R}$ clean while charging its batteries using C. Each robot has a limited battery size and a charging point in C as an assigned resting position. Additionally, there is a room utilisation plan (Figure 1b) containing the times when the rooms are in use, which can change on a daily basis. While a room is in use, no cleaning robot may be inside and, thus, cannot clean it.

The task is to create a cleaning *strategy* (or schedule) for B, constrained by the utilisation plan and charging needs. Moreover, this strategy should keep the rooms clean enough, while minimising battery consumption. To specify this task adequately, we define cleanliness in terms of *contamination*. Since

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 109–125, doi:10.4204/EPTCS.411.7

© Till Schnittka and Mario Gleirscher
This work is licensed under the
Creative Commons Attribution License.



(a) Room plan

Room	Time of use
$\mathscr{R}_1$	8:00 – 10:00
$\mathscr{R}_2$	12:00 – 14:00
$\mathscr{R}_3$	10:00 – 12:00
$\mathscr{R}_4$	15:00 – 16:00

(b) Room utilisation plan

Figure 1: Examples of a room plan and the per-room utilisation

we are dealing with floor cleaning robots, we limit our definition to floor surfaces. For example, hygiene guidelines for schools [8] and the associated standard DIN 77400 [3] recommend cleaning intervals for floor surfaces. We reflect this recommendation in our definition by *contamination rates and thresholds*. Each room has a certain contamination rate. Over time, total contamination accumulates in a room and can be reset by cleaning. Clearly, the total contamination of a room should not exceed a certain threshold.

Approach. We propose a quantitative stochastic approach to synthesise strategies for robot collectives with recurrent tasks, such that the strategies are robustly (i.e., under uncertainty) compliant to recurrence and safety constraints and optimisation goals. We consider (i) weighted stochastic models and strategy synthesis for (ii) optimally coordinating robot collectives while (iii) providing guarantees (e.g., recurrence, safety) on the resulting strategies (iv) under uncertainty (e.g., partial observability). We select POMDPs, a generalisation of Markov decision processes (MDPs), for our problem.

Related Work. Among the works employing POMDPs for optimal planning, Macindoe et al. [11] show strategy synthesis for human-robot cooperative pursuit games with robots and humans acting in turns. Moreover, Thomas et al. [16] combine PDDL-based high-level task scheduling and POMDP-based low-level navigation. They use a Kalman filter to predict the distribution of the POMDP's belief state based on discretised robot dynamics.

Using the PRISM model checker, Giaquinta et al. [4] show the synthesis of minimal-energy strategies for robots finding fixed objects. Object finding as an instance of navigation can be solved with memoryless strategies, that is, functions of the current state (here, positions of robot and object). Lacerda et al. [9] propose multi-objective synthesis of MDP policies satisfying *bounded co-safe LTL* properties using PRISM. Illustrated by a care robot, they employ a timed MDP and filter irrelevant transitions and states, resulting in a reduced MDP where time as a state variable preserves bounded properties. Basile et al. [2] use stochastic priced timed games and reinforcement learning (RL)-enhanced statistical model checking (with UPPAAL Stratego) to synthesise *safe, goal-reaching, and minimal-arrival-time* strategies for a *single* autonomous train operated under moving-block signalling.

El Mqirmi et al. [12] combine multi-agent RL and verification to coordinate robot collectives. An abstract MDP is generated from expert knowledge for optimal synthesis of a joint abstract strategy (using PRISM, STORM) under PCTL (safety) constraints. RL identifies a concrete strategy within these constraints by using shielding (i.e., only choosing actions compliant with the abstract strategy). Gu et al. [7] tackle state space reduction via RL to synthesise optimal navigation and task schedules for collectives (e.g., a quarry with autonomous vehicles) and timed games (in a UPPAAL Stratego extension) to check timed CTL properties (e.g., *liveness, safety, reachability*) of the synthesised strategies.

Vázquez et al. [17] developed a domain-specific language (DSL) for specifying tasks for collectives. Task allocation constraints are solved by ALLOY and plain MDPs are employed (via PRISM, EVOCHECKER) to synthesise *goal-reaching, minimum-travel-time* schedules.

Contributions. Our approach enhances works in optimal planning [11, 16] by a step of strategy verification against stochastic temporal properties. Object finding [4] corresponds to a *reachability* property, whereas continuous contamination and its removal to a *response* property. Moreover, object finding differs from *recurrent scheduling* in that it is static and can be realised with a simpler reward function and a smaller state space. Our problem involves a dynamic goal with robots operating 24 hours a day, having to coordinate their work continuously. Beyond bounded co-safe LTL [9], our approach supports *bounded response* properties $\mathbf{G}(\phi \rightarrow \mathbf{F}^{\leq T}\psi)$. The above works [9, 2, 12, 7] underpin the usefulness of stochastic abstractions for synthesis in various domains. Our use of POMDPs to hide parts of the stochastic process (e.g., contamination) and explicit concurrency (e.g., for many simultaneous robot movements) offers an alternative to obtaining small models for multi-agent synthesis under limited resources (e.g., time constraints to clean rooms). Additionally, we argue how the model of our case study—a collection of cleaning robots subjected to hygiene requirements—, while kept simple for illustrative purposes, scales and generalises to a range of similar scenarios in other application domains. Apart from our focus on recurrence and model reduction, a DSL [17] can wrap our approach into a practical workflow.

Overview. After giving key definitions in Section 2, we present our approach in Section 3. In the Sections 3.1 to 3.3, we explain the modelling, in Section 3.4 the treatment of collectives, in Section 3.5 the constrained POMDP synthesis problem, and, in the Sections 3.6 and 3.7, the extraction and verification of a strategy. We evaluate our approach in Section 4, discuss issues we encountered during modelling and synthesis in Section 5, and add concluding remarks in Section 6.

2 Preliminaries

Stochastic modelling is about describing uncertain real-world behaviour in terms of states and probabilistic actions producing transitions between these states. For stochastic reasoning (i.e., drawing conclusions about such behaviour), we use *probabilistic model checking*. This section introduces the *stochastic models*, *temporal logic*, and *tools* we employ for synthesis and verification.

Partially Observable Markov Decision Processes. Let $Dist(X)$ denote the set of discrete probability distributions over a set X, and $\mathbb{R}_{\geq 0}$ be the non-negative real numbers. Then, a POMDP [13] is given by

Definition 2.1. A POMDP is a tuple $M = (S, \bar{s}, A, P, R, \mathscr{O}, obs)$, where

- S is a set of states with $\bar{s} \in S$ being the initial state,
- A is a set of actions (or action labels),
- $P\colon S \times A \rightarrow Dist(S)$ is a (partial) probabilistic transition function,
- $R = (R_S, R_A)$ is a structure defining state and action rewards $R_S\colon S \rightarrow \mathbb{R}_{\geq 0}$ and $R_A\colon S \times A \rightarrow \mathbb{R}_{\geq 0}$,
- $\mathscr{O}$ is a finite set of observations, and
- $obs : S \rightarrow \mathscr{O}$ is a labelling of states with observations.

Moreover, $A(s) = \{a \in A \mid P(s,a) \text{ is defined}\}$ describes the actions *available* in s. A *path* in M is defined as a finite or infinite sequence $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ where $s_i \in S$, $a_i \in A(s_i)$, and $P(s_i,a_i)(s_{i+1}) > 0$ for all $i \in \mathbb{N}$. Let $last(\pi)$ be the last state of π. $FPaths_M$ and $IPaths_M$ denote all finite and infinite paths of M starting at state $\bar{s}$. Non-determinism in M is resolved through a *strategy* according to

Definition 2.2 (POMDP Strategy)**.** A strategy for a POMDP M is a map $\sigma : FPaths_M \to Dist(A)$, where

- for any $\pi \in FPaths_M$, we have $\sigma(\pi)(a) > 0$ only if $a \in A(last(\pi))$, and
- for any path $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ and $\pi' = s'_0 \xrightarrow{a'_0} s'_1 \xrightarrow{a'_1} \ldots$ satisfying $obs(s_i) = obs(s'_i)$ and $a_i = a'_i$ for all i, we have $\sigma(\pi) = \sigma(\pi')$.

We call σ *memoryless* if σ's choices only depend on the most recent state ($last(\pi)$), and *deterministic* if σ always selects an action with probability 1. Below, we consider memoryless deterministic strategies.

Probabilistic Linear Temporal Logic (PLTL). A POMDP M describes a stochastic process, such that every possible execution of that process corresponds to a *path* through M's transition graph. To draw qualitative conclusions about M and its associated strategies, we express properties in linear temporal logic (LTL). An LTL formula ϕ over atomic propositions AP follows the grammar

$$\phi ::= ap \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \phi \ \mathbf{U} \ \phi \tag{1}$$

where $ap \in AP$. In LTL, we make statements about M's path structure and specify admissible sets of paths. Informally, $\mathbf{X}\phi$ describes that ϕ holds in the next state of a given path, and $\phi \ \mathbf{U} \ \psi$ describes that ϕ holds until ψ occurs, or globally, if ψ never occurs. We allow the abbreviations $\mathbf{F}\phi \equiv \mathrm{T} \ \mathbf{U} \ \phi$, describing that ϕ holds at some point on a path, and $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$, describing that ϕ applies to the entire path.

To draw quantitative conclusions about M (or query probabilities and rewards), we use probabilistic LTL (PLTL), whose formulas ϕ are formed along (1) and by the two operators $\mathbf{P}$ and $\mathbf{R}$:

- $\mathbf{P}_{[\min|\max]\sim p|=?}[\psi]$ describes that the [minimum|maximum] probability of ψ (under all possible POMDP strategies) being valid is $\sim p$, and
- $\mathbf{R}^R_{[\min|\max]\sim r|=?}[\psi]$ expresses that the [minimum|maximum] expected reward R associated with ψ (under all possible POMDP strategies) meets the bound $\sim r$,

where ψ is an LTL formula, $\sim \in \{<, \leq, =, \geq, >\}$, and $=?$ is used for queries. Given a timer t in M and that all actions in A increment t by 1, we allow $\mathbf{F}^{\sim T}\psi \equiv \mathbf{F}(t \sim T \wedge \psi)$. In LTL, $M \models \phi$ expresses that all paths of M from $\bar{s}$ are permitted by ϕ, and, in PLTL, that a probability measure over M's paths satisfies ϕ. For convenience, we use ϕ to refer to both, the expression and the region in S where it evaluates to true. LTL and PLTL's semantics are explained in detail in, for example, [1, p. 231 and Sec. 6.2].

The PRISM Model Checker can check $M \models \phi$ using exact and approximate algorithms [14]. It supports a variety of stochastic models and logics and has its own languages for modelling and for specifying properties. In PRISM, a reward structure R is used as a parameter in $\mathbf{R}^R[\cdot]$. For POMDPs, PRISM can synthesise strategies in the form of Definition 2.2.

A PRISM *model* consists of a series of *modules*, each defining a fraction of a state s using its own variables. *Modules* can synchronise their transitions by sharing labels from A, such that transitions in several modules using the same label can only switch in a state $s \in S$ if each transition is enabled in s. An example module with state variable x and command `increase` is given in Listing 1.

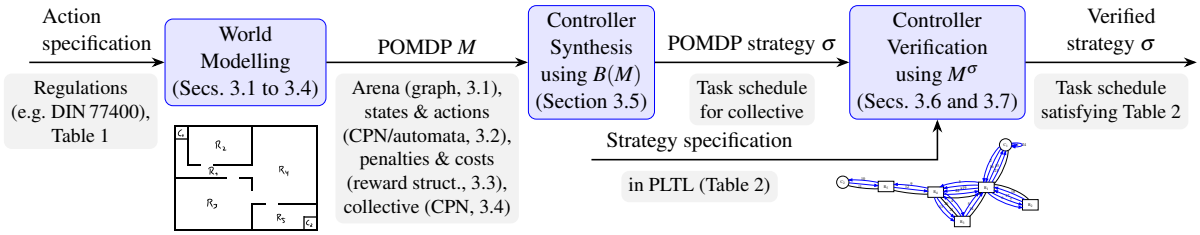


Figure 2: Overview of the proposed synthesis approach for robot collectives

```
module Example
  x : [0..2] init 0; // state variable x with range {0,1,2} and initial value 0
  [increase] x<2 -> (x'=x+1); // command increase increasing x by 1 if x < 2
endmodule
```

Listing 1: Example of a module in PRISM's guarded command language

Fixed-Grid Approximation in PRISM. When analysing a POMDP M, PRISM computes an approximate (finite-state) belief-MDP $B(M)$ [13], each *belief* being a probability distribution over the partially observable states (e.g., the possible room contamination). The size of $B(M)$'s state space, called *belief space* [10], is exponential in PRISM's grid-resolution parameter g controlling the approximation of the upper and lower bounds to be determined for $\mathbf{P}\,|\,\mathbf{R}^{\min\,|\,\max}$-properties using $B(M)$.

3 Developing Controllers by Example of the Cleaning Scenario

In this section, we first state our synthesis problem and then describe our approach to strategy synthesis and verification as illustrated in Figure 2.

Problem Statement. Our aim to synthesise a collective controller is specified in the LTL property

$$\underbrace{\mathbf{G}\big(\overbrace{(\omega \to \mathbf{F}^{\leq T}(\omega \wedge \phi_{\mathsf{r}}))}^{\text{reach task goal}} \wedge \overbrace{\mathbf{G}^{\leq T}\phi_{\mathsf{s}}}^{\text{keep safe}}\big)}_{\text{periodically achieve task safely (implied by our approach)}}\quad, \tag{2}$$

where ω is the *recurrence area* (including $\bar{s}$ and acting as a task invariant), ϕ_{r} specifies an *invariant-narrowing condition*,[1] ϕ_{s} specifies *task safety*, and $T > 1$ is the *recurrence interval* (an upper cycle-time bound). Among the controllers satisfying (2), we look for an optimal (e.g., one minimising energy consumption) and robust (e.g., under partial observability of stochastic room contamination) one.

Overview of Controller Development. First, a spatio-temporal abstraction of the cleaning scenario is modelled using a coloured Petri net (CPN) for coordination modelling and finite automata for describing robot-local behaviour. These aspects are translated into a reward-enhanced POMDP M (Sections 3.1 to 3.3) in support of multiple robots (Section 3.4), which uses probabilistic actions to reduce the state count of a hypothetical detailed model. Then, a strategy σ is synthesised for M (Section 3.5), which is used to derive a *deterministic*, *non-probabilistic*, and *integer-valued* model M^σ (Section 3.6). Finally,

[1] ϕ_{r} only has methodological relevance. It could, for example, be used to develop an increasingly strong invariant ω.

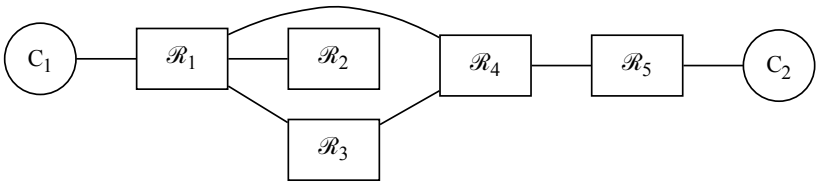


Figure 3: Example of a room plan graph

Table 1: Requirements of the cleaning scenario defining the `at` action implemented in three modules

Id.	**Action Specification** (Behavioural Requirement)	**Impl. in Module**
Ba	The robot either stays in the room it is currently in or moves to another room, whereby there must be an edge in the room plan graph between the current and next room.	`cleaner`
Bb	If the robot is on a charging station, its battery charge level is increased by the charging rate of the robot.	
Bc	If the robot is *not* on a charging station, its battery charge level is reduced by the robot's discharge rate.	
Da	The total contamination of each room increases by its contamination rate if there is no robot in that room.	`contamination`
Db	If the robot is in a room, the total contamination of that room is reset.	
Ta	The time counter is incremented by one.	`time`

M^{σ}, representing the high-level controller, is verified (Section 3.7) against strategy requirements that, due to current limitations in the formalisms and tools, cannot be checked directly during synthesis.

3.1 Spatio-temporal Abstraction

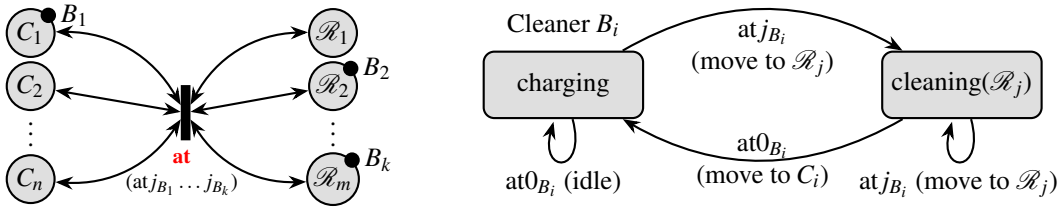
State Space. For the implementation of the problem (e.g., cleaning task), it is important to keep the number of states as small as possible. Therefore, large parts of the initial problem are abstracted.

Instead of a complete room plan with area assignment, the abstracted *environment* uses a graph that only contains the different rooms and charging stations. An example of such a graph for the room plan in Figure 1a can be seen in Figure 3. A pointer $B_i.x$, $i \in 0..k$, to a room or charging station in this graph is used to keep track of the position of robot B_i. The behaviour of the charging state $B_i.c$ of the robot's battery is described by a number of discrete charging levels and charge and discharge rates.

The total *contamination* is represented by a counter $\mathscr{R}_j.d$, $j \in 0..m$, whose maximum value is the contamination threshold $\mathscr{R}_j.threshold$. Since we want to avoid reaching a state with the contamination at this threshold, it is not necessary to model contamination beyond $\mathscr{R}_j.threshold$.

Actions and High-level Behaviour. Discrete values are used to model *time* as well, where the action `at`, as specified in Table 1 and described below, is performed at each discrete time step.

The CPN in Figure 4a provides a high-level description of the moves of the collective B across charging stations C and rooms $\mathscr{R}$ (Figure 3). The abstract `at` action (black bar) expands to a range of concrete POMDP actions $\texttt{at}\, j_{B_1} \ldots j_{B_k}$ with $j_{B_i} \in 0..m$. Whenever `at` is taken, any number of tokens (black dots, representing robots) on the places (grey circles, representing rooms and charging stations) can flow si-



(a) Coordination of $k \leq n$ cleaners (b) Control of cleaner B_i

Figure 4: Cleaner coordination (a) as a CPN and local control (b) as a finite automaton

multaneously, such that B_i can move from one place via `at` to an adjacent empty (possibly same) place.[2] Figure 4b outlines the control of a particular robot B_i. When composed (in parallel), as formalised in M by implicit constraints on the j-indices, simultaneous moves of several robots into a single place and jumps to non-adjacent places are prohibited by the coordination constraint in Figure 4a.[3]

3.2 Quantitative and Stochastic Abstraction

For the sake of simplicity, this section and the following will focus on a reduced problem with only one robot. The case of multiple robots will be reintroduced in Section 3.4.

As already mentioned, the PRISM-encoding of M is divided into three modules operating on three independent fragments of the state space S: The state of the robots, the room contamination, and time.

The `cleaner` module describes the behaviour of cleaning robot B_1. An integer is used to model the robot position $B_1.x$. For this, each room and charging station is mapped to an integer bijectively. For example, the room graph in Figure 3 can be described by the following relation: $C_0 \to 0, \mathscr{R}_1 \to 1, \mathscr{R}_2 \to 3, \mathscr{R}_3 \to 4, \mathscr{R}_4 \to 5, \mathscr{R}_5 \to 6, C_2 \to 7$. The battery status $B_1.c$ is also described as an integer whose upper bound is the maximum charge $B_1.maxcharge$ of B_1's battery, see Listing 2.

```
1 module cleaners
2   x : [0..m−1] init B1.start; // position of cleaner B1
3   c : [0..B1.maxcharge] init B1.ωchgthres; // battery status of B1
4   [at0] (x=0|x=1)
5         -> (x'=0) & (c=min(c+B1.chargerate, B1.maxcharge)); // charge B1
6   [at1] (x=0|x=1|x=2|x=3|x=4)
7         -> (x'=1) & (c=max(c-B1.dischargerate, 0)); // move to R1
8   ...
9 endmodule
```

Listing 2: A model fragment of the *cleaners* module highlighting its states and actions

For each charging station and each room there is a transition `atN` (where `N` is the integer assigned to the room), which models entering or staying in this room. The precondition for this transition is that the robot must already be in that room or a neighbouring room. If a robot enters or stays at a charging station, the charge increases by the charging rate; if a robot is in a room, the charge decreases by the discharge rate, see lines 5 and 7 respectively.

The `contamination` module describes the contamination status of $\mathscr{R}$. To reduce the number of states, contamination is modelled by booleans—the contamination flags $\mathscr{R}_j.d$, $j \in 1..m$—rather than in-

[2] We assume for any initial state $\bar{s} \in S$ that no more than one robot is at a particular place.

[3] This construction reduces S and P of M in comparison with using alphabetised synchronous composition.

tegers. The probability $\mathscr{R}_j.pr$ of $\mathscr{R}_j.d$ getting true is used to model the state in which the contamination of the corresponding room has reached its threshold $\mathscr{R}_j.threshold$. If there is no robot in $\mathscr{R}_j$, we set $\mathscr{R}_j.d =$ `true` with probability $\mathscr{R}_j.pr$ inversely proportional to the contamination threshold $\mathscr{R}_j.threshold$. If a robot visits or stays in $\mathscr{R}_j$ then $\mathscr{R}_j.d$ is set to `false`, see Listing 3.

```
module contamination // sequential stochastic contamination
  ℛ1.d : boolean init false; ℛ2.d : boolean init false;
  ...
  [at0] true -> 1−(∑i∈1..m ℛi.pr):  true
     + ℛ1.pr: (ℛ1.d'=true) + ℛ2.pr: (ℛ2.d'=true) + ...; // prob. contam. all while charg.
  [at1] true -> 1−(∑i∈1..m\1 ℛi.pr): (ℛ1.d'=false)
     + ℛ2.pr: (ℛ1.d'=false)&(ℛ2.d'=true) + ...; // clean ℛ1 and prob. cont. other rooms
  [at2] true -> 1−(∑i∈1..m\2 ℛi.pr): (ℛ2.d'=false)
     + ℛ1.pr: (ℛ2.d'=false)&(ℛ1.d'=true) + ...; // clean ℛ2 and prob. cont. other rooms
  ...
endmodule
```

Listing 3: A fragment of the `contamination` module (e.g. $\mathscr{R}_i.pr = 0.05$ for $i \in 1..m$)

The `time` module describes the progression of time and manages the switching to the error and final state (the model handles both the same). `time` also restricts the `atN` transitions so that they can only be used as long as the model is not in the error or final state. This is possible because a transition can only trigger if it can trigger in each module. Therefore, precondition in the `time` module can prevent a transition from triggering even though it is marked with `true` in the `cleaner` and `contamination` modules. The time is increased by one unit with each transition until it reaches T. At this point (due to the definition of `error_or_final`), only the `fin` transition can switch and the model ends in a loop, see Listing 4.

```
formula error_or_final = (c=0|!(t<T));
module time
  t : [0..T] init 0;
  [at0] !error_or_final -> (t'=min(t+1,T)); // charge
  [at1] !error_or_final -> (t'=min(t+1,T)); // move to ℛ1
  [fin] error_or_final  -> (t'=min(t+1,T)); // finish cycle
endmodule
```

Listing 4: A fragment of the `time` module

3.3 Choice of the Reward Function

Four requirements, a valid strategy must satisfy, can be derived from Formula (2) and Table 1:

FR At time T, all robots must be back in their initial location, so that the plan can be repeated.

ωC At time T, the battery of a robot must not be lower than its threshold charge level.

BC The battery of a robot must never be empty.

CT The total contamination of any room must never exceed its contamination threshold.

When just focusing MDP verification rather than synthesis, it would be sufficient to describe these as PLTL constraints. However, PLTL constraints cannot be used as queries for synthesising *strategies*, since generating strategies through PRISM requires each path to be able to fulfil all constraints eventually.

Using constraints that are violated on some of M's paths (e.g., if we require BC, any path leading to an empty battery eventually violates BC) will prevent PRISM from finding a reward-optimal strategy. This is a specific known limitation of the used formalism. Even though we cannot use PLTL constraints to encode all our requirements, the reward structure R can be used to prioritise selecting *strategies* that fulfil these requirements. The encoding of our requirements in R can be achieved by penalising states that do not fulfil some or all of the requirements, see Listing 5.

```
const a_lot = 10000000;
const a_bit = 10000;
rewards "penalties"
  c=0:                                  a_lot; // BC: battery empty
  t=T & (x!=B_1.start|c<B_1.ωchgthres):  a_lot; // FR: robot not at initial loc. at time T
  ℛ_1.d=true:                             a_bit; // Room 1's contamination flag is set
  ℛ_2.d=true:                             a_bit; // Room 2's contamination flag is set
  ...
endrewards
```

Listing 5: An example of the reward structure for the *state penalities*

We previously found it ineffective to penalise the contamination flags the same as the constraints FR, ωC, and BC. Whereas the latter can be determined from M's state, contamination flags only carry the probability of a requirement being violated. Hence, we apply lower penalties to the contamination flags. Further reward structures are used to model optimisation goals, such as energy consumption, see Listing 6a. However, the penalty for constraints is chosen such that it is not possible to offset the penalty of an invalid state by the reduced penalty for a less energy-consuming strategy.

```
rewards "energy consumption"
  t < T: B_1.maxcharge - c;
endrewards
```

(a) A fragment of the optimisation *rewards*

```
const a_lot = 10000000;
rewards "utilisation"
  x=2 & t>=8  & t<10: a_lot;
  x=2 & t>=12 & t<14: a_lot; ...
endrewards
```

(b) A fragment of room utilisation *rewards*

Listing 6: Fragments of the reward structure used for the cleaning scenario

Room Utilisation. As with the other requirements, a room utilisation profile to be respected by the cleaners (UT) can be softly specified using additional reward functions as shown in Listing 6b. However, UT as a safety property will later also be specified in PLTL and checked of the synthesised strategy.

3.4 Cooperation between Multiple Robots

To keep M simple, robots are not modelled as separate modules, but the state of the `cleaner` module is extended to include the positions of all robots (see Figure 4a). This simplification excludes all transitions from the model that would lead to conflicts in robot behaviour (e.g., the case where several robots clean the same room at the same time). Additionally, each robot has its own battery charge, see Listing 5.

```
x_1 : [0..m] init B_1.start;
x_2 : [0..m] init B_2.start;
...
c_1 : [0..B_1.maxcharge] init B_1.maxcharge;
c_2 : [0..B_2.maxcharge] init B_2.maxcharge;
...
```

Figure 5: The structure of the `cleaner` state

The `atN` actions for a single robot are now extended to `atN_N_...` actions, which then model the simultaneous movement of k robots, as illustrated in Figure 4a and implemented in Listing 7.

```
[at0_1] !error & (x1=0|x1=1) & (x2=0|x2=1|x2=2|x2=3|x2=4) // charge B1 and move B2 to R1
         -> (x1'=0) & (x2'=1)
          & (c1'=min(c1+B1.chargerate,B1.maxcharge))
          & (c2'=max(c2-B2.dischargerate,0));
[at1_2] !error & (x1=0|x1=1|x1=2|x1=3|x1=4) & (x2=1|x2=2) // move B1 to R1 and B2 to R2
         -> (x1'=1) & (x2'=2)
          & (c1'=max(c1-B1.dischargerate,0))
          & (c2'=max(c2-B2.dischargerate,0));
...
```

Listing 7: Two examples of `atN_N` actions

This solution increases the number of states per robot considerably, but the complexity of M is still within a practically verifiable range. Additionally, the reward structures that depend on the position and charge of a robot are adapted to include the position of all robots, as can be seen in Listing 8.

```
rewards "penalties"
  c1=0: a_lot;
  c2=0: a_lot;
  ...
  t=T & (x1!=B1.start|c1<B1.ωchgthres): a_lot;
  t=T & (x2!=B2.start|c2<B2.ωchgthres): a_lot;
  ...
endrewards
```

```
rewards "utilisation"
  x1=2 & t>=8  & t<10: a_lot;
  x2=2 & t>=8  & t<10: a_lot;
  x1=2 & t>=12 & t<14: a_lot;
  x2=2 & t>=12 & t<14: a_lot;
  ...
endrewards
```

Listing 8: Reward structure for a collective

3.5 Synthesising Strategies (under Uncertainty) for the Cleaning Scenario

PRISM's POMDP strategy synthesis works under certain limitations. As indicated in Section 3.3, it is not possible to use $\mathbf{R}^{R}_{\min=?}[\psi]$ for synthesis if $\mathbf{P}_{\min=?}[\psi] < 1$, that is, if M contains ψ-violating paths under some strategy σ. Hence, we choose a ψ that defines a state that all paths converge at, and synthesise a strategy that minimises the total reward (since we model R using penalties) up to that point. A commonality of all paths is the flow of time, so the reachability reward-based synthesis query

$$\sum_{R\in\{\text{penalties, energy consumption, utilisation}\}} \mathbf{R}^{R}_{\min=?}[\mathbf{F}\, t = T] \tag{3}$$

uses a target state where time t is equal to some maximum time T.

Additionally, a mapping *obs* needs to be specified, which defines the observations of M that σ can use to make choices. In this case, σ cannot use the contamination flags to make its choices. If σ could consider the contamination flag, it would not need to account for the accumulative probability; σ could just check if a contamination flag is true and act accordingly. We can hide the contamination flags from σ by defining *obs* to just include the position and charge of the robot and the time, see the listing on the right.

```
observables
  t,
  x1, x2, ...,
  c1, c2, ...
endobservables
```

PRISM allows the explicit generation of *deterministic* strategies. Such strategies are useful in this case, since, except for the final state, the model has no loops (i.e., time is always advancing). Because σ is deterministic, it can be thought of as a list of actions for each time step of the cleaning schedule, where the transition of each step of the strategy denotes the action of the robot at that time step within period T. Note that the observable environmental part of M is deterministic such that after applying σ, M^{σ} has exactly one path, hence, σ only depends on time t.

3.6 Creating an Induced Model from the Strategy

It is possible to create a cleaning schedule from the synthesised strategy σ. However, it is not yet possible to verify σ regarding the constraints listed in Section 3.3. This is because, up to this point, contamination was modelled in M only as a probabilistic factor. To verify the contamination constraint CT, below, we include a non-probabilistic contamination model in M' using counters to represent contamination.

Modelling the Contamination Value. To verify that the contamination value (modelled as a boolean sub-MDP of M) never actually reaches the thresholds $\mathscr{R}_j.threshold$, $j \in 1..m$, it is necessary to transform M into an MDP M' that accounts for the actual values $\mathscr{R}_j.d$. We accomplish this in M' by integer-valued contamination counters $\mathscr{R}_j.d$ (Listing 9) replacing the boolean variables $\mathscr{R}_j.d$ in M (Listing 3).

```
module contamination
  R1.d : [0..R1.threshold] init 0;
  R2.d : [0..R2.threshold] init 0;
  ...
  [at0_6] true -> (R1.d'=min(R1.d+R1.contaminationrate,R1.threshold))
                & (R2.d'=min(R2.d+R2.contaminationrate,R2.threshold)) & ...;
  [at0_1] true -> (R1.d'=0)
                & (R2.d'=min(R2.d+R2.contaminationrate,R2.threshold)) & ...;
  [at0_2] true -> (R1.d'=min(R1.d+R1.contaminationrate,R1.threshold))
                & (R2.d'=0) & ...;
  ...
endmodule
```

Listing 9: An example of the structure of the *contamination* module using discrete contamination values

Applying the Strategy. Apart from using contamination counters, our model does no longer contain probabilistic choices. Concretely, each probabilistic choice in M (branching to each possible selection of fully contaminated rooms, Listing 3) is replaced by an action in M' performing a simultaneous update of all contamination counters (Listing 9). We can use the generated strategy σ to derive the induced deterministic model M^σ from M', which acts according to the strategy. This step is done by modifying the preconditions of the `atN_N_...` actions of the `time` module to only be able to trigger when that action is chosen at the same point in the strategy. If, for example, within σ, the `at0_1` action is only chosen in time steps 8 and 10, we modify the `time` module, see the listing on the right.

```
module time
  ...
  [at0_1] !error_or_final
          & (t=8|t=10)
          -> (t'=min(t+1,T))
  ...
endmodule
```

Notes on the Relationship between M, M', and M^σ. The state space of M' is significantly larger than the one of M as the latter contains intermediate contamination $\mathscr{R}_j.d$ up to $\mathscr{R}_j.d = \mathscr{R}_j.threshold$. The fact that $\mathscr{R}_j$ gets contaminated in M corresponds to all shortest sequences of transitions with non-zero probability to a state where $\mathscr{R}_j.d = \text{true}$. The same fact in M' corresponds to all sequences of transitions leading to $\mathscr{R}_j.d = \mathscr{R}_j.threshold$. Hence, the time module in M^σ is a refinement of the time module in M and, due to synchronisation (via action labels), the resetting of contamination's (i.e., the cleaning) in M^σ is a refinement of the corresponding resets in M.

A property of M' preserving quantitative strategy correctness, that we left for future work, is to check whether the probabilities of the contamination flags set in M are greater than or equal to the hypothetical

Table 2: Requirements for validating the synthesised strategies (checks of M^σ by PRISM)

Id.	**Strategy Specification** (Behavioural Requirement)	**... expressed in PLTL**
FR	Cleaner B_i Finally Returns to its starting position.	$\bigwedge_{i\in[1..k]} \mathbf{P}_{\geq 1}[\mathbf{F}^{=T} B_i.x = B_i.start]$
ωR	Cleaner B_i has a final charge of at least $B_i.\omega chgthres$.	$\bigwedge_{i\in[1..k]} \mathbf{P}_{\leq 0}[\mathbf{F}^{=T} B_i.c < B_i.\omega chgthres]$
ωC	Contamin. of $\mathscr{R}_i$ is finally less than $\mathscr{R}_i.\omega contthres$.	$\bigwedge_{i\in[1..m]} \mathbf{P}_{\leq 0}[\mathbf{F}^{=T} \mathscr{R}_i.d < \mathscr{R}_i.\omega contthres]$
BC	Battery charge of Cleaner B_i is never 0.	$\bigwedge_{i\in[1..k]} \mathbf{P}_{\leq 0}[\mathbf{F}\, B_i.c = 0]$
CT	Contamination of $\mathscr{R}_i$ never exceeds $\mathscr{R}_i$'s threshold.	$\bigwedge_{i\in[1..m]} \mathbf{P}_{\leq 0}[\mathbf{F}\, \mathscr{R}_i.d \geq \mathscr{R}_i.threshold]$
UT	Room $\mathscr{R}_i$ is not cleaned while occupied.	$\bigwedge_{i\in[1..m]} \bigwedge_{T\in util(\mathscr{R}_i)} \bigwedge_{j\in[1..k]} \mathbf{P}_{\leq 0}[\mathbf{F}^{=T} B_j.x = \mathscr{R}_i]$

probabilities of the corresponding counters in M' and, thus, M^σ, reaching their thresholds. This property, when true, expresses that the flags are a sound (i.e., conservative) quantitative abstraction of the counters.

3.7 Strategy Verification via Verifying the Induced Model

Now, we use M^σ to check *recurrence* and *safety* from Formula (2), that is, $\omega \rightarrow \mathbf{F}^{\leq T}\omega$ and $\mathbf{G}^{\leq T}\phi_{\mathsf{s}}$.[4] In particular, we check their decomposed translations into PLTL requirements[5] listed in Table 2. The recurrence area ω is encoded by the state propositions in FR, ωR, and ωC, while safety ϕ_{s} is encoded by the state propositions in BC, CT, and UT. Note that the upper bound T of the recurrence interval is met by all paths in M^σ. $util(\mathscr{R}_i)$ is the set of time slots in which $\mathscr{R}_i$ is utilised.

For checking $\omega \rightarrow \mathbf{F}^{\leq T}\omega$, we define ω to be a (not necessarily maximum) region in S from where σ can be applied and ω is revisited after T steps. The requirements for σ need to be true for every initial state in ω. FR, ωR, and ωC ensure that applying σ leads to a state within ω. To specify ω, we use thresholds for the battery charge of robots ($B_i.c$ for every robot B_i) and the contamination of rooms ($\mathscr{R}_i.\omega contthres$ for every room $\mathscr{R}_i$). ω then characterises every state of M^σ where the battery charge and room contamination are within these thresholds and all robots are on their starting position. Recurrence can even be checked more easily for M^σ by selecting the worst state in ω, which is the state where every value lies exactly at the thresholds, and verifying the requirements in Table 2 for this state.

4 Experimental Evaluation

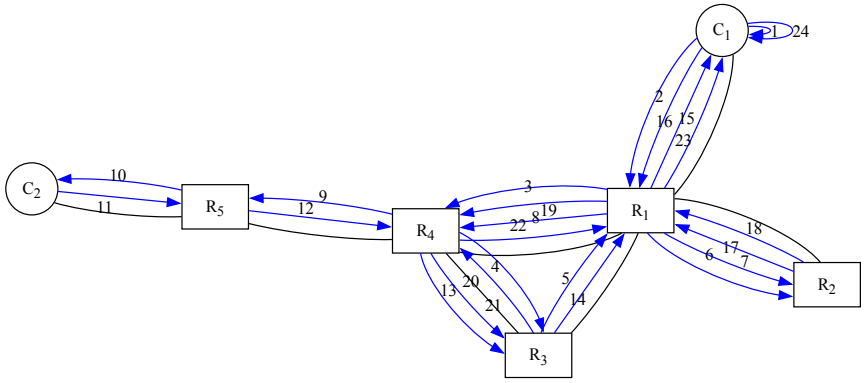
Our experimental evaluation addresses two research questions (RQs).

4.1 RQ1: Can we synthesise reasonable strategies for multiple robots?

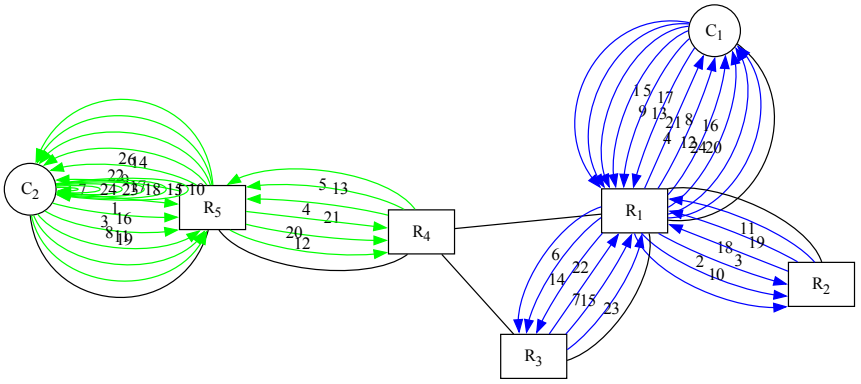
In the following, an instance of the example model with only one robot is considered first. The contamination rate is the same for all rooms, except that $\mathscr{R}_5.d$ has a threshold value $\mathscr{R}_5.threshold$ of 24 (based on a contamination rate of 1 h^{-1}), which is twice as high, whereas all other rooms have a threshold value of 12. We identified ω manually by examining the generated strategy. Using the method described in Section 3, a strategy was generated that meets all the requirements. This strategy is visualised in Figure 6a. Each action is shown with a blue arrow, at which the time step in which the action is to be executed is annotated. To develop a strategy for two robots, the battery charge was halved to keep the

[4] For the sake of simplicity of the example, we use $\phi_{\mathsf{r}} \equiv \top$ and can omit $\omega \rightarrow \mathbf{F}^{\leq T}\phi_{\mathsf{r}}$.

[5] All properties are expressed in quasi-LTL, that is, ACTL* allowing only one universal quantifier at the outermost level.



(a) Strategy for one robot



(b) Strategy for two robots (blue and green directed arcs)

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…	24
B_1	**C_1**	**$\mathscr{R}_1$**	**$\mathscr{R}_2$**	**$\mathscr{R}_1$**	C_1	**$\mathscr{R}_1$**	**$\mathscr{R}_3$**	**$\mathscr{R}_1$**	**C_1**	**$\mathscr{R}_1$**	**$\mathscr{R}_2$**	**$\mathscr{R}_1$**	C_1	**$\mathscr{R}_1$**	**$\mathscr{R}_3$**	**$\mathscr{R}_1$**	…	**C_1**
B_2	**C_2**	**$\mathscr{R}_1$**	C_2	**$\mathscr{R}_5$**	**$\mathscr{R}_4$**	**$\mathscr{R}_5$**	C_2	idle	**$\mathscr{R}_5$**	C_2	idle	**$\mathscr{R}_5$**	**$\mathscr{R}_4$**	**$\mathscr{R}_5$**	C_2	idle	…	**C_2**

(c) Execution of a 24h model cycle using the strategy in Figure 6b; (sub-)cycles indicated in bold

Figure 6: Synthesised strategies. Nodes represent charging stations and rooms; undirected arcs indicate room connections (doors); edge labels specify the execution order of particular `at` actions.

model less complex. The corresponding strategy can be seen in Figure 6b. This result turns out to be a *partition-based patrolling strategy*, considered effective under random disturbance [15, 143].

4.2 RQ2: How do model parameters influence the synthesis of recurrent strategies?

We evaluate the model using *a_bit*, $\mathscr{R}_i.pr$, and the fixed-grid resolution g as parameters. Figure 7 visualises the result using these parameters on a simplified model, which only contains one robot and omits room $\mathscr{R}_5$ and charger C_2. When building σ, we set $\mathscr{R}_i.pr = cumulative\_probability/|\mathscr{R}|$. The generated strategies were verified by iteratively assuming ω-thresholds (Table 2) from a set chosen appropriately.

Figure 7 contains four plots for resolutions $g = 1..4$. Correct non-recurrent strategies are represented by a blue dot, correct recurrent strategies by a green dot, and incorrect strategies in red.

We deemed the simplification of the model necessary to allow a timely execution of the test series, which contained 400 experiments in total (100 for each grid resolution). Detailed information about the

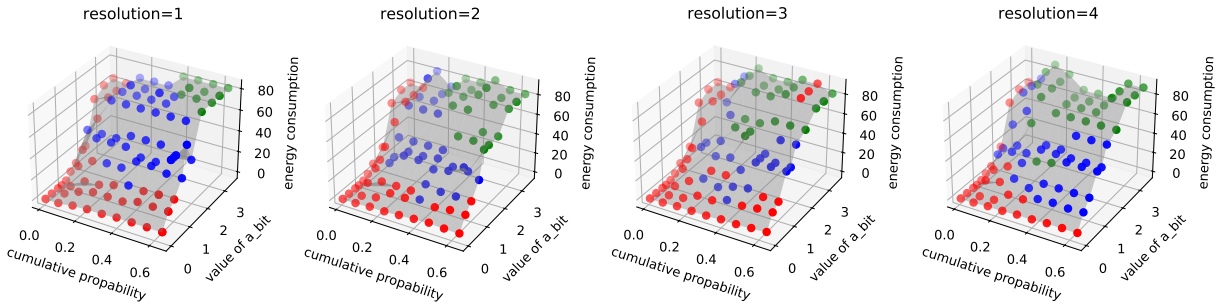


Figure 7: Parameterised evaluation. Non-recurrent strategies are marked in blue, recurrent strategies in green. The scale for *a_bit* is logarithmic.

time of the evaluation is shared at the end of the section.

As expected, a smaller cumulative contamination probability leads the strategy to de-prioritise cleaning the rooms regularly, since the probability of them being contaminated stays low. This, at some point, causes the strategy to not satisfy the recurrence criteria. Similarly, a lower value of *a_bit* causes the strategy to prioritise saving battery over resetting the contamination flags, which reduces the energy consumption significantly, but at some point at the cost of strategy correctness.

