Leibniz Transactions on Embedded Systems

Volume 8 | Issue 2 | December 2022
Special Issue on Distributed Hybrid Systems

Vol. 8, Issue 2 ISSN 2199-2002 https://www.dagstuhl.de/lites

ISSN 2199-2002

Published online and open access by
the European Design and Automation Association (EDAA) / EMbedded Systems Special Interest Group (EMSIG) and Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.
Online available at
`https://www.dagstuhl.de/dagpub/2199-2002`.

Publication date
December 2022

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://dnb.d-nb.de.

Digital Object Identifier
10.4230/LITES-v008-i002

Aims and Scope
LITES aims at the publication of high-quality scholarly articles, ensuring efficient submission, reviewing, and publishing procedures. All articles are published open access, i.e., accessible online without any costs. The rights are retained by the author(s).

LITES publishes original articles on all aspects of embedded computer systems, in particular: the design, the implementation, the verification, and the testing of embedded hardware and software systems; the theoretical foundations; single-core, multi-processor, and networked architectures and their energy consumption and predictability properties; reliability and fault tolerance; security properties; and on applications in the avionics, the automotive, the telecommunication, the medical, and the production domains.

Contact
Schloss Dagstuhl – Leibniz-Zentrum für Informatik
LITES, Editorial Office
Oktavie-Allee, 66687 Wadern, Germany
lites@dagstuhl.de
`http://www.dagstuhl.de/lites`

Contents

Papers

Introduction to the Special Issue on Distributed Hybrid Systems

Alessandro Abate
University of Oxford, UK

Uli Fahrenberg
EPITA Research Laboratory (LRE), Paris, France

Martin Fränzle
Carl von Ossietzky Universität Oldenburg, Germany

This special issue contains seven papers within the broad subject of *Distributed Hybrid Systems*, that is, systems combining hybrid discrete-continuous state spaces with elements of concurrency and logical or spatial distribution. It follows up on several workshops on the same theme which were held between 2017 and 2019 and organized by the editors of this volume.

The first of these workshops was held in Aalborg, Denmark, in August 2017 and associated with the MFCS conference. It featured invited talks by Alessandro Abate, Martin Fränzle, Kim G. Larsen, Martin Raussen, and Rafael Wisniewski. The second workshop was held in Palaiseau, France, in July 2018, with invited talks by Luc Jaulin, Thao Dang, Lisbeth Fajstrup, Emmanuel Ledinot, and André Platzer. The third workshop was held in Amsterdam, The Netherlands, in August 2019, associated with the CONCUR conference. It featured a special theme on distributed robotics and had invited talks by Majid Zamani, Hervé de Forges, and Xavier Urbain.

The vision and purpose of the DHS workshops was to connect researchers working in real-time systems, hybrid systems, control theory, formal verification, distributed computing, and concurrency theory, in order to advance the subject of distributed hybrid systems. Such systems are abundant and often safety-critical, but ensuring their correct functioning can in general be challenging. The investigation of their dynamics by analysis tools from the aforementioned domains remains fragmentary, providing the rationale behind the workshops: it was conceived that convergence and interaction of theories, methods, and tools from these different areas was needed in order to advance the subject.

2012 ACM Subject Classification Computing methodologies → Model verification and validation; Computer systems organization → Embedded and cyber-physical systems; Mathematics of computing → Stochastic processes; Theory of computation → Concurrency; Theory of computation → Distributed computing models; Theory of computation → Logic and verification; Theory of computation → Program reasoning; Theory of computation → Self-organization; Theory of computation → Timed and hybrid models

Keywords and Phrases Distributed hybrid systems

Digital Object Identifier 10.4230/LITES.8.2.0

Acknowledgements The editors acknowledge the *Chaire ISC : Engineering Complex Systems* and École polytechnique, in particular Éric Goubault, for financial support on some of the workshops, out of which this special issue was born, as well as for financing the special issue itself. We also wish to thank the editorial manager Michael Wagner for his excellent work, the authors of the papers in this issue, and the speakers and participants of the three workshops on distributed hybrid systems.

Published 2022-12-07

Editor Alessandro Abate, Uli Fahrenberg, and Martin Fränzle

Special Issue Special Issue on Distributed Hybrid Systems

1 Papers in this special issue

The first paper in this issue, by *Arvind Adimoolam* and *Thao Dang*, lies in the intersection of control theory, hybrid systems, and distributed systems. It is based on the PhD thesis of the first author and on several conference papers of both authors, and builds upon an invited talk by the second author at the DHS 2018 workshop. The work showcases the versatility of complex zonotopes for the analysis of networked control systems. Such systems are subject to network delays, packet dropouts, inaccurate sensing, and quantization errors, all of which have to be taken into account when analysing them. The authors show a result about the existence of complex zonotopic invariants and extend existing algorithms for stability verification of networked control systems.

The second paper, by *Pierre Courtieu*, *Lionel Rieg*, *Sébastien Tixeuil* and *Xavier Urbain*, concerns itself with distributed robotics, an area which itself is in the intersection of control theory and distributed systems. It builds upon an invited talk by the fourth author at the DHS 2019 workshop and presents the Pactole framework for formal modeling and analysis of protocols for mobile robotic swarms. Built on the Coq proof assistant, Pactole provides a uniform modeling language for protocol development which takes into account the many variations between models for distributed robotics and permits to devise certified proofs of possibility and impossibility results.

Article three in this issue, authored by *Uli Fahrenberg*, lies in the intersection of real-time systems and concurrency theory and introduces a new formalism of higher-dimensional timed automata. The interest is in modeling systems which exhibit both real-time behavior and concurrency, which is difficult or impossible in other existing frameworks. The author shows that the standard zone-based methods carry over from timed automata to higher-dimensional timed automata and also extends the model to higher-dimensional hybrid automata.

The fourth paper, by *Eduard Kamburjan*, *Stefan Mitsch* and *Reiner Hähnle*, introduces hybrid active objects for the modeling and analysis of hybrid systems, in order to address the challenges at the intersection of hybrid systems and concurrency. Building upon a talk at the DHS 2019 workshop, the proposed high-level programming-based approach extends the active-objects language ABS with features for modeling hybrid systems. The authors also extend the formal semantics of ABS and its runtime environment and implement a formal verification tool for hybrid ABS.

Article five in this issue, due to *Paul Kröger* and *Martin Fränzle*, builds upon a talk at the DHS 2018 workshop and identifies shortcomings of existing formal models for hybrid systems when it comes to describing the interactive behavior of multiple hybrid-state agents. It demonstrates rigorously that even the most expressive formal models of hybrid-state dynamics cannot precisely cover the emergent joint behavior of rationally acting agents and that existing models are thus bound to either provide significantly pessimistic or significantly optimistic verdicts about the overall system dynamics. Aligned with the overarching goal of developing pertinent theories for the behavioral analysis of distributed hybrid systems, the article proposes a novel model of Bayesian hybrid automata that considerably extends stochastic hybrid automata in order to overcome this deficiency.

The sixth paper, by *Ameneh Nejati* and *Majid Zamani*, concerns itself with networks of stochastic hybrid systems and the construction of control barrier certificates for these. The issue at stake is compositionality, thus, the synthesis of state-feedback controllers for interconnected systems based on certificates computed for subsystems. Building upon an invited talk by the second author at the DHS 2019 workshop, the paper proposes a dissipativity approach which takes into account the structure of the interconnection topology. The authors demonstrate the obtained results on comprehensive case studies.

The final article is titled "Real-Time Verification for Distributed Cyber-Physical Systems" and builds upon a talk at the DHS 2019 workshop. It proposes a real-time decentralised reachability approach for safety verification of a distributed multi-agent CPS, with the underlying assumption that all agents are time-synchronised with a low degree of error. Each agent periodically computes its local reachable set and exchanges this reachable set with the other agents with the goal of verifying the system safety, with local safety verification tasks based on their local clocks by analysing the messages it receives.

2 Conclusion

The fourth workshop on distributed hybrid systems was to be held in Vienna in 2020, associated with the CONCUR conference, but fell victim to the Covid pandemic. Whether the DHS workshop series will be revived in the future, and under which format, is uncertain, so this special journal issue may well be the conclusion of the DHS series.

One may thus rightfully pose the question as to the achievements of the workshops and this special issue, seen as a whole. The first purpose of the DHS workshops was to connect researchers working in real-time systems, hybrid systems, control theory, distributed computing, and concurrency theory, and as this special issue bears witness, that purpose has been achieved.

Another question is whether, through convergence and interaction of methods and tools from these different areas, the workshops have contributed to advance the subject of distributed hybrid systems itself. Many of the papers in this special issue concern themselves with research in the intersection of several of the above-mentioned areas, but it is of course difficult to assess how much the workshops themselves have contributed to these works.

Something that still has to emerge, and perhaps would be too much to expect from just three workshop editions, are new overarching theories which combine the subjects underlying distributed hybrid systems in new ways. The quest for such unifying theories becomes pronounced, given that distributed hybrid systems are at the heart of the recent push towards so-called smart environments, be it "smart cities" as denoting anticipated forms of interconnected intelligent urban structures, or "smart grids", "smart transportation", and "smart health" advancing energy supplies, transportation systems, and medical technology, or "Industry 4.0" revolutionising manufacturing technology. It thus is to be expected that such theories will materialize and will extend and generalize beyond specific domains. The influence of the DHS workshops, which have outlined manifold elements of an overarching theory as witnessed by the articles included in this volume, on the final theories-to-be cannot yet be assessed with full scrutiny.

Safety Verification of Networked Control Systems by Complex Zonotopes

Arvind Adimoolam
Indian Institute of Technology Kanpur, Rajeev Motwani Building, Kalyanpur, Kanpur, India

Thao Dang
VERIMAG, CNRS/University Grenoble Alpes, Bâtiment IMAG, 700 Avenue Centrale, Grenoble, France

Abstract

Networked control systems (NCS) are widely used in real world applications because of their advantages, such as remote operability and reduced installation costs. However, they are prone to various inaccuracies in execution like delays, packet dropouts, inaccurate sensing and quantization errors. To ensure safety of NCS, their models have to be verified under the consideration of aforementioned uncertainties. In this paper, we tackle the problem of verifying safety of models of NCS under uncertain sampling time, inaccurate output measurement or estimation, and unknown disturbance input. Unbounded-time safety verification requires approximation of reachable sets by invariants, whose computation involves set operations. For uncertain linear dynamics, two important set operations for invariant computation are linear transformation and Minkowski sum operations. Zonotopes have the advantage that linear transformation and Minkowski sum operations can be efficiently approximated. However, they can not encode directions of convergence of trajectories along complex eigenvectors, which is closely related to encoding invariants. Therefore, we extend zonotopes to the complex valued domain by a representation called *complex zonotope*, which can capture contraction along complex eigenvectors for determining invariants. We prove a related mathematical result that in case of accurate feedback sampling, a complex zonotope will represent an invariant for a stable NCS. In addition, we propose an algorithm to verify the general case based on complex zonotopes, when there is uncertainty in sampling time and in input. We demonstrate the efficiency of our algorithm on benchmark examples and compare it with a state-of-the-art verification tool.