In the evaluation, the optimal strategy per grid resolution uses less or equal energy with a larger grid resolution (84, 64, 64, and 36 for g of 1, 2, 3, and 4, respectively). There is also a difference in the number of distinct strategies that are generated between the different g, where the experiment with a resolution of 4 generates more unique strategies than the experiments with lower grid resolutions, where many parameters lead to the same generated strategy.

Finally, we can observe distinct areas of incorrect, non-recurrent and recurrent strategies that depend on these parameters, where the optimal recurrent strategy lies on the border between recurrent and non-recurrent strategies. While the states in ω are ordered for the choice of a worst case, the border area is at best an approximation of a Pareto front, the non-convex reward function defined by R combined with the belief-MDP approximation $B(M)$ may lead to optimal strategies remaining hidden from the search.

Beyond the reward structure $R^{utilisation}$ used for strategy pre-selection, checking the PLTL safety property UT (Table 2) ensures that the finally chosen strategy only cleans outside the room utilisation schedule (Figure 1b).

Some Key Data. The experiments were conducted on an AMD FX(tm)-8350 Eight-Core Processor with 32 GiB of RAM running Ubuntu 22.04.4 LTS. However, PRISM was restricted to one core and 12 GiB of RAM. The reduced model in Section 4.2 contains 4879 states and 35014 transitions, while the reduced model M^{σ} contains 23 states and 23 transitions. The verification consists of 13 PLTL formulas containing 25 propositions. We ran the experiments with parameters of $m = 4$, $\mathscr{R}_i.pr = 0.02, 0.04, \ldots, 0.16$ and a_bit = $1, 3, 6, 10, 17, 32, 100, 316, 1000, 3162$. The *cumulative probability* can be derived from $\mathscr{R}_i$: $\sum_{i=1,\ldots,m} \mathscr{R}_i.pr = 0.08, 0.16, \ldots, 0.64$. The strategy synthesis took about 5, 16, 60, and 200 seconds for a grid resolution of 1, 2, 3, and 4, respectively, while the verification took about one second for a given ω. Evaluating the entire test series took about 8 hours sequentially, although this process could be easily parallelised.

5 Discussion

Selecting the Recurrence Area ω. When evaluating the strategy, ω was chosen either by examining the strategy manually (Figure 6c) or by verifying a list of probable ωs. Further work may focus on finding probable ωs from the room layout, and generating strategies which fulfil these ωs.

Complexity of the Cleaning Scenario. For the evaluation, we considered a rather simple room layout. The performance of the above described method may be different with larger room graphs, more complex room layouts, a larger number of robots, and tighter restrictions on battery charge and room utilisation.

Moreover, our model allows us to find strategies that operate with a varying number of robots. Given that some robots remain idling all the time, our optimal synthesis could also be used to find the smallest subset $B_{\min} \subseteq B$ or minimal number $k_{\min} \leq k$ of robots for an optimal task performance.

The complexity of the model is heavily dependant on the number of rooms, the maximum time T, and the maximum charge of the robots. Following the comprehensive scheme in Section 4.2, we were able to calculate a 12-hour ($T = 12$) cleaning schedule for 3 robots with 11 rooms and a maximum charge of 6 in 15 hours. The corresponding belief-MDP $B(M)$ contains $\approx$ 690k states and $\approx$ 11.8m transitions.

Adjusting Grid-Resolution vs. Filtering Strategies. For industry-size POMDPs, a high resolution g can lead to an impractically high computational effort when solving the mostly NP-hard approximate analysis (i.e., verification, synthesis) problems. Hence, our approach is to keep g just fine enough to find some (not necessarily globally optimal) strategy σ and verify more nuanced properties of the quasi-MDP[6] M^σ derived from M by applying σ. In M^σ, verification is simpler (no belief-MDP $B(M)$ is computed), also the strategy (integrated in M^σ) can directly observe the outcome of each action and does not have to memorise a finite observation history. Despite the expansion of $\mathscr{R}_i.d$ to integers, M^σ's state space is expected to be smaller than $B(M)$'s state space for the applied values of g.

Parameter Selection. For the evaluation in Section 4.2, a set of values for the parameters *a_bit* and the *contamination probability* was chosen. Via g (Section 2), we reduced the resolution of the fixed grid (i.e., a wider grid width) to limit the number of states in the belief space approximation $B(M)$. However, our findings in Section 4.2 suggest that increasing the resolution, while keeping the cumulative contamination probability around 40 % and the value of *a_bit* around 300 leads to the synthesis of better strategies. However, these values may not be universally favourable for any room layout, and it may be possible to synthesise better strategies using a different set of parameters. Further work may focus on better ways of parameter selection.

Generalisation to Other Applications. The running example in our case study focuses on a cleaning robot collective. However, we think that our approach and model can be transferred rather straightforwardly to other spatio-temporal settings with recurrent tasks, for example,

- firefighting drone collectives tasked with repetitive sector-wise fire detection and water distribution and with partially observable quantities such as ground temperature and extinction level;
- geriatric care robot collectives tasked with recurrent monitoring and care-taking tasks (e.g., medication supply) with patient satisfaction and health status being partially observable;
- general patrolling collectives tasked with monitoring or supervising specific environments [15].

[6] non-probabilistic, deterministic, with full observability

6 Conclusion

We proposed an approach using weighted, partially observable stochastic models (i.e., reward-enhanced POMDPs) and strategy synthesis for optimally coordinating tasked robot collectives while providing recurrence and safety guarantees on the resulting strategies under uncertainty. Along with that, we discussed guidance on POMDP modelling and strategy synthesis. We focused on a cleaning robot scenario for public buildings, such as schools. Our notion of *correctness* combines (i) safe recurrence (e.g., repetitively accomplish the cleaning task while avoiding to collect penalties), (ii) robustness (e.g., correctness under worst-case contamination), and (iii) optimality (e.g., minimal energy consumption).

For scaling up strategy synthesis to scenarios beyond what can easily be tackled by stochastic game-based synthesis, we addressed the key challenge [6] of reducing the state space and the transition relation of a naïve model via partial observability (hiding details of stochastic room contamination) and by employing simultaneous composition (for robot movement). PRISM's grid-based POMDP approximation allowed us to adjust the level of detail of the belief space to synthesise strategies more efficiently. Furthermore, we softly encode the strategy search space using penalties and optimisation rewards and can, thus, shift the verification of more complicated properties to a later stage working with an unweighted and non-probabilistic behavioural model, again using a more detailed, numerical state and action space. However, decoupling synthesis from verification can require time-consuming experiments (Section 4.2) to identify regions of the parameter space for ensuring the existence of good recurrent strategies.

In future work, we will improve finding ω ensuring the existence of correct strategies (i.e., green dots in Figure 7). Ideally, we avoid defining ω explicitly (e.g., by hiding time). In a larger example, we want to allow invariant-narrowing with ϕ_r and observable stochasticity in the environment, such that σ can depend on arbitrary variables. The reset of the contamination flag on a room visit (Db) could be refined by a decontamination rate in M^σ. Moreover, we aim to use multi-objective queries to include further criteria (e.g., minimal contamination) for Pareto-optimal strategy choice. While PRISM imposes some limits on the combination of queries and constraints, we will need to see how we can use tools such as EVOCHECKER (as, e.g., used in [17]) for POMDPs. Also, we can further reduce the action set by taking into account trajectory intersections in the simultaneous movements (cf. Figure 4a). Finally, we want to connect the synthesis pipeline with code generation, such as demonstrated in our previous work [5].

References

[1] Christel Baier & Joost-Pieter Katoen (2008): *Principles of Model Checking*. MIT Press.

[2] Davide Basile, Maurice H. ter Beek & Axel Legay (2020): *Strategy Synthesis for Autonomous Driving in a Moving Block Railway System with UPPAAL Stratego*. In: *FORTE*, *LNPSE* 12136, Springer, pp. 3–21, doi:10.1007/978-3-030-50086-3_1.

[3] DIN (2015): *DIN 77400: Reinigungsdienstleistungen - Schulgebäude - Anforderungen an die Reinigung*. Standard, DIN. Available at `https://www.dinmedia.de/de/norm/din-77400/237208488`.

[4] Ruben Giaquinta, Ruth Hoffmann, Murray Ireland, Alice Miller & Gethin Norman (2018): *Strategy Synthesis for Autonomous Agents Using PRISM*, p. 220–236. Springer, doi:10.1007/978-3-319-77935-5_16.

[5] Mario Gleirscher, Radu Calinescu, James Douthwaite, Benjamin Lesage, Colin Paterson, Jonathan Aitken, Robert Alexander & James Law (2022): *Verified Synthesis of Optimal Safety Controllers for Human-Robot Collaboration*. *Sci. Comput. Program.* 218, p. 102809, doi:10.1016/j.scico.2022.102809. arXiv:2106.06604.

[6] Mario Gleirscher, Jaco van de Pol & James Woodcock (2023): *A Manifesto for Applicable Formal Methods*. *Softw. Syst. Model.* 22, pp. 1737–1749, doi:10.1007/s10270-023-01124-2. arXiv:2112.12758.

[7] Rong Gu, Peter G. Jensen, Cristina Seceleanu, Eduard Enoiu & Kristina Lundqvist (2022): *Correctness-guaranteed strategy synthesis and compression for multi-agent autonomous systems*. Sci. Comput. Program. 224, p. 102894, doi:10.1016/j.scico.2022.102894.

[8] Umweltbundesamt (Hrsg.) (2008): *Leitfaden für die Innenraumhygiene in Schulgebäuden*. Available at `https://www.umweltbundesamt.de/sites/default/files/medien/publikation/long/3689.pdf`.

[9] Bruno Lacerda, David Parker & Nick Hawes (2017): *Multi-Objective Policy Generation for Mobile Robots Under Probabilistic Time-Bounded Guarantees*. In: Automated Planning and Scheduling (ICAPS), 27th Int. Conf., pp. 504–512, doi:10.1609/icaps.v27i1.13865.

[10] William S. Lovejoy (1991): *Computationally Feasible Bounds for Partially Observed Markov Decision Processes*. Oper. Res. 39(1), p. 162–175, doi:10.1287/opre.39.1.162.

[11] Owen Macindoe, Leslie Pack Kaelbling & Tomás Lozano-Pérez (2012): *POMCoP: Belief Space Planning for Sidekicks in Cooperative Games*. In: AIIDE, The AAAI Press, pp. 38–43, doi:10.1609/aiide.v8i1.12510.

[12] Pierre El Mqirmi, Francesco Belardinelli & Borja G. León (2021): *An Abstraction-based Method to Check Multi-Agent Deep Reinforcement-Learning Behaviors*. In: AAMAS, pp. 474–482, doi:10.5555/3463952.3464012. arXiv:2102.01434.

[13] Gethin Norman, David Parker & Xueyi Zou (2017): *Verification and control of partially observable probabilistic systems*. Real-Time Systems 53(3), p. 354–402, doi:10.1007/s11241-017-9269-4.

[14] Dave Parker, Gethin Norman & Marta Kwiatkowska (2024): *PRISM Model Checker*. Available at `http://www.prismmodelchecker.org/manual/`.

[15] David Portugal & Rui Rocha (2011): *A Survey on Multi-robot Patrolling Algorithms*, pp. 139–146. Springer, doi:10.1007/978-3-642-19170-1_15.

[16] Antony Thomas, Fulvio Mastrogiovanni & Marco Baglietto (2021): *MPTP: Motion-planning-aware task planning for navigation in belief space*. Robot. Auton. Syst. 141, p. 103786, doi:10.1016/j.robot.2021.103786.

[17] Gricel Vázquez, Radu Calinescu & Javier Cámara (2022): *Scheduling of Missions with Constrained Tasks for Heterogeneous Robot Systems*. In: Proceedings Fourth International Workshop on Formal Methods for Autonomous Systems (FMAS) and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE), EPTCS 371, Open Publishing Association, pp. 156–174, doi:10.4204/eptcs.371.11. arXiv:2209.14040.

A Case Study on Numerical Analysis of a Path Computation Algorithm

Grégoire Boussu
Thales Research & Technology
Palaiseau, France
gregoire.boussu@thalesgroup.com

Nikolai Kosmatov
Thales Research & Technology
Palaiseau, France
nikolai.kosmatov@thalesgroup.com

Franck Védrine
Université Paris-Saclay, CEA, List
Palaiseau, France
franck.vedrine@cea.fr

Lack of numerical precision in control software — in particular, related to trajectory computation — can lead to incorrect results with costly or even catastrophic consequences. Various tools have been proposed to analyze the precision of program computations. This paper presents a case study on numerical analysis of an industrial implementation of the fast marching algorithm, a popular path computation algorithm frequently used for trajectory computation. We briefly describe the selected tools, present the applied methodology, highlight some attention points, summarize the results and outline future work directions.

1 Introduction

Numerical precision of algorithms has become an important concern for modern critical software. Accumulation of rounding errors can lead to serious issues in programs involving floating-point numbers. Such accumulated errors can significantly affect the accuracy of computations and lead to incorrect results. Even for a mathematically correct algorithm — considered in real numbers — its computer implementation can give inaccurate or incorrect results if this implementation does not properly take into consideration numerical precision aspects of the resulting computation in floating-point numbers. In critical software, in particular in control software related to trajectory computation, lack of numerical precision can lead to incorrect results with costly or even catastrophic consequences. Well-known examples include the Patriot missile failure in 1991[1] and the crash of Ariane 5 in 1996[2].

The fast marching algorithm [10] is a popular path computation algorithm frequently used for trajectory computation in autonomous systems. It answers the question of which path is optimal between two given nodes, that is, has the shortest time or, more generally, the smallest weight. The algorithm works in two steps. A first step performs a forward wave front propagation from the given origin point, computing the time the wave front will take to reach each point (of the plan, or grid, or graph). A second step uses the resulting computation to perform a backward propagation from the final point to the origin point in order to compute an optimal path. This algorithm has various applications for trajectory computation and image segmentation. The purpose of this work is to investigate the numerical precision of an industrial implementation by Thales of this algorithm over a discrete grid.

[1] See https://www-users.cse.umn.edu/~arnold/disasters/Patriot-dharan-skeel-siam.pdf.
[2] See https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html.

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 126–142, doi:10.4204/EPTCS.411.8

Numerical analysis of such trajectory computation algorithms is a very challenging and time-consuming task. Execution paths in the code are typically very long and go through many instructions. Each of them can have an impact on precision and robustness of the algorithm. Indeed, such a path can go through many *unstable branches*, that is, branches after a conditional expression for which a small imprecision of computation or a small variation of input values can change the truth value of the condition and lead to another branch in the code (ex: *else* instead of *then* branch of a conditional statement, or one more loop iteration), possibly impacting the rest of the algorithm. Rounding a floating-point number to an integer can have a similar impact when the resulting integer is later used in the code: if a value around 100.0 can be rounded to 99 or 100, it can potentially have a significant impact. Moreover, since the algorithm simulates a continuous real space by a discrete grid, a deviation at one node can easily involve different nodes and thus lead to a quite different result.

Various techniques and tools have been proposed to analyze the precision of program computations. They include dynamic analysis and static analysis techniques. In this work, we use three popular numerical analysis tools: Cadna [7] and Verrou [6] realizing (possibly unsound) dynamic analysis, and FLDLib [12] performing a combination of sound abstract interpretation and dynamic path exploration.

Contributions. This paper presents a case study on numerical analysis of an industrial implementation of the fast marching algorithm. While the considered implementation is currently not publicly available, the underlying algorithm is classic, therefore we believe that the presented methodology and findings can be of interest for other implementations of similar (and possibly other) algorithms. We briefly describe the selected tools, present the applied methodology combining several tools, highlight some attention points, summarize the results and outline ongoing and future work directions.

Outline. Section 2 presents the considered algorithm. Section 3 describes the verification methodology, the selected tools and our findings. Section 4 provides a conclusion and future work perspectives.

2 The Verification Target: the Fast Marching Algorithm

This section provides a simplified presentation of the problem and the implemented algorithm without giving all technical and theoretical details (which are not mandatory for understanding the paper). For a more thorough description of theory behind the Fast Marching Algorithm, one may refer to [11].

The problem under consideration for the study is named the *minimum-cost path problem*. On a finite graph with weighted edges, this problem can be stated as follows: which path to take between two specified vertices so that the sum of weights along this path is the lowest among all possible paths between the two nodes. When the weight is (seen as) the distance between the nodes, this problem is also called the *shortest path problem*, and the cost is (seen as) the time to reach the point. Different algorithms exist to solve the shortest path problem (e.g. Dijkstra, Bellman-Ford).

In our case, we are interested in the definition of the *minimum-cost path problem* in the continuous case: let us consider the problem in $\mathbb{R}^n$. A cost density function $\tau : \mathbb{R}^n \to (0,\infty)$ gives the cost at each point of the space. The *minimum-cost path problem* between A and B, two points in $\mathbb{R}^n$, is to find a path $c(s) : [0,\infty) \to \mathbb{R}^n$ that minimizes the cumulative cost (often interpreted as the arrival time) from A to B. The *cumulative cost* for a path c between A and M is:

$$T_c(M) = \int_0^l \tau(c(s))\, ds$$

where l is the length of path c between A and M, $c(0) = A$ and $c(l) = M$. Therefore, c_{sol} is a solution to the problem if and only if $c_{sol} \in \mathscr{C}$ where $\mathscr{C}$ is the set of all paths c between A and B with the minimum $T_c(B)$.

As stated in [10], if c_{sol} is a solution to the problem, it satisfies the equation below, named the Eikonal equation, for all $M \in c_{sol}$:

$$||\nabla T_{c_{sol}}(M)|| = \tau(M)$$

where ∇ denotes the gradient, and $||\cdot||$ denotes the Euclidean norm. This equation is, in particular, a way to describe the propagation of a wave front initiated in point A. The front speed at point M is given by $1/\tau(M)$, and $T_{c_{sol}}(M)$ is the time of arrival of the front from point A to point M.

In the presented definition of the problem, the value of τ at a given point depends only on the point's location. This case is qualified as *isotropic*. If τ also depends on the direction of the path at the point, the cost function is *anisotropic*. A method to solve this equation in the case of an isotropic problem discretized on a Cartesian grid was proposed by Sethian in 1995 [10] and has become the starting point for many extensions. This method, named *fast marching method* (FMM), shares many aspects with Dijsktra's algorithm. Once the equation is solved for all points of the grid, a second step is necessary to figure out the (or one of the) optimal path solution(s) to the minimum-cost path problem. We will go over the two steps sequentially.

2.1 First Step: Solving the Eikonal Equation

Basically, given a grid and a starting point A of the grid, Sethian's method allows one to calculate the arrival time $T_c(M)$ to any point M of the grid over a minimum-cost path c starting from point A. Each point M of the grid has 4 neighbors as shown in Fig. 1. Based on a relevant approximation scheme, $T_c(M)$ is calculated considering the possibilities that the wave about to reach the point M comes from North-East (with a contribution from the neighbors above and on the right), or South-East (with a contribution from the neighbors below and on the right), or South-West (with a contribution from the neighbors below and on the left, as shown in Fig. 1) or North-West (with a contribution from the neighbors above and on the left). Starting the algorithm with $T_c(A) = 0$ and $T_c(M) = \infty$ for all $M \neq A$, and picking the next point M to study in an appropriate order, the process progressively computes the arrival time $T_c(M)$ (or more precisely, its approximation due to the discretization) for all points M of the grid.

We can further explain the process using the interpretation of the equation with a wave front, illustrated in Fig. 2. The black points of the grid have their final value $T_c(M)$ computed, the gray ones have a tentative value $T_c(M)$ computed, and the white ones still have $T_c(M) = \infty$, as set at the initialization step.

The set of gray points is named the *narrow band*. Intuitively, at each step of the algorithm, the black points have already been reached by the wave, and at least one of the gray points will be reached next, before any of the white points will be reached. Just like in a classic implementation of Dijkstra's algorithm, a priority queue is used to store the gray points. When a point M is selected from the queue, its neighbors M' enter the queue (if they were not already part of it) and get their arrival time values $T_c(M')$ calculated or updated based on $T_c(M)$. The next point M to be selected in the queue is the one with the lowest tentative arrival time $T_c(M)$. When selected, such a gray point gets its tentative value $T_c(M)$ turned to the final one, and the point itself is removed from the queue and labeled as black. Intuitively, since the tentative arrival time of the wave to this point is the smallest one among the gray nodes, it cannot be reached even faster through some other node (for which the arrival time will necessarily be bigger) hence the computed arrival time to it is final. At the beginning of the algorithm, the priority queue is initialized with A.

The fast marching method has two interesting features:

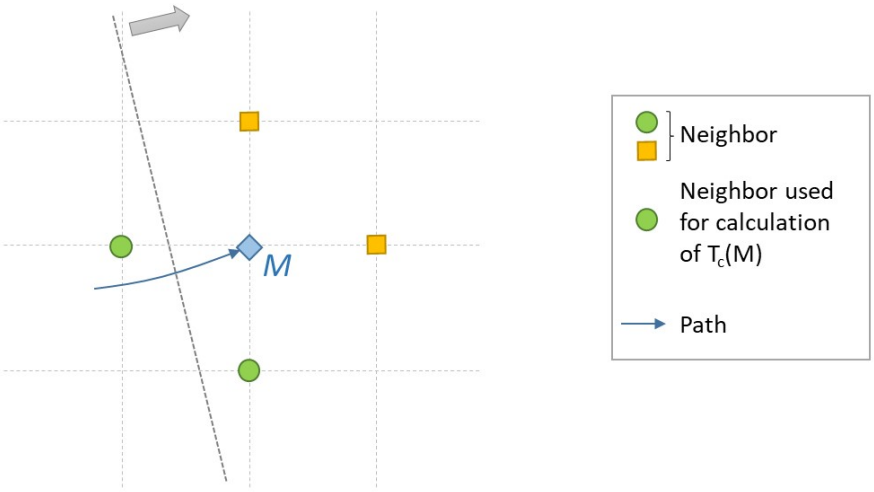


Figure 1: Neighbors selected for calculation of $T_c(M)$

- It is efficient in terms of computational complexity. Indeed, its complexity is similar to that of Dijsktra's algorithm, and is of $\mathcal{O}(n \lg(n))$, with n being the number of points of the grid.
- It can be proven that the FMM produces a solution that satisfies everywhere the discrete version of the Eikonal equation, leading to an approximation of its so-called *viscosity* solution (see for example [4] on viscosity solutions). So when the grid spacing tends to 0, the solution provided by the FMM algorithm tends to the continuous solution of the equation.

Though, as $T_c(M)$ is calculated based on the T_c of the neighbors of M, the calculation errors may propagate over the entire grid. Added up, these errors may lead to a discrepancy for points far from point A and impact the precision of the global result expected from using FMM. Studying the order of magnitude of this discrepancy is of great interest to be confident in the implementation of the algorithm.

2.2 Second Step: Finding an Optimal Path by a Backward Propagation

The theory provides a way to find an optimal path, thanks to a property of such a path: its direction is always normal to the wave front [1]. To produce the result, a so-called back-propagation from the final point (supposed to be on the grid) is realized, based on a gradient descent following the direction perpendicular to the wave front curve.

Though, once the arrival time values $T_c(M)$ are calculated for each point of the grid, we are still in a discrete space and the gradient calculation is not straightforward. The gradient descent can be approximated by selecting for each point its predecessor among the neighbors, the appropriate one being the (or one of the) neighbor(s) with the lowest $T_c(M)$. But this approach leads to a path made of following segments that can be perpendicular one to the next. Moreover, aggregating the length of each segment of the path will generally lead to a value overestimated compared to the optimal path length in a continuous space. It would be preferable to provide a visually smooth path with its length approximating the length of the viscosity solution of the Eikonal equation.

Such an alternative can be implemented with the following approach: starting from the final point of

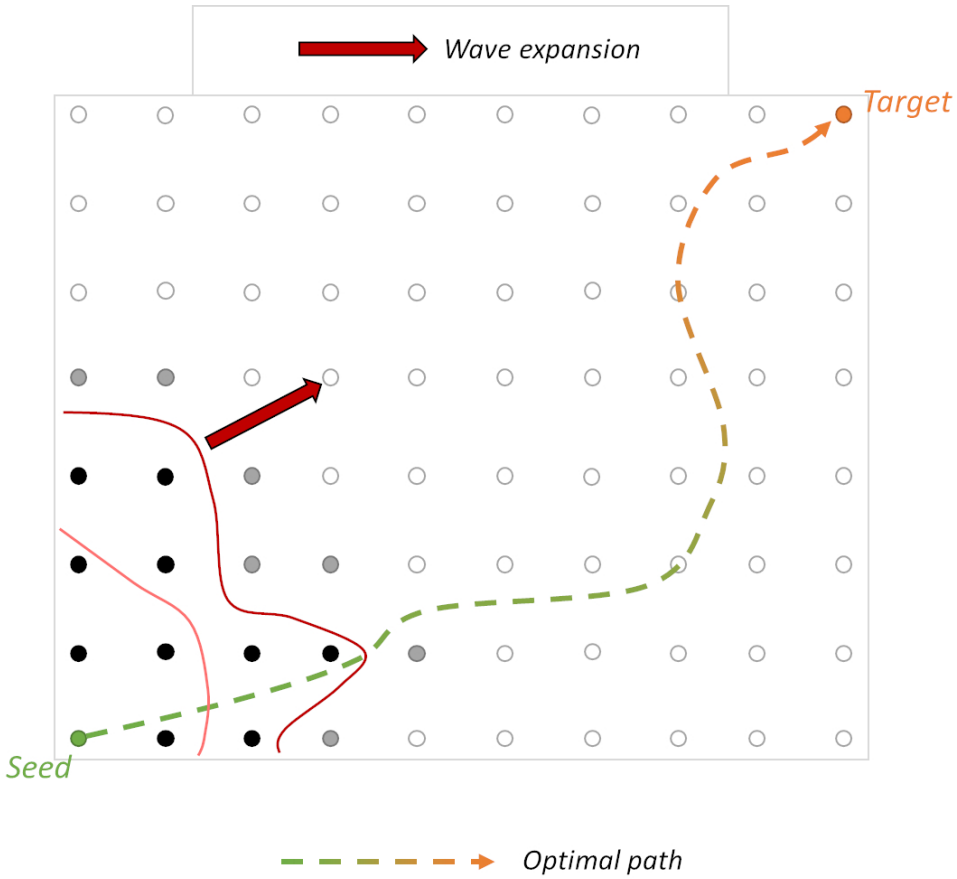


Figure 2: Propagation of the wave front

the path, a *pseudo*-gradient is calculated on each segment around this point, as shown in Fig. 3. The best point on the segments, i.e. the point (or one of the points) minimizing $\Delta T_c/distance$ (that is, maximizing the speed of the wave) is selected as the previous point of the approximated path. A similar approach is taken to find out the best point when the gradient is calculated from any point in the middle of a segment. This leads to a much smoother path, whose length provides a good approximation of the expected length of the viscosity solution.

Just as in the case of the fast marching algorithm, the calculation is made one point after the other. Therefore, the calculation errors due to the implementation can lead to an aggregated discrepancy. An analysis of sensitivity of the implementation to these errors is thus required.

2.3 Applications of These Algorithms

Many fields of application exist for these algorithms. The first is of course related to path calculation leading to the shortest time between two points, considering the speed of the mobile agent depending on its position in an area. A less obvious application could be image segmentation [3]. To allow for different and more specific situations, many extensions to the method have been developed. To name a few: the possibility to deal with anisotropic costs [8], or with time-dependent costs with no restriction on sign [2], or the extension taking into consideration constraints like minimum turning radius of the

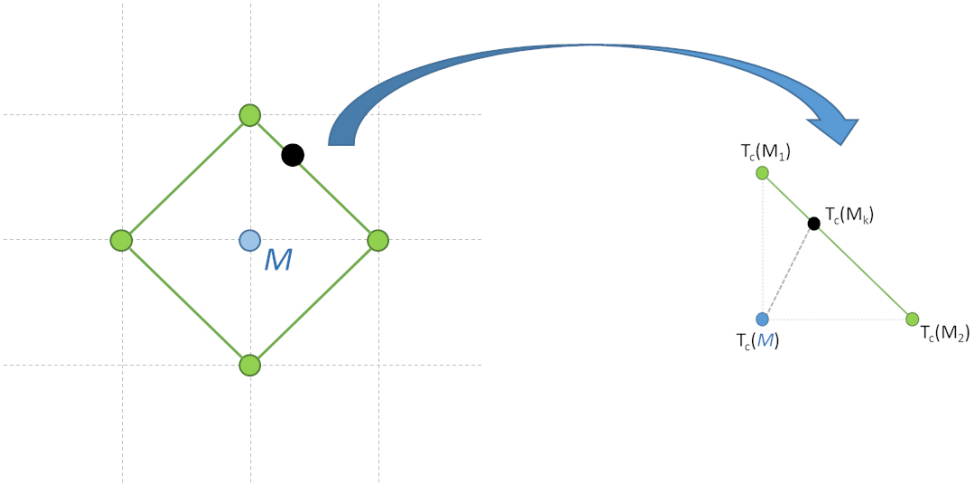


Figure 3: Calculation of pseudo-gradient on a segment

moving agent [9].

In our case, the fast marching method is a general approach to produce paths optimizing any kinds of criteria (or a mix of criteria). The most straightforward situation would be to aim at minimizing time to reach a location, while the area through which we can move is made of danger-free zones where the speed can be high, and others surrounded by dangers (mountains, ...) where the velocity should be reduced. Let us consider another example where time is not the criterion to optimize: the pilot of a plane wishes to avoid turbulence areas ahead (considered as static). He may want to find a good balance between disturbance due to very strong winds and the additional distance incurred by avoiding these areas. If the plane has a steady cruising speed, by defining the cost function τ with high values in the center of the turbulence areas and decreasing values towards the outside, the fast marching method can provide an appropriate path to follow (see Fig. 4).

For use cases where a lack of precision can generate additional risks (e.g. air traffic, autonomous drones), aggregated calculation errors can significantly impact the result of the computation. This concern motivated the current study.

3 The Verification Approach and Results

The target implementation of the path computation algorithm contains more than 6,200 lines of C++ code and provides several test cases. They include realistic test cases over a square grid with 200x200 nodes and obstacles simulated by higher weights, as illustrated by Fig. 4. Internally, the code uses some C++ STL (Standard Template Library) containers, like vectors, maps and priority queues. So, formal numerical analysis of this implementation and its adequacy with respect to the underlying mathematical formulas within a short period of time requires concentrating on successive research questions:

RQ1: What is the accuracy of the computed path cost?

RQ2: Is the computation robust (meaning that a small perturbation of inputs leads only to a small variation of outputs)?

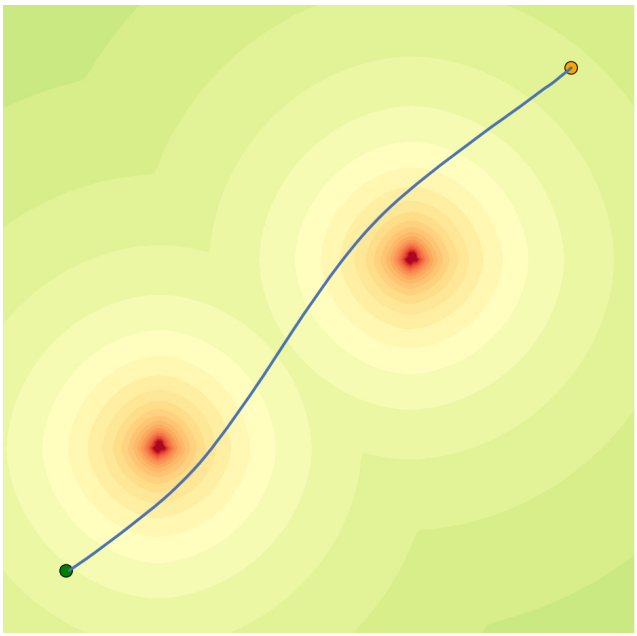


Figure 4: Result of calculation of a path avoiding turbulence areas

RQ3: How does the computed path compare to the path obtained on a more or less precise grid (say, with twice more or twice fewer points on each side)?

This paper focuses on RQ1 and RQ2, while RQ3 is left for future work. To address these questions, we decided to apply the following methodology that was successfully applied earlier on some simpler numerical use-cases, except the last item that is more related to deductive verification:

- instrument the tests with different numerical analysis libraries to identify the difficulties in obtaining relevant analysis results and then refine the verification objectives, such as accuracy requirements;
- enlarge the tests into analysis scenarios to check whether the analysis scales up and still provides precise results. Fine-grained analysis scenarios typically replace concrete input values by very small input intervals and then apply conservative interval operations; larger analysis scenarios can also be considered;
- apply modular formal verification to the components of the target implementation and assemble the reasoning results to provide a proof of global correctness.

After presenting the common instrumentation in the next section, we will apply Cadna to address RQ1, Verrou to address RQ1 and RQ2 by comparing with Cadna results, and FLDLib to address RQ1 and RQ2 to investigate the unstable branches that may have a significant impact on the robustness results.

3.1 A Common Instrumentation Mechanism for Different Verifications

The mechanism for building the target implementation of the fast marching algorithm uses the `cmake` tool; so a slight modification of the file `CMakeLists.txt` enables adding some new executable targets compiled with specific compilation flags. This feature enables the source code to be easily compiled with

analysis libraries into a single executable. For the verification purposes, this instrumentation mechanism competes with abstract interpreters when the abstractions to be used are generic (intervals, affine forms). But our case study also requires the elaboration of specific abstractions. So, to quickly explore and debug these newly created abstractions, an instrumentation based on C++ operator overloading and template/-macros mechanisms seemed to us more efficient than using an existing generic abstract interpreter.

Our default instrumentation mechanism replaces `double` and `float` types with data structures carrying analysis information like accuracy, as it is often done by instrumentation libraries [7, 12]. Such data structures implement an overload for the arithmetic operations (`+`, `-`, `*`, `/`, `pow`) to infer numerical properties like the accumulation of round-off errors in numerical computations. Integer types like `int` or `unsigned int` are not instrumented by default. But, the source code can use explicit intrumentation for these types by replacing `int` by `EnhancedInteger<int>` whenever it makes sense for some `EnhancedInteger` template class to define. The C++ compiler helps then to statically propagate these custom types on the source code since an operation manipulating `int` and `EnhancedInteger<int>` generates an error if its result is assigned to an `int` and not to an `EnhancedInteger<int>`.

For each analysis target, the file `CMakeLists.txt` adds specific compilation flags like `-I.../analysis_include -include std_header.h -DFLOAT_MY_ANALYSIS` to build the target. The directory `.../analysis_include` contains the file `std_header.h` that conditionally loads the appropriate analysis data structures for the flag `FLOAT_MY_ANALYSIS` and replaces the `double` type with the macro `double` defined by `#define double EnhancedFloatingPoint<double>`.

The research questions stated in the beginning of this section systematically compare two or more executions. These executions can (and do) follow different control flows (that is, different execution paths) in the target program. In our experiments, we instrument the code with three different strategies:

1. A single run of a synchronous analysis with a single control flow: this single analysis run propagates complete analysis information for every variable at every point of the execution path until the end of the program.
2. Multiple runs of asynchronous analyses with a single control flow: a run propagates partial analysis information until the end of the program. With multiple runs, the user can compute the analysis result as a model from the correlated input/output data.
3. A single run of a synchronous analysis with multiple control flows: this single analysis run propagates complete analysis information and thanks to additional local loops, it covers all possible execution paths (which corrects a weakness of Strategy 1 with an additional instrumentation and execution cost).

For the last strategy, we use `SPLIT/MERGE` macros introduced and used by FLDLib [12]. A pair of such macros (`SPLIT` and `MERGE`) define a so-called `SPLIT/MERGE` section: it expands into a local loop that iterates over all the reachable control flows of the `SPLIT/MERGE` section in order to analyze them one after another. A local memory defined in the `SPLIT` macro saves the memory before the section and restores it at the beginning of the loop body for an exploration of a new control flow. `SPLIT` also saves a control flow identifier for an exploration of a new execution path of the section. It then increments this identifier to cover another execution path in the next loop iteration. At the end of the loop, `MERGE` incrementally synchronizes the results of the local analyses to create a single analysis summary per variable. The analysis is then continued with this summary until the end of the program.

Beyond these generic principles, the instrumentation may encounter some problems listed below, which may require minor adaptations to the source code for analysis purposes. In practice, the first two problems are absent in the modern C++ implementation of the fast marching algorithm.

- dynamic allocations with C functions `malloc` and `free` should be replaced by C++ `new` operator with smart pointers (or `delete`): `EnhancedFloatingPoint` often has non-trivial constructor and destructor and the `malloc` and `free` functions do not call them unlike `new` and `delete`.
- C functions with variable number of arguments and specialized format specifiers (such as `scanf` and `printf("%e", ...)`) should be replaced with `std::cout`, `std::cin` calls because "%e" does not recognise `EnhancedFloatingPoint`.
- In a divide-and-conquer analysis approach, we typically instrument certain parts of the code and leave others unchanged. But, replacing `int` with `EnhancedInteger<int>` also generates many other replacements. In the case of the fast marching algorithm, the forward propagation part (Sect. 2.1) and the backward propagation for path generation (Sect. 2.2) share some common methods. However, replacing `int` with `EnhancedInteger<int>` is only required in the backward path generation. In this case we rename the original method as a template method in the private section of the class. Then, we duplicate the public method, one with `int` arguments and the other with `EnhancedInteger<int>` arguments. The bodies of the original method and its duplication just call the template private method. From the caller's perspective, the C++ "name lookup" generally generates correct calls.
- The second argument of binary operators whose first argument is of type `EnhancedInteger<int>` may be `int`, `unsigned`, `double`, `EnhancedFloatingPoint<double>`. The instrumentation needs precise overloaded operators to be called by all the constructs of the source code. In C++-03, providing an interface for this instrumentation that correctly connects the source operator with the correct overloading for all type combinations was a very complex task and ultimately produced a resulting interface that was difficult to manage. That is probably the reason why Cadna 2.1 does not support `long double`. Then, the *SFINAE* (Substitution failure is not an error) [14] feature allows the definition of such a robust interface, but this remains very technical with maintainance difficulties. Our libraries use recent C++-20 concepts to manage this `class` interactions, which makes the instrumentation more robust.

3.2 Results of the Approach based on Dynamic Analyses

To address RQ1, the objectives of the first analyses are

- ensuring that the code can be instrumented with dynamic analysis libraries (that are generally simple to use from the instrumentation point of view),
- obtaining initial quantitative accuracy properties to be refined later with more complex analyses,
- evaluating the robustness of the implementation: a small perturbation of input data should generate a small deviation in the outputs. If the implementation is not robust, we have no chance of proving formal functional properties, such as "the results depend in a limited way on the size of the grid".

Dynamic analysis with stochastic arithmetic meets these goals; that is why we use it as a first approach. To do this, we couple the instrumentation mechanism described in the previous section with stochastic analysis libraries in order to obtain accuracy and robustness results without any modification to the source code. Such analyses only require a test case and explore the impact of minor perturbations on the results after execution of the test scenario.

The Cadna[3] [7] library evaluates the accuracy of a code by propagating three executions in a single run (synchronous analysis with single control flow). This leads to maintaining three values (v_0, v_1, v_2 in

[3] See `https://www-pequan.lip6.fr/cadna`

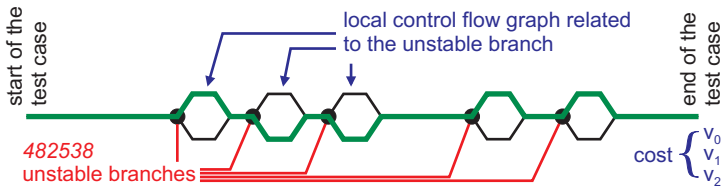


Figure 5: (Simplified) trace of the execution with Cadna (in green), where the cost of the path is evaluated by three values v_1, v_2, v_3.

Fig. 5) for each computed variable *var* instead of one. To evaluate potential impact of rounding errors, with this library each floating-point computation involving *var* is dispatched over v_0, v_1 and v_2. The ideal results are randomly rounded up or down — with a probability of 50% for up or 50% for down — instead of using the deterministic IEEE-754 rules implemented in the processor. The average of the three values provides the expectation of the computed result, while the standard deviation provides an estimate of the accumulation of round-off errors [13]. Such analysis is synchronous: the three executions are *forced* to follow the same control flow (shown in green in Fig. 5) and do not evaluate *unstable branches*, for which a possible imprecision can impact the result of a conditional test (and therefore the branch taken after it). Let us consider a comparison, say $d < d'$ between two `double` values d and d' instrumented by Cadna. It compares each of the three values v_0, v_1, v_2 obtained for the first value d with the corresponding value of the three values v'_0, v'_1, v'_2 obtained for the second value d'. Suppose that the first two comparisons return true and the last returns false. Cadna just reports an "`UNSTABLE BRANCHING`" and propagates the last execution into the then branch as well, even if it would naturally execute the else branch.

The instrumentation quickly succeeds on the target code with Cadna (thus fulfilling the first objective). The algorithm computes (in 0.556s) a path of 322 points as well as the cost of the path, stored in variable `cost`. Since `cost` is the value that the algorithm attempts to optimize, we expect it to be robust. The cost has an average of `0.392` and a relative error of 1.323×10^{-15} due to the accumulation of round-off errors. Cadna also reports the following warnings:

```
CRITICAL WARNING: the self-validation detects major problem(s). The results are NOT guaranteed.
There are 977194 numerical instabilities
1687 UNSTABLE MULTIPLICATION(S), ...
482538 UNSTABLE BRANCHING(S), 260343 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S)
```

The execution of the test case takes 0.182s and generates a shorter path of 320 points with a cost of 0.393257, that is outside the error range computed by Cadna. That confirms — as suggested by the warnings — that the Cadna results are not conclusive: unstable branches (not evaluated by Cadna) probably have a major impact on the path computation and therefore on the robustness of the algorithm.

The second analysis uses the Verrou[4] [6] tool[5]. Verrou evaluates the accuracy of a code during multiple runs by randomly rounding up or down every floating-point computation (asynchronous analysis with single control flow). Unlike Cadna, Verrou does not need additional memory: since the execution of the program perturbed by Verrou is non-deterministic, multiple runs provide multiple output values (see Fig. 6, where we show only four traces for readability). The average and the standard deviation of the output respectively provide the expected stochastic result and an error that is representative of

[4]See `https://github.com/edf-hpc/verrou`

[5]The Verificarlo [5] tool (see `https://github.com/verificarlo/verificarlo`) can be expected to produce similar results, but it was not used in this study.

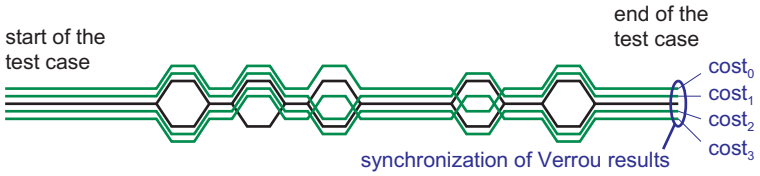


Figure 6: Four (simplified) traces (in green) from the 10 executions with Verrou. Each trace leads to computing a (possibly different) path and its cost

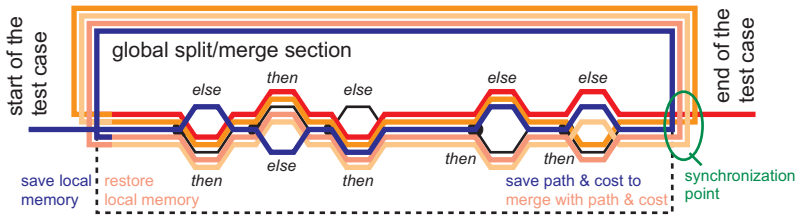


Figure 7: (Simplified) trace of the FLDLib analysis, in which the code of a SPLIT/MERGE section is executed several times for all executable control flows inside it and the results are consolidated at the end

the accumulation of round-off errors. With ten runs (performed in 16.123s), Verrou provides different lengths for the optimized path: 324, 307, 308, 307, 315, 315, 314, 317, 312, 300. The average of the ten values of `cost` is evaluated to 0.3922 and its standard deviation to 2.68×10^{-4}.

The error produced by Verrou seems to be more consistent than that of Cadna with respect to the original IEEE-754 floating-point execution. The multiple runs nevertheless do not contain the value of cost produced by the test case execution since $0.393257 \gg 0.3922 + 0.000268$. We relaunch the Verrou analysis several times and we systematically obtain an average and a standard deviation close to these values. That means that the floating-point execution takes a control path that is distinct from other control paths in terms of their impact on the `cost` value. At this point, the use of formal methods appears relevant to further investigate the relative instability of the floating-point execution.

3.3 Evaluating the Impact of Perturbations on the Control Flow and the Resulting Path

To further address RQ1 and investigate RQ2, the third analysis relies on the FLDLib[6] library [12] to provide a sound over-approximation of the accumulation of round-off errors by maintaining the ideal (in practice, a very precise machine) computation and the floating-point computation in parallel. FLDLib relies on SPLIT/MERGE sections (presented above) to analyze unstable branches by exploring each of the different control flows using abstract interpretation and by consolidating the observed results at the end. This analysis propagates affine forms for rounding errors and for the possible values on the test case. The mathematical representation of an affine form is $\alpha_0 + \sum_{i=1}^{n} \alpha_i \times \varepsilon_i$, where α_i are constant coefficients in $\mathbb{R}$ (approximated by floating-point values with a large mantissa) and ε_i are free variables in the interval $[-1.0, +1.0]$. The error symbols ε_i represent unknown values due to basic approximations of complex computations. The program variables can share some ε_i, which creates linear relationships between some of these variables. FLDLib also offers advanced features to reduce the size of the re-executed code with local synchronization annotations (see the FLDLib library documentation and [12] for more detail).

[6] See `https://github.com/fvedrine/fldlib`

```
1 double jm_res = (y - yMin) / dy;
2 unsigned int jm = (unsigned int) jm_res;
```

Figure 8: First unstable branch identified with FLDLib

In practice, we activate the FLDLib analysis after initialization of the mesh. Therefore, the grid is composed of points with floating-point coordinates considered exact, i.e. without any numerical error. With this assumption, the analysis only propagates affine forms over 4 execution paths. A simplified version of an execution trace of FLDLib is illustrated by Fig. 7. After several attempts, we have found the right settings: 319 bits for the internal mantissa of the coefficients of the affine forms and a limit of 30 shared symbols per expression. With an internal mantissa of 255 bits, the intervals representing the ideal computations are too wide at the end of the forward wave propagation (see Sect. 2.1). Therefore, the cost associated with the resulting path is too strongly over-approximated with the interval $[-\infty,+\infty]$: the algorithm performs at some moment a division by the distance between two points, and if the localization of these points is imprecise, a potential division by zero due to interval arithmetic gives this result. This first FLDLib experimentation (internal mantissa of 255 bits) is nevertheless interesting because, like with Cadna and Verrou before, it also produces a resulting path (with 278 points) that is different from the path produced by the floating-point execution of the test case (with 320 points).

FLDLib quickly identifies the location in the source code of a first unstable branching, for which it explores both branches in floating-point and ideal computation. It concerns the computation of the second instruction of Fig. 8 with the values `y=0.5`, `yMin=0`, `dy=0.005`.

In floating-point semantics, the value of `jm_res` is 100. In the ideal semantics, the value of `dy` is $5\times10^{-3}+1.0408\times10^{-18}$ since all the constants take the same floating-point value for both semantics; therefore, the analysis shows that the ideal value of `jm_res` belongs to the small interval $[100-2.082\times10^{-16},100-2.081\times10^{-16}]$. The value of `jm` is then 100 in floating-point semantics and 99 in ideal semantics, which creates an unstable branching. The analysis then separates the joined control flow of both semantics into two control flows and explores them separately. These control flows merge at the end of the source code after the computation of the path and its cost. The merge operation computes the numerical error from the substraction between the floating-point value and the ideal small interval of the cost, each value being inferred by the corresponding control flow. The result of this second FLDLib experimentation (internal mantissa of 319 bits) shows an interesting finding: the unstable branch has no impact on the cost and on the points of the path, even if the sorted priority queue (see the gray points of Fig. 2) is organized differently in the two control flows. Therefore, the resulting paths both have 320 points, like the floating-point execution and they return a relative error of 7.058×10^{-16} for the cost.

The duration of this second FLDLib analysis is 114 min after a limited exploration of only 4 execution flow paths. For one control flow path, the analysis encounters 351 296 unstable branches. Therefore, the estimated time for the complete analysis would be $2^{351\,296}\times114\,\text{min}/4$. Nevertheless, these preliminary results allow us to identify the first unstable branches and to show their absence of impact on the computed path and its cost.

As another finding, this second FLDLib analysis also provides the complete list of locations in the source code of the unstable branches encountered. This list has only 5 locations (which are executed multiple times due to loops), each with a unique calling context. The calling contexts show that the first 4 locations (one of which is the unstable branch of Fig. 8) belong to the forward wave propagation (Sect. 2.1) and that the last location belongs to the backward propagation dedicated to the path generation (Sect. 2.2).

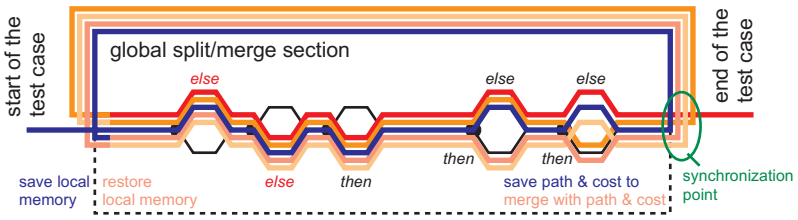


Figure 9: Trace of the FLDLib analysis forgetting unstable branches related to unsigned int conversion

To investigate the impact of other unstable branches, we modify the analysis with an analysis library that only targets the last unstable branch: the one that has a direct impact on the outcome of the path. The reason is that the first unstable branches listed potentially concern all the cells of the grid of Fig. 2 (as they are part of the forward wave propagation); therefore, they have little chance of having an impact on the final path, whereas the last unstable branch directly concerns the cells crossed by the resulting path (as it belongs to the path generation step). Concretely, the new analysis forces the ideal computation of the unstable branch created by the `unsigned int` conversion of Fig. 8 to be equal to the floating-point point computation. Indeed, we observe in Fig. 9 that for the first branches of the analysis paths, the control flow of ideal computations follows the floating-point control flow, which was not the case in Fig. 7. The duration of the new analysis increases significantly: 17 h 20 min instead of 114 min to cover four different branches[7].

The first iteration of this third FLDLib analysis follows a joined control flow for both semantics until the path and cost are computed, which was not the case in the previous analysis. The reported error for this iteration on cost is 4.71×10^{-13}, which is consistent with the Cadna results (1.323×10^{-15}) since FLDLib provides a guaranteed over-approximation while Cadna returns a stochastic estimation of the error. Then, as expected, the first unstable branch is found during the backward path generation. The second analysis iteration follows only the floating-point control flow and it gives the same result for cost as the reference execution. The third analysis iteration follows only the ideal control flow and the MERGE macro at the end of the `main` function creates an error of 4.66×10^{-13} as the maximum difference between the ideal cost and the floating-point cost. An important finding here is that this small error satisfies a sufficient stability criterion for our optimization algorithm, even if there are only two unstable branches evaluated.

The robustness of this conservative analysis, if it is confirmed on all paths despite the unstable branchings encountered by Cadna and qualified by Verrou, suggests that certain computations are redone in different parts of the algorithm, notably between the forward wave propagation (Sect. 2.1) and the backward propagation for path generation (Sect. 2.2). This would be another interesting finding for such kinds of algorithms, where some small steps are recomputed several times. Indeed, the stochastic analysis does not ensure the introduction of exactly the same perturbation if exactly the same computation is performed several times. Suppose such a redundant computation evaluates to a value *res* the first time, the stochastic analysis evaluates it to $res + \delta$ the second time, since the pertubations introduced by the analysis are not

[7]Here is an explanation for this time difference. The first analysis very early encounters an unstable branch that separates the floating-point control flow from the ideal control flow. The analysis of the floating-point control flow then propagates only constant values and the analysis of the ideal flow stops propagating affine forms related to the difference between the float and the ideal value since the floats are no longer present. Conversely, the new analysis has to propagate constants for floating-point values and affine forms for ideal values and for errors during longer execution fragments, which is costly, including the reduced product between the inferred error and the subtraction of ideal value and floating-point value.

	Cadna	Verrou	FLDLib: 4 paths over $2^{351\,296}$
instrumentation time	10 min	0 s	3 h
analysis time	0.556 s	16.123 s	17 h 20 min
cost error	1.323e-15	2.68e-4	4.71e-13
error of mean cost value wrt. reference execution	1.25e-3	1.27e-3	4.71e-13
indicative confidence in results (/10)	4	6	7
reason of error inconsistency	unstable branching not evaluated	original float execution not reached	no inconsistency for 4 paths

Figure 10: Analysis summary (the floating-point reference execution time without analysis being 0.182 s)

the same. Hence, the branch taken after the first computation may be different from the one taken after the second, whereas the deterministic IEEE-754 computation guarantees that it will be the same branch. This issue also occurs for FLDLib analysis in case of over-approximations. In this case, the evaluation result is $res + [a,b]$, but the analysis cannot guarantee that it is the same branch because the value chosen in $[a,b]$ the first time may be different from the value chosen in $[a,b]$ the second time. Since the intervals for the ideal values are very very small (319 bits of mantissa is equivalent to a precision of 4.68×10^{-97}) and since the main linear relationships are preserved between the variables, the analysis is likely to avoid certain over-approximations that would consider unreachable branches and generate false negatives.

Figure 10 shows a summary of our first experiments, which required little investment in annotations of the source code, but a lot of effort in the definition and configuration of the analyses. It gives an indicative (and subjective, based on our experience) level of confidence for the results of each tool. The instrumentation time corresponds to the time that was required for the authors to instrument the code. The analysis time shows the tool execution without the compilation steps. The cost error is the error output directly produced by the tools for the `cost` variable. It concerns the standard deviation of the cost values for the stochastic tools (Cadna and Verrou) and the conservative error for the formally guaranteed tool (FLDLib). The error of mean cost value is the difference between the mean of the cost values computed by the tools and the original floating-point evaluation of the variable `cost` computed by the code without any instrumentation. The indicative confidence in the corresponding analysis increases when both errors become closer. The reason of error inconsistency is given in the last line.

3.4 Ongoing Work on Formal Verification for Thin Numerical Scenarios

The aforementioned results are promising and make us believe that a complete formal robustness analysis for this case study is possible. But we need to cover all of the 351 296 unstable branches identified by the FLDLib analysis to know if the accuracy of the cost is rather close to 1.27×10^{-3} or 4.71×10^{-13} for this test case (cf. Fig. 10). This section presents our ongoing work in this direction.

For this purpose, we add local synchronization annotations around the detected unstable branches (see the resulting trace in Fig. 11). That means that the `unsigned int` variable receiving the conversion of the floating-point computation in Fig. 8 is conditionally defined. For instance, the evaluation of `jm_res` with the values `y=0.5`, `yMin=0`, `dy=0.005` can be seen as producing an integer defined as

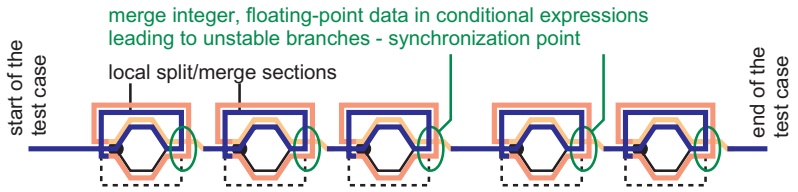


Figure 11: (Simplified) trace of the FLDLib analysis with local synchronization points

if b_0 *then* 100 *else* 99, where b_0 is a fresh and free logical variable in $\{\text{true}, \text{false}\}$. For this unstable branch, b_0 evaluates to true in the floating-point semantics and false in the ideal semantics. Further unstable branches have a more complex evaluation in ideal semantics.

For this propagation, we define a new conditional domain[8] in FLDLib that contains cascading conditional expressions or a simple integer value. This domain is implemented as an instantiation of `EnhancedInteger` template mentioned in Sect. 3.1. Therefore, it can represent domains like

$$\textit{if } b_0 \textit{ then } (\textit{if } b_1 \textit{ then } \ldots \textit{ else } \ldots) \textit{ else } ((\textit{if } b_2 \textit{ then } \ldots \textit{ else } \ldots))$$

The conditional domain also propagates to floating-point computations.

The initial code contains integer and floating-point values. Our automatic instrumentation (see Sect. 3.1) preserves floating-point constants but replaces floating-point variables with the default floating-point domain containing an affine form for the ideal computation and the accumulation of round-off errors, and an interval for the floating-point computations. Thus, each new domain potentially interacts with 3 different domains (conditional integer, conditional floating-point, affine forms) and the concepts of C++-20 are very useful for handling these interactions — adding the conditional domains to FLDLib required 12 kloc of C++ code.

A finalization of these new domains and their application for the robustness analysis of the case study is still ongoing. It will require a manual instrumentation of the code (that will probably take more than 8 hours) but can be expected to help analyze the target code.

Robustness analysis is a mandatory requirement before attempting to verify functional properties, such as the relative independence of the results with respect to the size of the grid (cf. RQ3). Checking these properties follows the same methodology as checking robustness. We proceed first with simple tests, then with formal analysis. We start by using the same test case, before attempting later a modular verification approach based on deductive methods.

4 Conclusion

Numerical analysis of software is important for critical programs, in particular related to trajectory computation used in autonomous systems. It is also a very challenging and time-consuming task. Indeed, precision and robustness of the algorithm can be impacted by many instructions, especially for programs with long execution paths and/or simulating a continuous real space by a discrete grid, for which a small perturbation of data can naturally lead to another behavior.

[8]This domain is not yet available in the public repository of FLDLib, but we plan to integrate it into the open-source repository of the tool in the near future.

This case study paper describes an industrial application of several modern numerical analysis tools to a real-life path computation algorithm with a realistic test case. We present the applied methodology and results. An important first step of the study is to ensure code instrumentability and to compare various analysis results to qualify the impact of unstable branches with stochastic methods (with tools like Cadna and Verrou). Next, we investigate unstable branches and formally ensure robustness with a formal analysis (using a tool like FLDLib). The results we obtained seem very promising: we managed to identify the unstable branches and the corresponding locations on the code that constitute important attention points for numerical analysis. Dynamic analysis tools (Cadna and Verrou) show that the relative error in the path computation is sufficiently small, and the algorithm is sufficiently robust. This conclusion should be confirmed by a formal analysis. A representative subset of unstable branches coming from different parts of the algorithm has been formally shown (with FLDLib) to ensure expected robustness properties, while the study for other branches is still in progress. So far, the analysis confirmed that the algorithm meets the user expectations in terms of accuracy and robustness.

Future Work. This case study suggests numerous future work perspectives. One perspective is to finalize the investigation of unstable branches. We plan to use the new conditional domains that were recently integrated into FLDLib and will be evaluated on this case study. Considering other realistic test cases and replaying the analyses for them is another work direction. Applying the described methodology on other industrial use cases is another perspective.

As a more ambitious long-term research objective, proving that the result does not depend on the size of the grid (RQ3) is a much more complex problem. Our plan is to apply a component-based divide-and-conquer approach on the source code. For each component, this requires formal instrumentation in order to propagate logical formulas instead of abstract domains. The starting point is the previous instrumentation of the code with its annotations for the synchronization of unstable branches. The methodology is inspired by the approach used in deductive verification, by first replacing the data structures of the code with classes representing formal properties. C++ operator overloading will propagate these properties across components using carefully designed verification unit scenarios. The engineer's objective will be to design unit scenarios (such as the postcondition/output invariant of a method/class is formally contained in the precondition/input invariant of the method/class that takes its results).

Acknowledgment. Part of this research (tooling improvement) was supported by the ANR InterFLOP project (grant ANR-20-CE46-0009). Part of this work was also supported by the ANR EMASS project (grant ANR-22-CE39-0014). We thank the designers and developers of numeric analysis tools of the InterFLOP community.

References

[1] R. Bellman & R. Kalaba (1965): *Dynamic Programming and Modern Control Theory*. Academic paperbacks, Elsevier Science.

[2] Elisabetta Carlini, Maurizio Falcone, Nicolas Forcadel & Régis Monneau (2008): *Convergence of a generalized fast marching method for a non-convex eikonal equation*. *SIAM Journal on Numerical Analysis* 46, pp. 2920–2952, doi:10.1137/06067403X.

[3] Da Chen, Jian Zhu, Xinxin Zhang, Minglei Shu & Laurent D. Cohen (2021): *Geodesic Paths for Image Segmentation With Implicit Region-Based Homogeneity Enhancement*. *IEEE Transactions on Image Processing* 30, pp. 5138–5153, doi:10.1109/TIP.2021.3078106.

[4] Michael G. Crandall, Hitoshi Ishii & Pierre-Louis Lions (1992): *user's guide to viscosity solutions of second order partial differential equations*. Available at `https://arxiv.org/abs/math/9207212`.

[5] Christophe Denis, Pablo de Oliveira Castro & Eric Petit (2016): *Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic*. In: *Symposium on Computer Arithmetic (ARITH)*, doi:10.1109/ARITH.2016.31.

[6] François Févotte & Bruno Lathuilière (2019): *Debugging and Optimization of HPC Programs with the Verrou Tool*. In Ignacio Laguna & Cindy Rubio-González, editors: *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), Denver, CO, USA, November 18, 2019*, IEEE, pp. 1–10, doi:10.1109/CORRECTNESS49594.2019.00006.

[7] Fabienne Jézéquel & Jean Marie Chesneaux (2008): *CADNA: a library for estimating round-off error propagation*. *Comput. Phys. Commun.* 178(12), pp. 933–955, doi:10.1016/J.CPC.2008.02.003.

[8] R. Kimmel & J. A. Sethian (1998): *Computing geodesic paths on manifolds.* *Proceedings of the National Academy of Sciences of the United States of America* 95(15), pp. 8431–8435, doi:10.1073/pnas.95.15.8431.

[9] Jean-Marie Mirebeau, Lionel Gayraud, Rémi Barrère, Da Chen & François Desquilbet (2023): *Massively parallel computation of globally optimal shortest paths with curvature penalization*. *Concurrency and Computation: Practice and Experience* 35(2), p. e7472, doi:10.1002/cpe.7472.

[10] J. A. Sethian (1996): *A Fast Marching Level Set Method for Monotonically Advancing Fronts*. *Proceedings of the National Academy of Sciences of the United States of America* 93(4), pp. 1591–1595, doi:10.1073/pnas.93.4.1591.

[11] J.A. Sethian (2001): *Evolution, Implementation, and Application of Level Set and Fast Marching Methods for Advancing Fronts*. *Journal of Computational Physics* 169(2), pp. 503–555, doi:10.1006/jcph.2000.6657.

[12] Franck Védrine, Maxime Jacquemin, Nikolai Kosmatov & Julien Signoles (2021): *Runtime Abstract Interpretation for Numerical Accuracy and Robustness*. In Fritz Henglein, Sharon Shoham & Yakir Vizel, editors: *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, *Lecture Notes in Computer Science* 12597, Springer, pp. 243–266, doi:10.1007/978-3-030-67067-2_12.

[13] J. Vignes (1993): *A stochastic arithmetic for reliable scientific computation*. *Mathematics and Computers in Simulation* 35(3), pp. 233–261, doi:10.1016/0378-4754(93)90003-D.

[14] Wikipedia: *Substitution failure is not an error*. Available at `https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error`.

Cross-Layer Formal Verification of Robotic Systems*

Sylvain Raïs[1,2], Julien Brunel[1], David Doose[1] and Frédéric Herbreteau[2]
[1] ONERA DTIS, Université de Toulouse, France
[2] Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, 33400, Talence, France
[1] firstname.lastname@onera.fr, [2] firstname.lastname@u-bordeaux.fr

Robotic systems are widely used to interact with humans or to perform critical tasks. As a result, it is imperative to provide guarantees about their behavior. Due to the modularity and complexity of robotic systems, their design and verification are often divided into several layers. However, some system properties can only be investigated by considering multiple layers simultaneously. We propose a cross-layer verification method to verify the expected properties of concrete robotic systems. Our method verifies one layer using abstractions of other layers. We propose two approaches: refining the models of the abstract layers and refining the property under verification. A combination of these two approaches seems to be the most promising to ensure model genericity and to avoid the state-space explosion problem.

1 Introduction

The design and development of modern robotic systems is a complex issue, as it brings together many fields of research. Moreover, these robotic systems are intended to interact with humans or to be deployed in critical sites. Therefore, it is essential to provide guarantees for the operation of these systems. Formal methods are widely used to assert the reliability of critical systems. They provide strong proof-based guarantees that the verified system behaves accordingly to the specifications. In the context of robotic systems, several modeling tools and formalisms have been developed to verify properties, either online [7] or offline [4, 6].

On the other hand, in order to improve the design of robotic systems, state-of-the-art approaches rely on multi-layer architectures as they provide powerful abstraction to develop each layer independently of the others. Such a design facilitates the development of robotic systems, improves their modularity and enables each layer to be (formally) verified separately. These advantages help to implement complex behaviors such as fault tolerance [9] and facilitate the reuse of robotic system code. Note that several multi-layer design standards exist within the robotics research community: five-layer pyramid design [3], four-layer design [13], three-layer pyramid design [10, 14], and more. Among these classical designs, the three-layer architecture shown in Figure 1 is a promising and widely used approach because it provides a modular design while minimizing the number of layers. In this architecture, the decision layer deals with the robot's decision-making and planning processes (e.g. a user interface or a "smart" program). The executive layer provides an abstract interface to the functional layer via the concept of skills [1, 13, 14, 10]. And the functional layer corresponds to low-level task processing.

*This study has received financial support from the French government in the framework of the France 2030 programme IdEx université de Bordeaux / RRI ROBSYS

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 143–150, doi:10.4204/EPTCS.411.9

© S. Raïs, J. Brunel, D. Doose & F. Herbreteau
This work is licensed under the
Creative Commons Attribution License.

Listing 1: An example of RobotLanguage design

```
skillset custom_robot {
    resource {
        motion { state { On Off } initial Off transition all }
        battery { state { Normal Critical } initial Normal transition all }
    }
    skill goto {
        input {distance: Integer}
        output position: Position
        precondition { (motion == Off) && (battery != Critical) }
        start motion -> On
        invariant {in_movement { guard motion == On }}
        interrupt {effect {motion -> Off}}
        success {arrived {effect {motion -> Off}}}
        failure {blocked {effect {motion -> Off}}}
    }
}
```

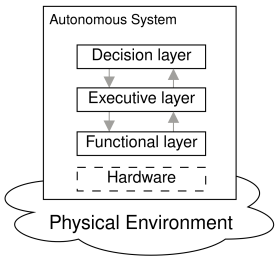


Figure 1: Three-layer architecture

In practice, it is impossible to verify the whole system at once, due to the complexity of robotic systems, or the incompatibility of certain theories that make verification undecidable. Specific formalisms and techniques have been developed for the design and the verification of each layer separately. However, the compartmentalization of the different analyses is an obstacle to complete system analysis because these formalisms cannot always be combined. In fact, some of the operating characteristics of the system must consider multiple layers in order to be studied.

The present work, which is part of a Ph.D. thesis, aims to provide an offline cross-layer verification method based on the three-layer design in Figure 1. Our method uses RobotLanguage [1] [5, 2, 1], an interesting framework for designing reliable robotic systems. RobotLanguage provides a formal language to model the executive layer, a formal offline verification of predefined properties on this model, and an automatic code generation from the model to implement this layer. Our approach is based on a RobotLanguage model of the executive layer, and extends it with abstract models of the other layers in order to verify properties of the whole system. In general, abstract models are not refined enough to verify robotic systems. We introduce two complementary approaches: one consists in refining the model, and the other consists in refining the property. We illustrate the relevance of our techniques on an example. Our method is not specific to the three-layer architecture in Figure 1, and can be used with other multi-layer designs.

2 Related Works

Several formal frameworks have been defined to support the design and the verification of robotic systems, such as RobotLanguage. PROSKILL [8] gathers the specifications of the decisional layer, the executive layer and a part of the functional layer (see Figure 1), and allows to verify temporal and timed properties both offline and online. However, PROSKILL provides a monolithic design for robotic systems and does not benefit from the advantages of a multi-layer design. Our method is based on the multi-layer design, preserving the modularity gained by this design, and thus fits well to our real robotic systems.

On the other hand, RobotLanguage comes with a tool, SkiNet [11], which provides a translation to Petri nets to perform offline formal verification of temporal properties. This tool has also been extended [12] to verify temporal properties online in order to address the state-explosion problem. However, SkiNet only verifies properties of the executive layer only, while our work aims to provide a multi-layer verification method.

[1] https://onera-robot-skills.gitlab.io/index.html

3 Cross-Layer Verification

RobotLanguage has been developed to design the executive layer of robotic systems. After a brief introduction, we describe the formalism used to model these systems. Next, we explain how to model each system layer and how to incorporate all models for formal verification. Finally, we present a method for systematically verifying multi-layer systems, illustrated with an example.

3.1 Introduction to RobotLanguage

Modern approaches to formal robotic system design are based on skills and resources[8, 5, 13, 10]. Skills are basic actions provided by the executive layer to implement complex behaviors in the decision layer. For example, Listing1 defines one skill: `goto`, which moves a robot a given distance. Resources represent physical features used by skills, such as `motion` and `battery` in Listing 1. The resource `battery` tracks levels, while `motion` monitors movement. In RobotLanguage, each group of skills and their shared resources forms a *skill set*, such as `custom_robot` in Listing 1.

The skill `goto` is an abstraction of the actual code executed at the functional layer. In RobotLanguage the system designer specifies conditions for starting a skill (`precondition`), conditions that should remain true during execution (`invariant`), and resource updates (`start`, `effect`).

RobotLanguage includes a toolset[2] that translates models into executable C++ code using the ROS2 middleware. This code creates one ROS2 node per skill set and several topics to manage communication between the executive and decision layers. In addition, the generated code verifies conditions (`precondition`, `invariant`) and applies effects (`start`, `effect`) specified in RobotLanguage. The programmer is responsible for implementing the functional layer in specific hook functions, whose prototypes are generated from the RobotLanguage design.

3.2 Modeling Formalism

In this paper, a model consists of a finite set $M = \{S_1, \ldots, S_k\}$ of finite labeled transition systems. Each transition system $S_i = (Q_i, q_i^0, \Sigma_i, T_i)$ consists of a finite set of states Q_i, a distinguished initial state q_i^0, a finite alphabet of events Σ_i and a transition relation $T_i \subseteq Q_i \times \Sigma_i \times Q_i$ where edges are labeled by events from Σ_i. Note that the transition systems may have common events on which they synchronize. Let $\Sigma = \bigcup_{i\in[1;k]} \Sigma_i$. A global state of M is a tuple $(q_1, \ldots, q_k)$ of states, one for each transition system in M. The initial global state is $(q_1^0, \ldots, q_k^0)$. There exists a global transition $(q_1, \ldots, q_k) \xrightarrow{a} (q_1', \ldots, q_k)$ with $a \in \Sigma$ if for each S_i such that $a \in \Sigma_i$, there exists a transition $(q_i, a, q_i') \in T_i$, and $q_i' = q_i$ for every S_i such that $a \notin \Sigma_i$. A global run is a sequence of global transitions starting from the initial global state.

As an example, consider the model consisting of two transition systems: S in Figure 2 and F in Figure 3a. These two transition systems synchronize on their common labels. Thus, any run in this model consists of asynchronous solid and zigzag transitions from S, or dotted and dashed transitions that synchronize S and F.

3.3 Executive Layer Modeling

First, we explain how to model the executive layer by describing the execution of a skill through the transition system in Figure 2. During its execution, the skill transitions through several states, depending on internal actions (plain transitions), or on interactions with the decision layer (zigzag transitions) or

[2] `https://onera-robot-skills.gitlab.io/`

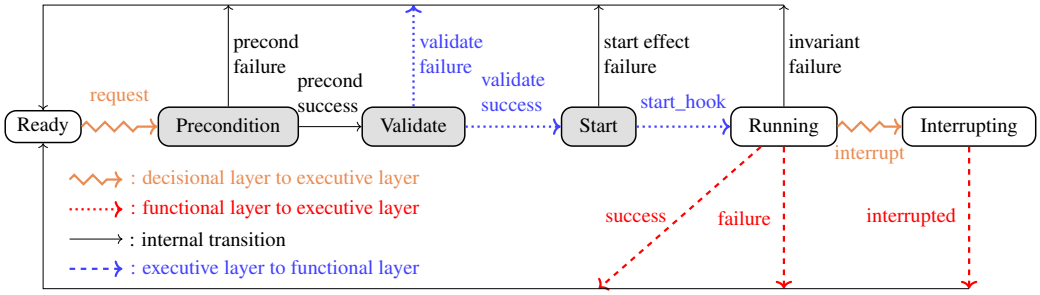


Figure 2: Control flow graph of a skill

with the functional layer (dashed/dotted transitions). The execution begins in the state `Ready`, and spans into three phases. First, on reception of `request` from the decision layer, the state of the system is checked at the executive layer (`precond`) and the functional layer (`validate`). If these conditions are satisfied, `start_hook` triggers the execution of the functional layer, switching the state to `Running`. Finally, the execution can terminate in a success or a failure, triggered by the functional layer, or it can be interrupted by the decision layer. In each case, the functional layer notifies the executive layer by calling `success`, `failure`, or `interrupted`. The execution can also stop if an invariant is violated. These invariants are monitored by the code that is automatically generated from the RobotLanguage design. We refer the reader to [1] for more details on the semantics of invariants in RobotLanguage which is beyond the scope of this paper.

For a given skill set like in Listing 1, a model using instances of the transition system in Figure 2 for each skill can formally verify some properties at the executive layer, such as "skill `goto` can be executed." However, this model is not refined enough to verify more specific properties, such as "skill `goto` cannot be executed infinitely often", which is expected to hold, since our RobotLanguage design in Listing 1 lacks a skill to recharge the battery.

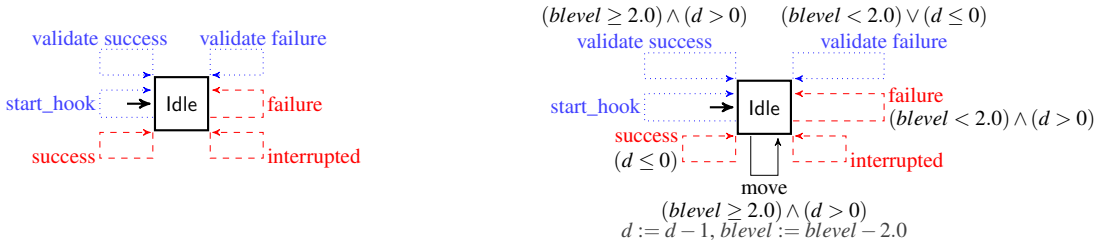
3.4 Multi-Layer Modeling

We aim to extend the model in Figure 2 (called S in the sequel) with models of the functional and decision layers. For now on, we will concentrate on the functional layer, as the approach that we present hereafter straightforwardly applies to the decision layer.

From Section 3.2, it comes that a model of the functional layer should conform to a *synchronization interface* that will enable communication between the model of the functional layer, and the model of the executive layer, through event synchronizations. More specifically, a model of the functional layer should synchronize on events validate success, validate failure, start hook, success, failure and interrupted with the transition in Figure 2.

The transition system F in Figure 3a shows a very abstract model of a functional layer that conforms to this synchronization interface. Note that F allows any sequence of the above mentioned events since its transitions can be crossed unconditionally. Hence, any sequence of events that is possible in S is also possible in the model $\{S,F\}$ where S and F synchronize on common labels. F can be seen as the generic most abstract functional layer model.

Figure 3b shows another model of the skill `goto` at the functional layer. Observe that this model also conforms to the synchronization interface. It further has an internal action `move` that does not synchronize with S: it is asynchronous. This model is described as a control graph with two variables: d which is the distance to travel, and *blevel* which is the battery level (both variables have finite domains). Note



(a) Default functional layer model

(b) Concrete functional layer model for skill `goto`

Figure 3: Transition systems modeling the skill `goto` at the functional layer

that the variables used at the functional layer partly model the robot's state, while the RobotLanguage resources used in the executive layer (Listing 1) are abstract knowledge of the robot's state, updated by monitoring the robot. The model for updating the `battery` resource according to the actual value of *blevel* is not shown for the sake of simplicity. Following our settings described in Section 3.2, we consider the transition systems F' that defines the semantics of control graph in Figure 3b. Its states are pairs $(d, blevel)$ of values of the two variables, and transitions $(d, blevel) \xrightarrow{a} (d', blevel')$ take into account the guards and updates on the variables. Now, observe that due to variables d and *blevel*, the model F' restricts the sequences of events that can occur in a run. For instance, validate success is not possible if the battery level is less than 2.0. F' can thus be seen as a refinement of F. As a result, some runs that exist in S do not exist any more in the model $\{S, F'\}$ that synchronizes S and F'.

Similarly, we can model the decision layer, with a synchronization interface that is defined by the events request and interrupt. We thus obtain a multi-layer model, that consists in transition systems for each skill (as in Figure 2) at the executive layer, for each skill at the functional layer (as in Figure 3a or 3b), as well as transition systems for each resource (as defined in Listing 1) and a transition system modeling the decision layer.

3.5 A Method for Cross-layer Verification

We aim at verifying that "skill `goto` cannot be executed infinitely often" taking into account a model of the functional layer. This property can be expressed in LTL as:

$$\mathsf{F}\,\mathsf{G}\ \texttt{not Running} \tag{1}$$

This formula specifies that after some finite amount of time, the robot will never be running. It does not hold when the functional layer is modeled as in Figure 3a. We present two approaches for verifying such specifications requiring a multi-layer model.

A first approach consists in considering the refined model of the functional layer in Figure 3b, where d represents the distance to travel, and *blevel* tracks the battery level. In Figure 3b, the black loop moves the robot one meter ahead, consuming two battery units at the same time. At some point, either the distance d reaches 0 which leads to a `success`, or the battery level gets below 2.0 which leads to a `failure`. Observe that the battery level *blevel* is set at the initialization of the model. Hence, the battery level eventually becomes insufficient to execute skill `goto`: it only allows "`validate failure`" and "`failure`" transitions. As a result, property (1), that is "skill `goto` cannot execute infinitely often", holds on the refined model.

A second approach consists in refining the specification. In this approach, we aim at verifying our property: "skill `goto` cannot execute infinitely often" on the model including the abstract representation

of the functional layer from Figure 3a, but *with some extra assumptions*. Coming back to our example, since our RobotLanguage design in Listing 1 does not include a skill to recharge the battery, we can expect the resource `battery` to be in state `Critical` after some finite amount of time. Hence, we can verify that "skill `goto` cannot execute infinitely often" *under the assumption* "eventually the `battery` is forever in state `Critical`". This approach consists in refining the LTL formula in (1) to verify the property only on runs which satisfy this assumption. This is formalized in (2), where `Critical` corresponds to the state of the resource `battery` in Listing 1. This formula ensures that if the `battery` eventually stays in state `Critical` forever, then, the skill `goto` is not executed infinitely often.

$$\mathsf{F}\,\mathsf{G}\,\texttt{Critical} \implies \mathsf{F}\,\mathsf{G}\,\text{not}\,\texttt{Running} \tag{2}$$

Observe that due to the precondition in Listing 1 the transition labeled "`precond success`" in Figure 2 can only be taken a finite number of times on any run such that the `battery` eventually stays in state `Critical` forever. As a result the property in (2) holds on the abstract model of the system with the functional layer modeled by the transition system in Figure 3a. Observe that this model does not need any extra variable and is thus much smaller than the model obtained with the first approach.

To validate our approaches, we have translated the transition systems and specifications corresponding to the two approaches, as formulas for the *Tatam* model-checker[3]. The RobotLanguage design in Listing 1 as well as the Tatam models underlying the two approaches above are available on a public repository[4]. As expected, we have first observed that the property "skill `goto` cannot execute infinitely often" does not hold on the abstract model of the functional layer in Figure 3a as the discharge of the battery is not taken into account. On the other hand, the two approaches above allow to prove that the property holds, either by providing a refined model of the functional layer, or by refining the specification.

We see these two approaches as complementary tools for cross-layer verification of robotic systems. Refining the property keeps the model small and simple. It also yields a simpler counter-example when a property is not satisfied. However, some properties require a more precise knowledge of the state of the robot. Then, the first approach should be used to refine (parts of) the model with as few details as possible in order to be able to verify the property under consideration.

4 Conclusion

This paper presents a method for cross-layer verification of robotic systems. Our approach consists in verifying one layer using abstractions of the others. We have proposed two approaches to prove a property. One consists in refining the models of the abstract layers, the other consists in refining the property. In practice, the combination of the two approaches seems to be the most promising since it allows to consider as few implementation details as possible in the model, while mitigating the state-space explosion problem.

As future work, we plan to implement our approach in a tool to formally verify RobotLanguage designs using a precise model of the executive layer and abstract models of the decision and functional layers. To obtain a full guarantee approach, we plan to extend our technique to prove that these abstract models correspond to the implementation of the corresponding layers.

[3]Tatam git repository: `https://github.com/DavidD12/tatam`

[4]`https://gitlab.com/sylvain.rais24/fmas_2024_s_rais_models`

References

[1] Alexandre Albore, David Doose, Christophe Grand, Jérémie Guiochet, Charles Lesire & Augustin Manecy (2023): *Skill-based design of dependable robotic architectures* 160, p. 104318. doi:10.1016/J.ROBOT.2022.104318.

[2] Alexandre Albore, David Doose, Christophe Grand, Charles Lesire & Augustin Manecy (2021): *Skill-Based Architecture Development for Online Mission Reconfiguration and Failure Management*, pp. 47–54. doi:10.1109/ROSE52553.2021.00015.

[3] V. Alcácer & V. Cruz-Machado (2019): *Scanning the Industry 4.0: A Literature Review on Technologies for Manufacturing Systems*. *Engineering Science and Technology, an International Journal* 22(3), pp. 899–919, doi:10.1016/j.jestch.2019.01.006. Available at `https://www.sciencedirect.com/science/article/pii/S2215098618317750`.

[4] Renato Carvalho, Alcino Cunha, Nuno Macedo & André Santos (2020): *Verification of system-wide safety properties of ROS applications*. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*, IEEE, pp. 7249–7254, doi:10.1109/IROS45743.2020.9341085.

[5] Christophe Grand Charles Lesire, David Doose (2020): *Formalization of Robot Skills with Descriptive and Operational Models*, pp. 7227–7232. doi:10.1109/IROS45743.2020.9340698.

[6] Lukas Johannes Dust, Rong Gu, Cristina Seceleanu, Mikael Ekström & Saad Mubeen (2023): *Pattern-Based Verification of ROS 2 Nodes Using UPPAAL*. In Alessandro Cimatti & Laura Titolo, editors: *Formal Methods for Industrial Critical Systems - 28th International Conference, FMICS 2023, Antwerp, Belgium, September 20-22, 2023, Proceedings*, 14290, Springer, pp. 57–75, doi:10.1007/978-3-031-43681-9_4.

[7] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon M. Moore, Qingzhou Luo, Aravind Sundaresan & Grigore Rosu (2014): *ROSRV: Runtime Verification for Robots*. In Borzoo Bonakdarpour & Scott A. Smolka, editors: *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, 8734, Springer, pp. 247–254, doi:10.1007/978-3-319-11164-3_20.

[8] Félix Ingrand (2024): *PROSKILL: A formal skill language for acting in robotics*. *CoRR* abs/2403.07770, doi:10.48550/ARXIV.2403.07770.

[9] André Leite, Andry Maykol Pinto & Aníbal Matos (2018): *A Safety Monitoring Model for a Faulty Mobile Robot*. *Robotics* 7(3), p. 32, doi:10.3390/ROBOTICS7030032.

[10] Mikkel Rath Pedersen, Lazaros Nalpantidis, Rasmus Skovgaard Andersen, Casper Schou, Simon Bøgh, Volker Krüger & Ole Madsen (2016): *Robot skills for manufacturing: From concept to industrial deployment*. *Robotics and Computer-Integrated Manufacturing* 37, pp. 282–291, doi:10.1016/j.rcim.2015.04.002.

[11] Baptiste Pelletier, Charles Lesire, David Doose, Karen Godary-Dejean & Charles Dramé-Maigné (2022): *SkiNet, A Petri Net Generation Tool for the Verification of Skillset-based Autonomous Systems*. In Matt Luckcuck & Marie Farrell, editors: *Proceedings Fourth International Workshop on Formal Methods for Autonomous Systems (FMAS) and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE), FMAS/ASYDE@SEFM 2022, and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE)Berlin, Germany, 26th and 27th of September 2022*, *EPTCS* 371, pp. 120–138, doi:10.4204/EPTCS.371.9.

[12] Baptiste Pelletier, Charles Lesire, Christophe Grand, David Doose & Mathieu Rognant (2023): *Predictive Runtime Verification of Skill-based Robotic Systems using Petri Nets*. In: *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*, IEEE, pp. 10580–10586, doi:10.1109/ICRA48891.2023.10160434.

[13] Francesco Rovida, Matthew Crosby, Dirk Holz, Athanasios Polydoros, Bjarne Großmann, Ronald Petrick & Volker Krueger (2017): *SkiROS—A skill-based robot control platform on top of ROS*, pp. 121–160. doi:10.1007/978-3-319-54927-9_4.

[14] Casper Schou, Rasmus Skovgaard Andersen, Dimitrios Chrysostomou, Simon Bøgh & Ole Madsen (2018): *Skill-based instruction of collaborative robots in industrial settings*. Robotics and Computer-Integrated Manufacturing 53, pp. 72–80, doi:10.1016/j.rcim.2018.03.008. Available at https://www.sciencedirect.com/science/article/pii/S0736584516301910.

Using Formal Models, Safety Shields and Certified Control to Validate AI-Based Train Systems *

Jan Gruteser Jan Roßbach Fabian Vu Michael Leuschel

Heinrich Heine University Düsseldorf
Faculty of Mathematics and Natural Sciences
Department of Computer Science
Düsseldorf, Germany

{jan.gruteser,jan.rossbach,fabian.vu,leuschel}@hhu.de

The certification of autonomous systems is an important concern in science and industry. The KI-LOK project explores new methods for certifying and safely integrating AI components into autonomous trains. We pursued a two-layered approach: (1) ensuring the safety of the steering system by formal analysis using the B method, and (2) improving the reliability of the perception system with a runtime certificate checker. This work links both strategies within a demonstrator that runs simulations on the formal model, controlled by the real AI output and the real certificate checker. The demonstrator is integrated into the validation tool ProB. This enables runtime monitoring, runtime verification, and statistical validation of formal safety properties using a formal B model. Consequently, one can detect and analyse potential vulnerabilities and weaknesses of the AI and the certificate checker. We apply these techniques to a signal detection case study and present our findings.