2012 ACM Subject Classification Computer systems organization → Reliability
Keywords and Phrases Safety Verification; Networked Control System; Reachability Analysis; Invariant; Complex Zonotope
Digital Object Identifier 10.4230/LITES.8.2.1
Received 2020-08-29 **Accepted** 2021-10-15 **Published** 2022-12-07
Editor Alessandro Abate, Uli Fahrenberg, and Martin Fränzle
Special Issue Special Issue on Distributed Hybrid Systems

1 Introduction

The computerized control of spatially distributed components through communication channels is called *networked control*. The emergence of fast and reliable communication networks has enabled efficient networked control of many applications in aerospace, automation, manufacturing and robotics [17, 26, 34, 37, 38]. Advantages of networked control systems (NCS) include reduced installation cost due to absence of wiring and remote operability. However, they are prone to inaccuracies in execution such as delays, packet dropouts and errors in sensing and quantization. To mitigate the risk of system failure, we need to verify the safety requirements of NCS in the presence of such inaccuracies. In this paper, we tackle the problem of verifying unbounded time safety of linear networked control systems with a uncertain sampling period for feedback input, inaccurate sensors for output estimation and disturbance input.

Safety verification involves proving that the set of reachable states of the system are contained within a specified safe set. However, exactly computing the set of reachable states of models containing linear differential equations with discontinuous switching between states or vector fields,

called *affine hybrid systems*, is computationally intractable [13]. Instead, a common approach is to overapproximate this set of reachable states to prove safety. There has been a lot of research on overapproximating the set of reachable states of affine hybrid systems in both bounded and unbounded time [19, 32, 35]. But networked control systems have time triggered switching which can be handled accurately by these methods. The reason is explained below.

Challenge of safety verification of NCS

In an NCS, there is time triggered switching of states with uncertainty in the switching time (sampling time). The approach followed by most techniques [19, 32, 35] to overapproximate reachable sets of hybrid systems involves computing intersection between reachable sets and guard conditions by some set representation like polytopes [35], zonotopes [21], ellipsoids [10], or polynomial sub-level sets like barrier certificates [32]. In the case of NCS, the guard is on the time of switching, i.e., the clock variable. For very simple hybrid systems having constant vector fields, there is linear relationship between clock variables and other state variables. Then, we can expect to reliably overapproximate the intersection of guard on the clock variable and reachable states containing value of clock variable. But an NCS is more complex involving linear differential equations with time triggered switching. There is an exponential relationship between the clock variable and the reachable states in NCS. Such exponential relationship can not be captured using the well known set representations mentioned above. Instead, a more effective way is proposed in [8, 16, 18, 25, 31] to handle time triggered switching by dividing the switching time into very small sub-intervals and using matrix exponentials to map the reachable states in different sub-intervals. The above methods for stability verification of NCS do not consider additive input in the dynamics. However, our dynamics is more general where we consider consider additive disturbance input. The above methods [8, 16, 18, 25, 31] use ellipsoids and H-polytopes which do not efficiently handle additive input in high dimensions. Ellipsoids provide poor accuracy for approximating Minkowski sum with input sets, which is required in reachability analysis. Similarly, the complexity of H-polytopes exactly representing Minkowski sums blows up at least exponentially in the dimension of state space [28]. Therefore, we need better set representations to accurately overapproximate the reachable sets at various switching times.

Solution: Generalizing zonotope to complex zonotope

In this context, an effective set representation is a *zonotope* [20], described as a linear combination of real vectors called *generators*, whose combining coefficients are bounded in absolute value. Its advantage is that linear transformation and the Minkowski sum can be efficiently computed. However, we are required to overapproximate the unbounded time reachable set of NCS that typically involves computing *invariants*. Computing invariants by a set representation requires encoding the directions of convergence of reachable sets towards equilibrium. In NCS, some of the directions for convergence can be along the eigenvectors of the dynamics, which can be complex valued vectors (Theorem 7). But (simple) zonotopes, which are confined to the real valued domain, can fail to capture such complex valued directions of convergence of trajectories. Therefore, our goal is to extend them to the complex number domain to obtain a new set representation called *complex zonotope*. Complex zonotopes retain the merit of usual zonotopes that linear transformation and the Minkowski sum operations can be computed efficiently. Additionally, they can capture the contraction of reachable sets based on the complex eigenstructure, which is not possible using a real valued zonotope. We provide mathematical evidence (Theorem 7) to support the latter claim. Furthermore, their real projections are geometrically more expressive than usual zonotopes and can represent some non-polytopic sets.

Using complex zonotopes, we propose an algorithm to verify linear safety constraints of a linear NCS with uncertain sampling period, inaccurate output estimation and disturbance input. The algorithm is a semi-decision procedure which can either terminate when the system is verified to be safe or fail to terminate. In the latter case, the user can set a threshold number of iterations for termination to get a bounded time result. Using the algorithm, we successfully applied our algorithm to verify benchmark examples of NCS [23, 29] with high dimensions (≥ 12 dimensions (including state and controller input variables), while the simple zonotope based version of our algorithm and a state-of-the-art verification tool [19] both failed to verify them.

In summary, we make the following contributions in this paper.

1. We introduce the complex zonotope set representation, a geometrically more expressive representation than zonotope, to handle invariant computation in the presence of additive input and time triggered switching.
2. We propose a theoretical result (Theorem 7) about the existence of complex zonotopic invariants based on eigenvectors of the NCS dynamics.
3. We extend the previous algorithms for stability verification of NCS [8, 16, 18, 25, 31] without additive input to safety verification in the presence of additive input. We use complex zonotopes containing eigenvectors for computing invariants of NCS more effectively when there is additive input.
4. As a proof-of-concept, we compare the performance of our complex zonotope with a real zonotope containing concatenation of real and imaginary parts of the complex template. We compare the performance on high dimensional (>9 state space) benchmarks examples in literature. We show that while our complex zonotope is successful in verification, the real zonotope either fails to compute an invariant or computes one with very large bounds above the safety threshold. We also compare with another state-of-the-art tool [19], which also failed to verify the benchmarks.

This paper is an extended version of part of our work presented in the conferences [1–4] and the PhD thesis [5]. The extensions and modifications made in this paper are explained in Section 2.

Organization. In Section 2, we review previous research related to our work and draw some comparison. In Section 3, we formalize NCS with uncertainty in sampling time, inaccurate output estimation and unknown open loop input. We explain the relation between safety verification and invariant computation at sampling times for an NCS. In Section 4, we introduce the complex zonotope representation as a generalization of usual zonotopes to complex valued domain. We describe a result that shows how a complex zonotope can specify invariants based on eigenstructure. We discuss operations on complex zonotopes that are later used to verify safety properties. In Section 5, we describe the procedure for verification using complex zonotopes. The experiments on some benchmark examples and results are discussed in Section 6. We begin by describing in the following the mathematical notation used in this paper.

1.1 Notation

The set of real numbers is represented by $\mathbb{R}$, integers by $\mathbb{Z}$, and their positive subsets by $\mathbb{R}_{\geq 0}$ and $\mathbb{Z}_{\geq 0}$, respectively. The set of complex numbers is $\mathbb{C}$. Given a subset $\mathbb{S}$ of real or complex numbers we denote the set of n-dimensional vectors from $\mathbb{S}$ as $\mathbb{S}^n$ and $n \times m$ matrices from $\mathbb{S}$ as $\mathbb{S}^{n \times m}$. The i^{th} component of a vector x is x_i while the element of the i^{th} row and the j^{th} column of a matrix X is X_{ij}. The numbers of rows of matrix X and size of vector x are $\text{rows}(X)$ and $\text{rows}(x)$, respectively. The number of columns of X is $\text{cols}(X)$. Given any two real vectors x, y such that $\text{rows}(x) = \text{rows}(y)$, we say $x \leq y$, if $\forall i \in \{1, \ldots, \text{rows}(x)\}$, $x_i \leq y_i$. The diagonal matrix

containing a vector x along its diagonal is denoted by $\mathcal{D}(x)$. The identity matrix of size $n \times n$ is denoted by $\mathcal{I}_n$ and a vector of ones of size n is denoted by $[1]_{n\times 1}$. The real part of a complex number x is $\mathrm{Re}(x)$ and the imaginary part is $\mathrm{Im}(x)$. The absolute value of a complex number x is $|x| = \sqrt{\mathrm{Re}(x)^2 + \mathrm{Im}(x)^2}$. The infinity norm of a complex vector x is $\|x\|_\infty = \max_{i=1}^{\mathrm{rows}(x)} |x_i|$. For a complex matrix X, $|X|$ is the matrix containing the absolute values of the elements of X, i.e., $|X|_{ij} = |X_{ij}|$. For any $a, b \in \mathbb{R} \bigcup \{\infty, -\infty\}$ we use the following notation for intervals.

$$(a,b) = \{x \in \mathbb{R} \mid a < x < b\}, \quad (a,b] = \{x \in \mathbb{R} \mid a < x \leq b\}$$
$$[a,b) = \{x \in \mathbb{R} \mid a \leq x < b\}, \quad [a,b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$$

Given two same dimensional subsets of complex vector space $U \in \mathbb{C}^n$ and $V \in \mathbb{C}^n$, we define their Minkowski sum as follows.

$$U \oplus V = \{u + v \mid u \in U,\ v \in V\}$$

Given a complex matrix $M \in \mathbb{C}^{n\times n}$ and a non-negative real matrix $\Upsilon \in \mathbb{R}^{n\times n}_{\geq 0}$, we define a neighborhood of M whose difference with M is less than Υ, as follows.

$$M \bigtriangledown \Upsilon = \left\{ M + \widehat{M} \;\middle|\; \widehat{M} \in \mathbb{C}^{n\times n},\ \left|\widehat{M}\right| \leq \Upsilon \right\}.$$

2 Related work

This paper is a journal extension of part our work part on safety and stability verification using complex zonotope presented in the conferences [1–4] and a PhD thesis [5]. The present work adapts the algorithm used for stability verification [2, 4] and tackles the problem of safety verification of NCS. In this context, we provide a sufficient condition for checking inclusion of a set of complex zonotopes whose templates are in the neighborhood of a complex zonotope, inside another complex zonotope. This generalizes inclusion checking between two complex zonotopes from the conference papers [1, 5] to inclusion checking between a set of complex zonotopes and another complex zonotope. Our conference paper on safety verification [1] proposes an approximation of intersection of complex zonotope with linear constraints which can be coarse because complex zonotopes are not closed under intersection with linear constraints. If using this method for an NCS with uncertain sampling time, intersection of complex zonotopes with sampling time constraints can be inaccurately approximated. Instead, in this paper we make use of matrix exponential maps to compute an *invariant*.