1 Introduction and Motivation

Artificial intelligence (AI) is increasingly used in safety-critical applications such as autonomous driving [33] and autonomous flying [26, 20]. While AI can be effective for many challenging tasks, it also introduces new risks and concerns. This leads to new challenges regarding certification and ensuring the safety of AI components (see, e.g., Peleska et al. [28] in the context of autonomous railway systems).

This work deals with systems that employ an AI perception system, such as image recognition. Those systems include autonomous vehicles and autonomous railway systems. In earlier work [14], we formally verified a steering system, assuming the perception system works perfectly. However, as the perception system is imperfect, we also created simulations with (hand-coded) probabilities for all kinds of erroneous detections. We then applied Monte Carlo simulation to estimate the likelihood of safety-critical errors. As a concrete case study, we applied those techniques to an AI-based railway system [14] using ProB [23] and SimB [35]. In this paper, we move towards using the *real* AI within these simulation and validation runs, rather than using estimated error rates. As outlined by Myllyaho et al. [25], fully virtual simulation enables validation of the system in dangerous situations without real danger. Although the validity of the simulator is difficult to verify, simulation still helps as a validation method to initially assess the quality of the system under evaluation.

The AI perception system itself is based on the widely-used YOLO [29] architecture, making the system difficult to verify with formal methods alone. To address this, we implemented a runtime certificate checker using classical computer vision algorithms to verify the output of the AI [31]. This checker can be certified using classical techniques (e.g., [4]).

*This research is part of the KI-LOK project funded by the "Bundesministerium für Wirtschaft und Energie"; grant # 19/21007E. The work of Fabian Vu is part of the IVOIRE project funded by "Deutsche Forschungsgemeinschaft" (DFG) and the Austrian Science Fund (FWF) grant # I 4744-N.

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 151–159, doi:10.4204/EPTCS.411.10

© J. Gruteser, J. Roßbach, F. Vu & M. Leuschel
This work is licensed under the
Creative Commons Attribution License.

This work presents a real-time demonstrator linking the formal model, the AI, and certified control, extending an approach used to validate reinforcement learning agents [34]. AI now controls the simulation directly by executing events for the perception system in the formal model. With our approach, we capture real AI behaviour and can use two runtime monitoring techniques: 1) the formal B model acts as a safety shield for the steering system (e.g., to detect false negatives), 2) and the certified control monitors the perception system (detecting false positives). We demonstrate this methodology on a railway case study [14], and discuss our findings and the challenges of the approach.

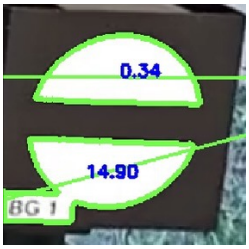
B Method, ProB, and SimB. The B method [1] is a state-based formal method for specifying and verifying software systems. The B method is based on first-order logic and set theory and has been used industrially for over 25 years [3] to generate software that is "correct by construction" [7, 10], and for system-level safety modelling. For the latter, the B method has been used for many railway applications, such as ETCS Hybrid Level 3 [16, 24] and CBTC systems [5, 6]. In this article, we use the B method to model autonomous train control in a shunting yard, based on the model from [14].

ProB [23] is an animator, constraint solver, and model checker for formal models. It supports various formal languages including the formal B method. SimB [35] is a simulator built on ProB's animator, supporting real-time simulation and Monte Carlo simulation. SimB can be linked with external software components [34]. We use ProB and SimB in this article to run the steering system and the safety shield, and also for validating the entire system.

Certified Control. Certified control [19] is a runtime monitoring approach to ensure the safety of the perception system in autonomous vehicles. Unlike conventional monitoring methods, certified control does not rely on independent perceptions. Instead, a controller provides a *certificate* containing all essential information to prove formal properties. This certificate may be generated by a sophisticated AI algorithm, which does not need to be formally verified. Using this certificate, the runtime monitor verifies if the specified criteria hold for the provided data. This monitor can, in contrast to the AI system, be comparatively small and deterministic. The architecture establishes a trusted foundation that can potentially be subjected to a rigorous formal verification process.

In previous tests, this technique almost eliminated all false positive detections, in exchange for rejecting some true positives [31]. The bounding box of the sign detected by the YOLO model is cropped from the image and passed to the runtime monitor, which uses various computer vision techniques, e.g. contour detection, to validate the sign for expected features. If the desired features are not recognised, the detection is rejected. Figure 1 shows a successfully validated Sh1, and a Sh0 sign that would be rejected by the checker.

(a) Validated Sh1 Sign



(b) Falsely rejected Sh0 Sign

Figure 1: Runtime Monitor Example

2 Linking Formal Model, AI, and Certified Control

This section describes how we link together the formal model, the AI, and the certificate checker. In the formal model, we formally specify and verify the steering system which includes safety shields to prevent unsafe operations based on the AI's perception and the known environment. In previous

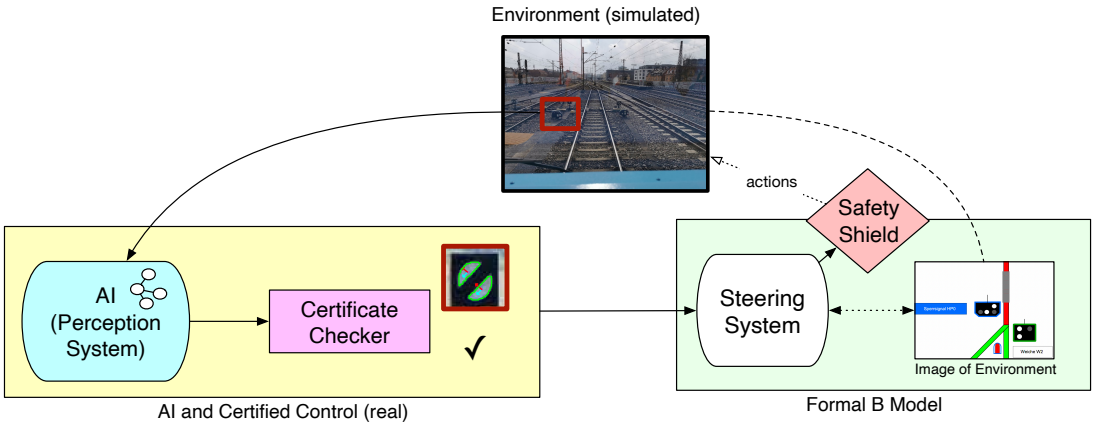


Figure 2: Simulation of the Formal B Model with AI-based Perception System and Certificate Checker in an Environment

work [14] on validating an AI-based train control system, we encoded probabilities for false positive and false negative detections of the AI by hand. Now, we use *real* AI components and real certificate checkers for simulation. Figure 2 gives an overview of how we link the formal model, the AI, and certified control inside a real-time demonstrator with runtime monitoring/verification.

In our case study, the AI-based perception system processes the environment in form of images at runtime. Ideally, various techniques should be employed to ensure that the AI is trained correctly and performs well (see [30] and references therein). Additionally, we use certified control to monitor the perception system, and detect false positives (i.e., the monitor will reject detections which it cannot confirm). The output of the perception system and the certificate checker are then synchronised with the formal model. The simulated environment, including the image provided to the AI, must correspond to the formal model's current state. This is a significant challenge as discussed in Section 3.3. The formal model contains events for both the steering system and the perception system. The model should be safe under the assumption that the perception system works perfectly. Furthermore, the formal model can be used as a basis for a safety shield which enforces safe actions on the steering system. The shield can disallow unsafe actions, like driving through a detected stop signal. The safety shield can also detect false negatives. For example, when the AI detects no signals but the formal model "knows" that a signal must be visible at the current location, it can enforce a safe fallback action (like stopping).

3 Case Study: AI-based Signal Detection

We apply the presented technique to a case study provided by our project partners (see [14]). For this case study, we developed a formal B model [14], consisting of an environment, the steering system, and the perception system. The environment includes obstacles, points (aka switches) and signal states, field elements and movements of the steered train. The formal model abstracts away the AI-based perception system by events that represent possible outcomes of the object detection, including correct, false positive and false negative detections.

The objective, or the "mission order", is to drive autonomously from the starting position through a small shunting yard to the destination without dangerous situations or at least as safely as human drivers (cf. [14, **PROB1-2**]). The focus of this work is the detection of signal aspects during the shunting movement.

3.1 Implementation

We collected images from multiple videos of the case study track containing the various signal aspects, e.g., stop signals, permission signals, and no signals along the route to capture an interactively changing environment. Based on the position of the train and the state controlled by the formal model, an image with the appropriate signalling aspect is randomly selected from the corresponding collection. For our experiments, we assume that a signal becomes visible as soon as the train is closer than 10 distance units (freely selectable). The procedure for a simulation step is as follows:

1. Pick an image randomly depending on the current train location and signal states in the B model
2. Pass the image to our fine-tuned YOLOv8 model (cf. [31]); based on the result: if no signal has been detected: ignore and do not execute any operation in the B model; if a signal has been detected:
   - Correct signal (corresponding B event is enabled): execute `VIS_DetectCorrectSignal`
   - Wrong stop signal: execute `VIS_DetectWrongStopSignal`
   - Wrong permission signal: execute `VIS_DetectWrongPermissionSignal`
3. Execute event for environment change, e.g., switch signals or activate derailer, with a probability of 25% or move train forwards and update controller with a probability of 75% (environment changes should occur less frequently than train movements)

The simulation runs in a loop until reaching the ending condition which is later explained in our experiments.

The controller of the steering system is updated after each detection to recompute the maximum allowed movement distance. Since the simple object detection AI does not provide positioning information, we place all detections in the formal model at a fixed distance in front of the train. In the second step, we (optionally) apply the certificate checker which monitors the output of the AI by accepting or rejecting its detections.

For the AI to run the simulation, we use SIMB's interface for external simulation [34].

3.2 Experiments and First Results

For initial experiments, we encoded a safety shield in the B model that only allows for signal detections at known positions of signals. If no signal is detected at an expected position, the B operations for train movements are disabled so that the train falls back to safe mode and stops in front of the signal. This assumes that we have a map of the shunting yard and know at which locations the signals are located. We then analyse the behaviour using SIMB's real-time simulation which is now controlled by the AI and the certificate checker. For better understanding, we use the domain-specific VisB visualisation [36] from previous work [14]. Both tools are part of PROB2-UI [2]; an illustration is shown in Figure 3.

With SIMB, we also run Monte Carlo simulation with 500 runs for all combinations with/without safety shield, and with/without certified control. We also investigated the effect of not applying the certificate checker to stop signals, so that these cannot be falsely rejected and the train always enters a safe state (stop). The termination condition has been defined so that a simulation stops when a safety-critical situation occurs or when the train can no longer proceed, either due to its arrival at a stop signal or reaching the end. For each execution run, we estimate the maximum (safe) *distance* travelled to validate that the train does move forward (not driving at all would be 100 % safe, but not useful). Furthermore,

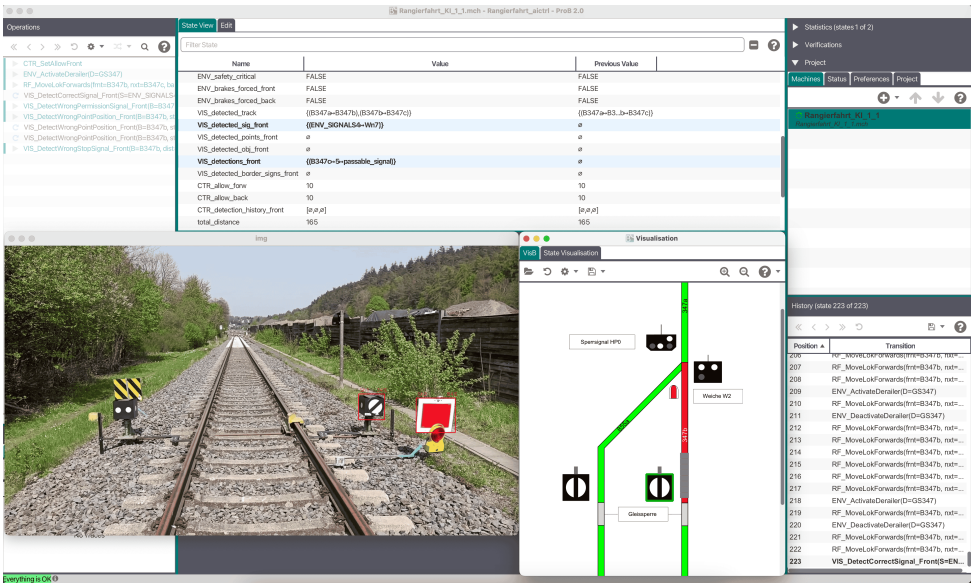


Figure 3: Real-time Simulation in ProB2-UI together with Domain-Specific Visualisation and corresponding image; Signal is detected correctly.

we estimate the likelihood of a safety-critical situation where an accident might occur, similar to [14], using the safety properties **SAF1-5** [1]. The results are shown in Table 1.

Table 1: Results after 500 Simulations (NoStop: certificate checker is not applied to stop signals, False/Correct Det.: total number of activated operations for false/correct detections in all simulations)

Controller	No safety shield			Safety Shield: Known signal positions		
Cert. Control	No	NoStop	Yes	No	NoStop	Yes
Distance	6.4 (0.0 %)	6.1 (0.0 %)	221.2 (63.2 %)	247.8 (82.4 %)	247.0 (80.4 %)	216.8 (63.0 %)
Safe	100 %	100 %	63.2 %	82.8 %	80.4 %	63.0 %
False Det.	500	500	0	20611	20993	0
Correct Det.	-	-	2335	9155	8929	2094

With certified control, it can be obtained that *all* false detections are correctly rejected (in our simple environment). Unfortunately, we also observe a significant decrease in correct detections with certified control due to false rejections (as shown in the last row of Table 1). When using the safety shield, there are still many false detections, as all detections are forwarded to the controller but then ignored if they were not expected. Without safety shield and without certified control it can be observed that the train always stops early before the first wrong detected stop signal and therefore does not make progress (the train does not even arrive at a position where it could detect a signal correctly). This is because our detection model produces quite a lot of false positive detections of stop signals. As expected, these *false positive* detections can be effectively avoided by both the safety shield and the certificate checker. This is reflected in significantly improved values for the distance travelled in the first row of Table 1.

[1] Note that compared to the results in [14], we simulate *real* AI behaviour at runtime, instead of encoding fixed probabilities for how the AI, i.e., the perception system could behave.

Surprisingly, the results of the combination of the safety shield and certified control are worse than those without certified control. With ProB, we identify the cause by inspecting the execution runs simulated in SimB. We found that *falsely rejected* detections by the certificate checker can still lead to safety-critical situations when a signal has been correctly detected as a permission signal but then falls back to stop. In this case, the safety shield no longer applies and a false negative detection can still cause the train to overrun the signal. Moreover, we identified particular scenarios, in which the checker encounters difficulties in certifying a correct stop signal detection, resulting in unexpectedly many false rejections (e.g. Fig. 1b).

In general, it can be observed that the likelihood of safety-critical situations is still high. This might be the case because of two reasons: (1) our custom AI model does not produce perfect results, and (2) there are still false negative detections. Our experiments revealed that the safety shield of known signal positions is still not enough to tackle these.

3.3 Challenges

The main benefit of our methodology is that we can link the execution of a real AI model to a formal model and check its behaviour using formal properties and statistical validation techniques. However, there are still many challenges and limitations.

A major issue is to match the real environment provided to the AI with the environment's state in the formal model. This requires an interactive simulation environment with control of the environment (signals, points, etc.) and control of the train, to ensure that the actuators controlled by the formal model are taken into account by the simulated environment. We have conducted some successful experiments with a commercial train simulator, but since it is not designed for discrete states, as required by the formal model, the handling is complicated (apart from the fact that a new instance of the simulator would have to be set up for each simulation run). Simply using a video as input is not sufficient either, as the environment remains static and we have no control over the movement of the train. In our current experiments, we avoid this problem by sampling images from videos fitting to the current state. Although this is sufficient to demonstrate the concept, we have not yet simulated real train rides. This requires a simulation environment for configurable scenarios (with ProB we are already able to load and visualise flexible scenarios, e.g., via a standardised data exchange format such as railML [15]). While there are already established tools in the automotive sector, e.g., CARLA [8], such tools are still rare in the railway sector, but are under development [9, 13, 37].

4 Related Work

There are many approaches to verifying neural networks [21, 32, 18]. In practical application, however, it is challenging for these techniques to scale to large neural networks. Another technique is robustness checks [11, 12] which also work on neural networks. Robustness checks aim to ensure the safety of the AI directly, while this work employs safety boxes around the AI. For instance, the perception system could be unsafe, but is monitored by a certificate checker. Similarly, the steering system could make unsafe decisions based on the perception, but is monitored by a safety shield encoded in the formal model.

Another work presented by Pasareanu et al. [27] abstracts away perception components, and replaces them with a probabilistic component that estimates their behaviour. In particular, the probabilities are derived from confusion matrices computed for the underlying neural network. Finally, the verification

is done by probabilistic model checkers such as PRISM [22] and STORM [17]. To improve safety, Pasareanu et al. [27] employ run-time guards which are used as runtime monitors.

Instead of estimating the probabilities for real AI behaviour, this work simulates real AI behaviour at runtime. This means that the perception system operates at runtime with real images and provides the detection to the validation tools ProB and SimB. Alternatively, we could have extracted confusion matrices and encoded them as probabilities into the simulation.

5 Conclusion and Outlook

This work successfully links a formal model, AI, and certified control to a real-time demonstrator. We demonstrated the technique in an AI-based train system. The methodology consists of the following steps: (1) formally specify and verify the steering system using formal methods, (2) encode safety shields in the formal model to prevent unsafe operations, (3) use *real* AI for simulation, (4) add a runtime certificate checker of the AI outputs to reduce false positive detections, and (5) link all components to simulate the formal model with the AI and the certificate checker's output. Using the tools ProB and SimB, we then evaluate the performance of the AI with (and without) certified control and safety shield.

With SimB, we can make statistical statements about the formal properties of the whole system. We analysed the likelihood of unsafe situations and identified weaknesses in our AI and certificate checker. With our approach, we can identify issues early during development. The results are then used to improve the AI or the certificate checker, followed by further validation.

In our case study, we identified false negatives of certified control as a cause of unsafe situations. In future, we thus need to improve our safety shield, to better protect against such false negative detections and reduce the error rates to levels required for certification. Other future improvements are to link our tool with a realistic simulation environment, e.g., in the form of co-simulation.

Acknowledgements. We thank Hitachi for providing the case study and parts of the video material.

References

[1] Jean-Raymond Abrial & A. Hoare (2005): *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, doi:10.1017/CBO9780511624162.

[2] Jens Bendisposto, David Geleßus, Yumiko Jansing, Michael Leuschel, Antonia Pütz, Fabian Vu & Michelle Werth (2021): *ProB2-UI: A Java-Based User Interface for ProB*. In: *Proceedings FMICS*, *LNCS* 12863, Springer, pp. 193–201, doi:10.1007/978-3-030-85248-1_12.

[3] Michael J. Butler, Philipp Körner, Sebastian Krings, Thierry Lecomte, Michael Leuschel, Luis-Fernando Mejia & Laurent Voisin (2020): *The First Twenty-Five Years of Industrial Use of the B-Method*. In: *Proceedings FMICS*, LNCS 12327, pp. 189–209, doi:10.1007/978-3-030-58298-2_8.

[4] CENELEC (2011): *Railway Applications – Communication, signalling and processing systems – Software for railway control and protection systems*. Technical Report EN50128, European Standard.

[5] Mathieu Comptier, David Déharbe, Julien Molinero Perez, Louis Mussat, Pierre Thibaut & Denis Sabatier (2017): *Safety Analysis of a CBTC System: A Rigorous Approach with Event-B*. In: *Proceedings RSSRail*, pp. 148–159, doi:10.1007/978-3-319-68499-4_10.

[6] Mathieu Comptier, Michael Leuschel, Luis-Fernando Mejia, Julien Molinero Perez & Mareike Mutz (2019): *Property-Based Modelling and Validation of a CBTC Zone Controller in Event-B*. In: *Proceedings RSSRail*, pp. 202–212, doi:10.1007/978-3-030-18744-6_13.

[7] Daniel Dollé, Didier Essamé & Jérôme Falampin (2003): *B dans le transport ferroviaire. L'expérience de Siemens Transportation Systems.* Technique et Science Informatiques 22(1), pp. 11–32, doi:10.3166/tsi.22.11-32.

[8] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López & Vladlen Koltun (2017): *CARLA: An Open Urban Driving Simulator*. CoRR abs/1711.03938, doi:10.48550/arXiv.1711.03938. arXiv:1711.03938.

[9] Gianluca D'Amico, Mauro Marinoni, Federico Nesti, Giulio Rossolini, Giorgio Buttazzo, Salvatore Sabina & Gianluigi Lauro (2023): *TrainSim: A Railway Simulation Framework for LiDAR and Camera Dataset Generation*. IEEE Transactions on Intelligent Transportation Systems 24(12), pp. 15006–15017, doi:10.1109/TITS.2023.3297728.

[10] Didier Essamé & Daniel Dollé (2007): *B in Large Scale Projects: The Canarsie Line CBTC Experience*. In: Proceedings B (B2007), LNCS 4355, Springer, Besancon, France, pp. 252–254, doi:10.1007/11955757_21.

[11] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri & Martin Vechev (2018): *Ai2: Safety and robustness certification of neural networks with abstract interpretation*. In: 2018 IEEE symposium on security and privacy (SP), IEEE, pp. 3–18, doi:10.1109/SP.2018.00058.

[12] Divya Gopinath, Guy Katz, Corina S. Păsăreanu & Clark Barrett (2018): *DeepSafe: A Data-Driven Approach for Assessing Robustness of Neural Networks*. In: Proceedings ATVA, LNCS 11138, Springer, pp. 3–19, doi:10.1007/978-3-030-01090-4_1.

[13] Jürgen Grossmann, Nicolas Grube, Sami Kharma, Dorian Knoblauch, Roman Krajewski, Mariia Kucheiko & Hans-Werner Wiesbrock (2023): *Test and Training Data Generation for Object Recognition in the Railway Domain*. In: SEFM 2022 Collocated Workshops, LNCS 13765, Springer, pp. 5–16, doi:10.1007/978-3-031-26236-4_1.

[14] Jan Gruteser, David Geleßus, Michael Leuschel, Jan Roßbach & Fabian Vu (2023): *A Formal Model of Train Control with AI-based Obstacle Detection*. In: Proceedings RSSRail, LNCS 14198, Springer, pp. 128–145, doi:10.1007/978-3-031-43366-5_8.

[15] Jan Gruteser & Michael Leuschel (2024): *Validation of RailML Using ProB*. In: Proceedings ICECCS 2024, LNCS 14784, Springer, pp. 245–256, doi:10.1007/978-3-031-66456-4_13.

[16] Dominik Hansen, Michael Leuschel, Philipp Körner, Sebastian Krings, Thomas Naulin, Nader Nayeri, David Schneider & Frank Skowron (2020): *Validation and real-life demonstration of ETCS hybrid level 3 principles using a formal B model*. Int. J. Softw. Tools Technol. Transf. 22(3), pp. 315–332, doi:10.1007/s10009-020-00551-6.

[17] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann & Matthias Volk (2022): *The probabilistic model checker Storm*. STTT 24(4), pp. 589–610, doi:10.1007/s10009-021-00633-z.

[18] Xiaowei Huang, Marta Kwiatkowska, Sen Wang & Min Wu (2017): *Safety Verification of Deep Neural Networks*. In: Proceedings CAV, LNCS 10426, Springer, pp. 3–29, doi:10.1007/978-3-319-63387-9_1.

[19] Daniel Jackson, Valerie Richmond, Mike Wang, Jeff Chow, Uriel Guajardo, Soonho Kong, Sergio Campos, Geoffrey Litt & Nikos Aréchiga (2021): *Certified Control: An Architecture for Verifiable Safety of Autonomous Vehicles*. CoRR abs/2104.06178, doi:10.48550/arXiv.2104.06178. arXiv:2104.06178.

[20] Ismet Burak Kadron, Divya Gopinath, Corina S. Păsăreanu & Huafeng Yu (2022): *Case Study: Analysis of Autonomous Center Line Tracking Neural Networks*. In: Proceedings VSTTE 2021, LNCS 13124, Springer, pp. 104–121, doi:10.1007/978-3-030-95561-8_7.

[21] Guy Katz, Clark Barrett, David L Dill, Kyle Julian & Mykel J Kochenderfer (2017): *Reluplex: An efficient SMT solver for verifying deep neural networks*. In: Proceedings CAV, LNCS 10426, Springer, pp. 97–117, doi:10.1007/978-3-319-63387-9_5.

[22] Marta Kwiatkowska, Gethin Norman & David Parker (2011): *PRISM 4.0: Verification of probabilistic real-time systems*. In: Proceedings CAV, LNCS 6806, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_-47.

[23] Michael Leuschel & Michael Butler (2008): *ProB: an automated analysis toolset for the B method*. *STTT* 10(2), pp. 185–203, doi:10.1007/s10009-007-0063-9.

[24] Michael Leuschel & Nader Nayeri (2023): *Modelling, Visualisation and Proof of an ETCS Level 3 Moving Block System*. In: *Proceedings RSSRail*, LNCS 14198, Springer, pp. 193–210, doi:10.1007/978-3-031-43366-5_12.

[25] Lalli Myllyaho, Mikko Raatikainen, Tomi Männistö, Tommi Mikkonen & Jukka K. Nurminen (2021): *Systematic literature review of validation methods for AI systems*. *Journal of Systems and Software* 181, p. 111050, doi:10.1016/j.jss.2021.111050.

[26] Kenzo Nonami, Farid Kendoul, Satoshi Suzuki, Wei Wang & Daisuke Nakazawa (2010): *Autonomous flying robots: unmanned aerial vehicles and micro aerial vehicles*. Springer Science & Business Media, doi:10.1007/978-4-431-53856-1.

[27] Corina S. Păsăreanu, Ravi Mangal, Divya Gopinath, Sinem Getir Yaman, Calum Imrie, Radu Calinescu & Huafeng Yu (2023): *Closed-Loop Analysis of Vision-Based Autonomous Systems: A Case Study*. In: *Proceedings CAV*, LNCS 13964, Springer, pp. 289–303, doi:10.1007/978-3-031-37706-8_15.

[28] Jan Peleska, Anne E Haxthausen & Thierry Lecomte (2022): *Standardisation considerations for autonomous train control*. In: *Proceedings ISoLA*, *LNCS* 13704, Springer, pp. 286–307, doi:10.1007/978-3-031-19762-8_22.

[29] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick & Ali Farhadi (2016): *You Only Look Once: Unified, Real-Time Object Detection*. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 779–788, doi:10.1109/CVPR.2016.91.

[30] Jan Roßbach, Oliver De Candido, Ahmed Hamman & Michael Leuschel (2024): *Evaluating AI-based Components for Autonomous Railway System*. In: *Proceedings KI 2024*, *LNAI* 14992, Springer, pp. 190–203, doi:10.1007/978-3-031-70893-0_14.

[31] Jan Roßbach & Michael Leuschel (2023): *Certified Control for Train Sign Classification*. *EPTCS* 395, pp. 69–76, doi:10.4204/eptcs.395.5.

[32] Wenjie Ruan, Xiaowei Huang & Marta Kwiatkowska (2018): *Reachability Analysis of Deep Neural Networks with Provable Guarantees*. In: *Proceedings IJCAI*, pp. 2651–2659, doi:10.24963/ijcai.2018/368.

[33] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine et al. (2020): *Scalability in perception for autonomous driving: Waymo open dataset*. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2446–2454, doi:10.48550/arXiv.1912.04838.

[34] Fabian Vu, Jannik Dunkelau & Michael Leuschel (2024): *Validation of Reinforcement Learning Agents and Safety Shields with ProB*. In: *NASA Formal Methods Symposium*, *LNCS* 14627, Springer, pp. 279–297, doi:10.1007/978-3-031-60698-4_16.

[35] Fabian Vu, Michael Leuschel & Atif Mashkoor (2021): *Validation of Formal Models by Timed Probabilistic Simulation*. In: *Proceedings ABZ*, *LNCS* 12709, Springer, pp. 81–96, doi:10.1007/978-3-030-77543-8_6.

[36] Michelle Werth & Michael Leuschel (2020): *VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics*. In: *Proceedings ABZ*, LNCS 12071, Springer, pp. 260–265, doi:10.1007/978-3-030-48077-6_21.

[37] Michael Wild, Jan Steffen Becker, Günter Ehmen & Eike Möhlmann (2023): *Towards Scenario-Based Certification of Highly Automated Railway Systems*. In: *Proceedings RSSRail*, *LNCS* 14198, Springer, pp. 78–97, doi:10.1007/978-3-031-43366-5_5.

Model Checking for Reinforcement Learning in Autonomous Driving: One Can Do More Than You Think!

Rong Gu
Mälardalen University
Västerås, Sweden
rong.gu@mdu.se

Most reinforcement learning (RL) platforms use high-level programming languages, such as OpenAI Gymnasium using Python. These frameworks provide various API and benchmarks for testing RL algorithms in different domains, such as autonomous driving (AD) and robotics. These platforms often emphasise the design of RL algorithms and the training performance but neglect the correctness of models and reward functions, which can be crucial for the successful application of RL. This paper proposes using formal methods to model AD systems and demonstrates how model checking (MC) can be used in RL for AD. Most studies combining MC and RL focus on safety, such as safety shields. However, this paper shows different facets where MC can strengthen RL. First, an MC-based model pre-analysis can reveal bugs with respect to sensor accuracy and learning step size. This step serves as a preparation of RL, which saves time if bugs exist and deepens users' understanding of the target system. Second, reward automata can benefit the design of reward functions and greatly improve learning performance especially when the learning objectives are multiple. All these findings are supported by experiments.

1 Introduction

With the advance of hardware and artificial intelligence (AI), *Autonomous Driving (AD)* has become more and more realistic around us. However, although automotive companies are running road tests for their AD vehicles over millions of miles a year, accidents still keep occurring [28], like crashes involving Tesla's driver-assistance system [23], and a fatal crash caused by a self-driving car of Uber [4]. Such accidents damage people's confidence in AD dramatically as the public usually cannot accept even a single accident caused by an AD vehicle. One of the reasons is that people even experts cannot fully understand why AD vehicles behave in a certain way as the AI components are different from conventional hardware and software systems and they can be unpredictable. Without knowing the reason behind an AD vehicle's every action, it is impossible to gain trust in the machine. Formal methods (FMs) are widely accepted for their ability to analyse safety-critical systems with mathematics-based methods. In recent years, scientific studies and projects have been conducted where FMs play an important role in providing safety assurance on AD systems [24][25][12][26][27][10]. However, FMs have limits when adopted in AD systems, such as the usability barrier due to the complex mathematical models and scalability due to the notorious state-space explosion. Another problem with using FMs in AD systems is the lack of tools that can provide visualisation of models, counterexamples, and analytical results.

Thanks to our previous work, two state-of-the-art tools in both FMs and AD research communities have been integrated, namely *CommonUppRoad* [13]. This new tool combines the model checker UPPAAL [18] with CommonRoad [2], an open-source toolset for AD development, testing, and visualisation. Users of CommonUppRoad can load and configure an AD scenario and specify the planning goal by programming it in Python. The tool then converts everything into a formal model that is recognisable

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 160–177, doi:10.4204/EPTCS.411.11

© Rong Gu
This work is licensed under the
Creative Commons Attribution License.

and verifiable by UPPAAL. Although UPPAAL is well-known for its ability of symbolic and statistical model checking of timed automata, the latest version of UPPAAL[1] also provides functions for controller synthesis. Specifically, UPPAAL can compute a so-called *strategy* that controls which transitions (aka, actions) to choose at each of the states. In this way, the model state space is restricted by the strategy and the exhaustive-search-based synthesis in UPPAAL guarantees that the strategy fulfils properties, e.g., safety. Additionally, UPPAAL also provides reinforcement learning (RL) algorithms to optimise a strategy. For example, a safe strategy can permit multiple actions at a state, but only one of them moves the AD vehicle towards the goal. RL can capture this action by accumulating rewards of state-action pairs and eventually control the AD vehicle to always choose the actions with the highest reward, i.e., the goal-reaching actions. Additionally, the fact that the learning is performed under the control of a safe strategy ensures the learning process as well as the result are still safe, and this is not given by pure RL. One of the barriers to using UPPAAL in the AD domain is that the strategies of UPPAAL are not visualised properly. One cannot get to know the moving trajectories of an AD vehicle under the control of a UPPAAL strategy. CommonUppRoad compensates for this disadvantage by leveraging CommonRoad's ability to visualise the driving scenarios and its rich database of real-world traffic scenarios.

Despite all the advantages, CommonUppRoad has unsolved problems. First, the synthesis of safe strategies is done via an exhaustive search of the state space. This is very similar to the symbolic model checking of UPPAAL. Therefore, the state-space explosion of symbolic model checking also exists in the synthesis of safe strategies. According to our experiments [13], the computation time rises exponentially when the maximum execution time of the AD vehicle increases linearly. Second, the motion model of AD vehicles in CommonUppRoad is not precise enough to represent continuous actions. For example, turning in CommonUppRoad only has two speeds, i.e., $\pm 0.1\ d/s$, which is not faithful to the vehicular dynamics in CommonRoad. Third, when the tool returns "no result", it means the state space does not contain a sequence of actions that satisfies the safety property, such as never colliding. However, it is hard to analyse where the bug originates because it can be caused by the sensor error or the control logic of the strategy. Last, besides the safety guarantee, model checking (MC) does not benefit RL in CommonUppRoad. However, there are many aspects that the former can do for the latter. For example, the design of the reward function is a dominant factor influencing learning efficiency and effectiveness. However, there is a lack of an automatic validation framework for reward functions in the context of AD [1]. The author believes MC can contribute to establishing such a framework.

This paper introduces how MC benefits RL in the AD context. First, the author proposes new model templates for AD vehicles and their driving scenarios in the new CommonUppRoad framework. By using the new model templates, analysis of sensor accuracy and a quick check on the existence of solutions can be conducted prior to RL. This step is highly beneficial but neglected by most studies. Second, the new model templates allow multiple timesteps of sensing and decision-making, as well as finer granularities of action discretisation, which can be taken as *continuous* actions in practice. Last, the author shows how symbolic and statistical model checking benefits the design and evaluation of reward functions. Statistical model checking and the corresponding model templates enable statistical analysis of probabilistic models, which are important for AD design and testing in uncertain environmental models. In a nutshell, the contributions of this paper are as follows:

- New model templates supporting "continuous actions" – a fine granularity of discretisation, and the corresponding model conversion from CommonRoad to UPPAAL.
- Model pre-analysis before RL. The analysis shows the data accuracy that RL can tolerate, suitable periods for sensing and decision-making, and the possible existence of an optimal motion plan. A

[1] UPPAAL 5 on uppaal.org

safety shield can also be automatically generated prior to RL such that the state-space exploration of RL is restricted to safe regions.

- Reward automata. This model and the corresponding analysis benefit RL designers in understanding the system model and reward functions. With reward automata, the learning performance can be greatly improved especially when the learning objectives are multiple.

The remainder of the paper is organised as follows. Section 2 introduces CommonUppRoad, which is the foundation of this study. Section 3 describes the new model templates before Section 4 introduces how model checking strengthens reinforcement learning. Section 5 presents the experiments and results. Section 6 presents the related work and Section 7 concludes the paper and introduces future work.

2 Preliminary

In this section, the author briefly introduces the aspects of CommonUppRoad that are necessary for understanding this paper. Interested readers are referred to the literature [13] for a detailed description of the tool including experimental results.

2.1 Models of AD Vehicles and Environment

In CommonUppRoad, users can load and configure an AD scenario that is stored as an XML file in CommonRoad [2], one of the fundamental tools of CommonUppRoad. Essentially, a scenario contains a network of roads, static vehicles and traffic signs on the roads, a planning goal, and a group of moving vehicles that can follow predefined trajectories or behave reactively to other vehicles. Static elements in the scenario, such as static obstacles, are presented as a group of XY coordinates in the XML file, representing their shapes (e.g., rectangle) and positions in a 2D environment. The states of moving vehicles consist of their positions, orientations, velocities, and accelerations at each of the time points until the maximum time. The scenario XML file stores moving trajectories as sequences of states that are sorted from the beginning to the maximum time. An example of moving trajectories is shown in Fig.1.

CommonRoad provides Python functions for visualising and parsing the XML file. CommonUppRoad calls these functions in the model conversion from AD scenarios to formal models, that is, timed games (TG) in UPPAAL. As a class of formal model, TG has syntax and semantics. Briefly, the syntax is the formal presentation of the model structure and the semantics is an interpretation of the syntactical model. Fig.1 has the snippets of two TG, which depict their syntactic presentations. The box on the bottom left is the AD-action TG. Although it is only partially shown in the figure (for the sake of page limit), one can see that the turning action, an edge from location `Choose` to location `Turn_Const`, is discrete. In UPPAAL, integers can be assigned to edges via a *select* statement, distinguishing syntactically the same edge at the semantic level. For example, in the AD-action TG, a select statement (`d:int[0,1]`) associates an integer `d`, whose value can be 0 or 1, to the edge from location `Choose` to location `Turn_Const`, meaning that turning has two angular speeds. Similarly, other actions that are syntactically represented by one edge can have multiple semantic transitions. Controller synthesis means to choose from those transitions at each of the states such that the TG can satisfy properties regardless of how the environment's actions, that is, the dashed edges, are performed. For example, the moving-obstacle TG in Fig.1 only has one location and three dashed edges, meaning that after initialisation, moving obstacles keep changing their state variables continuously, updating their discrete variables and making decisions periodically. If the planning goal is to reach a destination, then motion planning here

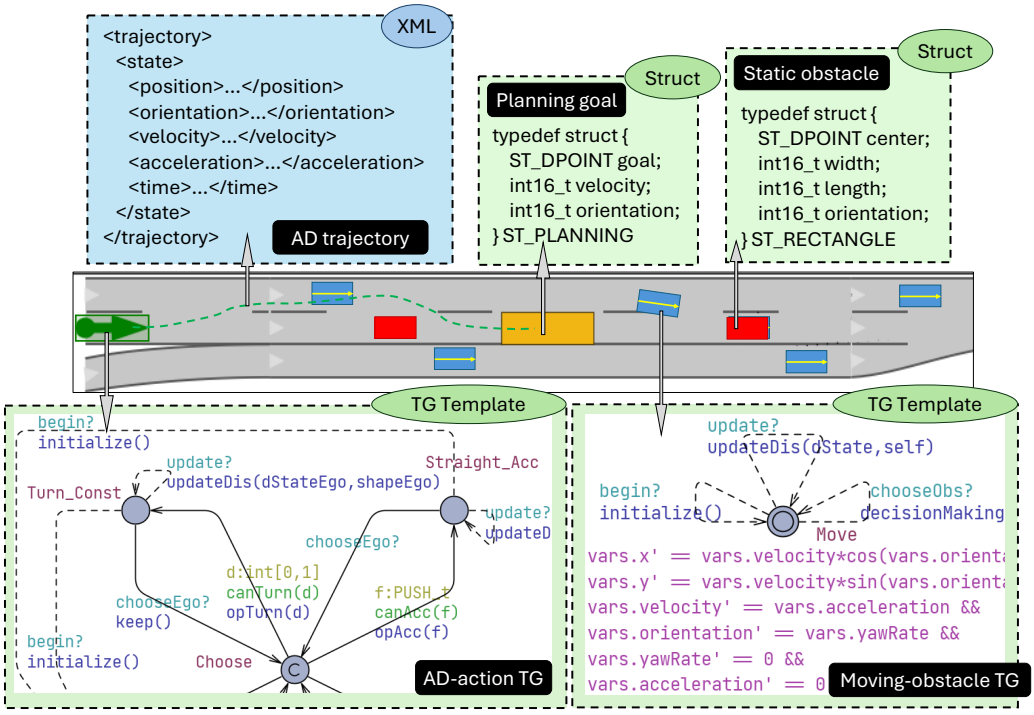


Figure 1: CommonUppRoad, a platform for model-checking-based AD motion planning, verification, and visualisation. Figure adopted from the literature [13].

means to choose actions at states of the AD-action TG such that the destination is reachable no matter how the actions in the moving-obstacle TG are taken.

2.2 Motion Planning Methods and Visualisation

CommonUppRoad has two kinds of motion-planning methods, i) exhaustive-search-based synthesis (ESS), and ii) reinforcement-learning-based synthesis (RLS).

2.2.1 Exhaustive-Search-Based Synthesis (ESS)

As synthesis is about finding a combination of actions of the AD vehicle, a natural method is to exhaustively explore the model state space and collect the desired execution traces, i.e., sequences of state-action pairs. ESS follows this idea but the problem is that timed automata have infinite and uncountable states because of the continuous variable, *time*. UPPAAL uses *zones* to represent the infinite values of time such that the symbolic state space of timed automata is finite and countable [5]. ESS uses the symbolic state space and the exploration is on the fly, that is, checking the property while exploring the state space simultaneously. Specifically, CommonUppRoad uses the following query in UPPAAL, where `A[]` means all the states of all the traces in the state space, and functions `collide()` and `offroad()` are functions for judging if a collision or going off the road ever happens, respectively. Therefore, Query (1) aims to collect all the state-action pairs where those two functions return *false*.

```
strategy safe = control:  A[] !collide() && !offroad()        (1)
```

When Query (1) is executed, UPPAAL explores the entire symbolic state space of the model of the AD vehicle and environment, checks if `collide()` or `offroad()` ever returns *false*, and stores the execution

traces where the check passes. If a strategy is synthesised, it is represented as a function mapping actions to each of the states. Since the state-space exploration is exhaustive, ESS is *sound* and *complete*, that is, the synthesised strategy must be correct, i.e., free of colliding and offroad situations (soundness), and if a correct strategy exists, the method must be able to find it (completeness). Henceforth, we call strategies computed via ESS safe strategies. A safe strategy is *permissive* as it contains all the safe state-action pairs, including the inefficient ones. For example, when an AD vehicle needs to turn left at an intersection, a permissive strategy would allow the AD vehicle to wait unnecessarily long, e.g., close to the maximum execution time, and then turn. Therefore, permissive strategies can be optimised.