A number of set representations have also been developed for verification of hybrid systems and also numerical programs. Linear relationships between state variables can be represented using polytopes [15] and their variants, such as template polyhedra [35], hypercubes [36], octagons [30], zonotopes [20] and tropical polyhedra [9]. Non-linear relationships between state variables can be encoded by ellipsoids [11, 27], polynomial templates [7, 33] and polynomial zonotopes [6, 12]. Similarly, a barrier certificate [32] can be used to separate the reachable set from an unsafe set using a sub-level set of a suitable function, such as a polynomial.

The efficiency of a set representation in verification depends on the efficiency of computing common operations used in reachability analysis. For a linear networked control system with additive uncertainty, linear transformation and the Minkowski sum are the main operations involved in the computation of the reachable sets. The zonotope representation [20] can be very efficient for *bounded time* reachability of a linear networked control system because the linear transformation and Minkowski sum over zonotopes can be computed efficiently. However, approximating *unbounded time* reachable sets typically requires computing invariants. Efficiently

computing invariants requires encoding directions for convergence of state trajectories, which can depend on complex valued eigenvectors. But real valued zonotopes can not capture the convergence along complex valued eigenvectors. Therefore, we generalize zonotopes to complex zonotopes that can incorporate possibly complex eigenvectors among its generators in a way that the reachable set approximation can converge along the complex eigenvectors.

This generalization of zonotope to complex zonotope can express some non-polytopic sets in addition to polytopic zonotopes in the real domain, and hence more expressive. This generalization is similar in spirit to quadratic zonotope [6] or more generally polynomial zonotope [12] since both represent constraints on variables. However, while complex zonotope involves a linear combination of some ellipsoids and line segments, polynomial zonotope involves a non-linear combination of only line segments. Therefore, both represent very different classes of non-polyhedral sets. Moreover, we show in this paper that an infinite parametrized family of invariant complex zonotopes can be represented efficiently for any stable linear system using eigenvectors. However, there is no known guarantee of existence of invariant polynomial zonotopes, except trivially the equilibrium, for stable linear systems.

We remark that not many set representations can handle additive disturbance efficiently. Ellipsoids are not closed under the Minkowski sum [10], which can result in reduction of approximation accuracy in the presence of additive disturbance. The work of Allamigeon et. al [10] proposed an over-approximation of the Minkowski sum of ellipsoids based on the Löwner order, still the exact Minkowski sum can not be represented by an ellipsoid. Although polytopes are closed under Minkowski sum, there can be exponential blowup of complexity of the resulting half-space representation [28]. In contrast, zonotope and its extension to complex zonotope can exactly represent the Minkowski sum and its computation is also very efficient.

We draw inspirations from the algorithms for stability verification of sampled data systems which compute invariants [8, 16, 18, 25, 31] either as sub-level sets of Lyapunov functions or polytopes. In our work, we extend the use invariants to verify safety in addition to stability. Furthermore, our complex zonotope representation can efficiently handle additive disturbance due to efficient computation of the Minskowski sum. This is an advantage over sub-level sets of Lyapunov functions where the Minkowski sum can not be represented accurately, and also over polytopes whose representation is of exponential complexity [28].

3 Networked control systems

In a networked control system (NCS), a controller input is exchanged over a network between different components as information packets. The controller input is sampled at discrete time instants, while it remains constant between successive sampling times. But the sensors which estimate the output may be inaccurate and the sampling period can be uncertain, possibly due to packet dropouts. In this paper, we consider systems with linear dynamics, linear feedback input from the controller, uncertainty in sampling period and inaccurate output estimation modeled by additive error and additive disturbance input. This system can also be categorized as a hybrid system because of periodic reset of feedback input and constraints on its sampling period.

The system is modeled as follows. The state of the entire system at time $t \in [0, \infty)$ is $x_t \in \mathbb{R}^n$, the feedback input exchanged between components is $u_t \in \mathbb{R}^m$, disturbance input is $v_t \in V \subseteq \mathbb{R}^m$, output is $y_t \in \mathbb{R}^p$, additive error estimation is $w_t \in W \subseteq \mathbb{R}^p$, $\tau_{\min}$ is a lower bound on feedback sampling period, $\tau_{\max}$ is an upper bound on sampling time, the set of initial states is Ω and $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{m \times n}$, $C \in \mathbb{R}^{p \times n}$, $D \in \mathbb{R}^{p \times m}$, $F \in \mathbb{R}^{m \times p}$ are real matrices related to the system dynamics described below.

$$\exists\,(t_k)_{k=0}^{\infty},\ \forall k \in \mathbb{Z}_{\geq 0} t_k \in \mathbb{R}_{\geq 0}$$

$$\forall t \notin \bigcup_{k=0}^{\infty} \{t_k\},\ \frac{\partial x_t}{\partial t} = Ax_t + Bu_t,\quad \frac{\partial u_t}{\partial t} = 0 \tag{1}$$

$$\forall k \in \mathbb{Z}_{\geq 0},\quad y_{t_k} = Cx_{t_k} + Du_{t_k} + w_{t_k} \tag{2}$$

$$u_{t_k} = Fy_{t_k} + v_{t_k} \tag{3}$$

$$(t_{k+1} - t_k) \in [\tau_{\min}, \tau_{\max}],\ t_0 = 0,\ x_0 \in \Omega \tag{4}$$

We denote the combined vector of the state of the plant and controller input at any time t as $z_t = \left[x_t^T, u_t^T\right]^T$. We call a sequence of combined states at sampling times $(z_{t_k})_{k=0}^{\infty}$ as a sampling time trajectory which, by simple manipulation of the above equations, can be shown to be equivalently governed by the following dynamics.