2.2.2 Reinforcement-Learning-Based Synthesis (RLS)

RLS explores the model state space randomly via simulations instead of exhaustive exploration. Note that states in RLS are not symbolic anymore as they have concrete values of the state variables, including time. Scores of the state-action pairs are computed through a reward function and accumulated during the random exploration. After user-defined rounds of simulation (aka, learning episodes), learning finishes with a score table containing the pairs and their scores. Following the control of a learned strategy means always choosing the action with the highest/lowest score at each of the states. The advantage of RLS is that state-space explosion does not exist as the exploration is not exhaustive now, but the method is neither sound nor complete. We need methods to provide correctness guarantees on the learning results. CommonUppRoad uses safety strategies synthesised by UPPAAL running Query (1) to achieve this goal. Specifically, RL executes the following query in UPPAAL, where `maxE(reward)` means the learning objective is to maximise rewards, `MAXT` is the maximum time of one learning episode, `<>` is a temporal operator meaning the existence of a state in any of the traces, `goal()` is a function to judge if the AD vehicle reaches the destination, and `under safe` means the random selection of state-action pairs must be within the pair set of strategy `safe`. Hence, the safe strategy serves as a safety shield for learning and must be synthesised prior to running Query (2), and the latter aims to sample and compute the scores of the pairs from traces that have a state reaching the destination and the state space must be restricted by the safe strategy (aka, safety shield).

```
strategy reachSafe = maxE(reward)[<=MAXT]: <> goal() under safe      (2)
```

Strategies can be visualised in CommonUppRoad as animated moving trajectories[2]. Specifically, CommonUppRoad simulates the model by using the following query, where `x`, `y`, and `velocity` are the representative variables that are in the state structure of trajectories in Fig.1. Query (3) randomly simulates the model under the control of strategy `reachSafe`, samples values of the variables in the curly brackets, and stores them in a log file, which is parsed in CommonUppRoad for visualisation. Strategy visualisation is another important feature that CommonUppRoad provides because there was no easy way to visualise UPPAAL strategies as moving trajectories.

```
simulate [<=MAXT]{x, y, velocity, ...} under reachSafe      (3)
```

2.2.3 Unsolved Problems in CommonUppRoad

Modelling of Continuous Actions. As shown in Fig.1, the AD-action TG only has discrete actions. For instance, turning only has two possible angular speeds in CommonUppRoad but the vehicle dynamics in CommonRoad can have continuous actions, e.g., a continuously changing angular speed represented by

[2]Trajectory animations are posted online: sites.google.com/view/commonupproad/experiment

a real number. However, formally modelling continuous actions is not trivial. Once the model includes continuous variables, safe shields as Query (1) are not supported anymore unless those variables are hybrid clocks. Hybrid clocks are special variables in UPPAAL. They are continuous and their changing rates are described by ordinary differential equations (ODE), but they cannot be used in guards (i.e., boolean expressions on edges) or invariants (i.e., boolean expressions on locations) as the values of hybrid clocks are not supposed to change the model behaviour. In other words, hybrid clocks are abstracted away from symbolic analysis, and thus UPPAAL can still use zones to form a symbolic state space of the model. Therefore, although moving obstacles have nonlinear and continuous dynamics, their variables are modelled as hybrid clocks in UPPAAL and thus CommonUppRoad can still synthesise safety shields for reinforcement learning.

Modelling AD-action TG is not the same. If one models the AD vehicle's actions as hybrid clocks and ODE, they would not be taken into the symbolic state space and thus safety-shield synthesis would have no actions to learn from. This is not the problem of the tool, UPPAAL, but rather a theoretical limit. Models with both discrete and continuous components are hybrid automata, and the reachability problem of hybrid automata is undecidable in general [14]. Hence, there is no model checker that supports exhaustive verification of hybrid automata so far. Hybrid clocks provide a way to model continuous actions but one needs methods to represent those continuous actions symbolically.

Model pre-analysis before RL. Like most studies of AD motion planning, CommonUppRoad attempted to synthesise an AD controller without analysing the model itself. However, there are at least two questions before synthesis starts: i) are the data perceived by the digital controller of an AD vehicle accurate enough, and ii) does a valid motion plan exist in the model state space? Question i) comes from the fact that the controller of an AD vehicle is a piece of software that periodically reads data about the surrounding environment and sends signals to control the vehicle. As a piece of software, no matter how accurate the sensors are, it inevitably truncates real numbers to floating-point numbers, which have a limited length of digits. Additionally, sensors cannot be perfect. When a new measuring method is introduced into an AD system, one may want to investigate the intervals of sensor errors before running motion planning. Question ii) comes as a following concern after question i). Given the current sensor error and motion primitives (aka, the atomic motions used in RL), does a valid solution exist? If one can get a negative answer to this question quickly, there is no need to run RL at all. Therefore, a model analysis before RL can be greatly beneficial.

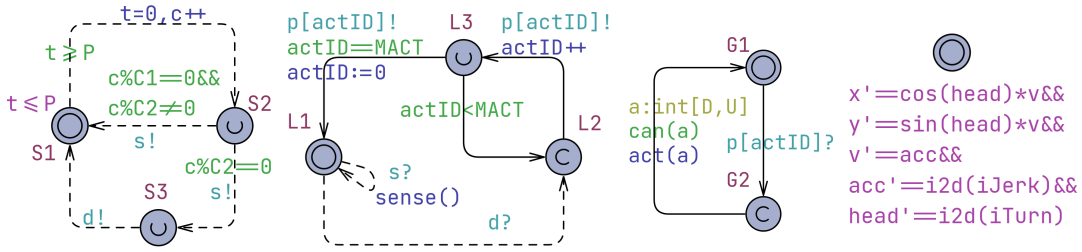
3 New Model Templates of AD Vehicles and Environment

Before introducing what model checking can do for reinforcement learning in the context of AD vehicles, the author describes the new model templates of CommonUppRoad in this section. The new model templates allow symbolic and statistical model checking, which play the fundamental role in the model pre-analysis and reward automata design (Section 4).

3.1 Model Templates of AD Vehicles

Fig.2 depicts three UPPAAL model templates of the AD vehicles, i.e., controller, action, and dynamics. AD vehicles are cyber-physical systems that consist of digital and physical components. Fig.2(a) - Fig.2(c) are the digital components describing the control logic.

Timer is a timed automaton calling other models, e.g., `Controller`, periodically. This is for modelling CPU periodically calling the processes of the controlling software and reading sensors. In Fig.2(a),



(a) Timer (b) Controller (c) Action (d) Dynamics

Figure 2: UPPAAL model templates of AD vehicles. In (a), (b), and (c), `s`, `d`, and `p` are broadcast synchronisation channels, `sense()`, `can()`, and `act()` are C-like code functions, `a`, `c`, and `actID` are integers, and `P`, `C1`, `C2`, `D`, `U`, and `MACT` are constant integers, and `t` is a clock. In (d), variables `x`, `y`, `v`, `acc`, and `head` are hybrid clocks, and `iJerk` and `iTurn` are integers, equations like `v'==acc` define the derivatives of the hybrid clocks, and `i2d` is a C-like code function.

the model leaves location `S1` every `P` time unit and comes to an urgent location `S2` meaning that the next transition starts immediately when reaching this location. Transitions from `S2` have two options: i) going back to `S1` directly, ii) going back to `S1` via another urgent location `S3`. In case i), the transition is synchronised with `Controller` on channel `s` meaning that the `Controller` reads data from sensors by calling function `sense()`. In case ii), the first transition to location `S3` is the same as case i), which is followed by a transition to location `S1` synchronised on channel `d` meaning that the `Controller` starts to make a decision of actions after reading the sensor data. Note that locations `S2` and `S3` are both urgent, meaning that the operations of sensor-reading, decision-making, and performing actions happen instantaneously at the end of each period. Constants `C1` and `C2` are for distinguishing the periods of cases i) and ii), which are also used in the model pre-analysis before running RL (Section 4.1).

Controller and **Action** are also timed automata. When `Timer` invokes sensor-reading, `Controller` transits via a self-loop edge of its initial location `L1`, meaning that `Controller` does nothing but reads data from sensors in this period. When `Timer` invokes decision-making, `Controller` transits to a committed location `L2`, meaning that the next transition must start from this location immediately. Next, `Controller` goes to an urgent location `L3` synchronising on channel `p[actID]`, in which `actID` is an integer identifying actions. In Fig.2(c), `Action` transits to a committed location `G2` synchronising on channel `p[actID]` too. The difference is that now location `G2` is committed, so transitions from `L3` in `Controller` must wait until the ones in `Action` finish, meaning that actions are atomic and cannot be interrupted. From `G2`, `Action` goes back to `G1` via multiple actions although only one edge appears from `G2` to `G1`. The author labels this edge with a `select` statement, which assigns different integers to an edge. In this way, an edge can represent multiple transitions at the semantic level. Here, `a:int[D,U]` means an integer `a` from constant `D` to constant `U` is selected, where `a` is a variable associated to this action, and `D` and `U` are the lower and upper boundaries of `a`, respectively. This design is for modelling continuous actions.

In the field of control theory, real numbers are often used to present continuous actions and their dynamics. However, it is impossible to represent real numbers in digital systems precisely, because digital systems use floating-point numbers to represent real numbers. A truncation is inevitable when the real number is not rational or its digit lengths exceed the limit of the digital system, e.g., 32 bits or 64 bits. In the model-checking world, floating-point numbers are not welcome as they would overly bloat a model's state space and floating-point computations are unstable due to the *cancellation* effects [16]. For example, if a model contains floating-point numbers (aka, type `double`), UPPAAL does not allow

symbolic analysis, e.g., symbolic model checking and ESS.

To avoid this problem, floating-point numbers in the new model templates are either abstracted away from the symbolic state space, e.g., hybrid clocks, or represented as integers. Specifically, the *base* and *exponent* of floating-point numbers are defined as constant integers, and the *significand* of each floating-point number is an integer, which is used to represent this floating-point number. For example, when the base is 10 and the exponent is -4, 21200 represents 2.12.

$$2.12 = \underbrace{21200}_{significand} \times \overbrace{\underbrace{10^{-4}}_{base^{exponent}}}^{scale} \tag{4}$$

In the model `Action`, constants `D` and `U` are the integer representations of two floating-point numbers, that is, the lower and upper boundaries of continuous variable `a`. When the period of decision-making comes, `Action` needs to choose a value from `D` to `U` for variable `a`, which indicates selecting a continuous action. One can set the granularity of continuous actions by changing the exponent of floating-point numbers. Although this representation is an approximation of real numbers, it is how digital systems work and allows symbolic analysis in UPPAAL.

Dynamics is a hybrid automaton (Fig.2(d)), where variables `x`, `y`, `v`, `acc`, and `head` are hybrid clocks, indicating the AD vehicle's X and Y coordinates, velocity, acceleration, and heading, respectively. Equations, such as `v'==acc`, define the derivatives of the hybrid clocks. The rate of acceleration (aka, jerk) and turning are modelled as integers (i.e., `iJerk` and `iTurn`) because these two variables are changed by actions, and the integers are the significands of the floating-point numbers associated to continuous actions. Therefore, before they are used in any computation, such as the equations of derivatives, they must be transformed back to floating-point numbers. This is done in function `i2d`, where the significand multiplies the scale and becomes a floating-point number. Oppositely, another function `d2i` transforms a floating-point number into an integer by dividing the former by the scale.

3.2 Numerical Integration

Since the continuous variables in `Dynamics` are hybrid clocks, one cannot use them in symbolic analysis in UPPAAL. However, functions like `sense()` are supposed to detect the values of these variables. Thanks to the integer representations of floating-point numbers, the author implements a function for calculating the values of continuous variables by using numerical integration. Algorithm 1 shows the numerical integration in function `sense()`.

Algorithm 1: sense(): numerical integration

```
1  Function sense()
2      int steps := 1/G                          // G is the granularity of integration
3      double x := i2d(iX)                          // convert integer iX to double x
4      double y, v, acc, head ...                // convert the rest integers to double
5      double unit = C1 · G                         // C1 is defined in Timer, Fig.2(a)
6      while steps ≠ 0 do
7          acc := acc + iJerk · unit
8          head := head + iTurn · unit
9          v := v + acc · unit
10         x := x + v · cos(head) · unit
11         y := y + v · sin(head) · unit
12         steps := steps - 1
13     iX := d2i(x)                                    // convert double x to integer iX
14     iY, iV, iAcc, iHead ...                   // convert the rest double to integers
15     return
```

Line 3 and line 4 convert variables like `iX` to floating-point numbers. From line 7 to line 12, the results of the ODE in the model `Dynamics` are approximated by the numerical integration. Specifically, the floating-point numbers are updated step by step with a changing unit $C1 \cdot G$, where $C1$ is the length of the sensing period and G is a predefined sampling granularity of the integration points. Although the numerical integration only approximates the integral, it reflects what the digital system perceives via sensors, that is, periodically updated discrete variables. Between two consecutive sampling periods, the variables' values remain unchanged in the controller. After the numerical integration, line 13 and line 14 convert the floating-point numbers back to their integer representations, which can be used in functions like `collide()` and `offroad()`.

3.3 Model Template of Moving Obstacles

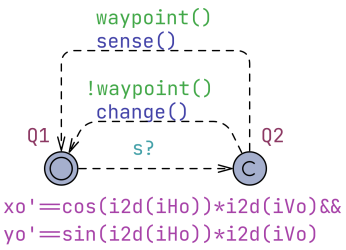


Figure 3: Obstacle

Moving obstacles in CommonRoad usually follow predefined trajectories (Fig.1), therefore the model template of moving obstacles is relatively simple. Fig.3 is the model template, where `xo` and `yo` are the only two hybrid clocks because the velocity (`iVo`) and heading (`iHo`) of a moving obstacle immediately changes when it reaches a waypoint on the trajectory. One may argue that the model template is not faithful to real moving obstacles because the velocity and heading of an object in the real world must be continuous. However, this model template is based on two conditions: i) the moving obstacle follows a predefined trajectory, and ii) the controller of the AD vehicle is digital. Condition i) is explained, so the moving obstacle can only change its velocity and head at the waypoints of the predefined trajectory. Condition ii) is also true because our AD vehicle is a cyber-physical system. Therefore, the information on moving obstacles is discrete from the AD vehicle's point of view. Besides, this modelling removes unnecessary details, which simplifies verification and synthesis. One can easily add probabilistic behaviour to moving obstacles, like the uncertain velocity due to erroneous sensors of AD vehicles. During learning and statistical model checking, non-deterministic transitions in the model template are interpreted as stochastic transitions with a uniform probability distribution by UPPAAL. However, this paper does not focus on such behaviours and interested readers are referred to the website of CommonUppRoad[3].

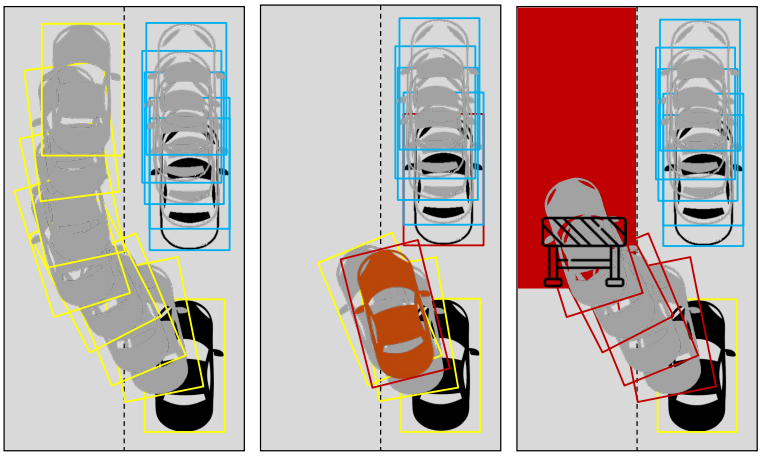
4 Model Checking for AD Reinforcement Learning

In this section, the author introduces three aspects that model checking (MC) can do for reinforcement learning (RL) in the context of AD vehicles. First, MC enables model pre-analysis and quick checking for the existence of an optimal motion plan. Second, to achieve the best learning performance, MC helps to choose a suitable decision-making period, construct reward automata, and synthesise a safety shield for the learning process. Last, MC can verify the learning results.

4.1 Pre-analysis of AD Vehicle and Scenario Models

When facing a motion-planning problem such as the one in Fig.4, the first task is probably designing a good searching algorithm to find the optimal state-action pairs in the model state space. However, such solutions neglect two important aspects that should be considered before the motion planning starts.

[3] sites.google.com/view/commonupproad



(a) Lane changing (b) A missing frame (c) No valid motion plan

Figure 4: An illustration of potential problems when running RL without pre-analysis.

First, all AD vehicles are cyber-physical systems, which means the controller is a piece of software that periodically reads sensor data, makes decisions, and sends signals to actuators to control the physical processes of the vehicle. Between two periods, the vehicle moves according to the latest controlling signals. For example, Fig.4(a) depicts the discrete frames capturing the AD vehicle's trajectory, which is safe because the yellow boxes do not overlap with the blue boxes, i.e., the collision detection ranges of the AD vehicle and moving obstacle, respectively. However, such a trajectory does not guarantee a safe lane-changing manoeuvre. If the sensing periods are too long, the AD vehicle may miss critical frames of collision (Fig.4(b)). If the decision-making periods are too short, learning may spend too much time on trivial behaviour. Additionally, computer systems inevitably lose accuracy when representing real numbers. The mismatch of data types would result in unexpected behaviour.

Fig.4(c) shows another scenario where running RL is meaningless because a safe motion plan does not even exist. Therefore, one may want to ensure the existence of an optimal motion plan before consuming any resource for learning. However, checking the existence of an optimal motion plan can be as difficult as finding one, because motion planning usually has multiple objectives, such as safety, progress, comfort, and traffic rules conformance. However, it is relatively easy to decide the non-existence of valid motion plans because safety is the first and foremost target of motion planning. If one can quickly conclude the non-existence of safe motion plans, one does not even need to run RL. Symbolic model checking is a powerful tool for this job.

Table 1: UPPAAL queries for model pre-analysis, where `fabs` returns the absolute value of a floating-point number, `cv` is a hybrid clock and `iv` is an integer, `THD` is the threshold of sensor errors.

	Query	Explanation
Q_a	`Pr[<=MAXT]{fabs(cv-i2d(iv))≥THD}`	Probability of sensor errors
Q_b	`E[]!collide()&&!offroad()`	Existence of a safe path
Q_c	`E<>!collide()&&!offroad()&&goal()`	Existence of a safe and reachable path
Q_d	`strategy safe=control:A[] !collide()&&!offroad()`	Safety shield for RL

As aforementioned in Section 3.1, the controller model has two periods, one for reading data from sensors and one for decision-making. One can define the lengths of those two periods and execute the UPPAAL queries in Table 1 for model pre-analysis. In Q_a, `cv` is a hybrid clock representing a

continuous variable in the physical component of the AD vehicle, i.e, model `Dynamics` in Fig.2(d), and `iv` is an integer used in the controlling software, i.e., model `Controller` in Fig.2(b). When the difference between these two variables exceeds a threshold, it means the information perceived by the controlling software is too far away from the ground truth, and thus a sensor error is discovered. Q_a uses statistical model checking and returns the probability of sensor errors occurring. For example, when `cv` represents the physical distance between the AD vehicle and the front car, `iv` represents the same distance but calculated by using discrete frames, and the difference between `cv` and `iv` is too large, it indicates the sensing periods are too coarse and a colliding scenario similar to Fig.4(b) may exist.

Q_b and Q_c use symbolic model checking, which exhaustively explores the entire symbolic state space of the model. If Q_b is satisfied, an absolutely safe path exists in the state space where no collision or offroad event happens, otherwise, no safe path exists and thus RL is not needed any more. However, the path found by Q_b does not necessarily reach the goal. Q_c adds a condition `goal()`, which returns *true* when the AD vehicle reaches the destination. If Q_c is satisfied, a path reaching the destination safely is found, otherwise, a safe and reachable path does not exist and thus RL is not needed either (Fig.4(c)). Q_d is for synthesising a safe strategy that guarantees the AD vehicle is safe regardless of how other vehicles move in the environment. This is the same query for synthesising safety shields for RL, i.e., Query (1). Q_b - Q_d are symbolic analysis, which excludes hybrid clocks. Hence, if a sensor error is indicated by Q_a, Q_b - Q_d are not needed because even if a symbolic safe path and a shield are found, they are not necessarily safe in real scenarios. If Q_a returns a value lower than the tolerant level of sensor errors, but Q_b or Q_c is not satisfied, one may want to shorten the decision-making periods, change or add the AD actions, or extend the time limit. All of these changes can be easily configured in the Python code of CommonUppRoad. One does not need to know the formal model templates (Fig.2) behind the Python code, which makes CommonUppRoad user-friendly to researchers outside the FMs community.

Now if Q_a - Q_c all show positive results but Q_d fails, it means the moving obstacles, like the front car in Fig.4, can make the AD vehicle deviate from the path of Q_b or Q_c. One can modify the behaviour of the AD vehicle or shorten the decision-making periods so that the AD vehicle has more chances to avoid moving obstacles. In some cases, however, the AD vehicle only needs to make decisions at a few critical time steps, but RL with a fixed learning step size makes it unnecessarily complex. Having different lengths of sensing and decision-making periods makes RL in CommonUppRoad adaptable to simple and complex scenarios. One can start with a large decision-making period but a standard sensing period, and run Q_d to see if a safety shield exists. If it does exist, one can decompose the motion-planning problem into sub-problems by dividing it at the time points of decision-making. Next, one can fine-tune the motion plan for each of the sub-problems by using RL. We refer interested readers to our previous work about this concatenated motion planning and verification [21].

4.2 Model Checking Facilities Multi-Objective Learning

Autonomous driving usually involves multiple objectives. Naturally, safety and reachability are two major objectives, which require the AD vehicle to reach the destination without collision or going off the road. Additionally, comfort and traffic rule conformance are also important objectives of AD. In some applications, timing and efficiency are also non-negligible requirements. Although the objectives of AD are multiple, the goal of RL is simple, i.e., maximising the cumulative reward. Therefore, the design of AD reward functions must take into account all the objectives, which is not trivial because those objectives can be contradictory. For example, RL rewards AD vehicles for progressing towards the goal and punishes them for collisions. When a permanent obstacle blocks the only path to the goal, RL may eventually motivate the AD vehicle to crash into the obstacle if the cumulative waiting penalty exceeds

the collision penalty. This erroneous behaviour stems from AD engineers' insufficient understanding of reward functions in the AD context. Formal models, however, provide rigorous semantics of the AD vehicles' behaviour as well as reward functions, which would greatly help the AD engineers design multi-objective reward functions and even verify them before running RL.

In this paper, the author selects three AD objectives in the literature [1], that is, safety, progress, and comfort, and proposes different methods to cope with them. Additionally, the reward functions of all objectives are integrated into one automaton, which is similar to the concept of *reward machines* in the literature [15]. This design has many benefits. First, it enables the users of CommonUppRoad to consider the various driving contexts in one model. Under different contexts, e.g., different weather conditions, the reward functions can be different. Second, a reward automaton allows users to verify the design of reward functions before RL. One can observe the change of rewards by exploring the model's symbolic state space step by step, or model check properties, such as the penalty for unsafe behaviour is always larger than the summation of rewards of other objectives.

4.2.1 Reward Automata

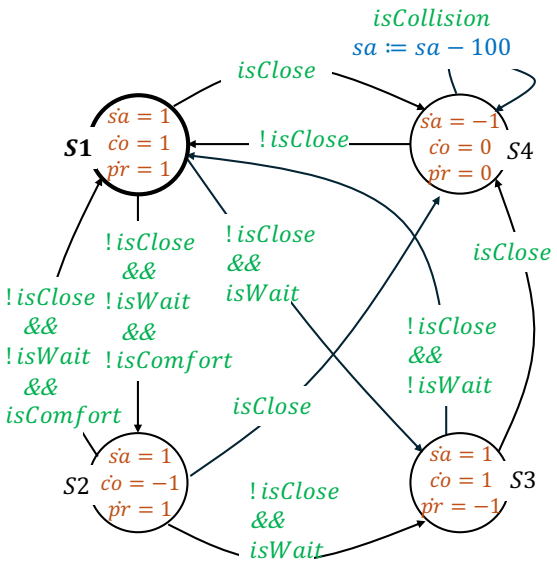


Figure 5: An example of reward automata. Hybrid clocks *sa*, *co*, and `pr` represent the rewards for safety, comfort, and progress, respectively.

Fig.5 shows an example of reward automata, which considers the aforementioned three objectives in the following order of priority: *safety>progress>comfort*. State `S1` is the initial state, where the AD vehicle moves normally and the three rewards increase steadily at the rate of one. Once the distance between the AD vehicle and an obstacle is lower than a threshold, i.e., `isClose` is *true*, the reward automaton transits to state `S4`, where `sa` decreases and the other two rewards stay the same. This indicates that once an unsafe situation occurs, no reward should increase. This prohibits the cumulative rewards from becoming larger than the penalty for unsafe behaviour. Similarly, the reward automaton can transit to states `S2` and `S3`, where the rewards for comfort and progress decrease, respectively. State `S4` has a self-loop, where a collision happens and `sa` is divided by a large value, e.g., 100. This design follows the recommendation from the literature [1], i.e., a safety reward should include a sparse penalty for collisions and a continuous dense term that penalises dangerous behaviour. However, if one synthesises a safety shield for RL first (Q_d in Table 1), this self-loop is not necessary because the safety shield eliminates all collisions and offroad events.

One can verify reward automata together with the AD vehicle model, which helps in understanding and improving reward functions. Table 2 lists some exemplary UPPAAL queries for reward automata verification. Q_e - Q_g verify if the reward automaton transits to the right state when the AD vehicle's behaviour presents the corresponding features. For example, Q_g verifies whether the reward automaton goes to the right state to decrease the comfort reward when the acceleration or the angular speed becomes larger than 80% of its maximum value, meaning the AD vehicle is accelerating or turning too much and makes passengers feel uncomfortable. However, the reward automaton only does this transition when

the Boolean variable `comf`, i.e., comfort, is false, and other Boolean variables `prog` and `safe` are true, because the objective *comfort* has the lowest priority among all objectives. Q_f is designed similarly. Counterexamples returned from these queries greatly help the AD vehicle designers to see the problems of their reward functions and improve them before RL starts.

Table 2: Examplary UPPAAL queries for the verification of reward automata `RA`, where `safe`, `prog`, and `comf` are Boolean variables, `iX` is the integer representation of a hybrid clock `x`, `TD` is the distance threshold of collision, `S` is a non-negtive integer, `MA` and `MT` are maximum acceleration and turning speed, respectively, p_1 and p_2 are real numbers between 0 and 1, $\texttt{AD}_N$ and $\texttt{Obs}_N$ are the positions of the AD vehicle and an obstacle at the $\texttt{N}_{th}$ period, respectively, `GO` is the goal area, $\texttt{Dis}(\square_1,\square_2)$ computes the distance between two rectangles, which represent objects in the environment, and `w1` - `w3` are weights.

	Query	Explanation
Q_e	`A[] !safe imply RA.S4,` `safe=Dis(`$\texttt{AD}_N$`,`$\texttt{Obs}_N$`)`$> S \cdot TD$	If AD vehicle and an obstacle gets too close, the safety reward is punished.
Q_f	`A[] !prog&safe imply RA.S3,` `prog=Dis(`$\texttt{AD}_N$`,GO)>Dis(`$\texttt{AD}_{N+1}$`,GO)`	If the distance from AD to the goal increases, the progress reward is punished.
Q_g	`A[] !comf&prog&safe imply RA.S2,` `comf=iAcc<`p_1`·MA&iTurn<`p_2`·MT`	If acceleration or angular velocity is too large, the comfort reward is punished.
Q_h	`A[] RA.S4 imply` `w1·iSa+w2·iCo+w3·iPr`$\leq$`0`	The safety reward cannot be compensated by other rewards when unsafe behaviour occurs.

5 Experiments

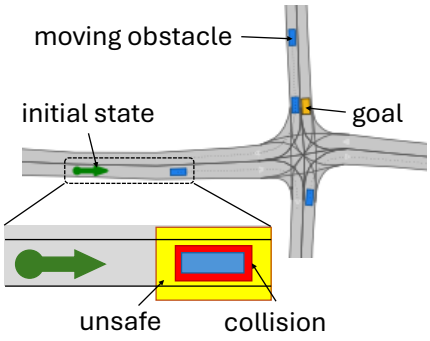


Figure 6: Experimental AD scenario

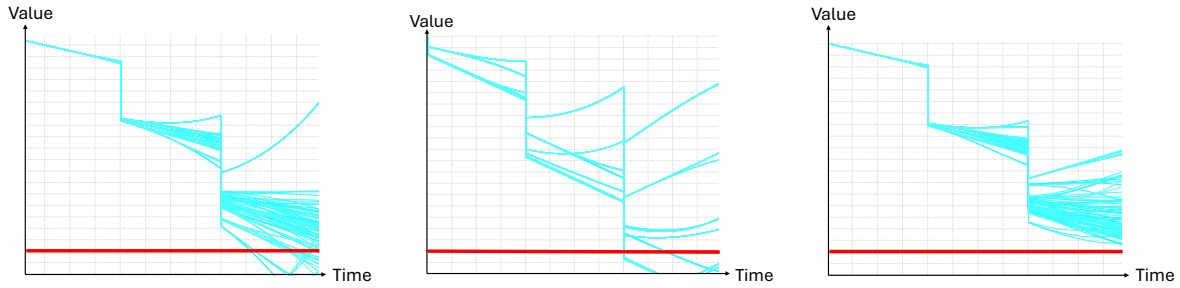
The experiments in this section aim to demonstrate the strengths of MC-enhanced RL in two aspects: i) the necessity of model pre-analysis, and ii) the benefits of MC in designing reward automata[4]. The selected AD scenario is depicted in Fig.6, where the AD vehicle needs to enter the intersection, turn left, and drive to the goal area safely. Surrounding obstacles, there are two critical ranges: *unsafe* and *collision*. Once the distance between the AD vehicle and an obstacle is less than `TD` (resp. 3·`TD`), a collision (resp. unsafe) situation happens.

To show the necessity of model pre-analysis, the author intentionally decreases the sensor accuracy and extends the sensing periods. Specifically, the exponent (see Equation (4)) is decreased from four to one such that the integer representations of floating-point numbers preserve only one digit after the decimal. The sensing periods are set to be two time units such that sensors may miss critical frames. First, the author executes Query (1) to synthesise a safety shield, namely `safe`, and verify the model by the following queries.

$$\texttt{Pr[}\leq\texttt{MAXT](<> Dis(AD,Obs)}\geq 3\cdot\texttt{TD)}\ \texttt{under safe} \tag{5}$$

$$\texttt{simulate[}\leq\texttt{MAXT;100](3·TD,Dis(AD,Obs))}\ \texttt{under safe} \tag{6}$$

[4]The model for experiments: sites.google.com/view/commonupproad/experiment. Run it in UPPAAL 5.1.0-beta5.



(a) Exponent: 1, sensing period: 1 (b) Exponent: 4, sensing period: 2 (c) Exponent: 4, sensing period: 1

Figure 7: Model pre-analysis result (Query (6)). The blue lines are the distances between the AD vehicle and an obstacle in 100 simulations. The red line is the threshold of unsafe distance, i.e., $3 \cdot$`TD`.

Fig.7 shows the results of Query (6) in different settings and 100 simulations. In the first two settings, the AD vehicle's distance to an obstacle can be less than the threshold even though the model is under the control of the safety shield. This shows that the exponent and sensing period are inadequate. Fig.7(c) has no cases exceeding the threshold, and thus the author chooses this setting in the following experiments.

There are several RL algorithms implemented in UPPAAL, such as Q learning. One can also call their own RL algorithm in UPPAAL from an external library [12]. The next experiment uses two synthesis methods: Q learning with/without a safety shield, using a reward automaton or a reward function. First, the author executes Query (7), in which `dv1` and `cv1`, etc., are state variables used in RL, and `rf` can be a reward function (Equation (8)) or a summation of hybrid clocks (Equation (9)), where `safe`, `prog`, and `comf` are Boolean variables (Table 2), and `sa`, `pr`, and `co` are hybrid clocks (Fig.5). As strategy `reach` may not be safe or reach the goal, the author verifies it against Queries (10) and (11).

```
strategy reach = maxE(rf)[<=MAXT]{dv1,...}->{cv1,...}:  <> time>=MAXT    (7)
```

$$\texttt{rf} = 10 \cdot \texttt{safe} + (5 + 100 \cdot \mathit{goal}()) \cdot \texttt{prog} + \texttt{comf} \tag{8}$$

$$\texttt{rf} = \texttt{sa} + \texttt{pr} + \texttt{co} \tag{9}$$

```
A[] !collide() && !offroad() under reach                                  (10)
```

```
A<> goal() under reach                                                    (11)
```

In comparison, the author also synthesises a safety shield (Query (1)) before RL. The synthesis query now is Query (2) such that the learning process and result are guaranteed to be safe. The computation time is in Table 3, where the column *learning episodes* refers to the number of simulation rounds that RL used in the experiment. The experiment is conducted on a Macbook Pro with an Apple M2 Pro chip and 16 GB memory. The OS is Sonoma 14.6.1 and UPPAAL's version is 5.1.0-beta5. Safety-shield synthesis costs around 30 seconds, which is much longer than what the pre-analysis takes (i.e., Q_b and Q_c), that is, within 3 seconds together. In the cases where safety-shield synthesis takes too long, the pre-analysis would be even more beneficial. The computation time of RL with a reward function and without a safety shield (aka, RF) costs the longest time because it does not have a safety shield to restrict the model space and the reward function does not guide the state-space exploration as efficiently as the reward automaton.

The author would like to discuss more about the benefits of using reward automata (RA) and reward functions (RF). Table 3 indicates that the computation time of RL with a safety shield costs a similar time when using RA or RF. However, the design of RF benefited heavily from RA in the experiment.

For example, the author chose the weights in Equation (8) by observing the behaviour of the reward automaton and verifying queries in Table 2. Those queries helped the author understand the mistakes of weights because the priority order of the objectives was broken, e.g., the accumulated rewards of progressing exceeded the punishment of unsafe situations. In other words, without reward automata, the design of RF would take much longer time in the trial-and-error process. However, the computation times of queries in Table 2 were less than 20 seconds in the experiment.

Table 3: RL with/without safety shields (SS), using reward automata (RA) or reward functions (RF)

Combination	Queries	Computation time	Learning episodes
SS&RA	Query (1)	34.8 s	/
	Query (2)	15.1 s	20
SS&RF	Query (1)	34.8 s	/
	Query (2)	16.0 s	20
RA	Query (7)	15.1 s	20
RF	Query (7)	**310.8 s**	**500**

6 Related Work

In the formal methods (FMs) community, verification of reinforcement learning in autonomous driving (AD) has drawn wide interest. Khaitan et al. [17] propose a curriculum learning approach for training a deep reinforcement learning agent. The performance of the curriculum is tested on the task of traversing unsignalised intersections with the CommonRoad framework. Naumann et al. [22] propose a motion planning method through probabilistic analysis under occlusions and limited sensor range, and use a real-world scenario with actual existing occlusions in CommonRoad for the validation. Liu et al. [19] address specification-compliant motion planning for AD vehicles based on set-based reachability analysis with automata-based model checking. The effectiveness of the methods is demonstrated with scenarios from the CommonRoad benchmark suite. In comparison, this study emphasises FMs' strong support for RL. The MC-based model pre-analysis and reward automata as well as the corresponding verification are the first attempts. The support of continuous states and actions, and symbolic and statistical model checking in one model are also profoundly beneficial for deepening users' understanding of the target system.

Another usability barrier of FMs in AD systems is the steep learning curve of the mathematics-based techniques. Researchers have developed tools to overcome this barrier, such as Kronos [8], LTSim [7], and SpaceEx [9]. Bersani *et al.* [6] present PuRSUE (Planner for RobotS in Uncontrollable Environments), which supports users to configure their robotic applications and automatically generate their controllers by using UPPAAL. Gu et al. [11] develop a tool called *MALTA*, which uses UPPAAL as a backend mission planner for AD vehicles under complex road conditions. These tools mainly suffer from a common problem: state-space explosion. Alur et al. [3] propose a compositional method for synthesizing reactive controllers satisfying Linear Temporal Logic specifications for multi-agent systems. Muhammad et al. [20] also use the concept of compositional planning for synthesising energy-optimal motion plans. The new model templates allow different periods for decision-making and sensing, which greatly eases the computational effort for synthesising safety shields and learning. The experiment results show that the new method performs better than the first version of CommonUppRoad [13], which demonstrates the improvement of the new model templates.

7 Conclusion

This paper demonstrates how model checking (MC) can strengthen reinforcement learning (RL) in the domain of autonomous driving (AD). This study is built upon CommonUppRoad, a platform combining CommonRoad and UPPAAL. The author proposes new model templates for AD vehicles, driving scenarios, and reward automata. The new model templates contain discrete and continuous components and support symbolic and statistical model checking. Thanks to the new features, the new version of CommonUppRoad proposes model pre-analysis prior to RL and reward automata for designing reward functions. The Model pre-analysis can help RL designers find bugs in sensor accuracy and determine period lengths, which are imperative for RL. Reward automata are greatly beneficial for multi-objective RL. The experiments demonstrate the necessity of model pre-analysis and the profoundly improved performance of RL with safety shields and reward automata.

One of the future works is investigating the possibility of breaking soft rules to achieve important objectives. How reward automata can help in this setting is unknown. Another direction is scenario generation and critical scenario identification. A combination of MC and RL would be greatly helpful in discovering collision scenarios in a huge database of scenarios.

Acknowledgments

The author acknowledges the support of the Swedish Knowledge Foundation via the project SATISFIES - Holistic Synthesis and Verification for Safe and Secure Autonomous Vehicles, grant nr. 20230047.

References

[1] Ahmed Abouelazm, Jonas Michel & J Marius Zoellner (2024): *A Review of Reward Functions for Reinforcement Learning in the context of Autonomous Driving*. arXiv preprint arXiv:2405.01440, doi:10.48550/arXiv.2405.01440.

[2] Matthias Althoff, Markus Koschi & Stefanie Manzinger (2017): *CommonRoad: Composable benchmarks for motion planning on roads*. In: *2017 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, pp. 719–726, doi:10.1109/IVS.2017.7995802.

[3] Rajeev Alur, Salar Moarref & Ufuk Topcu (2018): *Compositional and symbolic synthesis of reactive controllers for multi-agent systems*. Information and Computation 261, pp. 616–633, doi:10.1016/j.ic.2018.02.021.

[4] BBC (September 16th, 2020): *Uber's self-driving operator charged over fatal crash*. `https://www.bbc.com/news/technology-54175359`.

[5] Johan Bengtsson & Wang Yi (2003): *Timed automata: Semantics, algorithms and tools*. In: *Advanced Course on Petri Nets*, Springer, pp. 87–124, doi:10.1007/978-3-540-27755-2_3.

[6] Marcello M Bersani, Matteo Soldo, Claudio Menghi, Patrizio Pelliccione & Matteo Rossi (2020): *PuRSUE-from specification of robotic environments to synthesis of controllers*. Formal Aspects of Computing 32(2), pp. 187–227, doi:10.1007/s00165-020-00509-0. Springer.

[7] Stefan Blom, Jaco van de Pol & Michael Weber (2010): *LTSmin: Distributed and symbolic reachability*. In: *International Conference on Computer Aided Verification*, Springer, doi:10.1007/978-3-642-14295-6_31.

[8] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis & Sergio Yovine (1998): *Kronos: A model-checking tool for real-time systems*. In: *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer, doi:10.1007/BFb0055357.

[9] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang & Oded Maler (2011): *SpaceEx: Scalable verification of hybrid systems*. In: *International Conference on Computer Aided Verification*, Springer, doi:10.1007/978-3-642-22110-1_30.

[10] Kunal Garg, Songyuan Zhang, Oswin So, Charles Dawson & Chuchu Fan (2024): *Learning safe control for multi-robot systems: Methods, verification, and open challenges*. *Annual Reviews in Control* 57, p. 100948, doi:10.1016/j.arcontrol.2024.100948.

[11] Rong Gu, Eduard Baranov, Afshin Ameri, Cristina Seceleanu, Eduard Paul Enoiu, Baran Cürüklü, Axel Legay & Kristina Lundqvist (2024): *Synthesis and Verification of Mission Plans for Multiple Autonomous Agents under Complex Road Conditions*. *ACM Trans. Softw. Eng. Methodol.* 33(7), doi:10.1145/3672445.

[12] Rong Gu, Peter G Jensen, Cristina Seceleanu, Eduard Enoiu & Kristina Lundqvist (2022): *Correctness-guaranteed strategy synthesis and compression for multi-agent autonomous systems*. *Science of Computer Programming* 224, p. 102894, doi:10.1016/j.scico.2022.102894.

[13] Rong Gu, Kaige Tan, Andreas Holck Høeg-Petersen, Lei Feng & Kim Guldstrand Larsen (2024): *CommonUppRoad: A Framework of Formal Modelling, Verifying, Learning, and Visualisation of Autonomous Vehicles*. Available at `https://arxiv.org/abs/2408.01093`.

[14] Thomas A Henzinger, Peter W Kopke, Anuj Puri & Pravin Varaiya (1995): *What's decidable about hybrid automata?* In: *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pp. 373–382, doi:10.1145/225058.225162.

[15] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano & Sheila A McIlraith (2022): *Reward machines: Exploiting reward function structure in reinforcement learning*. *Journal of Artificial Intelligence Research* 73, pp. 173–208, doi:10.1613/jair.1.12440.

[16] Franjo Ivančić, Malay K Ganai, Sriram Sankaranarayanan & Aarti Gupta (2010): *Numerical stability analysis of floating-point computations using software model checking*. In: *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, IEEE, pp. 49–58, doi:10.1109/MEMCOD.2010.5558622.

[17] Shivesh Khaitan & John M Dolan (2022): *State dropout-based curriculum reinforcement learning for self-driving at unsignalized intersections*. In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp. 12219–12224, doi:10.1109/IROS47612.2022.9981109.

[18] Kim G Larsen, Paul Pettersson & Wang Yi (1997): *UPPAAL in a nutshell*. *International journal on software tools for technology transfer* 1, pp. 134–152, doi:10.1007/s100090050010.

[19] Irani Liu & Matthias Althoff (2023): *Specification-compliant driving corridors for motion planning of automated vehicles*. *IEEE Transactions on Intelligent Vehicles*, doi:10.1109/TIV.2023.3289580.

[20] Naeem Muhammad, Gu Rong, Seceleanu Cristina, Guldstrand Larsen Kim, Nielsen Brian & Albano Michele (2024): *Energy-Optimized Motion Planning for Autonomous Vehicles Using UPPAAL Stratego*. In: *The 18th International Symposium on Theoretical Aspects of Software Engineering*, Springer, doi:10.1007/978-3-031-64626-3_21.

[21] Muhammad Naeem, Rong Gu, Cristina Seceleanu, Kim Guldstrand Larsen, Brian Nielsen & Michele Albano (2024): *Energy-Efficient Motion Planning for Autonomous Vehicles Using Uppaal Stratego*. In: *International Symposium on Theoretical Aspects of Software Engineering*, Springer, pp. 356–373, doi:10.1007/978-3-031-64626-3_21.

[22] Maximilian Naumann, Hendrik Konigshof, Martin Lauer & Christoph Stiller (2019): *Safe but not over-cautious motion planning under occlusions and limited sensor range*. In: *2019 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, pp. 140–145, doi:10.1109/IVS.2019.8814251.

[23] The Washington Post (June 10th, 2023): *17 fatalities, 736 crashes: The shocking toll of Tesla's Autopilot*. `https://www.washingtonpost.com/technology/2023/06/10/tesla-autopilot-crashes-elon-musk/`.

[24] KKS funded Project (2015 - 2023): *DPAC - Dependable Platforms for Autonomous systems and Control*. `https://www.es.mdh.se/dpac/`.

[25] KKS funded Project (2024 - 2026): *Holistic Synthesis and Verification for Safe and Secure Autonomous Vehicles*. https://www.mdu.se/en/malardalen-university/research/research-projects/holistic-synthesis-and-verification-for-safe-and-secure-autonomous-vehicles?

[26] José Manuel Gaspar Sánchez, Truls Nyberg, Christian Pek, Jana Tumova & Martin Törngren (2022): *Foresee the unseen: Sequential reasoning about hidden obstacles for safe driving*. In: *2022 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, pp. 255–264, doi:10.1109/IV51971.2022.9827171.

[27] Qisong Yang, Thiago D Simão, Nils Jansen, Simon H Tindemans & Matthijs TJ Spaan (2023): *Reinforcement Learning by Guided Safe Exploration*. *arXiv preprint arXiv:2307.14316*, doi:10.3233/FAIA230598.