$$z_{t_{k+1}} = R_{(t_{k+1}-t_k)} z_{t_k} + J_1 v_{t_{k+1}} + J_2 w_{t_{k+1}} \quad \text{where}$$
$$\forall \tau \in [\tau_{\min}, \tau_{\max}],\ R_\tau = A_r \exp\left(A_c \tau\right),$$
$$A_r = \begin{bmatrix} \mathcal{I}_n & 0 \\ FC & FD \end{bmatrix},\ A_c = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix},\ J_1 = \begin{bmatrix} 0 \\ \mathcal{I}_m \end{bmatrix},\ J_2 = \begin{bmatrix} 0 \\ F \end{bmatrix} \tag{5}$$

▶ **Example 1** (Damped harmonic oscillator)**.** We consider a damped harmonic oscillator where the feedback driving force is communicated over a network. The negative feedback driving force is $-x_1$ where x_1 is the position of the oscillator. In this case, we we have the following matrices for the dynamics.

$$A = \begin{bmatrix} 0 & 1 \\ 0 & -0.5 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad D = 0 \quad F = -1$$

Then the equivalent matrices for the dynamics of combined states, i.e., A_c, A_r, J_1, J_2 and R_t for any $t \in [0, \infty)$ are the following.

$$A_c = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -0.5 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad A_r = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$
$$J_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad J_2 = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \quad R_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \exp\left(\begin{bmatrix} 0 & t & 0 \\ 0 & -0.5t & t \\ 0 & 0 & 0 \end{bmatrix}\right)$$

3.1 Safety of NCS

Given a set of safe states $\mathcal{S} \subseteq \mathbb{R}^{n+m}$, we say that an NCS is safe if the state of plant, controller and feedback input of every trajectory at all times lies within $\mathcal{S}$. In other words, the set Γ of all reachable states defined as follows should lie within $\mathcal{S}$.

$$\Gamma = \left\{ \begin{bmatrix} x_\tau \\ u_\tau \end{bmatrix} \;\middle|\; \tau \in [0, \infty),\ (x_t, u_t)_{t \in [0,\infty)} \text{ satisfies (1)-(4)} \right\}$$

We shall show below that the safety of an NCS is guaranteed by the existence of an *invariant* set at sampling times that obeys certain conditions. We will later use this result to develop an algorithm for NCS safety verification.

▶ **Definition 2** (Sampling time invariant set)**.** A set Ψ of states is called a sampling time invariant if

$$\forall t \in [\tau_{\min}, \tau_{\max}],\ R_t\Psi \oplus J_1 V \oplus J_2 W \subseteq \Psi.$$

▶ **Theorem 3** (Relation between safety and sampling time invariant)**.** *For any $S \subseteq \mathbb{R}^{n+m}$, we have the reachable set Γ is included in $\mathcal{S}$, if there exists a sampling time invariant Ψ such that the following is true*

$$\forall \tau \in [0, \tau_{\max}],\ \exp(A_c\tau)\,\Psi \subseteq \mathcal{S} \tag{6}$$
$$\Omega \subseteq \Psi \tag{7}$$

Proof. Let us consider that at some sampling time t_k, a state $z_{t_k} \in \Psi$. Then at the next sampling time t_{k+1}, we get the following based on (5):

$$z_{t_{k+1}} \in R_{(t_{k+1}-t_k)}\Psi \oplus J_1 V \oplus J_2 W$$

We have $(t_{k+1} - t_k) \in [\tau_{\min}, \tau_{\max}]$ according to the dynamics of NCS. Therefore, according to the Definition 2 of sampling time invariant, we get $z_{t_{k+1}} \in \Psi$. This means that any state originating inside Ψ will remain inside Ψ at every sampling time. Since the initial set is contained in Ψ, *i.e.*, $\Omega \subseteq \Psi$, we get that for all possible trajectories of the system, the state remains within Ψ at the sampling time. Now, we have to prove that between any two sampling times, the state remains within the safe set.

Let $t = t_k + \tau$ be any time point where t_k is the latest sampling time before t, that is, $\tau \in [0, \tau_{\max}]$. Then the combined state reached at $t_k + \tau$ is given by

$$z_t = \exp(A_c\tau)\, z_{t_k}.$$

As we have shown that $z_{t_k} \in \Psi$, we get $z_t \in \exp(A_c\tau)\,\Psi$. It follows from (6), we get $z_t \in \mathcal{S}$, which proves the theorem. ◀

According to the above theorem, the safety of NCS can be verified by finding a sampling time invariant satisfying (6). The eigenstructure of the dynamics is closely related to sampling time invariants, which will be explained later in Theorem 7. Therefore, we introduce complex zonotope as a set representation in the next section, which enables us to use eigenvectors of the dynamics to find invariants. Complex zonotope also has other advantages, such as efficient computation of linear transformation and the Minkowski sum.

4 Complex zonotope

A simple zonotope is a set of points which are linear combinations of real vectors such that the combination coefficients are bounded in absolute values. Under this representation, the linear transformation and Minkowski sum can be exactly and quickly computed. This is useful for efficient bounded time reachability of a linear system as discussed in Girard *et al.* [20]. The mathematical definition of a simple zonotope or a real zonotope is as follows:

▶ **Definition 4** (Simple/Real zonotope)**.** Let P be a real valued matrix and $c \in \mathbb{R}^{\text{rows}(P)}$ be a real vector. The following is a real zonotope centered at c.

$$\mathcal{Z}(P,\ c) = \left\{ P\zeta + c \;\middle|\; \zeta \in \mathbb{R}^{\text{cols}(P)},\ \forall i \in \{1, \ldots, \text{cols}(P)\}\ \ |\zeta_i| \leq 1 \right\}$$

We have discussed in the previous section that unbounded time safety verification of NCS is related to computing sampling time invariants. For a stable NCS with real eigenvectors, a set contracts along its eigenvectors. By incorporating such eigenvectors among the generators of a zonotope, we can find sampling time invariants. However, when the eigenvectors are complex valued, real zonotopes may not be able to represent the directions for convergence among its generators. Therefore, we extend real zonotopes to the complex number domain to represent complex valued eigenvectors among their generators to find positive invariants. We show in a later lemma that incorporating complex eigenvectors allows representing invariants for a stable linear system with possibly complex eigenvalues.

▶ **Definition 5** (Complex zonotope)**.** Let $P \in \mathbb{C}^{\text{rows}(P)\times\text{cols}(P)}$ be a complex matrix, $c \in \mathbb{R}^{\text{rows}(P)}$ be a real vector and $s \in \mathbb{R}^{\text{cols}(P)}_{\geq 0}$ be a non-negative real vector. The following is a complex zonotope centered at c with template P and scale vector s.

$$\mathcal{Z}\left(P,\ c,\ s\right) = \left\{ P\zeta + c \;\middle|\; \zeta \in \mathbb{C}^{\text{cols}(P)},\ |\zeta| \leq s \right\}$$

While real zonotopes are a subclass of polytopes, real projections of complex zonotopes are more general and include non-polytopic sets in addition to polytopic zonotopes. Geometrically speaking, real projections of complex zonotopes are the Minkowski sum of some embedded ellipses and line segments.

▶ **Example 6.** Let us consider a complex zonotope $\mathcal{Z}\left(P,\ c,\ s\right)$ where

$$P = \begin{bmatrix} 1+2\imath & 1 & 2+\imath \\ 1-2\imath & 1 & 2-\imath \end{bmatrix}, \quad c = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad s = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

The real projection generated of $\mathcal{Z}\left(\begin{bmatrix} 1+2\imath \\ 1-2\imath \end{bmatrix},\ \begin{bmatrix} 0 \\ 0 \end{bmatrix},\ 1\right)$ is the ellipse $\left\{(x,y) \in \mathbb{R}^2 \;\middle|\; (x+y)^2 + \frac{(x-y)^2}{4} \leq 4\right\}$, real projection of $\mathcal{Z}\left(\begin{bmatrix} 2+\imath \\ 2-\imath \end{bmatrix},\ \begin{bmatrix} 0 \\ 0 \end{bmatrix},\ 1\right)$ is the ellipse $\left\{(x,y) \in \mathbb{R}^2 \;\middle|\; (x-y)^2 + \frac{(x+y)^2}{4} \leq 4\right\}$ and the real projection of $\mathcal{Z}\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix},\ \begin{bmatrix} 0 \\ 0 \end{bmatrix},\ 1\right)$ is the line segment $\left\{(x,y) \in \mathbb{R}^2 \mid -2 \leq x+y \leq 2\right\}$. So, the non-polytopic real projection of $\mathcal{Z}\left(P,\ c,\ s\right)$ in this example is the Minkowski sum of the two ellipses and one line segment as shown in Figure 1.

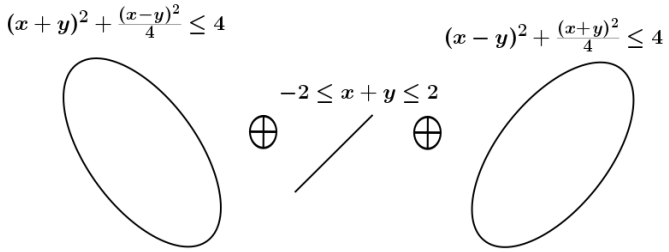


Figure 1 Complex zonotope as the Minkowski sum of two ellipsoids and a line segment

The following theorem states that when the sampling time period is certain, there is no additive disturbance input and the sampling time transformation matrix $R_{\tau_{\min}}$ is stable, we can find different sampling time invariant complex zonotopes by incorporating eigenvectors of the sampling time dynamics in its template and arbitrarily varying the scale factors.

▶ **Theorem 7** (Invariance based on eigenstructure). *Let us consider that* $\tau_{\min} = \tau_{\max} = t$, $V = \{0\}$ *and* $W = \{0\}$. *Let* $E \in \mathbb{C}^{(n+m)\times(n+m)}$ *contain the eigenvectors of* R_t *as its columns whose corresponding complex eigenvalues are* $e \in \mathbb{C}^{(n+m)}$. *If* $0 < \|e\|_\infty \leq 1$, *then for any scale vector* $s \in \mathbb{R}^n_{\geq 0}$, *the complex zonotope* $\mathcal{Z}(E,\ 0,\ s)$ *is a sampling time invariant.*

Proof. Let $x \in \mathcal{Z}(E,\ 0,\ s)$. Based on the complex zonotope representation, there exists $\zeta \in \mathbb{C}^n, |\zeta| < s$ such that $x = E\zeta$. It then follows that

$$R_t x = R_t E\zeta = E\mathcal{D}(e)\,\zeta \tag{8}$$

Let $\alpha = \mathcal{D}(e)\,\zeta$. So $R_t x = E\alpha$ where we derive the following bound on th magnitude of α_i for any $i \in \{1, \ldots, n\}$.