[28] Xinhai Zhang, Jianbo Tao, Kaige Tan, Martin Törngren, Jose Manuel Gaspar Sanchez, Muhammad Rusyadi Ramli, Xin Tao, Magnus Gyllenhammar, Franz Wotawa, Naveen Mohan et al. (2022): *Finding critical scenarios for automated driving systems: A systematic mapping study*. *IEEE Transactions on Software Engineering* 49(3), pp. 991–1026, doi:10.1109/TSE.2022.3170122/mm1.

Creating a Formally Verified Neural Network for Autonomous Navigation: An Experience Report*

Syed Ali Asadullah Bukhari Thomas Flinkow Medet Inkarbekov
Barak A. Pearlmutter Rosemary Monahan
Department of Computer Science, Maynooth University, Maynooth, Ireland
{ali.bukhari,thomas.flinkow,medet.inkarbekov,rosemary.monahan}@mu.ie barak@pearlmutter.net

The increased reliance of self-driving vehicles on neural networks opens up the challenge of their verification. In this paper we present an experience report, describing a case study which we undertook to explore the design and training of a neural network on a custom dataset for vision-based autonomous navigation. We are particularly interested in the use of machine learning with differentiable logics to obtain networks satisfying basic safety properties by design, guaranteeing the behaviour of the neural network after training. We motivate the choice of a suitable neural network verifier for our purposes and report our observations on the use of neural network verifiers for self-driving systems.

1 Introduction and Motivation

A shift from a purely rule-based software to a learning-based approach for control of autonomous driving systems is evident in recent years [26, 21]. This change can be attributed primarily to the advances in Deep Neural Networks (DNNs) and their improved ability to handle the complexity of environments compared to conventional autonomous navigation methods. Moreover, the availability of hardware accelerators and GPUs on edge devices at low cost and power has also progressed the use of DNNs for these systems. In addition, the computing ability coupled with the presence of various on-device sensors, such as Lidar and cameras, makes it possible to achieve the task of controlling vehicles without the need of human intervention [38]. In particular, vision-based systems have been successful for this purpose as on-board cameras can be used to help auto navigation as well as providing real time video monitoring, as is often required in these systems. Some notable examples of vision-based systems employing DNNs include obstacle avoidance, path following, and object detection [27, 47].

The increased reliance of self-driving vehicles on neural networks opens up the challenge of their verification. The safety-critical nature of these vehicles calls for their complete verification before they are put to use in a real environment. In general, the verification of any safety-critical component in any system is viewed as a crucial task, as an overlooked corner case may lead to a catastrophic condition or an irreparable loss. However, the presence of neural networks in a system makes the verification task further challenging [28].

Many efforts have been made to verify neural networks, with verifiers like VNN [40], Marabou [19] and α, β-CROWN [46, 43, 44, 42, 45, 20, 29] (the winner of the recent VNN-COMP neural network verification competitions [2, 23, 3]). These verifiers have shown promising results for verification of robustness properties of neural network models used for objects classification. Robustness, in the case of image classification, can be described as the ability of the neural network to retain the prediction label of an input image in response to a small change in the input image.

*This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant number 20/FFP-P/8853.

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 178–190, doi:10.4204/EPTCS.411.12
© S. A. A. Bukhari, T. Flinkow, M. Inkarbekov, B. A. Pearlmutter & R. Monahan

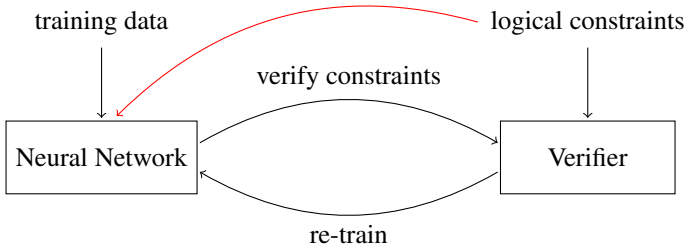


Figure 1: Differentiable logics allow for the use of logical constraints during training by translating them into additional loss terms. Note that in this paper, we do not use continuous verification, that is, we do not re-train the network after the verification; instead, we only try to evaluate what influence training with logical constraints has on the verification afterwards.

While the aforementioned verifiers usually assume a trained network with fixed weights, another area of research is the design of correct-by-construction neural networks. One approach in this direction is *differentiable logics* [9, 41, 35]. The fundamental principle of machine learning is to minimise a so-called *loss function* which indicates how wrong the network output is compared to the desired output. Differentiable logics are used to transform a logical constraint ϕ into an additional logical loss term $\mathcal{L}_\phi$ to minimise when learning. This loss term is in addition to the standard loss (such as mean-squared error loss $\mathcal{L}_{\text{MSE}}$). Therefore, the optimisation objective is of the form $\mathcal{L} = \mathcal{L}_{\text{MSE}} + \lambda \mathcal{L}_\phi$. It is important to balance the different loss terms, and for this we employ gradient normalisation (GradNorm) [6], an adaptive loss balancing approach which treats λ as an additional learnable parameter $\lambda(t)$. GradNorm has been shown to outperform grid search, the conventional algorithm used in machine learning for hyperparameter tuning.

Figure 1 shows the standard loop of training a network, verifying it, and re-training the network if necessary. Differentiable logics allow the integration of constraints as additional loss terms into the training process. Multiple mappings have been defined in the literature to translate logical constraints into loss terms, which allow for real-valued truth values, but are also differentiable almost everywhere for use with standard gradient-based methods. Prominent examples include DL2 [9] and fuzzy logic based mappings [41, 35]. In our experiments, we use the Gödel fuzzy logic $[\![\cdot]\!] : [0,1] \to [0,1]$ which translates as follows:

- conjunction as $[\![x \wedge y]\!] = \min(x,y)$,
- disjunction as $[\![x \vee y]\!] = \max(x,y)$, and
- implication as $[\![x \to y]\!] = \begin{cases} 1, & x < y \\ y, & \text{else.} \end{cases}$

We chose to use this logic translation as it provides a desirable property for the translation process i.e. that the operators have strong derivatives almost everywhere. Previous experimental results [10] suggest that the choice of a logic does not have a major impact.

In this paper we present an experience report, describing a case study which we undertook to explore the following challenges: (1) design and training of a neural network on a custom dataset for vision-based autonomous navigation, (2) use of machine learning with differentiable logics to obtain networks satisfying basic safety properties, (3) obtaining formal guarantees with formal verification of the neural network after training. We motivate the choice of a suitable neural network verifier for our purposes and report our observations on the use of neural network verifiers for self-driving systems. As can be seen in Section 2,

to the best of our knowledge, such a setup of using formal methods at multiple stages of the machine learning pipeline has not been published before.

2 Related Work

The verification of autonomous driving systems, especially on edge devices, remains a largely under-explored area. Despite the rapid progress in autonomous vehicle technology [25], few studies have addressed the verification challenges in resource-constrained environments. This section provides an overview of some foundational contributions in this domain.

Sun et al. [37] present a framework for formal verification of safety properties in autonomous systems controlled by neural networks. Their approach focuses on ensuring that a robot can safely navigate environments with polyhedral obstacles by constructing a finite state abstraction of the system and applying reachability analysis. The introduction of imaging-adapted partitions in their work is particularly relevant to the verification of robotic cars, as it simplifies the modelling of LiDAR-based perception, making the verification process more manageable.

Habeeb et al. [13] develop a procedure for the safety verification of camera-based autonomous systems, which includes a falsification approach that collects unsafe trajectories to assist in retraining neural network controllers. Their concept of image-invariant regions is noteworthy, as it enables reasoning about trajectories at the level of regions rather than individual positions, potentially reducing the complexity of the verification process for systems relying on visual inputs.

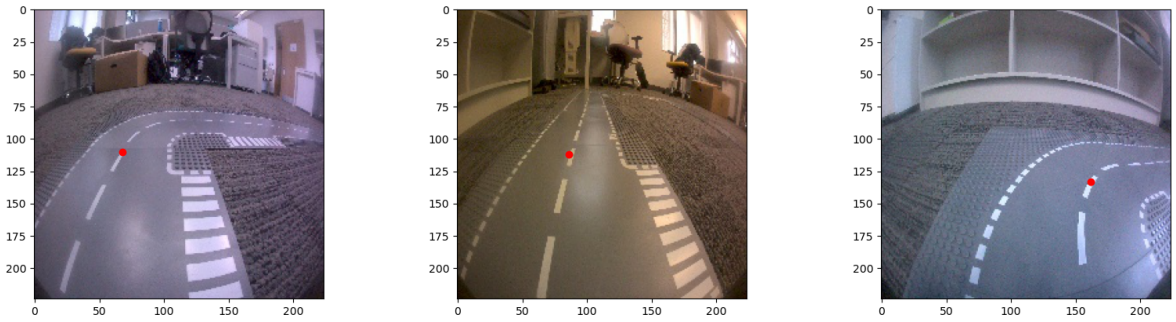
Cleaveland et al. [7] propose a risk verification framework for stochastic systems with neural network controllers, focusing on estimating the risk of failure under stochastic conditions. Their work is significant in handling the verification of systems where environmental perturbations might occur, which is critical for autonomous vehicles operating in unpredictable real-world environments. The empirical validation of their framework using autonomous vehicles demonstrates its applicability to scenarios where robustness to environmental changes is essential.

Ivanov et al. [15] present a benchmark for assessing the scalability of verification tools in the context of an autonomous racing car controlled by a neural network. Their work highlights the challenges in verifying systems with high-dimensional inputs, such as LiDAR data, and underscores the importance of addressing the gap between simulated and real-world performance, known as the sim2real gap, in verification efforts. The limitations identified in their study, particularly regarding sensor faults in real-world environments, emphasise the need for more scalable and robust verification methods.

While these approaches address key verification aspects, they fall short in addressing the specific requirements for edge devices used in autonomous driving systems. Further research is necessary to bridge this gap and provide effective solutions for resource-constrained environments.

3 Case Study: A Verified Neural Network for Autonomous Navigation

The goal of this case study is to build a formally verified regression neural network that detects the centre of the track on which the vehicle is travelling. We train the network in two different ways. First, we train in a standard manner, training only on the dataset. Our second approach aims to achieve a correct-by-construction network by training using the constraints that we want the network to satisfy after training. In this approach we use differentiable logics.



(a) label = 68 ,110 (b) label = 86, 112 (c) label = 162, 133

Figure 2: Samples from the LEGO road dataset showing various lighting conditions and track configurations. The red dots located at label coordinates show the centre of the path to be followed in the frame.

3.1 Experimental Setup

For demonstration purposes, we have used the JetBot kit available at [36]. JetBot [16] is an open-source AI robotic vehicle based on NVIDIA Jetson Nano [24]. Jetson Nano is an edge device equipped with GPUs to train and run neural networks at a lower power range of 5 W to 10 W. It can be easily interfaced with several sensor modules including a camera, making it suitable for demonstrating a vision-based autonomous navigation application. The JetBot kit features a Leopard Imaging 136 FOV Camera [14] (3280×2464 pixels).

3.2 Data Collection

Various neural network models and training data are readily available for a range of computer vision tasks, thus providing a possible foundation for development of vision-based autonomous navigation systems. Additionally, there are multiple open source architectures for autonomous car navigation such as Openpilot [8]. A lane-keeping dataset for autonomous plane taxiing was presented in [11], where they also verified the neural network component. In this case study, we generate our own dataset as our focus is on a meaningful prototype, rather than developing a state-of-the-art lane keeping autonomous vehicle. We build upon the example provided with JetBot, available at [17].

More specifically, we have built a reference track using readily available Lego road plates (with a centred line). The data is collected by placing the robotic car at various positions on the tracks and taking images with the onboard camera facing down on the track. The images are labelled with x and y coordinates indicating the centre of the path to be followed in the frame. These coordinates are used to calculate the driving parameters for the JetBot motor. The dataset consists of 385 images stored in a resolution of 224×224 pixels. A few samples from the dataset are shown in Fig. 2. The dataset is publicly available at `https://github.com/tflinkow/fmas2024`

3.3 Neural Network Architecture

Scalability is an important issue for neural network verifiers due to the high-dimensional inputs, the large number of neurons in the network, and the use of non-linear activation functions. Hence, in order to make verification tractable, finding a sufficiently small neural network architecture is essential in this case study. For that reason, we first chose to downscale the original 224×224 pixel RGB images to 112×112 pixels

Table 1: Summary of the proposed neural network.

Network Layer	Input Shape	Output Shape	Kernel (if applicable)	No. of Trainable Parameters
	Size × *Size* × *Channel* × *Batch*		Size, Stride, Padding	
Conv2D	$112 \times 112 \times 1 \times 1$	$112 \times 112 \times 1 \times 1$	3×3, 1×1, 1×1	10
ReLU	$112 \times 112 \times 1 \times 1$	$112 \times 112 \times 1 \times 1$	—	—
MaxPool2D	$112 \times 112 \times 1 \times 1$	$56 \times 56 \times 1 \times 1$	2×2, 2×2, none	—
Conv2D	$56 \times 56 \times 1 \times 1$	$56 \times 56 \times 1 \times 1$	3×3, 1×1, 1×1	10
ReLU	$56 \times 56 \times 1 \times 1$	$56 \times 56 \times 1 \times 1$	—	—
MaxPool2D	$56 \times 56 \times 1 \times 1$	$28 \times 28 \times 1 \times 1$	2×2, 2×2, none	—
Linear	784×1	256×1	—	200, 960
ReLU	256×1	256×1	—	—
Linear	256×1	128×1	—	32, 896
ReLU	128×1	128×1	—	—
Linear	128×1	2×1	—	258
Tanh	2×1	2×1	—	—

and convert them to grey scale before passing them to the network, thus allowing for a noticeably smaller network architecture.

The neural network we propose consists of two convolutional layers with ReLU activations (each followed by a max-pooling layer) followed by three fully connected layers, the first two with ReLU activations, and the last layer followed by a tanh activation. Further details about each layer and other network parameters are listed in Table 1. The network contains 234,136 parameters. Our proposed architecture is shown in Fig. 3.

In the following, we consider our network to approximate the function

$$\mathcal{N} : \mathbb{R}^{112\times 112} \to \mathbb{R}^2, \tag{1}$$

where the inputs are images of size 112×112, and the outputs are the x, y-coordinates of the centre of the track in the image (according to the image label).

3.4 Verification Properties

The property we wish to verify is a standard local robustness property [5] for neural networks, formally defined as

$$\mathsf{Robustness}(\boldsymbol{x}_0, \varepsilon, \delta) := \forall \boldsymbol{x}. \, ||\boldsymbol{x}_0 - \boldsymbol{x}||_\infty \leq \varepsilon \implies ||\mathcal{N}(\boldsymbol{x}_0) - \mathcal{N}(\boldsymbol{x})||_\infty \leq \delta. \tag{2}$$

This property checks that for slight perturbations $\boldsymbol{x}$ within some bound ε of a given input image $\boldsymbol{x}_0$, the neural network $\mathcal{N}$ should give roughly the same output, i.e., the measure of difference between the network's outputs $\mathcal{N}(\boldsymbol{x}_0)$ and $\mathcal{N}(\boldsymbol{x})$ should be within an acceptable threshold δ.

As explained in [9], learning properties of the form $\forall \boldsymbol{x}. \, ||\boldsymbol{x}_0 - \boldsymbol{x}||_\infty \leq \varepsilon \implies \phi$ can be approximated by finding a counterexample (i.e., the worst perturbation) $\boldsymbol{x}^*$ for ϕ within the ε-neighbourhood of $\boldsymbol{x}_0$ with Projected Gradient Descent (PGD) [22] and using it in training. This approach is similar to adversarial training. However the perturbation found using PGD is only used as a counterexample for the logical constraint, and not for calculating the mean squared error loss term. Thus in turn adversarial examples do

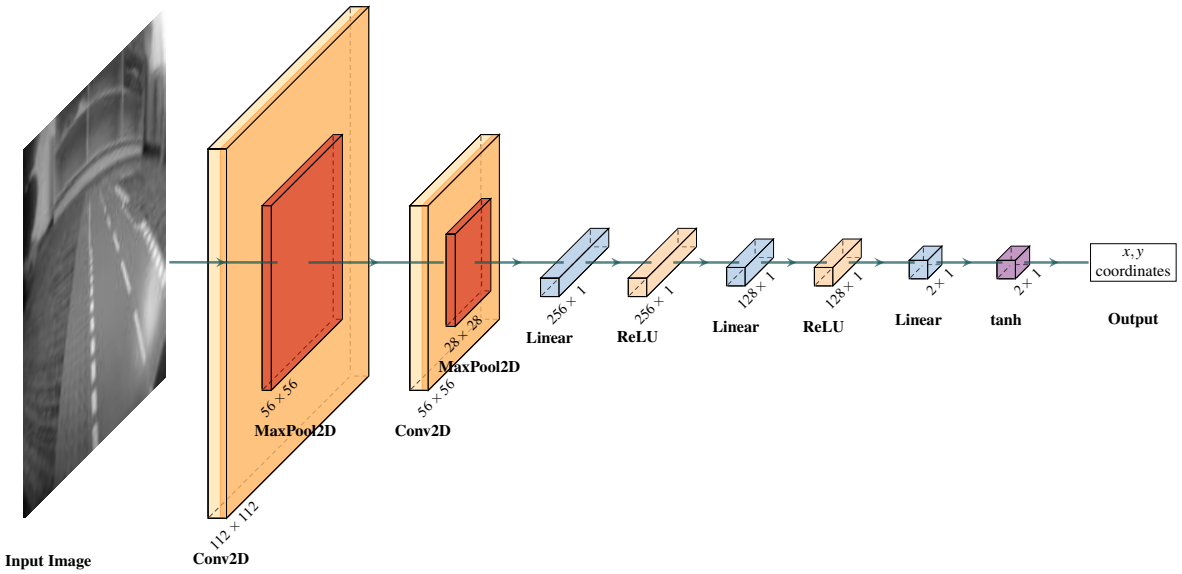


Figure 3: The proposed neural network architecture for extracting the path centre in an input image of size 112×112 pixels.

not improve the performance of the network; they are only used to make the network satisfy the constraint more. The code we use to achieve this in training is based on [10].

4 Results and Observations

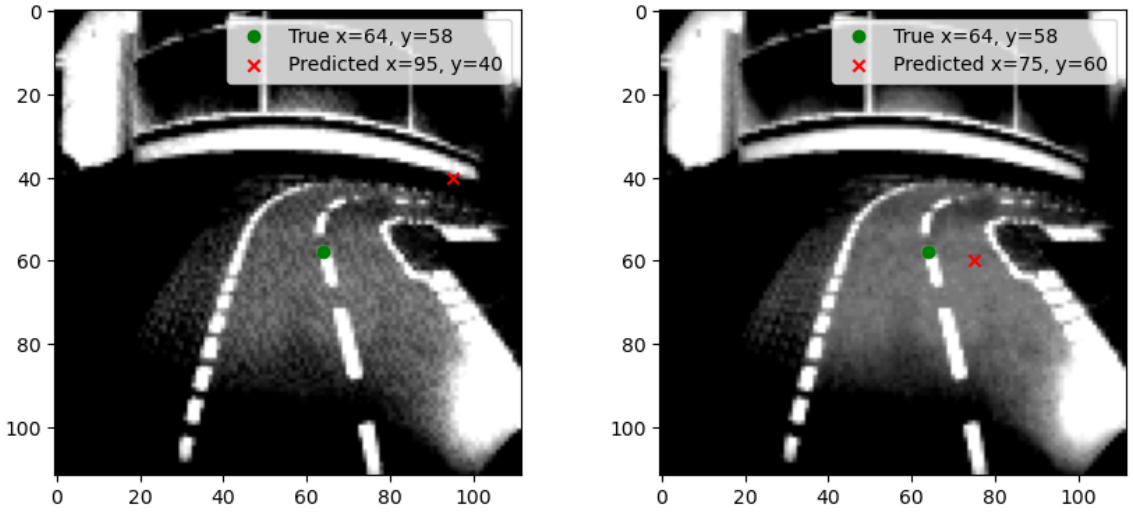
We performed two variations of training of the suggested neural network on our collected data. Firstly, we trained the network in a standard manner (called *vanilla* in the following) using only our collected data. Secondly, we trained our network on the collected data but added constraints (called *constrained-training* in the following). Both training runs were executed for 100 *epochs* with a *batch size* of 16.

4.1 Performance

To evaluate the performance, we look at prediction loss (i.e. mean squared error) and constraint accuracy (i.e. the number of times the constraint was satisfied out of all the predictions). Figs. 4 and 5 illustrate that including constraints in the training process leads to improved performance on adversarial examples, as well as drastically improving constraint accuracy. However, as mentioned in [12], training with constraints does not guarantee their satisfaction. Instead, we aimed to obtain formal guarantees that both networks verify the constraints after training.

4.2 Verification of Robustness

We used α, β-CROWN to verify that the network satisfies the robustness property in Eq. (2). However, we found that the tool was not suitable for the verification of regression tasks (as opposed to classification tasks) without modification. We also used DNNV [30], a framework providing a unified interface



(a) Vanilla (b) Constrained-training

Figure 4: Comparison of predictions of models trained without (vanilla) and with logical constraints (constrained-training) on an adversarial example in epoch 45 of 100.

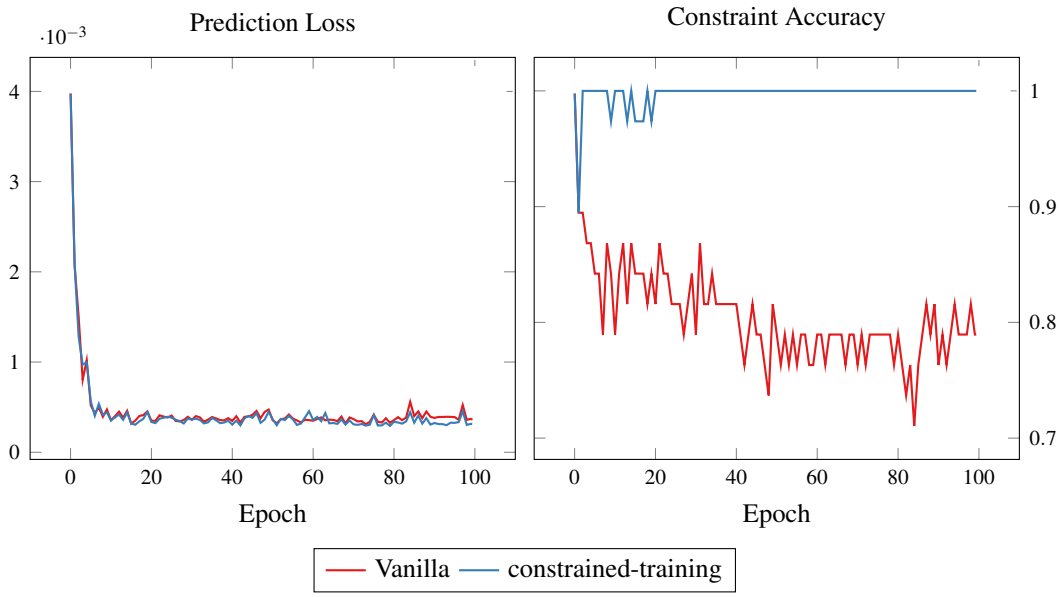


Figure 5: The prediction loss (i.e., mean squared error; lower is better) and constraint accuracy (higher is better) for a model trained only on data (vanilla) and a model trained on both data and logical constraints with differentiable logics (constrained-training).

Listing 1: The robustness property from Eq. (2) specified in DNNV's property specification language.

```
from dnnv.properties import *

N = Network("N")
x = Image("data/image_0.npy")

epsilon = Parameter("epsilon", float, default=(48. / 255))
delta = Parameter("delta", float, default=0.1)

Forall(
    x_,
    Implies(
        ((x - epsilon) <= denormalise(x_) <= (x + epsilon)),
        (abs(N(x_)[0][0] - N(x)[0][0]) <= delta) &
        (abs(N(x_)[0][1] - N(x)[0][1]) <= delta)
    ),
)
```

Listing 2: Output of DNNV for verification of robustness property.

```
dnnv.verifiers.bab
  result: BabTranslatorError(Unsupported computation graph detected)
  time: 0.3495

dnnv.verifiers.eran
  result: unknown
  time: 4.9732

dnnv.verifiers.nnenum
  result: NnenumError(Return code: 1)
  time: 0.7476

dnnv.verifiers.reluplex
  result: ReluplexTranslatorError(Unsupported computation graph detected)
  time: 0.3622
```

to interact with verifiers such as Reluplex [18], Marabou [19], BaB [4], ERAN [32, 33, 34, 31], and nnenum [1]. DNNV provides a Python-based domain-specific language for expressing properties. Our translation of the constraint Eq. (2) is shown in Listing 1. DNNV was not straightforward to install, requiring older versions of Python along with outdated requirements for packages such as numpy. The only verifiers that we were able to install successfully within DNNV were BAB, ERAN, Reluplex and nnenum. Verifying the property shown in Listing 1 led to an inconclusive result for ERAN, whereas Reluplex and BAB were not able to attempt verification due to unsupported operations, and nnenum returned a generic error. The tool's output for these scenarios is shown in Listing 2.

We also used the Matlab Toolbox for Neural Network Verification (NNV) [40] for verification of the robustness property. NNV provides a relatively simplier interface that accepts several neural network formats. The tool has an `estimateNetworkOutputBounds` function that estimates lower and upper output bounds of the network, supporting a change in input within the specified bounds. For a small perturbation ε in input X, the input bounds are $X-\varepsilon$ and $X+\varepsilon$. For the input image in Fig. 6, Table 2

Figure 6: An image (converted to grey scale) from the LEGO dataset taken as a running example.

Table 2: Lower and upper output bounds of the proposed neural network computed by NNV against perturbations in the normalised input image from Figure 6.

Input Perturbations ε (Normalised Image)	**Network Configuration**			
	Vanilla		Constrained-training	
	x	y	x	y
0.001	[9 – 107]	[19 – 97]	[8 – 107]	[15 – 97]
0.01	[1 – 112]	[1 – 112]	[1 – 112]	[1 – 112]

shows the lower and upper bounds (x, y coordinates) estimated by NNV against the perturbations in the normalised input for different network training configurations. It can be observed from Table 2 that the `estimateNetworkOutputBounds` function does return any tighter bounds for the network's output, showing the sensitivity of the network to input changes.

4.3 Lessons Learned

Supporting formal guarantees for neural networks appears qualitatively different from traditional formal verification. While formal verification techniques for neural networks are general enough to allow the verification of various properties over certain regions of the input space, in practice, finding these regions can be problematic—for example, requiring a network to be robust usually applies only for specific images, not all possible images. As this is difficult to specify for all but the most simple, low-dimensional problems (such as ACAS Xu [18]), verification is usually limited to verifying local properties at input points contained in the available data. This has important implications for the use of the network in formally verified systems—such as the lack of real guarantees for unseen data.

The architecture of the neural network has extensive consequences for the whole pipeline, from affecting the training time with differentiable logics, to making verification difficult or potentially impossible. As explained in [39], max-pooling layers are typically too complicated for verification. The same applies to tanh layers, as many tools support only ReLU activation functions, fully connected, and convolutional layers, which explains why Reluplex and BaB failed to verify our property, as seen in Listing 2.

Regarding verification tools, the effort in installation, interfacing tools with the network to be verified, and getting the tool running in terms of computational resources is substantial. Tools were difficult to install due to their (sometimes outdated) dependencies, and the data and network often required conversion to an acceptable format to be compatible with the tools expectations. We note also that neural network verification tools focus primarily on networks performing classification rather than regression, as evidenced by the large number of examples provided for the earlier case.

Lastly, considering the autonomous navigation example in our case study and the dataset design, we would like to explore using further labelled data, e.g., the left and right edges of the track could be included in addition to the centre of the track. This opens up more interesting logical constraints to use in training.

5 Conclusion

In this paper we presented our experience of creating a formally verified neural network for autonomous navigation. While we gained many insights into the currently available tools and approaches, the challenge of creating verification-friendly neural networks (in general and) for autonomous driving systems is still an open problem, requiring expertise from both the formal verification and the neural network communities.

We investigated the design and training of a neural network on a custom dataset for vision-based autonomous navigation in a fashion that integrates standard training methods and logic-based formal methods. In particular, we used differentiable logics to constrain training to yield networks satisfying safety properties like robustness. This approach can be integrated into a pipeline in which the networks thus trained are then checked for compliance using formal verification of the post-training neural network.

References

[1] Stanley Bak (2021): *Nnenum: Verification of ReLU Neural Networks with Optimized Abstraction Refinement*. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz & Ivan Perez, editors: *NASA Formal Methods*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 19–36, doi:10.1007/978-3-030-76384-8_2.

[2] Stanley Bak, Changliu Liu & Taylor Johnson (2021): *The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results*, doi:10.48550/arXiv.2109.00498. arXiv:2109.00498.

[3] Christopher Brix, Stanley Bak, Changliu Liu & Taylor T. Johnson (2023): *The Fourth International Verification of Neural Networks Competition (VNN-COMP 2023): Summary and Results*, doi:10.48550/arXiv.2312.16760. arXiv:2312.16760.

[4] Rudy R. Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli & Pawan K. Mudigonda (2018): *A Unified View of Piecewise Linear Neural Network Verification*. In: *Advances in Neural Information Processing Systems*, 31, Curran Associates, Inc.

[5] Marco Casadio, Ekaterina Komendantskaya, Matthew L. Daggitt, Wen Kokke, Guy Katz, Guy Amir & Idan Refaeli (2022): *Neural Network Robustness as a Verification Property: A Principled Case Study*. In Sharon Shoham & Yakir Vizel, editors: *Computer Aided Verification*, Lecture Notes in Computer Science, Springer International Publishing, pp. 219–231, doi:10.1007/978-3-031-13185-1_11.

[6] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee & Andrew Rabinovich (2018): *GradNorm: Gradient Normalization for Adaptive Loss Balancing in Deep Multitask Networks*. In: *Proceedings of the 35th International Conference on Machine Learning*, PMLR, pp. 794–803.

[7] Matthew Cleaveland, Lars Lindemann, Radoslav Ivanov & George J. Pappas (2022): *Risk verification of stochastic systems with neural network controllers*. *Artificial Intelligence* 313, p. 103782, doi:10.1016/j.artint.2022.103782.

[8] CommaAI (2024): *Commaai/Openpilot*. comma.ai. Available at `https://github.com/commaai/openpilot`.

[9] Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang & Martin Vechev (2019): *DL2: Training and Querying Neural Networks with Logic*. In: *Proceedings of the 36th International Conference on Machine Learning*, PMLR, pp. 1931–1941.

[10] Thomas Flinkow, Barak A. Pearlmutter & Rosemary Monahan (2024): *Comparing Differentiable Logics for Learning with Logical Constraints*. arXiv:2407.03847. (under review).

[11] Daniel J. Fremont, Johnathan Chiu, Dragos D. Margineantu, Denis Osipychev & Sanjit A. Seshia (2020): *Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VerifAI*. In: *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*, Springer-Verlag, Berlin, Heidelberg, pp. 122–134, doi:10.1007/978-3-030-53288-8_6.

[12] Eleonora Giunchiglia, Mihaela Catalina Stoian & Thomas Lukasiewicz (2022): *Deep Learning with Logical Constraints*. In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, International Joint Conferences on Artificial Intelligence Organization, Vienna, Austria, pp. 5478–5485, doi:10.24963/ijcai.2022/767.

[13] P. Habeeb, Nabarun Deka, Deepak D'Souza, Kamal Lodaya & Pavithra Prabhakar (2023): *Verification of Camera-Based Autonomous Systems*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42(10), pp. 3450–3463, doi:10.1109/TCAD.2023.3240131.

[14] Leopard Imaging Inc. (2024): *LI-IMX219-MIPI-FF-NANO-H136—Leopard Imaging Inc.* Available at `https://leopardimaging.com/product/platform-partners/nvidia/nvidia-jetson-nano/nano-mipi-camera-kits/li-imx219-mipi-ff-nano/li-imx219-mipi-ff-nano-h136/`.

[15] Radoslav Ivanov, Taylor J. Carpenter, James Weimer, Rajeev Alur, George J. Pappas & Insup Lee (2020): *Case study: verifying the safety of an autonomous racing car with a neural network controller*. In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, HSCC '20, Association for Computing Machinery, New York, NY, USA, doi:10.1145/3365365.3382216.

[16] JetBot. (2024): *JetBot*. Available at `https://jetbot.org/master/index.html`.

[17] JetBot (2024): *Road Following—JetBot*. Available at `https://jetbot.org/master/examples/road_following.html`.

[18] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian & Mykel J. Kochenderfer (2017): *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*. In Rupak Majumdar & Viktor Kunčak, editors: *Computer Aided Verification*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 97–117, doi:10.1007/978-3-319-63387-9_5.

[19] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer & Clark Barrett (2019): *The Marabou Framework for Verification and Analysis of Deep Neural Networks*. In Isil Dillig & Serdar Tasiran, editors: *Computer Aided Verification*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 443–452, doi:10.1007/978-3-030-25540-4_26.

[20] Suhas Kotha, Christopher Brix, J. Zico Kolter, Krishnamurthy Dvijotham & Huan Zhang (2023): *Provably Bounding Neural Network Preimages*. In A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt & S. Levine, editors: *Advances in Neural Information Processing Systems*, 36, Curran Associates, Inc., pp. 80270–80290. Available at `https://proceedings.neurips.cc/paper_files/paper/2023/file/fe061ec0ae03c5cf5b5323a2b9121bfd-Paper-Conference.pdf`.

[21] Yifang Ma, Zhenyu Wang, Hong Yang & Lin Yang (2020): *Artificial intelligence applications in the development of autonomous vehicles: A survey*. *IEEE/CAA Journal of Automatica Sinica* 7(2), pp. 315–329, doi:10.1109/JAS.2020.1003021.

[22] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras & Adrian Vladu (2018): *Towards Deep Learning Models Resistant to Adversarial Attacks*. In: *International Conference on Learning Representations*. Available at `https://openreview.net/forum?id=rJzIBfZAb`.

[23] Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu & Taylor T. Johnson (2022): *The Third International Verification of Neural Networks Competition (VNN-COMP 2022): Summary and Results*, doi:10.48550/arXiv.2212.10376. arXiv:2212.10376.

[24] NVIDIA (2024): *Jetson Nano Brings the Power of Modern AI to Edge Devices | NVIDIA*. Available at https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/.

[25] Budi Padmaja, CH VKNSN Moorthy, N Venkateswarulu & Myneni Madhu Bala (2023): *Exploration of issues, challenges and latest developments in autonomous cars*. Journal of Big Data 10(1), p. 61, doi:10.1186/s40537-023-00701-y.

[26] Dean A. Pomerleau (1988): *ALVINN: An Autonomous Land Vehicle in a Neural Network*. In: Advances in Neural Information Processing Systems (NIPS), 1, Morgan-Kaufmann, pp. 305–15. Available at https://proceedings.neurips.cc/paper_files/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf.

[27] Ratheesh Ravindran, Michael J. Santora & Mohsin M. Jamali (2020): *Multi-object detection and tracking, based on DNN, for autonomous vehicles: A review*. IEEE Sensors Journal 21(5), pp. 5668–5677, doi:10.1109/JSEN.2020.3041615.

[28] Sanjit A Seshia, Dorsa Sadigh & S Shankar Sastry (2022): *Toward verified artificial intelligence*. Communications of the ACM 65(7), pp. 46–55, doi:10.1145/3503914.

[29] Zhouxing Shi, Qirui Jin, Zico Kolter, Suman Jana, Cho-Jui Hsieh & Huan Zhang (2024): *Neural Network Verification with Branch-and-Bound for General Nonlinearities*. arXiv preprint arXiv:2405.21063.

[30] David Shriver, Sebastian Elbaum & Matthew B. Dwyer (2021): *DNNV: A Framework for Deep Neural Network Verification*. In Alexandra Silva & K. Rustan M. Leino, editors: Computer Aided Verification, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 137–150, doi:10.1007/978-3-030-81685-8_6.

[31] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel & Martin Vechev (2019): *Beyond the Single Neuron Convex Barrier for Neural Network Certification*. In: Advances in Neural Information Processing Systems, 32, Curran Associates, Inc.

[32] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel & Martin Vechev (2018): *Fast and Effective Robustness Certification*. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, Curran Associates Inc., Red Hook, NY, USA, pp. 10825–10836.

[33] Gagandeep Singh, Timon Gehr, Markus Püschel & Martin Vechev (2018): *Boosting Robustness Certification of Neural Networks*. In: International Conference on Learning Representations.

[34] Gagandeep Singh, Timon Gehr, Markus Püschel & Martin Vechev (2019): *An Abstract Domain for Certifying Neural Networks*. Proceedings of the ACM on Programming Languages 3(POPL), pp. 1–30, doi:10.1145/3290354.

[35] Natalia Ślusarz, Ekaterina Komendantskaya, Matthew Daggitt, Robert Stewart & Kathrin Stark (2023): *Logic of Differentiable Logics: Towards a Uniform Semantics of DL*. In: EPiC Series in Computing, 94, EasyChair, pp. 473–493, doi:10.29007/c1nt.

[36] Sparkfun (2024): *SparkFun JetBot AI Kit v3.0 Powered by Jetson Nano—KIT-18486—SparkFun Electronics*. Available at https://www.sparkfun.com/products/18486.

[37] Xiaowu Sun, Haitham Khedr & Yasser Shoukry (2019): *Formal verification of neural network controlled autonomous systems*. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 147–156.

[38] Sebastian Thrun (2010): *Toward robotic cars*. Commun. ACM 53(4), p. 99–106, doi:10.1145/1721654.1721679.

[39] Hoang-Dung Tran, Stanley Bak, Weiming Xiang & Taylor T. Johnson (2020): *Verification of Deep Convolutional Neural Networks Using ImageStars*. In Shuvendu K. Lahiri & Chao Wang, editors: Computer Aided Verification, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 18–42, doi:10.1007/978-3-030-53288-8_2.

[40] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak & Taylor T. Johnson (2020): *NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems*. In Shuvendu K. Lahiri & Chao Wang, editors:

Computer Aided Verification, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 3–17, doi:10.1007/978-3-030-53288-8_1.

[41] Emile van Krieken, Erman Acar & Frank van Harmelen (2022): *Analyzing Differentiable Fuzzy Logic Operators*. *Artificial Intelligence* 302, p. 103602, doi:10.1016/j.artint.2021.103602. arXiv:2002.06100.

[42] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh & J. Zico Kolter (2021): *Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification*. *Advances in Neural Information Processing Systems* 34.

[43] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin & Cho-Jui Hsieh (2020): *Automatic perturbation analysis for scalable certified robustness and beyond*. *Advances in Neural Information Processing Systems* 33.

[44] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin & Cho-Jui Hsieh (2021): *Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers*. In: *International Conference on Learning Representations*. Available at `https://openreview.net/forum?id=nVZtXBI6LNn`.

[45] Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh & J. Zico Kolter (2022): *General Cutting Planes for Bound-Propagation-Based Neural Network Verification*. *Advances in Neural Information Processing Systems*.

[46] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh & Luca Daniel (2018): *Efficient Neural Network Robustness Certification with General Activation Functions*. In: *Advances in Neural Information Processing Systems*, 31, Curran Associates, Inc.

[47] Jingyuan Zhao, Wenyi Zhao, Bo Deng, Zhenghong Wang, Feng Zhang, Wenxiang Zheng, Wanke Cao, Jinrui Nan, Yubo Lian & Andrew F. Burke (2024): *Autonomous driving system: A comprehensive survey*. *Expert Systems with Applications* 242, p. 122836, doi:10.1016/j.eswa.2023.122836. Available at `https://www.sciencedirect.com/science/article/pii/S0957417423033389`.

Open Challenges in the Formal Verification of Autonomous Driving

Paolo Burgio Angelo Ferrando Marco Villani
University of Modena and Reggio Emilia
Department of Physics, Informatics and Mathematics
Modena, Italy
forename.surname@unimore.it

In the realm of autonomous driving, the development and integration of highly complex and heterogeneous systems are standard practice. Modern vehicles are not monolithic systems; instead, they are composed of diverse hardware components, each running its own software systems. An autonomous vehicle comprises numerous independent components, often developed by different and potentially competing companies. This diversity poses significant challenges for the certification process, as it necessitates certifying components that may not disclose their internal behaviour (black-boxes). In this paper, we present a real-world case study of an autonomous driving system, identify key open challenges associated with its development and integration, and explore how formal verification techniques can address these challenges to ensure system reliability and safety.

1 Introduction

The Society of Automotive Engineers (SAE) defines the design goals of autonomous driving across six distinct levels, ranging from Level 0 (L-0) to Level 5 (L-5), as outlined in [27]. These levels represent a spectrum of automation: L-0 denotes no automation, followed by L-1 which includes driver assistance, L-2 for partial automation, L-3 for conditional automation, L-4 for high automation, and culminating in L-5, which signifies full automation. Each level reflects the increasing capability of autonomous systems and their interaction with human drivers. Currently, most commercially available vehicles operate at L-2 automation. This level encompasses features such as adaptive cruise control and lane-keeping assistance, enabling the vehicle to assist the driver while still requiring constant supervision and active engagement. A few manufacturers are exploring L-3 systems, which offer conditional automation under specific circumstances. For example, some modern vehicles equipped with L-3 systems can handle highway driving autonomously, including lane-keeping, speed regulation, and adaptive cruise control, but require the driver to take over when exiting highways or in complex urban environments. Despite such advancements, the industry is still in the early stages of fully implementing higher levels of automation.

According to a recent survey on the subject [17], achieving L-5 autonomy requires the appropriate integration of technologies and efficient communication channels. Realising the full potential of automated driving demands a **reliable**, robust, and widespread mobile network. In this work, we focus on the first item on the list; that is, we are interested in making the components of autonomous driving, as well as their interactions, (more) reliable. To achieve this, we start with a case study of a real-world autonomous driving system and address the issues to enhance its reliability from a formal perspective.

Taking inspiration from [22], we treat the autonomous driving system as a component-based system composed of black-box components that we are not interested in opening (or cannot open). Instead, we focus on how to achieve the formal verification of the resulting heterogeneous system. That is, how the

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 191–200, doi:10.4204/EPTCS.411.13

© P. Burgio, A. Ferrando & M. Villani
This work is licensed under the
Creative Commons Attribution License.

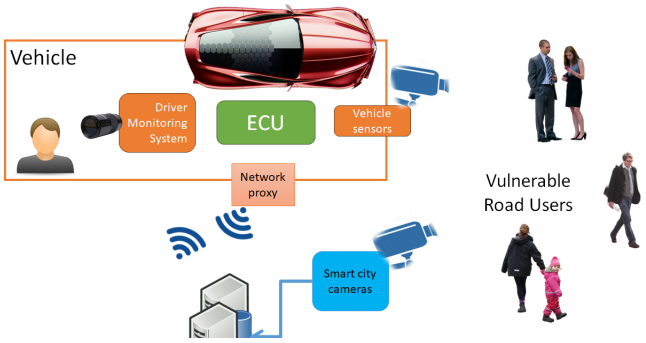


Figure 1: The vehicle architecture and use case

components interact with each other, and how we can verify (and perhaps even enforce) correct behaviour according to well-known standards in autonomous driving.

This paper presents a real-world case study in autonomous driving (Section 2), highlighting key challenges. Section 3 examines how formal methods, particularly formal verification, can address these issues and be integrated into autonomous systems, with Section 4 discussing their limitations. Finally, Section 5 concludes the paper and outlines future directions.

2 Autonomous Driving Case Study

As a motivational example, we present a case study from the AI4CSM Project[1], funded by the European Commission [33]. This study features a L-3/L-4 autonomous vehicle equipped with advanced sensor fusion capabilities, exemplifying a next-generation automotive platform [29, 35]. The vehicle can interpret both driver status (e.g., drowsiness, distraction) using in-vehicle cameras, and external environmental conditions using on-board sensors and data from city sensors, as illustrated in Figure 1.

In the simplest scenario, the vehicle operates at a low level of automation (*i.e.*, L-2/L-3), with the driver maintaining control. If the vehicle detects a potentially dangerous situation, such as drowsiness or imminent collisions, it triggers a secure takeover strategy, transitioning to L-4 and executing a safety manoeuvre. For our study, we focus on the sensor fusion component, where the perception module running on the on-board Electronic Control Unit (ECU) aggregates information from heterogeneous data streams. Specifically, we implemented a system to monitor the driver using camera-based behavioural analysis, coupled with on-board cameras to inspect the surrounding environment. Additionally, we enhanced the vehicle's perception capabilities by incorporating data from a smart city prototype area, namely the Modena Automotive Smart Area (MASA)[2]. Its structure is shown in Figure 2.

This area includes smart cameras mounted on poles that detect vulnerable road users and analyse or predict their movement trajectories. These data are streamed to the vehicle through the smart city's 4G-5G wireless connectivity. The vehicle's centralised ECU, also known as the Domain Controller, processes this information to determine the most appropriate response, such as emergency braking, complex manoeuvres, or issuing driver warnings. For research purposes, we implemented this use case on a Citroen Mehari. We will now explore the main open challenges that must be addressed to facilitate the industrialisation of these complex systems.

[1] https://ai4csm.eu/
[2] https://www.automotivesmartarea.it/

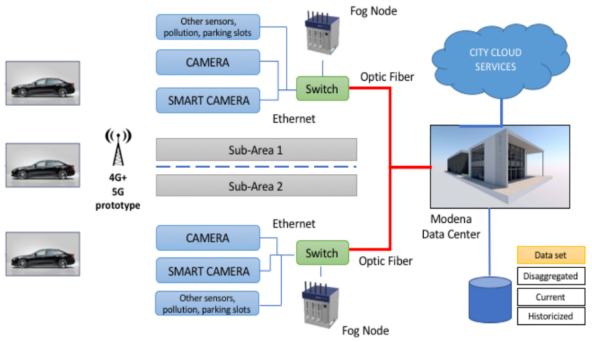


Figure 2: The Modena Automotive Smart Area.

Aggregating probabilistic data sources. One primary open challenge arises from the inherent nature of most software components used for perception. This stage is the first in any autonomous driving stack, where raw sensor data (in our scenario, RGB cameras) are processed to interpret and analyse the driver or the car's surroundings. Numerous algorithms and approaches could be employed, most of which [29, 35, 5, 19, 24] heavily rely on machine learning, deep learning, or, more generally, on heuristics and statistical methodologies to handle the complexity of raw data frames. Additionally, most systems have a hierarchical structure, consisting of sub-components arranged in pipelines. Figure 3 illustrates this decomposition for our camera-based behavioural analysis used for monitoring the driver.

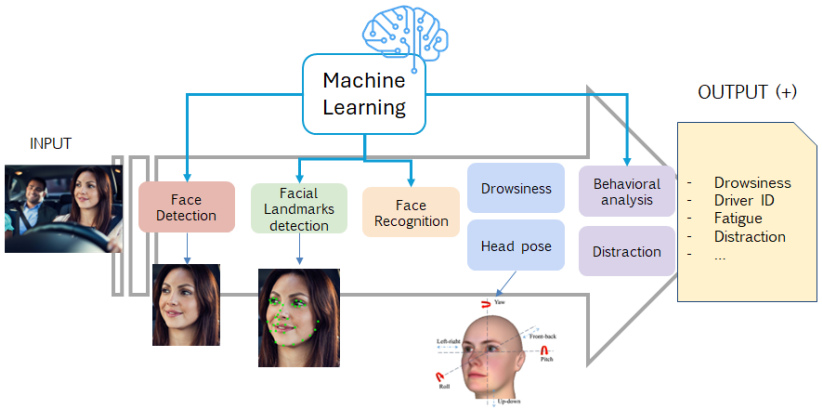


Figure 3: Scheme of the behavioural DMS pipeline.

Every block of this system has a specific performance metric, typically expressed in Frames-Per-Second (FPS), and a nominal accuracy, indicating the reliability of the information produced by the (sub)component. Our Driver Monitoring System (DMS) is a white-box component, developed in-house, which allows us complete access to its internals for tuning and modifications. However, in realistic industrial scenarios, most software modules will be developed by different companies and often implemented as "black-box" ECUs. Therefore, we identify the first open challenge.

Open Challenge 1: the need to compose a hierarchy of probabilistic software modules to formally measure and derive the overall system's resulting accuracy.

Deploy on embedded systems. Deploying intelligence in automation use cases requires two key components: powerful computational hardware and numerous sensor modules to accurately interpret

the surrounding and in-cabin environment in a timely manner. This presents a significant challenge for automotive engineers, who must integrate TOPS-greedy[3] software components onto power-efficient boards, ideally featuring many-core data processors such as those from NVIDIA Orin [25] or AMD XILINX [7, 1]. Figure 4 illustrates the target architecture of next-generation ECUs.

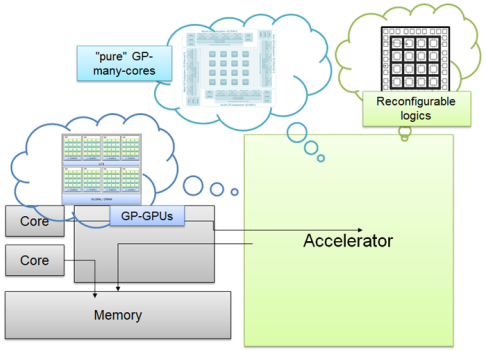


Figure 4: Generic architecture next-generation ECUs.

These systems employ multi-core host platforms, which include both Real-Time and non-Real-Time core ISAs (Instruction Set Architectures), which define the set of instructions a processor can execute. These are coupled with data-crunching architectures such as GPGPUs [25], reconfigurable arrays [1], or application-specific circuitry to implement processing algorithms directly in hardware. Such a complex architecture presents two main challenges.

Open Challenge 2: to devise efficient strategies for mapping software components onto the available computing cores, exploiting redundancy and voting schemes to enhance overall system reliability.

Intuitively, most of the algorithms we employ can potentially run on various cores, and finding the optimal mapping must be handled in the most efficient manner.

3 Formal Methods to the Rescue

In the previous section, we discussed a real-world case study in autonomous driving, highlighting challenges in integrating autonomous systems into road infrastructure and progressing towards L-5 capabilities. Here, we explore how formal methods, particularly formal verification, address these challenges. For a comprehensive overview of formal verification techniques in autonomous systems, see [21].

3.1 Open Challenge 1: Heterogeneous Composition of Untrustworthy Components

The first challenge involves managing components with varying levels of reliability. Some components are open-source (white-box), allowing full access and modifications, while others are closed-source (black-box), restricting access to their internal workings. To address this challenge, we propose exploiting formal verification techniques. Specifically, as outlined in [20], we employ formal verification methods focusing on three key areas (called also recipes): verification of decision-making components, AI-based components, and the enforcement of safety claims.

[3]They require a high number of Tera Operations Per Second (TOPS) to process complex algorithms, such as those used in artificial intelligence and sensor fusion.

To address this challenge, we propose the use of heterogeneous verification techniques [22, 4, 6, 26]. These techniques are based on the Assume-Guarantee principle, where each component of the heterogeneous system is defined in terms of its *assumptions* (what the component expects from the system to function correctly) and its *guarantees* (what the component provides to the system upon correct execution). This methodology allows us to abstract away the implementation details of the various system components, enabling the system designer to focus on their integration. As long as the assumptions and guarantees of a component are documented and made available, it can be implemented as either a white-box or black-box component. By employing these verification techniques, it is possible to formally verify the proper integration of multiple components, potentially developed by different parties [22, 4, 6, 26].

In addition to Assume-Guarantee reasoning, model checking and formal methods can play a crucial role in verifying component integration and ensuring system reliability [8, 3, 10, 16]. Component-Based Software Engineering (CBSE) methodologies also provide a framework for assembling reliable systems from diverse components [30]. Moreover, adhering to safety and certification standards, such as ISO 26262, is essential for validating the safety of automotive software systems [14].

To complete the verification process, we envision the use of Runtime Verification (RV) [2], a technique for monitoring and analysing the execution of a system at runtime to ensure it adheres to specified properties. RV can check and enforce adherence to all assumptions and guarantees of the components. As highlighted in [22], the Assume-Guarantee verification methodology focuses on verifying the resulting distributed system and the integration of its components. However, it relies on the assumption that the assumptions and guarantees of each component (which may be black-boxes) are satisfied. To bridge this gap and provide a robust verification technique suitable for the heterogeneous nature of the autonomous driving domain, we need to employ additional verification methods to ensure the proper behaviour of individual components. By confirming that each component behaves correctly, we can validate the entire system's integration and maintain its formal assurances.

It is important to note that the use of RV in this context is not entirely straightforward. The components of an autonomous driving system may exhibit a certain level of uncertainty, meaning that the information they provide may not always be precise. For example, as illustrated in Figure 3, the steps that process the camera input to determine the driver's level of drowsiness, distraction, fatigue, etc., are inherently uncertain. These steps rely on Machine Learning models, which offer results with varying levels of confidence. Additionally, this uncertainty is not limited to the current information provided but may also encompass temporal aspects. For instance, a component might predict that the driver will fall asleep in five minutes, with a given level of confidence. Due to these factors, it is unrealistic to rely solely on standard RV approaches for verifying component conformance. Instead, techniques that incorporate RV with uncertainty must be considered, such as those discussed in [34, 32, 11]. For a comprehensive survey on this topic, the reader may refer to [31]. Additionally, the complexity of verifying machine learning and AI components within autonomous systems presents unique challenges. Ensuring the reliability of non-deterministic algorithms requires specialised verification techniques [23]. Addressing these challenges will enable the development of robust, reliable, and safe autonomous driving systems.

3.2 Open Challenge 2: Efficient Strategies for Mapping the Distributed Computation

Addressing this challenge involves the use of formal verification techniques to ensure that the mapping strategies are both efficient and reliable (since we are in a real-time system). Formal verification can be employed to systematically verify that the software components are optimally distributed across the computing cores, and that redundancy and voting mechanisms are correctly implemented to enhance fault tolerance and system robustness. By formally verifying these strategies, we can guarantee the sys-

tem meets its performance and reliability requirements, even in the presence of component failures or uncertainties. Indeed, fault-tolerant designs, which incorporate redundancy and voting schemes, play a crucial role in mitigating the impact of component failures. The study presented in [9] provides valuable insights into the application of formal verification techniques to validate the correctness of these designs. By systematically verifying that fault-tolerant hardware meets specified reliability requirements, the authors demonstrate the effectiveness of formal methods in identifying design errors that traditional testing might overlook. Although [9] does not originate from the domain of autonomous driving, it provides a valuable foundation for addressing the open challenge discussed here. The paper presents methodologies for formal verification of fault-tolerant hardware designs, which are crucial for ensuring the reliability and robustness of systems with heterogeneous components. By adapting these verification techniques, we can systematically validate the correctness and reliability of the complex, integrated systems used in autonomous vehicles.

Formal verification can be used to prove that the redundancy and voting mechanisms are correctly implemented and that they effectively enhance system reliability; for example, in [28], the authors discuss fault-tolerance techniques that include redundancy and efficient scheduling policies. Formal verification ensures that these techniques are correctly applied, thereby enhancing the reliability of the system.

The work in [15] provides a comprehensive framework for the formal verification of distributed Resource Management (RM) schemes in many-core systems using probabilistic model checking. This research is particularly relevant to our work in the context of autonomous driving systems, which also require efficient resource allocation across multiple computing cores. The authors demonstrate the use of the PRISM model checker [18] to analyse and compare the performance and reliability of different RM schemes. They emphasise the limitations of traditional simulation methods, which are inherently exhaustive, and advocate for formal verification to ensure completeness and accuracy. In our study, similar formal verification techniques can be applied to optimise the mapping of software components onto the available computing cores in autonomous vehicles. By leveraging the probabilistic analysis methods described in [18], we could systematically evaluate the robustness and performance efficiency of our proposed resource management strategies in autonomous driving systems.

4 Limitations of Applying Formal Methods in Autonomous Systems

While Formal Methods provide a promising approach for integrating heterogeneous components and efficient mapping in autonomous systems, there are significant practical limitations to their use. The main challenges include the need for specialised knowledge, scalability issues, interpretability of results, and difficulties in handling uncertain environments, as well as cost-benefit trade-offs in development.

One key limitation is the high barrier to entry, as FM often requires deep expertise in formal logic and verification techniques. This specialised skill set is not commonly available within standard engineering teams, and the process of specifying and verifying systems can be time-consuming. Despite the development of new tools aimed at making FM more accessible, their capabilities are still evolving and often require substantial refinement to meet industry needs.

Scalability and complexity also present major obstacles. Autonomous systems comprise numerous interacting components, leading to a state space that grows exponentially, making exhaustive verification computationally expensive or infeasible. Techniques like compositional reasoning and modular verification attempt to manage this, but they require careful abstraction, which may oversimplify or overlook critical behaviours. Moreover, model checking can be computationally intensive, particularly when applied to real-time or resource-constrained systems, and Assume-Guarantee reasoning hinges on accurate

assumptions that are challenging to guarantee in practice.

Another practical challenge is the interpretability of verification results. Outputs from FM tools, such as model checkers or runtime verifiers, may highlight specification violations without providing clear solutions, requiring domain expertise to resolve. When AI and machine learning components are involved, this issue is further complicated by their probabilistic and non-deterministic nature, making the analysis of verification results particularly difficult.

The dynamic and uncertain environment in which autonomous systems operate adds further complexity. Perception modules relying on sensor fusion and AI-based decision-making are context-dependent and produce inherently uncertain outputs. Traditional FM approaches struggle to define precise specifications in such cases. Techniques that incorporate probabilistic reasoning or uncertainty-aware models are being explored [34, 32, 11], but their practical applicability to real-world systems remains under active research and development.

Integrating FM into existing development workflows also poses a significant challenge, as it requires a careful cost-benefit analysis. The process of formally specifying, modeling, and verifying components demands significant time and resources, potentially extending the development lifecycle. While FM provides strong safety and reliability assurances – critical for autonomous vehicles – these benefits must be weighed against their scalability and adaptability to system updates and changes.

Lastly, verifying AI and ML components remains an open challenge, as their non-deterministic behaviour does not fit within traditional FM frameworks. Specialised techniques are needed to manage varying levels of confidence and probabilistic outputs in AI models [23]. Thus, hybrid approaches combining formal verification with testing and validation are necessary for comprehensive system assurance.

In summary, FM offers a rigorous foundation for ensuring system reliability and safety in autonomous vehicles, but practical limitations such as scalability, required expertise, result interpretation, and integration into dynamic environments must be addressed to enable their effective use in real-world applications.

5 Conclusions and Future Work

In this short paper, we present a real-world case study in the autonomous driving domain, identify key open challenges, and discuss how formal verification techniques can address these issues.

We focus on two primary challenges hindering the achievement of L-5 automation in autonomous vehicles. The first is the heterogeneous composition of black-box components, which affects system reliability. The second challenge involves the proper mapping of computations within a highly dynamic, real-time distributed system. We propose how existing formal verification techniques can be leveraged to address these issues and explore their implications, along with potential adaptations for integration into autonomous driving systems.

Our aim is to highlight these open challenges in the autonomous driving domain and demonstrate how formal verification techniques can theoretically enhance system reliability. For future work, we intend to build upon the insights reported here by applying some of the discussed techniques and methodologies to our case study, leveraging tools like those proposed in [12, 13] to further explore their practical applicability and impact on system robustness.

Acknowledgements

The authors have received funding from ECSEL JU project AI4CSM (GA N.101007326) and the Chips JU project ShapeFuture (GA N.101139996).

References

[1] AMD Xilinx (2022): *Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*. Available at `https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html`. Accessed on October 2024.

[2] Ezio Bartocci, Yliès Falcone, Adrian Francalanza & Giles Reger (2018): *Introduction to Runtime Verification*. In Ezio Bartocci & Yliès Falcone, editors: *Lectures on Runtime Verification - Introductory and Advanced Topics*, *Lecture Notes in Computer Science* 10457, Springer, pp. 1–33, doi:10.1007/978-3-319-75632-5_1.

[3] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen & Joseph Sifakis (2010): *Compositional verification for component-based systems and application*. *IET Softw.* 4(3), pp. 181–193, doi:10.1049/IET-SEN.2009.0011.

[4] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto L. Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger & Kim G. Larsen (2018): *Contracts for System Design*. *Found. Trends Electron. Des. Autom.* 12(2-3), pp. 124–400, doi:10.1561/1000000053.

[5] Roberto Cavicchioli, Riccardo Martoglia & Micaela Verucchi (2022): *A Novel Real-Time Edge-Cloud Big Data Management and Analytics Framework for Smart Cities*. *Journal of Universal Computer Science* 28(1), p. 3 – 26, doi:10.3897/jucs.71645.

[6] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai & Cesare Tinelli (2016): *CoCoSpec: A Mode-Aware Contract Language for Reactive Systems*. In Rocco De Nicola & Eva Kühn, editors: *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, *Lecture Notes in Computer Science* 9763, Springer, pp. 347–366, doi:10.1007/978-3-319-41591-8_24.

[7] Kwon Neung Cho, Jeongeun Kim, Do Young Choi, Young Hyun Yoon, Jung Hwan Oh & Seung Eun Lee (2021): *An FPGA-Based ECU for Remote Reconfiguration in Automotive Systems*. *Micromachines* 12, doi:10.3390/mi12111309. Available at `https://www.mdpi.com/2072-666X/12/11/1309`.

[8] E. M. Clarke, O. Grumberg & D. A. Peled (1999): *Model Checking*. MIT Press.

[9] Luis Entrena, Antonio J. Sanchez-Clemente, Luis Ángel García-Astudillo, Marta Portela-García, Mario García-Valderas, Almudena Lindoso & Roberto Sarmiento (2023): *Formal Verification of Fault-Tolerant Hardware Designs*. *IEEE Access* 11, pp. 116127–116140, doi:10.1109/ACCESS.2023.3325616.

[10] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga & Saddek Bensalem (2015): *Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation*. *Softw. Syst. Model.* 14(1), pp. 173–199, doi:10.1007/S10270-013-0323-Y.

[11] Angelo Ferrando & Vadim Malvone (2022): *Runtime Verification with Imperfect Information Through Indistinguishability Relations*. In Bernd-Holger Schlingloff & Ming Chai, editors: *Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, *Lecture Notes in Computer Science* 13550, Springer, pp. 335–351, doi:10.1007/978-3-031-17108-6_21.

[12] Angelo Ferrando & Vadim Malvone (2024): *Hands-on VITAMIN: A Compositional Tool for Model Checking of Multi-Agent Systems*. In Marco Alderighi, Matteo Baldoni, Cristina Baroglio, Roberto Micalizio & Stefano Tedeschi, editors: *Proceedings of the 25th Workshop "From Objects to Agents", Bard (Aosta), Italy, July 8-10, 2024*, *CEUR Workshop Proceedings* 3735, CEUR-WS.org, pp. 148–160. Available at `https://ceur-ws.org/Vol-3735/paper_12.pdf`.

[13] Angelo Ferrando & Vadim Malvone (2024): *VITAMIN: A Compositional Framework for Model Checking of Multi-Agent Systems*. *CoRR* abs/2403.02170, doi:10.48550/ARXIV.2403.02170. arXiv:2403.02170.

[14] International Organization for Standardization (2018): *ISO 26262-1:2018. Road vehicles – Functional safety*.

[15] Shafaq Iqtedar, Osman Hasan, Muhammad Shafique & Jörg Henkel (2016): *Formal probabilistic analysis of distributed resource management schemes in on-chip systems*. In Luca Fanucci & Jürgen Teich, editors: *2016*

Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016, IEEE, pp. 930–935. Available at https://ieeexplore.ieee.org/document/7459441/.

[16] Daniel Karlsson, Petru Eles & Zebo Peng (2007): *Formal verification of component-based designs*. *Des. Autom. Embed. Syst.* 11(1), pp. 49–90, doi:10.1007/S10617-006-9723-3.

[17] Manzoor Ahmed Khan, Hesham El-Sayed, Sumbal Malik, Muhammad Talha Zia, Muhammad Jalal Khan, Najla Alkaabi & Henry Alexander Ignatious (2023): *Level-5 Autonomous Driving - Are We There Yet? A Review of Research Literature*. *ACM Comput. Surv.* 55(2), pp. 27:1–27:38, doi:10.1145/3485767.

[18] Marta Z. Kwiatkowska, Gethin Norman & David Parker (2002): *PRISM: Probabilistic Symbolic Model Checker*. In Tony Field, Peter G. Harrison, Jeremy T. Bradley & Uli Harder, editors: *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, London, UK, April 14-17, 2002, Proceedings*, *Lecture Notes in Computer Science* 2324, Springer, pp. 200–204, doi:10.1007/3-540-46029-2_13.

[19] Songtao Liu, Di Huang & andYunhong Wang (2018): *Receptive Field Block Net for Accurate and Fast Object Detection*. In: *The European Conference on Computer Vision (ECCV)*, doi:10.1007/978-3-030-01252-6_24.

[20] Matt Luckcuck (2023): *Using formal methods for autonomous systems: Five recipes for formal verification*. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 237(2), pp. 278–292, doi:10.1177/1748006X211034970.

[21] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon & Michael Fisher (2019): *Formal Specification and Verification of Autonomous Robotic Systems: A Survey*. *ACM Comput. Surv.* 52(5), pp. 100:1–100:41, doi:10.1145/3342355.

[22] Matt Luckcuck, Marie Farrell, Angelo Ferrando, Rafael C. Cardoso, Louise A. Dennis & Michael Fisher (2022): *A Compositional Approach to Verifying Modular Robotic Systems*. *CoRR* abs/2208.05507, doi:10.48550/ARXIV.2208.05507. arXiv:2208.05507.

[23] G. Marcus & E. Davis (2019): *Rebooting AI: Building Artificial Intelligence We Can Trust*. Pantheon Books.

[24] Yuqi Nie, Nam H. Nguyen, Phanwadee Sinthong & Jayant Kalagnanam (2023): *A Time Series is Worth 64 Words: Long-term Forecasting with Transformers*. In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*, OpenReview.net. Available at https://openreview.net/forum?id=Jbdc0vTOcol.

[25] NVIDIA (2022): *NVIDIA Jetson AGX Orin Series*. Available at https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf. Accessed: October 2024.

[26] Ivan Ruchkin, Joshua Sunshine, Grant Iraci, Bradley R. Schmerl & David Garlan (2018): *IPL: An Integration Property Language for Multi-model Cyber-physical Systems*. In Klaus Havelund, Jan Peleska, Bill Roscoe & Erik P. de Vink, editors: *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, *Lecture Notes in Computer Science* 10951, Springer, pp. 165–184, doi:10.1007/978-3-319-95582-7_10.

[27] I SAE (2021): *Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles j3016 202104*. *Society of Automotive Engineers* 41.

[28] Sepideh Safari, Mohsen Ansari, Heba Khdr, Pourya Gohari-Nazari, Sina Yari-Karin, Amir Yeganeh-Khaksar, Shaahin Hessabi, Alireza Ejlali & Jörg Henkel (2022): *A Survey of Fault-Tolerance Techniques for Embedded Systems From the Perspective of Power, Energy, and Thermal Issues*. *IEEE Access* 10, pp. 12229–12251, doi:10.1109/ACCESS.2022.3144217.

[29] Society of Automotive Engineers (SAE) (2024): *Sensor fusion expanding in step with advancing vehicle sophistication*. Available at https://www.sae.org/news/2024/02/sensor-fusion-trends. Accessed on October 2024.

[30] Clemens A. Szyperski, Dominik Gruntz & Stephan Murer (2002): *Component software - beyond object-oriented programming, 2nd Edition*. Addison-Wesley component software series, Addison-Wesley. Available at https://www.worldcat.org/oclc/248041840.

[31] Rania Taleb, Sylvain Hallé & Raphaël Khoury (2023): *Uncertainty in runtime verification: A survey*. *Comput. Sci. Rev.* 50, p. 100594, doi:10.1016/J.COSREV.2023.100594.

[32] Rania Taleb, Raphaël Khoury & Sylvain Hallé (2021): *Runtime Verification Under Access Restrictions*. In Simon Bliudze, Stefania Gnesi, Nico Plat & Laura Semini, editors: *9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021*, IEEE, pp. 31–41, doi:10.1109/FORMALISE52586.2021.00010.

[33] Ovidiu Vermesan, Reiner John, Patrick Pype, Gerardo Daalderop, Kai Kriegel, Gerhard Mitic, Vincent Lorentz, Roy Bahr, Hans Erik Sand, Steffen Bockrath & Stefan Waldhör (2021): *Automotive Intelligence Embedded in Electric Connected Autonomous and Shared Vehicles Technology for Sustainable Green Mobility*. *Frontiers in Future Transportation* 2, doi:10.3389/ffutr.2021.688482.

[34] Shaohui Wang, Anaheed Ayoub, Oleg Sokolsky & Insup Lee (2011): *Runtime Verification of Traces under Recording Uncertainty*. In Sarfraz Khurshid & Koushik Sen, editors: *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, *Lecture Notes in Computer Science* 7186, Springer, pp. 442–456, doi:10.1007/978-3-642-29860-8_35.

[35] Zhangjing Wang, Yu Wu & Qingqing Niu (2020): *Multi-Sensor Fusion in Automated Driving: A Survey*. *IEEE Access* 8, pp. 2847–2868, doi:10.1109/ACCESS.2019.2962554.

Formalizing Stateful Behavior Trees

Serena S. Serbinowska
0000-0002-9259-1586
Vanderbilt University
Nashville TN, USA
serena.serbinowska@vanderbilt.edu

Preston Robinette
0000-0002-4906-2179
Vanderbilt University
Nashville TN, USA
preston.robinette@vanderbilt.edu

Gabor Karsai
0000-0001-7775-9099
Vanderbilt University
Nashville TN, USA
gabor.karsai@vanderbilt.edu

Taylor T. Johnson
0000-0001-8021-9923
Vanderbilt University
Nashville TN, USA
taylor.johnson@vanderbilt.edu

Behavior Trees (*BT*s) are high-level controllers that are useful in a variety of planning tasks and are gaining traction in robotic mission planning. As they gain popularity in safety-critical domains, it is important to formalize their syntax and semantics, as well as verify properties for them. In this paper, we formalize a class of *BT*s we call Stateful Behavior Trees (*SBT*s) that have auxiliary variables and operate in an environment that can change over time. *SBT*s have access to persistent shared memory—often known as a blackboard—that keeps track of these auxiliary variables. We demonstrate that *SBT*s are equivalent in computational power to Turing Machines when the blackboard can store mathematical (unbounded) integers. We also identify conditions where *SBT*s have computational power equivalent to finite state automata, specifically where the auxiliary variables are of finitary types. We present a domain specific language (DSL) for writing *SBT*s and adapt the tool BehaVerify for use with this DSL. This new DSL in BehaVerify supports interfacing with popular *BT* libraries in Python, and also provides generation of Haskell code and nuXmv models, the latter of which are used for model checking temporal logic specifications for the *SBT*s. We include examples and scalability results where BehaVerify outperforms another verification tool (MoVe4BT) by a factor of 100.

1 Introduction

A Behavior Tree (*BT*) is a high-level controller that shares similarities with a hierarchical state machine, yet distinguishes itself by offering greater flexibility and modularity in defining behaviors. At its core, a *BT* organizes various behaviors within a tree structure, where leaf nodes encapsulate distinct behaviors and higher-level nodes define the control flow. This hierarchical arrangement facilitates the design of complex behaviors and is both scalable and adaptable to changing circumstances or requirements.

*BT*s were originally created for video game development and were devised to enhance the autonomy and realism of Non-Playable Characters (NPCs). An NPC is an entity within a video game that operates under the control of the game's Artificial Intelligence (AI). *BT*s are useful for specifying such behaviors. The explainability and versatility of *BT*s have also led to their widespread adoption in areas like robotics and AI. Accordingly, *BT*s have been used for a variety of tasks, such as for controlling wheeled-legged robots [10] and bipedal locomotion robots [16], in vision measurement systems of road users [24], and the management swarms [18, 20]. Additional applications can be found in this survey [19].

As *BT*s continue to be adopted to address new and existing challenges in various domains, especially in real-world, safety-critical domains such as robotics, it is increasingly important to formalize their structure and behaviors. Such formalization is crucial for the verification of safety and liveness specifications, ensuring the systems behave reliably and as intended under all conditions. Toward this end, we provide

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods
for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 201–218, doi:10.4204/EPTCS.411.14

a formalization we call Stateful Behavior Trees (*SBT*s). *SBT*s are a class of *BT*s that have auxiliary variables and operate in an environment. The primary contributions of this work are the following.

1. We formalize a novel class of models we call *SBT*s that operate in an environment and have global variables stored in persistent shared memory.
2. We demonstrate equivalence of *SBT*s to Turing Machines and Finite State Automata under syntactic assumptions, which is of critical importance for model checking *BT*s.
3. We present a domain specific language (DSL) for writing *SBT*s implemented in an entirely reworked software tool called BehaVerify [25].
4. We compare the entirely reworked BehaVerify [25] to MoVe4BT [23] in different verification examples and outperform in each; in one, BehaVerify outperformed by a factor of over 100.

2 Related Work

In this section, we discuss relevant literature, focusing on the verification of *BT*s and domain specific languages (DSLs) for *BT*s. We highlight the contributions of this paper within these contexts.

Our Prior Work In our prior work we presented BehaVerify [25]. That version of BehaVerify [25] took as input a Py Trees [27] object and walked the tree to create a nuXmv [5] model. The created nuXmv [5] model was incomplete; it had composite and decorator nodes, but the leaf nodes were 'stubs' for the user to fill in. The same was true of variables. The new version of BehaVerify [25] utilizes a Domain Specific Language (DSL). It takes as input a *BT* specified using the DSL and produces as output a nuXmv [5] model, a Py Trees [27] implementation of the *BT*, or a Haskell implementation of the *BT*. Crucially, the nuXmv [5] model is now complete; there are no 'stubs' for the user to fill in as all the variables and leaf nodes are fully and completely generated.

Existing Behavior Tree Frameworks To our knowledge, there are no existing DSLs for *BT*s, but there are several related libraries and frameworks. [15] lists a variety of different DSLs, but we believe these would more correctly be classified as library implementations of *BT*s (e.g. BehaviorTree.CPP [1] and Py Trees [27]). PROMISE [14] is a DSL inspired by *BT*s, but it is not a DSL for *BT*s. MoVe4BT [23], which we compare against, uses an xml style for specifying *BT*s.

Verifying Behavior Trees There are several existing formal verification works for *BT*s. [2] utilizes SPOT [12] for verification of *BT*s, but is limited to atomic propositions and boolean operators. Furthermore, the examples provided seemed to take over an hour to run for very small trees. [6] does runtime verification for a fragment of Timed Propositional Temporal Logic (TPTL), but not design-time model verification. We compared against BTCompiler in our previous paper [25]. To the best of our knowledge, the only other existing and available tools for model verification of *BT*s are ArcadeBT [17] and MoVe4BT [23].

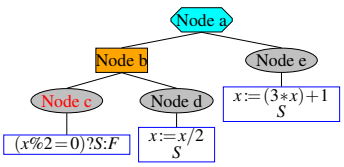
ArcadeBT [17] is an automatic verification method for *BT*s that verifies safety properties by encoding the *BT* using Linear Constrained Horn Clauses (LCHCs). To do this, ArcadeBT [17] includes an implementation of *BT*s in C++ that can be automatically converted to LCHCs and verified using Z3 [22]. The tool has been evaluated on trees with up to 18 nodes. By comparison, our new DSL and implementation in BehaVerify handles trees with 20000 nodes (see Section 7) and supports verification of linear temporal logic (LTL) and computation tree logic (CTL) allowing for both liveness and safety to be verified.

MoVe4BT [23] allows for the verification of LTL specifications over nodes, but it cannot verify LTL specifications written as predicates over variables. In contrast, the implementation of *SBT* verification in BehaVerify developed in this paper supports LTL specifications over variables and nodes. MoVe4BT [23] supports nodes with true parallelism, while BehaVerify does not. However, Py Trees [27], a popular implementation of *BT*s that BehaVerify targets, does not support true parallelism. A more detailed comparison of the tools can be found in Section 7, including experimental evaluations that demonstrate BehaVerify is able to verify trees 100 times bigger than MoVe4BT [23] (20000 vs 200 nodes).

BTs with State and Theoretical Foundation for Verification While not universal, it is common for *BT*s to interact with memory, referred to here as a *blackboard*. [4] compares pure *BT*s (*BT*s without a blackboard) to unrestricted *BT*s (*BT*s with a finite blackboard). While the unrestricted *BT*s are strictly more powerful than the pure *BT*s, [3] points out that this violates the 'reactive' nature of *BT*s ([3] states "An architecture is reactive if its decision making depends only on the current state of the environment"). Instead of using blackboards, the authors of [3] advocate for combining *BT*s with Stateful Components, thereby preserving the benefits of *BT*s without loss of computational power. Regardless, practical major implementations of *BT*s (such as Py Trees [27] and its Robotic Operating System (ROS) extension PyTreesRos, BehaviorTree.cpp [1], and Unreal Engine [13]) all feature the blackboard. As such, there is a practical need for a framework that addresses *BT*s with blackboards.

Other researchers take a different approach and treat *BT*s as deterministic functions with control systems, as seen in the works by [29], [9], [28], [26], and [8], with [21] analyzing the potential of *BT*s as alternatives to Controlled Hybrid Dynamical Systems. While these undoubtedly describe *BT*s with state, there are crucial differences between this style and our formalization of *SBT*s. The issue at hand is that 'state' is an ambiguous term; it could refer to either memory or to the environment. For instance, it was not assumed that pure *BT*s do not function in a persistent environment; rather, the assumption was that the *BT* does not leverage its own persistent memory to augment its behavior. In essence, pure *BT*s are functional; if presented with the same set of inputs, they will produce the same outputs. In this sense, the control system approach utilizes pure *BT*s without memory; the state represents the environment rather than memory. In contrast, *SBT*s consider both a blackboard *and* the environment. Moreover, the environment in the control system model was fully under control of the *BT*. While this may be a reasonable assumption in certain contexts, it makes it impossible to model uncertainty (such as the presence of wind for drones). As such, our formalization allows for nondeterministic updates. This is extended to the environment, which is allowed to change and develop according to user defined rules that can utilize nondeterminism.

Computational Power Finally, we consider the computational power of the resulting models. [7] informally says *BT*s are the same as Finite State Machines (*FSM*s), but does not explicitly state any assumptions, restrictions, etc. On the other hand, [4] creates a hierarchy of Teleo-reactive programs (TR), Decision Trees (DT), *BT*s, and *FSM*s. They conclude that if you provide a TR, DT, or *BT* with access to a finite blackboard that they can freely read from/write to, then they are equivalent to a *FSM*. Giving a *FSM* access to a finite blackboard does not increase the computational power of the model, as this is the equivalent of adding a finite number of additional states. We take this a step further and consider the power of a *SBT* with an infinite blackboard (a blackboard capable of storing variables of unbounded size) and conclude that such a model has the computational power of a Turing Machine.



Tick	1	1	1	1	1	1	1	2	2	2	2	2	2	2
t	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Active	a	b	c	b	d	b	a	a	b	c	b	a	e	a
Returns	-	-	*S*	-	*S*	*S*	*S*	-	-	*F*	*F*	-	*S*	*S*
x	6	6	6	6	3	3	3	3	3	3	3	3	10	10

Figure 1: A *BT* for the Collatz conjecture (hailstone sequence) consisting of a selector node (a), a sequence node (b), a check (c), and two actions (d, e). We use the ternary operator $i?j:k$ to mean if i then j else k. We use % for modulo. Tick indicates the number of times the tree has been ticked. t is used to track the number of times we have changed active nodes. Active is used to track where we are in the tree. Returns indicates what the active node returns; a - indicates the node did not return. If a node is finished, then it returns one of S, F, or R.

3 Behavior Tree Overview

We will utilize Figure 1 to provide an intuitive explanation of *BT*s. We will then provide additional details.

In Figure 1 we have a basic *BT* for calculating a hailstone sequence from a starting value. To calculate such a sequence, start with a positive integer and use the following rules: if the number is even, divide by 2; otherwise, multiply by 3 and add 1. The root of the tree (a) is a selector node. As the name implies, this node 'selects' a child. In this case, we want to select either 'divide by 2' or 'multiply by 3 and add 1'. (b) is a sequence node; aptly named once more, this node executes a sequence, aborting if a failure is encountered. Here our sequence is 'number is even' followed by 'divide by 2'. (c) is a check node; it checks if a condition is true or false. (d) and (e) are action nodes; they actually do stuff.

Our tree starts when it receives an external signal (a *tick*). This causes the root (a) to become active. Execution will now, mostly, follow a *depth first traversal* (DFT). (a) hasn't yet selected an option, so it follows DFT and makes (b) active. (b) hasn't completed the sequence or encountered a failure, so it follows DFT and makes (c) active. (c) checks if x is even; since 6 is even it returns Success (S). (b) is active again, but it still hasn't completed a sequence or encountered a failure, so it follows DFT and makes (d) active. (d) executes the action and halves the value of x and returns S. (b) is active again and the sequence successfully finished, so S is returned. (a) is finally active again and it has selected an option, so it returns S. The first tick is now over. The tree will now do nothing until it receives another tick. When it does, (a) again becomes active, then (b), then (c). This time, however, x is not even, so (c) returns Failure (F). Thus the sequence failed, so (b) returns F. (a) is now active again, but still hasn't selected an option, so (e) becomes active. (e) executes the action; x becomes 10 and S is returned. (a) has now selected an option, so it returns S. The second tick is now over.

We now provide some more concrete requirements for *BT*s that were not covered by the example. There must be a path from the root to every other node in the tree. Each node has exactly one parent, except the root which has no parent. There are four possible states for nodes: Success (S), Failure (F), Running (R), and Invalid (I). Each node becomes I when a new tick arrives. When a node finishes executing it returns one of S, R, or F. Finally, there are three types of nodes: leaf, decorator, and composite.

Leaf Nodes Leaf nodes are nodes that do not have any children, and they either check a condition (check node) or do an action (action node). Check nodes return S if the associated condition is true and F otherwise; they do not do anything else. Action nodes can perform various actions. Furthermore, they are allowed to return S, F, or R and can utilize conditions to determine what to return. In Figure 1, (c) is a check node while (d) and (e) are action nodes.

Decorator Nodes A decorator node always has exactly one child. For our purposes, decorator nodes are used to change the output of a child node without requiring the child node to be modified. Some common decorator types are inverter, which swaps S and F, and R_Is_F which turns R into F.

Composite Nodes Composite nodes control the execution flow through a BT. There are three types of composite nodes: selector, sequence, and parallel. The children are ordered and for convenience we will use a left-to-right order.

1. Selector or fallback nodes, try to 'select' a child. A child is 'selected' if it returns S. Each child is activated in order, from left to right, until one of them returns S or R. At that point, the selector returns the same status. If every child returns F, the selector returns F.
2. Sequence nodes are identical to selector nodes, except S and F are swapped. While this is true and useful to note, it is more practical to think of them in a more distinct manner. Sequence nodes are used to execute a sequence to the end or a failure point. Each child is activated in order, from left to right, until one of them returns F or R. At that point, the sequence returns the same status. If every child returns S, then the sequence returns S.
3. Parallel nodes will not appear in this paper, but it is still important to mention them. As the name implies, parallel nodes activate all their children simultaneously. We do not support this behavior and neither does Py Trees [27], the Python implementation that BehaVerify targets. Instead, our parallel nodes activate each child in order, one at a time, left to right. Unlike selector and sequence nodes, there is no early termination condition for parallel nodes; each child *will* be activated. Once all the children have returned, the parallel node consults a policy to determine what to return.

4 Formal Definition of Stateful Behavior Trees

Here we provide a formal definition of a SBT. In service of this task, we start by defining a tree. A rooted tree is a triple (V,r,E) such that

- V is a finite set representing the vertices of the tree.
- $r \in V$ is a vertex representing the root.
- Let VS be the set of all finite sequences $vs = [v_0, v_1, \ldots, v_n]$ such that $\forall j,k \in \mathbb{Z}$ s.t. $(0 \leq j,k \leq n), (v_j, v_k \in V \wedge (v_j = v_k \implies j = k))$. That is to say, the elements of the sequence are unique vertices. If a and b are elements in a sequence, we will use $<,>,\leq,\geq$ to indicate relative order of the sequence. For example $a < b$ means that a appears before b in the sequence.
- $E : V \mapsto VS$ is a function from vertices to sequences of vertices (the children). It must also meet the following requirements
  - $\forall v \in V, r \notin E(v)$ (the root has no parent).
  - $\forall v,v' \in V, v \neq v' \implies E(v) \cap E(v') = \varnothing$. Each vertex has at most one parent.
  - $\forall v \in V, \exists [v_0,v_1,\ldots,v_n]$ s.t. $v_0 = r \wedge v_n = v \wedge \forall j \in \mathbb{Z}$ s.t. $0 \leq j < n, v_{j+1} \in E(v_j)$. There exists a path from the root to each vertex.

  These conditions ensure that the tree is actually a tree.

4.1 Stateful Behavior Tree

A SBT is a tuple $(V,r,E,S_{SBT},s_{SBT},\Sigma_{SBT},\delta_{SBT})$ such that

- (V,r,E) is a tree.
- S_{SBT} is a set representing the possible states of the blackboard of SBT. We discuss the implications of this set being infinite vs finite in Subsection 4.2.
- $s_{SBT} \in S_{SBT}$ is the initial state of the blackboard.
- Σ_{SBT} is a set representing the possible inputs (the environment).
- ST is the set of all functions $st : V \mapsto \{S,R,F,I\}$. Each $st \in ST$ is a function that maps each vertex to a status. ST is not an element of the tuple; it arises from the elements.
- $\delta_{SBT} : V \times ST \times S_{SBT} \times \Sigma_{SBT} \mapsto 2^{V \times ST \times S_{SBT}}$. Here $2^{V \times ST \times S_{SBT}}$ is the power set of $V \times ST \times S_{SBT}$. The function maps to sets to allow for the expression of nondeterminism. This function must also obey the following:

$$\forall v,v' \in V, \forall st,st' \in ST, \forall s,s' \in S_{SBT}, \forall a \in \Sigma_{SBT}, (v',st',s') \in \delta_{SBT}(v,st,s,a) \implies$$
$$\left(\begin{array}{c} (v = r \wedge st(v) \neq I \implies (v' = r \wedge s = s' \wedge (\forall v'' \in V, st'(v'') = I))) \wedge \\ (v \neq r \implies \forall v'' \in V, (v'' = v) \vee st(v'') = st'(v'')) \wedge \\ (v' = v = r \vee v' \in E(v) \vee v \in E(v')) \wedge \\ (v \in E(v') \implies st'(v) \neq I) \wedge \\ (v' \in E(v) \implies st(v) = st'(v) = st(v') = I) \wedge \\ (v' \in E(v) \implies \forall v'' \in E(v), st(v'') \neq I \vee v' \leq v'') \end{array}\right)$$

  We will refer to v as the active node and v' as the next node while explaining the above. If the root is active and not I, then we reset the status of the tree without changing anything else. In all other cases, only the status of the active node can be updated. The next node is either the root, the child of the active node, or the parent of the active node. If the next node is the parent of the active node, then the next status of the active node will not be I. If the active node is the parent of the next node, then the status of the active node and the next status of the active node are I and the status of the next node is I. If the active node is the parent of the next node, then all children that appear earlier in the sequence of active node's children are not I. These rules ensure that we move through the tree in the appropriate order.

Let $[a_0,a_1,...]$ be a sequence of inputs from Σ_{SBT}. Then a SBT trace is a sequence $[(v_0,st_0,s_0),(v_1,st_1,s_1),...]$ such that $v_0 = r$, $s_0 = s_{SBT}$, $\forall v \in V, st_0(v) = I$, and $\forall j \in \mathbb{Z}, j \geq 0 \implies (v_{j+1},st_{j+1},s_{j+1}) \in \delta_{SBT}(v_j,st_j,s_j,a_j)$.

4.2 Translating Stateful Behavior Trees to Finite State Machines

Assuming that SBT has a finite alphabet set and a finite set of states, we will translate it into a nondeterministic Finite State Machine (FSM). A FSM is a tuple $(S_{FSM},s_{FSM},\Sigma_{FSM},\delta_{FSM})$.

- S_{FSM} is a set of states and $s_{FSM} \in S_{FSM}$ is the initial state.
- Σ_{FSM} is a set of possible inputs.
- $\delta_{FSM} : S_{FSM} \times \Sigma_{FSM} \mapsto 2^{S_{FSM}}$ is the transition function.

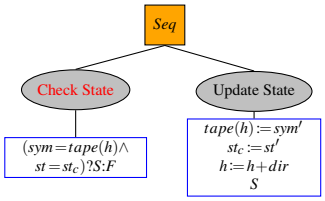


Figure 2: Assume that $f(sym,st)=(sym',st',dir)$, where f is the transition function for the Turing Machine. Then this means that if the Turing Machine is in state st and reads sym from the tape head, it will write sym' to the tape head, transition to st', and move the tape head according to dir. The subtree captures this behavior.

Let $[a_0,a_1,...]$ be a sequence of inputs from Σ_{FSM}. Then a *FSM* trace is a sequence $[s_0,s_1,...]$ such that $s_0=s_{FSM}$ and $\forall j\in\mathbb{Z}, j\geq 0 \implies s_{j+1}\in\delta_{FSM}(s_j,a_j)$.

Let $SBT=(V,r,E,S_{SBT},s_{SBT},\Sigma_{SBT},\delta_{SBT})$. Assume S_{SBT} and Σ_{SBT} are finite. Furthermore, let ST be the set of all functions $st: V\mapsto\{S,R,F,I\}$ (as defined earlier). Since V is a finite set, ST is a finite set. Then $V\times S_{SBT}\times ST$ is finite as well. Let n be the number of elements in $V\times S_{SBT}\times ST$. Create a one-to-one mapping $BTSM$ from $V\times S_{SBT}\times ST$ to the integer interval $[0,1,\ldots,n-1]$. Let $\delta_{FSM}:[0,1,...,n-1]\times\Sigma_{SBT}\mapsto 2^{[0,1,\ldots,n-1]}$ be defined such that

$$\forall v\in V,\forall s\in S_{SBT},\forall st\in ST,\forall a\in\Sigma_{SBT},\forall(v',s',st')\in\delta_{SBT}(v,s,st,a)$$
$$BTSM(v',s',st')\in\delta_{FSM}(BTSM(v,s,st),a).$$

Then $([0,1,\ldots,n-1],BTSM(r,s_{SBT},st_0),\Sigma_{SBT})$ is an equivalent *FSM*, where $st_0\in ST$ such that $\forall v\in V, st_0(v)=I$. By translating the *SBT* to a *FSM*, we allow for verification with tools such as nuXmv [5].

Turing Complete In our translation, we assumed that S_{SBT} was finite. If the blackboard can store one or more infinite variables (e.g. true integers), then S_{SBT} is not finite and *SBT*s are Turing Complete. To see this, consider the following:

- While the 'tape' of a Turing Machine (TM) is infinite, the alphabet of symbols that can appear in each cell is finite. Assume that there are n such symbols. Then the 'tape' can clearly be represented as an integer in base n where each cell is represented by a digit.
- There are finitely many states that the Turing Machine can be in. Thus, these can be enumerated and stored using a finite integer.
- The location of the 'tape head' can be stored using an unbounded integer.

Thus by storing two unbounded integers and a bounded integer in the blackboard, we can fully capture the state and tape of a TM. All that remains is to reproduce the transition function of the TM. Because the transition function is a function from the set of finite states and finite alphabet to the set of finite states, finite alphabet, and a tape motion (Left, Right), it can easily be captured using a BT. For each possible input to the transition function that has a defined output, create a 3-node subtree as seen in Figure 2. Finally, add a selector root node with the 3-node subtrees as children.

Turing Incomplete A TM that only has access to a finite tape is not Turing Complete. Similarly, if we restrict the blackboard to storing finitely many finite variables, the *SBT* ceases to be Turing Complete, as seen by the fact that a translation to a *FSM* exists.

5 DSL and Implementation Details

In this section, we provide details regarding the BehaVerify DSL used to specify *SBT*s. The grammar presented in Grammar 1 differs from the actual DSL (see [1]) for the following reasons:

1. **Syntactic sugar.** For example, the actual DSL allows for fixed size arrays. In practice, this is equivalent to utilizing a large number of variables, but is more convenient (especially when combined with some basic loop functions).
2. **Visual divisions.** The actual DSL uses far more pronounced visual dividers between sections (e.g., a case result is written as 'case { Code } result { Code, Code, ... }'). This ensures that a specification written using the implementation is readable. However, it also injects a great deal of 'text' into the grammar of the DSL, making it more difficult to parse here.
3. **Other features.** The actual DSL is still actively being developed for new features, many of which are not relevant to this paper (e.g., hyperproperties). Such features were omitted.
4. **Format.** The actual DSL is written for use with textX [11] rather than with Backus-Naur Form.

⟨*SBT*⟩ ::= ⟨*Enums*⟩ ';' ⟨*Consts*⟩ ';' ⟨*BLVars*⟩ ';' ⟨*ENVVars*⟩ ';' ⟨*Env*⟩ ';' ⟨*Chks*⟩ ';' ⟨*Acts*⟩ ';' '{' ⟨*Node*⟩ '}' ';' ⟨*Specs*⟩ ';'

⟨*Status*⟩ ::= 'success' | 'running' | 'failure'

⟨*Code*⟩ ::= ⟨*Int*⟩ | ⟨*Boolean*⟩ | ⟨*String*⟩ | ⟨*ID*⟩ | '(' ⟨*Function*⟩ ')'
#ID is used to reference variables, constants, etc

⟨*CodeList*⟩ ::= ⟨*Code*⟩ | ⟨*Code*⟩ ',' ⟨*CodeList*⟩

⟨*Function*⟩ ::= 'add' ',' ⟨*Code*⟩ ',' ⟨*CodeList*⟩ | 'not' ',' ⟨*Code*⟩ | ...

⟨*Enums*⟩ ::= ε | '{' ⟨*String*⟩ '}' ⟨*Enums*⟩ #ε is the empty string

⟨*Const*⟩ ::= ⟨*ID*⟩ ':=' ⟨*Int*⟩ | ⟨*ID*⟩ ':=' ⟨*Boolean*⟩ | ⟨*ID*⟩ ':=' ⟨*String*⟩

⟨*Consts*⟩ ::= ε | '{' ⟨*Const*⟩ '}' ⟨*Consts*⟩ #ε is the empty string

⟨*Domain*⟩ ::= 'BOOLEAN' | '[' ⟨*Code*⟩ ',' ⟨*Code*⟩ ']' | '{' ⟨*CodeList*⟩ '}'
#Code is used to allow constants and expressions

⟨*CaseResult*⟩ ::= ⟨*Code*⟩ '?' ⟨*CodeList*⟩ #If Case (left), then Result (right).
#Choose nondeterministically if multiple Results

⟨*Assign*⟩ ::= '{' ⟨*CaseResult*⟩ '}' ⟨*Assign*⟩ | '{' ⟨*CodeList*⟩ '}'
#Try each CaseResult until one works. Default to CodeList if all fail

⟨*RCaseResult*⟩ ::= ⟨*Code*⟩ '?' ⟨*Status*⟩
#Same as CaseResult but for status. Deterministic.

⟨*RAssign*⟩ ::= '{' ⟨*RCaseResult*⟩ ',' ⟨*RAssign*⟩ '}' | '{' ⟨*Status*⟩ '}'

[1] https://github.com/verivital/behaverify/blob/main/metamodel/behaverify.tx

⟨*Statement*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*Assign*⟩ #Update variable ID using Assign

⟨*Statements*⟩ ::= ε | ‘{’ ⟨*Statement*⟩ ‘}’ ⟨*Statements*⟩ #ε is the empty string

⟨*Var*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*Domain*⟩ ‘,’ ⟨*Assign*⟩

⟨*BLVars*⟩ ::= ε | ‘{’ ⟨*Var*⟩ ‘}’ ⟨*BLVars*⟩ #ε is the empty string

⟨*ENVVars*⟩ ::= ⟨*BLVars*⟩

⟨*Envs*⟩ ::= ⟨*Statements*⟩

⟨*Chk*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*Code*⟩ #Code must resolve to a Boolean
#ID is the NodeType. Allows for reusage.

⟨*Chks*⟩ ::= ε | ‘{’ ⟨*Chk*⟩ ‘}’ ⟨*Chks*⟩ #ε is the empty string

⟨*Act*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*Statements*⟩ ⟨*RAssign*⟩ ⟨*Statements*⟩
#ID is the NodeType. Allows for reusage.

⟨*Acts*⟩ ::= ε | ‘{’ ⟨*Act*⟩ ‘}’ ⟨*Acts*⟩ #ε is the empty string

⟨*NodeBody*⟩ ::= ‘`sequence`’ ‘{’ ⟨*Children*⟩ ‘}’ | ‘`selector`’ ‘{’ ⟨*Children*⟩ ‘}’
| ‘`parallel one`’ ‘{’ ⟨*Children*⟩ ‘}’ | ‘`parallel all`’ ‘{’ ⟨*Children*⟩ ‘}’
| ‘`inverter`’ ‘{’ ⟨*Node*⟩ ‘}’ | ‘`X_is_Y`’ ⟨*Status*⟩ ⟨*Status*⟩ ‘{’ ⟨*Node*⟩ ‘}’
| ⟨*ID*⟩ ‘:=’ ⟨*ID*⟩ #Name of leaf node (left) and NodeType (right)

⟨*Node*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*NodeBody*⟩

⟨*Children*⟩ ::= ‘{’ ⟨*Node*⟩ ‘}’ | ‘{’ ⟨*Node*⟩ ‘}’ ⟨*Children*⟩

⟨*Spec*⟩ ::= ‘`LTL`’ ‘{’ ⟨*Code*⟩ ‘}’ | ‘`CTL`’ ‘{’ ⟨*Code*⟩ ‘}’ | ‘`Invar`’ ‘{’ ⟨*Code*⟩ ‘}’

⟨*Specs*⟩ ::= ε | ⟨*Spec*⟩ ⟨*Specs*⟩ #ε is the empty string

Grammar 1: Representation of BehaVerify DSL with slight changes. We avoid defining basic types such as Int or ID (a letter followed by letters or digits).

Note that both Grammar 1 and the actual DSL allow for nonsensical statements (e.g. (add, 1, 'dog')). It is possible to create a grammar to exclude such cases, but the practical implementation proved both cumbersome to maintain and slow in practice. Instead, we made a semantic checker for basic type checking along with other cases not covered by the grammar structure (e.g. ensuring that identifiers are unique). Listing 1 is an example of how Grammar 1 would be used to create the *SBT* in Figure 4.

Listing 1: A basic example of a *SBT* specified using Grammar 1

```
;;//no enumerations or constants
{x, [(neg, 1), 5], {0, 1}};
//^Bl var x, nondeterministically initialized to 0 or 1
{y, {0, 1}, {(eq, x, 0) ? 0} {1}};
//^Env var y, initialized to 0 if x is 0, otherwise 1
{y, {(eq, y, 0) ? 1} {0}};//between ticks, swap y value
{cN, (geq, (add, x, y), 3)};
```

```
//^declare check cN, checks if x+y is more than 3
{aN1, {x, {(add, x, 1)}}, {success}}
//^declare action aN1, adds 1 to x (; only on last action)
{aN2, {x, {(sub, y, 1)}}, {success}};
//^declare action aN2, set x to y-1
{a, sequence {{b := aN1}{c := cN}{d := aN2}}};
//^tree structure
;//no specifications
```

Specifications The user may write specifications for the *SBT* using Linear Temporal Logic (LTL), Computational Tree Logic (CTL), or using Invariants over first order logic with standard connectives (and, or, etc.). In the case of LTL and CTL, temporal functions may be used that are otherwise unavailable. We provide a brief overview on LTL, as we use it in Section 7 to specify the desired outcomes and confirm that they occur (or provide a violation).

LTL operates on traces (sequences of states). Let $tr = [s_0, s_1, s_2, ...]$ be a trace for a *FSM*. When considering such a trace, time t refers to s_t. In general, we are interested in whether an LTL formula is true for the entire trace; this is the same as asking if the LTL formula is true at time 0. Below we provide a minimal Grammar 2 and then an overview of the presented functions and some syntactic sugar.

$\langle LTL \rangle ::= \langle a \rangle$ #First Order Logic Formula
| $\neg \langle LTL \rangle \mid \langle LTL \rangle \vee \langle LTL \rangle$ #Boolean operators; in practice we allow more operators
| $\bigcirc(\langle LTL \rangle) \mid (\langle LTL \rangle) \mathscr{U} (\langle LTL \rangle)$ #Temporal operators next and until

Grammar 2: Minimal LTL Grammar.

- $\bigcirc(\phi)$ (next) is true at time t if ϕ is true at time $t+1$.
- $(\phi_1) \mathscr{U} (\phi_2)$ (until) is true at time t if $\exists t''$ such that $t \leq t''$ and ϕ_2 is true at t'' and $\forall t'$ such that $t \leq t' < t''$, ϕ_1 is true at t'.
- $(\phi_1) \mathscr{M} (\phi_2)$ (strong release) is true at time t if $\exists t''$ such that $t \leq t''$ and ϕ_2 is true at t'' and $\forall t'$ such that $t \leq t' \leq t''$, ϕ_1 is true at t'.
- $\square(\phi)$ (globally) is true at time t if $\forall t'$ such that $t \leq t'$, ϕ is true at time t'.
- $\Diamond(\phi)$ (finally) is true at time t if $\exists t'$ such that $t \leq t'$, ϕ is true at time t'.

6 Fastforwarding Execution

Recall the example execution in Figure 1. The execution presented was intuitive and clear, but also highlighted a clear drawback: it took 14 time steps to complete 2 ticks in a 5 node tree. This is not ideal for verification. The total encoding, presented in our previous paper [25], was created to address this issue. The experiments conducted in [25] clearly demonstrated the performance concerns associated with stepping through nodes one at a time and demonstrated that the total encoding is an effective method by which to mitigate this. However, that encoding required the user to edit the resulting model by hand: an arduous task requiring not only expertise in nuXmv but in how BehaVerify encoded the model. We have now addressed this issue (the user need only provide a specification file; everything else is handled automatically) and will explain our solution below.

Fastforwarding and a review of the total encoding The goal is to handle the entire tick in one step, rather than stepping through each node one at a time. To that end, the total encoding represented the status of each node as a function of other nodes. This is shown in Figure 3. Both the encoding and the process of automatically creating the total encoding were part of our prior work. However, our prior work required the user to manually handle the creation of functions such as $user_B$ in nuXmv, along with creating appropriate variable updates in nuXmv. These limitations were the result of:

1. A lack of a DSL. Our prior work handled existing Py Tree objects to create the tree. Unfortunately, this was not conducive to specifying how leaf nodes behave.
2. The complexity of variables in a total encoding. Suppose the variable x is 0 at the start of the tick and 1 at the end; what value of x does the function $user_C$ use?

This is where our new work comes into play. The user uses our DSL to specify leaf nodes, defining what status they return and how they change variables, and BehaVerify takes that information and automatically creates an improved total encoding allowing for the fastforwarding of execution, complete with variable updates and functions for leaf nodes, no additional input from the user required. BehaVerify resolves the issue with variables through the use of variable stages. Each variable has at least one stage. For each possible update to a variable, an additional stage is created representing the variable after the update. Thus, the number of stages a variable has is equal to the number of possible updates to that variable during a single tick plus one. This can be seen in Figure 4. Each stage describes the value of a variable during a portion of the tick; BehaVerify tracks which stage the variable is in and references the appropriate stage in other functions, thus resolving the issue posed above.

Specification Writing We found that fastforwarding often simplifies specification writing. Consider Figure 4. Suppose we want to write that during each tick, (d) returns S (note that this specification is false; during the first tick (d) is I). With fastforwarding, this can be written as an invariant condition, namely $status(d) = S$. If we are not using fastforwarding, we must write this using LTL

$$\square(status(a) = I \implies ((status(a) = I) \mathscr{U} (status(d) = S))).$$

This specification is far more complicated, because we now have to define the duration of a tick. In this case, we accomplish this task by realizing that since (a) is the root, the tick ends when (a) returns. As such, this specification says that it is always the case that if (a) is I, then (a) will stay I until (d) returns S.

Note that sequential properties can also be written with fastforwarding, though may require a little more forethought. For instance, suppose we want to specify that (b) eventually returns S and until that

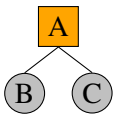
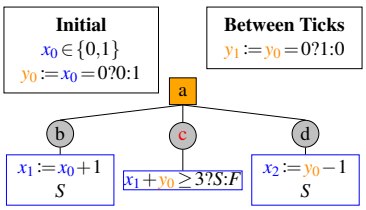


Figure 3:
(L) 3 node tree.
(R) total encoding.
Note: $x?y:z$ means if x, then y, else z.

- act is a function that describes if a node is active or not during a given tick. $status$ is a function that describes the status of a node during a given tick.
- $act(A) \triangleq \top$, $act(B) \triangleq act(A)$, $act(C) \triangleq act(A) \wedge (status(B) = S)$
- $status(A) \triangleq \begin{cases} I \text{ if } \neg act(A) \\ F \text{ if } status(B) = F \vee status(C) = F \\ R \text{ if } status(B) = R \vee status(C) = R \\ S \text{ if } status(B) = S \wedge status(C) = S \end{cases}$
- Let $user_B$ be a function specified by the user that depends on variables and outputs a status. Then $status(B) \triangleq act(B)?user_B(vars):I$.
- Let $user_C$ be a function specified by the user that depends on variables and outpus a status. Then $status(C) \triangleq act(C)?user_C(vars):I$.

Tick	1	1	1	1	1	2	2	2	2	2	2	2	2
t	0	1	2	3	4	5	6	7	8	9	10	11	12
Active	a	b	a	c	a	a	b	a	c	a	d	a	a
Returns	-	*S*	-	*F*	*F*	-	*S*	-	*S*	-	*S*	*S*	-
x	0	1	1	1	1	1	2	2	2	2	1	1	1
y	0	0	0	0	0	1	1	1	1	1	1	1	0



Tick	1	2
a	*F*	*S*
b	*S*	*S*
c	*F*	*S*
d	*I*	*S*
x_0	0	1
x_1	1	2
x_2	1	1
y_0	0	1
y_1	1	0

Figure 4: Here x is a blackboard variable and y is an environment variable (an input). The input changes between ticks, so y_1 does not appear in the tree. The upper table shows what stepping through the tree one node at a time would look like. The lower table shows what fastforwarding looks like. The tree itself includes stage subscripts that were added for readability; the user would normally create a tree without knowledge of the stages, as BehaVerify handles stage creation. Note that when fastforwarding, the value of x_2 is set to that of x_1 if (d) does not execute. Furthermore, note that the value of x at the end of the tick (x_2) is the value of x at the start of the next tick (x_0).

happens, (c) does not returns *S*. Since (b) occurs before (c) in the tree, this specification can be written as $status(c) \neq S \mathscr{U} status(b) = S$. The reverse, namely that (c) eventually returns *S* and until that happens (b) does not return *S*, is slightly trickier. One might be tempted to write $status(b) \neq S \mathscr{U} status(c) = S$, but this accepts the case where (b) and (c) both return *S* for the first time on the same tick. To account for this, one should use $\mathscr{M}$ instead of $\mathscr{U}$. In general, this example illustrates the point well; it is entirely possible to describe sequential events when writing specifications for fastforwarded execution, but one must be mindful of the structure of the tree. Finally, it is important to note that some sequential specifications are trivially true, and their verification is not a major objective for BehaVerify. For instance, if (b) and (c) are both active during a tick, then because (b) is before (c) in a depth first traversal of the tree, node (b) was active before node (c). BehaVerify does generate a list of nodes in order of depth first traversal; if one wishes to confirm such specifications, they may consult this order.

Optimizations An immediate concern raised by introducing stages is how this effects model size. After all, if we went from storing a single variable with 10 states to storing 3 variables with 10 states each, then the model is now spending 1000 states on this single variable. Fortunately, through optimizations, we can generally avoid this issue. Specifically, if a variable update is deterministic, then the stage is a function of the previous stage and doesn't increase the model size. Furthermore, even if an update is nondeterministic, we can sometimes avoid an increase in model size. Consider a very simple model with one variable, x, which is updated nondeterministically twice per tick. Then, without optimizations, we would have three stages x_0, x_1, and x_2, and each stage would increase the size of the model. Here x_0 is the value of x at the start of the tick, x_1 the value after the first update, and x_2 the value after the second update. Note that the next value of x_0 is the current value of x_2. This is true for all variables; the next value of stage 0 is equal to the current value of the last stage. Furthermore, each other stage depends only on current values. In this example, since nothing depends on the value of x_2 other than the next value of x_0, we can safely remove the last stage and simply make it so the next value of x_0 is the value that x_2 would have been assigned. It is important to note that if a user writes a specification that checks the value of x at the end of a tick, this optimization *would* change the result; we detect such cases and automatically disable the optimization for the variable. Furthermore, if another variable or node depends on the value of x_2, this optimization would

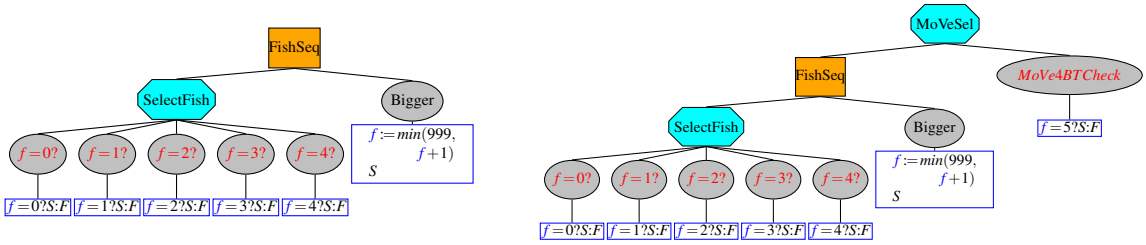


Figure 5: Left is original, right is reworked for MoVe4BT. Number of nodes is counted using right tree (10 in this case). f is the size of the 'biggest fish' so far. Each check returns S if the f is equal to its number and F otherwise. If any check returns S, the left half of the tree returns S and f is incremented.

change the result and would therefore be disabled. In such cases, we can instead try to combine the 0 stage with the 1 stage using a similar process, with similar caveats. We can attempt this with the first and last stage of each variable.

7 Verification Results for Stateful Behavior Trees

Here we present formal verification results and include comparisons to MoVe4BT [23] that demonstrate BehaVerify scales significantly better in the size of the tree and overall state space. Additionally, we present an interesting example demonstrating that BehaVerify is capable of finding complex counterexamples. All results were generated on a Dell Inc. OptiPlex 7040 with 64 GiB of Memory with an Intel i7–6700 CPU @ 3.40GHz with 8 cores. The code used and instructions for reproducing the results are available [2].

We note here that BehaVerify takes a specification file written using the DSL and produces a nuXmv model. The timing results for BehaVerify are based *solely* on the time it takes nuXmv to run on the generated model. We do not include the time it takes for BehaVerify to generate the nuXmv model. This is because we wanted to compare our encoding to that of the competing tool, MoVe4BT, and we felt this was best demonstrated through a comparison of the model checking aspect. However, we also note that the compile times do not meaningfully change the outcome of the results; compiling the simple robot experiment takes fractions of a second even with a 30 by 30 grid and the same is true for the bigger fish experiment with 1000 nodes.

Scaling Tree Experiment: Bigger Fish The bigger fish experiment (see Figure 5) scales the tree while the blackboard and environment are unchanged. f is an integer between 0 and 1000 inclusive and is initially 0. The upper limit was increased for the tests on 10000 and 20000 nodes. We intended to verify $\Diamond(\Box(f=n))$ (variable attains and maintains a value), but MoVe4BT does not support LTL specifications over variables; you are restricted to checking the statuses of nodes. We modified the tree and created a node (named *MoVe4BTCheck*) to check the value of the variable (see Figure 5 for details). We then tried to verify $\Diamond(\Box(MoVe4BTCheck=S))$. In BehaVerify, this specification states that eventually, during each tick the check returns S. However, because MoVe4BT does not utilize fastforwarding (see Section 6 for details), MoVe4BT interprets this to mean that eventually the check is always active, which is false. We instead had to settle for verifying $\Diamond(MoVe4BTCheck=S)$. This means that instead of verifying that the variable attains the value and maintains it, MoVe4BT only verifies that it attains it. It is crucial to

[2] https://github.com/verivital/behaverify/tree/main/REPRODUCIBILITY/2024_FMAS_SBT

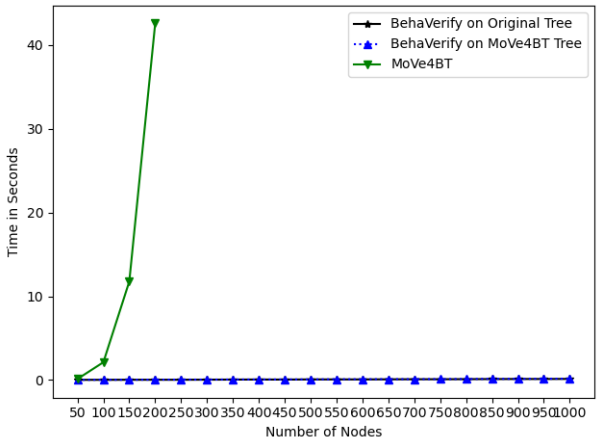


Figure 6: Time to verify $\Diamond(\Box(f=n))$ (original) and $\Diamond(f=n)$ (changed) where n is the number of nodes minus 5. Starting at 250 nodes total, MoVe4BT ran for over a minute, producing a blank screen with no results; we interpret this as a timeout. We ran BehaVerify with 10000 and 20000 nodes, taking 8.20 and 32.32 seconds. At 200 and 20000 nodes, BehaVerify reported 196 and 19996 reachable states. BehaVerify reports similar runtimes for both trees.

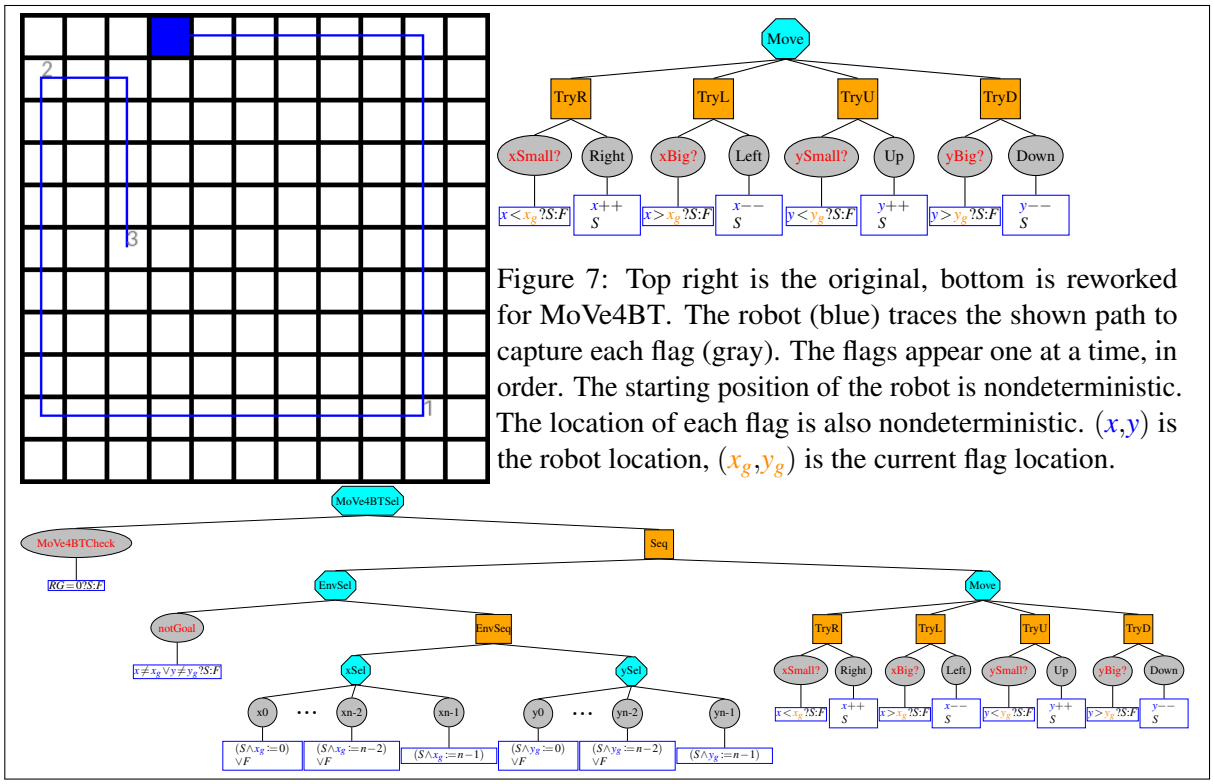


Figure 7: Top right is the original, bottom is reworked for MoVe4BT. The robot (blue) traces the shown path to capture each flag (gray). The flags appear one at a time, in order. The starting position of the robot is nondeterministic. The location of each flag is also nondeterministic. (x,y) is the robot location, (x_g,y_g) is the current flag location.

stress that BehaVerify is capable of verifying the original condition; in fact, we include timing results for both the original and new condition. Figure 6 shows BehaVerify scales well with tree complexity, while MoVe4BT does not. The specification is true. If the model is changed so it is false (by removing a leaf in the chain), both tools produce a counterexample.

Scaling Blackboard Experiment: Simple Robot The simple robot experiment (see Figure 7) scales the blackboard while the tree is constant. A robot on a n by n board tries to reach a goal. Once reached, a new goal is generated. We verify that eventually 3 goals are reached. The experiment scales by increasing n from 2 to 30 in increments of 4. We compare to MoVe4BT for this experiment. MoVe4BT has no

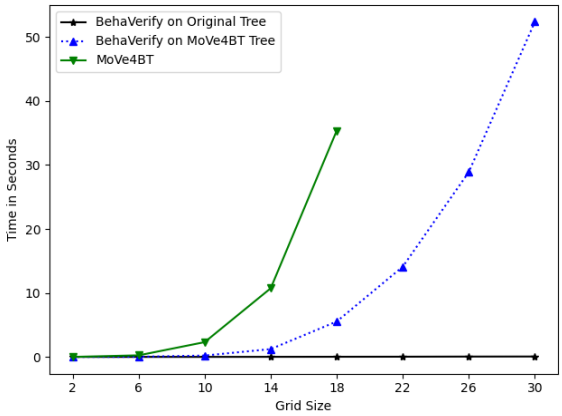


Figure 8: Time in seconds to verify that finally, there are no remaining goals: $\Diamond(RG = 0)$. Our tool ran on both the original tree and the changed tree. Starting at a 22 by 22 grid, MoVe4BT ran for over a minute and then produced a blank screen; we interpreted this as a timeout. At 18 by 18, BehaVerify reported about 2^{19} reachable states. At 30 by 30, BehaVerify reported about 2^{22} reachable states.

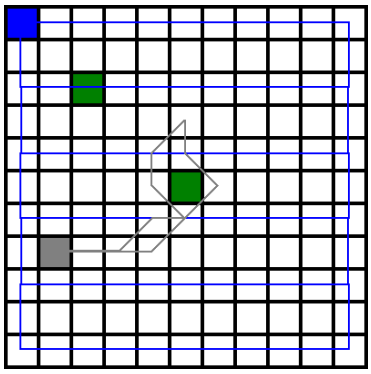


Figure 9: The target (gray) is able to avoid the drone (blue) by circling a tree (green) that obstructs vision but not movement. The target's initial position and movement are nondeterministic; the drone's are not. Note that this was a counterexample generated by nuXmv from the generated model.

concept of an environment so the environment has to become part of the *BT*. Furthermore, MoVe4BT cannot assign values to variables nondeterministically; instead MoVe4BT nodes can nondeterministically choose one of *S*, *R*, *F*, and a single accompanying 'program' with updates. Thus to achieve the effect of nondeterminism, we had to use a series of nodes. Each node could nondeterministically choose to change the value or not. Thus the tree is more complex and also grows in size with the blackboard. Finally, we had to introduce an extra node (called *MoVe4BTCheck*) so that MoVe4BT could verify the condition at all. Refer to Figure 7 for a visual comparison of the trees. See Figure 8 for timing results.

Moving Target Experiment For this experiment, a drone is searching for a mobile target on a grid. The drone has limited vision, which is further obscured by trees (the trees do not prevent movement). We conduct two experiments for this setup: one where the target can move every 5 turns and one where the target can move every 10 turns. When the target can move every 10 turns, we verify that eventually the drone finds the target. When the target can move every 5 turns, we generate a counterexample (see Figure 9). In both cases, the specification being verified is that we eventually 'see' the target. Given the complexity of encoding the simple robot environment for MoVe4BT and the fact that this experiment is more complex and nondeterministic, we did not attempt to recreate it in MoVe4BT.

Reasoning About Results We believe fastforwarding (see Section 6) is the main reason BehaVerify outperforms MoVe4BT. Based on the traces that MoVe4BT produces, we believe that MoVe4BT jumps from leaf to leaf during execution. This provides a performance boost over stepping through every single node in the tree, but it is not enough. Consider the bigger fish experiment. With 20000 nodes,

there are 19997 leaf nodes. During the first tick, 2 leaf nodes will be active. During the second tick, there will be 3. Finally, there will be 19996 active leafs. Thus, MoVe4BT would have to step through $2+3+\cdots+19996=199{,}930{,}005$ leaf nodes resulting in a very long trace. By comparison, the BehaVerify trace would have less than 20000 states. Next, we consider the simple robot experiment, where changing the tree had a huge impact. This is because MoVe4BT does not support nondeterministic variable assignments; as such we had to create a series of nondeterministic nodes. Thus instead of having one nondeterministic update for x and one for y, we had $n-1$ for each. This caused the number of total states to jump from 2^{23} to 2^{80} while the number of reachable states was relatively unchanged. The changed tree is close to a worst case scenario for BehaVerify; many variable updates, all of them nondeterministic. Even in this worst case scenario, BehaVerify significantly outperformed MoVe4BT.

8 Conclusions and Future Work

We introduced and formally defined *SBT*s and demonstrated they are Turing Complete under certain assumptions. We presented a DSL for specifying *SBT*s implemented in BehaVerify. Our experiments demonstrate BehaVerify can complete a verification task on a tree with 20000 nodes in the time that MoVe4BT, a different verification tool, verifies a property on a tree with 200 nodes, demonstrating two orders of magnitude scalability improvement. Potential future work includes improving our encoding of array variables, developing a graphical user interface for the creation of *SBT*s, visualization of counterexamples provided by nuXmv, general performance improvements, and expanding support for unbounded variable types in nuXmv for bounded model checking (BMC).

Acknowledgements

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) through grant numbers 2220426 and 2220401, the Defense Advanced Research Projects Agency (DARPA) under contract number FA8750-23-C-0518, and the Air Force Office of Scientific Research (AFOSR) under contract numbers FA9550-22-1-0019 and FA9550-23-1-0135. This paper was also supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR), and the Army Research Office (ARO). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of AFOSR, DARPA, or NSF.

References

[1] Auryn Robotics: *Tutorial 02: Blackboard and Ports*. Available at https://www.behaviortree.dev/docs/tutorial-basics/tutorial_02_basic_ports.

[2] Oliver Biggar & Mohammad Zamani (2020): *A Framework for Formal Verification of Behavior Trees With Linear Temporal Logic*. *IEEE Robotics and Automation Letters* 5(2), pp. 2341–2348, doi:10.1109/LRA.2020.2970634.

[3] Oliver Biggar, Mohammad Zamani & Iman Shames (2020): *A principled analysis of Behavior Trees and their generalisations*. *CoRR* abs/2008.11906, doi:10.48550/arXiv.2008.11906. arXiv:2008.11906.

[4] Oliver Biggar, Mohammad Zamani & Iman Shames (2021): *An Expressiveness Hierarchy of Behavior Trees and Related Architectures*. *IEEE Robotics and Automation Letters* 6(3), pp. 5397–5404, doi:10.1109/lra.2021.3074337.

[5] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri & Stefano Tonetta (2014): *The nuXmv Symbolic Model Checker*. In: *CAV*, pp. 334–342. Available at http://dx.doi.org/10.1007/978-3-319-08867-9_22.

[6] Michele Colledanchise, Giuseppe Cicala, Daniele E. Domenichelli, Lorenzo Natale & Armando Tacchella (2021): *Formalizing the Execution Context of Behavior Trees for Runtime Verification of Deliberative Policies*. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE Press, pp. 9841–9848, doi:10.1109/IROS51168.2021.9636129.

[7] Michele Colledanchise & Petter Ögren (2014): *How Behavior Trees modularize robustness and safety in hybrid systems*. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1482–1488, doi:10.1109/IROS.2014.6942752.

[8] Michele Colledanchise & Petter Ögren (2016): *How Behavior Trees generalize the Teleo-Reactive paradigm and And-Or-Trees*. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 424–429, doi:10.1109/IROS.2016.7759089.

[9] Michele Colledanchise & Petter Ögren (2017): *How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees*. *IEEE Transactions on Robotics* 33(2), pp. 372–389, doi:10.1109/TRO.2016.2633567.

[10] Alessio De Luca, Luca Muratore & Nikos G. Tsagarakis (2023): *Autonomous Navigation With Online Replanning and Recovery Behaviors for Wheeled-Legged Robots Using Behavior Trees*. *IEEE Robotics and Automation Letters* 8(10), pp. 6803–6810, doi:10.1109/LRA.2023.3313052.

[11] I. Dejanović, R. Vaderna, G. Milosavljević & Z. Vuković (2017): *TextX: A Python tool for Domain-Specific Languages implementation*. *Knowledge-Based Systems* 115, pp. 1–4, doi:10.1016/j.knosys.2016.10.023. Available at https://www.sciencedirect.com/science/article/pii/S0950705116304178.

[12] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard & Henrich Lauko (2022): *From Spot 2.0 to Spot 2.10: What's New?* In: *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, *Lecture Notes in Computer Science* 13372, Springer, pp. 174–187, doi:10.1007/978-3-031-13188-2_9.

[13] EpicGames (2021): *Behavior tree overview*. Available at https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/.

[14] Sergio García, Patrizio Pelliccione, Claudio Menghi, Thorsten Berger & Tomas Bures (2020): *PROMISE: High-Level Mission Specification for Multiple Robots*. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, Association for Computing Machinery, New York, NY, USA, pp. 5–8, doi:10.1145/3377812.3382143.

[15] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski & Swaib Dragule (2023): *Behavior Trees and State Machines in Robotics Applications*. *IEEE Transactions on Software Engineering*, pp. 1–24, doi:10.1109/TSE.2023.3269081.

[16] Zhaoyuan Gu, Nathan Boyd & Ye Zhao (2022): *Reactive Locomotion Decision-Making and Robust Motion Planning for Real-Time Perturbation Recovery*. In: *2022 International Conference on Robotics and Automation (ICRA)*, pp. 1896–1902, doi:10.1109/ICRA46639.2022.9812068.

[17] Thomas Henn, Marcus Völker, Stefan Kowalewski, Minh Trinh, Oliver Petrovic & Christian Brecher (2022): *Verification of Behavior Trees using Linear Constrained Horn Clauses*. In Jan Friso Groote & Marieke Huisman, editors: *Formal Methods for Industrial Critical Systems*, Springer International Publishing, Cham, pp. 211–225, doi:10.1007/978-3-031-15008-1_14.

[18] Qian Huang, Xianming Ma, Kun Liu, Xinyi Ma & Weijian Pang (2022): *Autonomous Reconnaissance Action of Swarm Unmanned System Driven by Behavior Tree*. In: *2022 IEEE International Conference on Unmanned Systems (ICUS)*, pp. 1540–1544, doi:10.1109/ICUS55513.2022.9986758.

[19] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren & Christian Smith (2022): *A survey of Behavior Trees in robotics and AI*. *Robotics and Autonomous Systems* 154, p. 104096, doi:10.1016/j.robot.2022.104096.

[20] Seungwoo Jeong, Taekwon Ga, Inhwan Jeong & Jongeun Choi (2022): *Behavior Tree-Based Task Planning for Multiple Mobile Robots using a Data Distribution Service*. In: *2022 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, pp. 1791–1798, doi:10.1109/AIM52237.2022.9863364.

[21] Alejandro Marzinotto, Michele Colledanchise, Christian Smith & Petter Ögren (2014): *Towards a unified behavior trees framework for robot control*. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5420–5427, doi:10.1109/ICRA.2014.6907656.

[22] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[23] Huang Peishan, Hong Weijiang, Chen Zhenbang & Wang Ji: *MoVe4BT: Modeling & Verification For BT*. Available at https://move4bt.github.io/. Accessed: 2023-12-14.

[24] Fangbo Qin, De Xu, Blake Hannaford & Tiantian Hao (2023): *Object-Agnostic Vision Measurement Framework Based on One-Shot Learning and Behavior Tree*. *IEEE Transactions on Cybernetics* 53(8), pp. 5202–5215, doi:10.1109/TCYB.2022.3181054.

[25] Serena S. Serbinowska & Taylor T. Johnson (2022): *BehaVerify: Verifying Temporal Logic Specifications For Behavior Trees*. In: *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, Springer-Verlag, Berlin, Heidelberg, pp. 307–323, doi:10.1007/978-3-031-17108-6_19.

[26] Christopher I. Sprague & Petter Ögren (2022): *Continuous-Time Behavior Trees as Discontinuous Dynamical Systems*. *IEEE Control Systems Letters* 6, pp. 1891–1896, doi:10.1109/LCSYS.2021.3134453.

[27] Daniel Stonier: *PyTrees Module API*. Available at https://py-trees.readthedocs.io/en/devel/modules.html. Accessed: 2023-12-14.

[28] Petter Ögren (2020): *Convergence Analysis of Hybrid Control Systems in the Form of Backward Chained Behavior Trees*. *IEEE Robotics and Automation Letters* 5(4), pp. 6073–6080, doi:10.1109/LRA.2020.3010747.

[29] Petter Ögren & Christopher I. Sprague (2022): *Behavior Trees in Robot Control Systems*. *Annual Review of Control, Robotics, and Autonomous Systems* 5(Volume 5, 2022), pp. 81–107, doi:10.1146/annurev-control-042920-095314.