$$|\alpha_i| = |e_i\zeta_i| = |e_i|\,|\zeta_i|$$

and since $\|e\|_\infty \leq 1$, we have

$$|\alpha_i| \leq |\zeta_i| \leq s_i.$$

As $|\alpha| \leq s$, we get $R_t x = E\alpha \in \mathcal{Z}(E,\ 0,\ s)$. Since this is true for all $x \in \mathcal{Z}(E,\ 0,\ s)$, we can establish that $R_t\mathcal{Z}(E,\ 0,\ s) \subseteq \mathcal{Z}(E,\ 0,\ s)$. ◀

▶ Remark (Advantage of complex zonotope over real zonotope). The above theorem is true for complex zonotopes but not for real zonotopes because real zonotopes can not have complex valued vectors as generators. Let us consider that the dynamics with constant sampling time has complex eigenstructure where the rank of eigenvectors is the total dimension $n + m$. The above theorem means that using complex zonotopes we can find a sampling time invariant set containing any initial set. However, using real zonotopes, we may not be able to find such a sampling time invariant set because real zonotopes capture contraction along complex eigenvectors. This is one main advantage of using complex zonotopes apart from being geometrically more expressive than real zonotopes. The below example illustrates how a complex zonotope sampling time invariant can be found by using the complex eigenvectors as generators. But a real zonotope invariant containing an initial set can not be found by using imaginary and real parts of eigenvectors and the coordinate directions as generators.

▶ **Example 8.** Let us consider the damped harmonic oscillator NCS in Example 1. Let us consider an initial set $\Omega = [-0.85, 0.85] \times [0, 0]$. Let us consider a complex matrix $P = \begin{bmatrix} -0.5774 & -0.5774 & -0.0051 \\ 0.1298 - 0.5626\iota & 0.1298 + 0.5626\iota & 0.1020 \\ 0.5774 & 0.5774 & -0.9948 \end{bmatrix}$. For $s = \begin{bmatrix} 29 \\ 0.8 \\ 0.8 \end{bmatrix}$, we get that $\mathcal{Z}\left(P,\ \begin{bmatrix} 0 \\ 0 \end{bmatrix},\ s\right)$ is a sampling time invariant of the NCS such that $\Omega \subseteq \mathcal{Z}(P,\ c,\ s)$. Now let us take the real matrix $G = \begin{bmatrix} \mathrm{Re}(P) & \mathrm{Im}(P) & \mathcal{I}_3 \end{bmatrix}$, which contains the real and imaginary parts of the complex matrix P as real generators and also the identity matrix. We used convex optimization (Algorithm 1) to search for a scaling factor h and center c such that the real zonotope $\mathcal{Z}(G,\ c,\ h)$ is a sampling time invariant containing the initial set. But our search failed to find it. Similarly, we tested with 5 uniformly randomly generated real valued templates, but all those real templates also could not be successful in finding an invariant. This illustrates that using complex zonotope containing eigenvectors as generators can let us find invariants in examples where real zonotopes may fail.

The result in Theorem 7 is only true when there is no uncertainty in sampling period. To handle a more general case where there is uncertainty in sampling period, we can incorporate eigenvectors of multiple matrices R_t for different values of $t \in [\tau_{\min}, \tau_{\max}]$. Then optimization can be used to find an appropriate scale factor that guarantees invariance, which we shall explain in Section 5.

As mentioned earlier, like real zonotopes, complex zonotopes are also closed under linear transformation and the Minkowski sum and they can be computed efficiently. This property is useful for efficiently representing sets of successor states of an NCS and is exploited in computing the operations used in our proposed NCS safety verification algorithm, namely Algorithm 1.

▶ **Lemma 9** (Linear transformation and the Minkowski sum)**.** *Let us consider complex zonotopes* $\mathcal{Z}(P,\ c,\ s) \subseteq \mathbb{C}^n$ *and* $\mathcal{Z}(Q,\ e,\ r) \subseteq \mathbb{C}^n$ *and real matrices* $A, B \in \mathbb{R}^{n\times n}$*. Then the following is true:*

$$A\mathcal{Z}(P,\ c,\ s) \oplus B\mathcal{Z}(Q,\ e,\ r) = \mathcal{Z}\left(\begin{bmatrix}AP & BQ\end{bmatrix},\ Ac+Be,\ \begin{bmatrix}s\\ r\end{bmatrix}\right). \tag{9}$$

Proof. Let us consider $x \in \mathcal{Z}(P,\ c,\ s)$ and $y \in \mathcal{Z}(Q,\ e,\ r)$. Then there exist $\zeta, \alpha \in \mathbb{C}^n$ such that $|\zeta| \le s$, $|\alpha| \le r$, $x = P\zeta + c$ and $y = Q\alpha + e$. We derive the following:

$$Ax + By = AP\zeta + Ac + BQ\alpha + Be$$
$$\begin{bmatrix}AP & BQ\end{bmatrix}\begin{bmatrix}\zeta\\ \alpha\end{bmatrix} + (Ac+Be).$$

We have $\left|\begin{bmatrix}\zeta\\ \alpha\end{bmatrix}\right| \le \begin{bmatrix}s\\ r\end{bmatrix}$. Therefore, $Ax + By \in \mathcal{Z}\left(\begin{bmatrix}AP & BQ\end{bmatrix},\ Ac+Be,\ \begin{bmatrix}s\\ r\end{bmatrix}\right)$. Since this is true for all $x \in \mathcal{Z}(P,\ c,\ s)$ and $y \in \mathcal{Z}(Q,\ e,\ r)$, we obtain (9). ◀

In order to verify sampling time invariance of a complex zonotope given an interval of uncertainty, we need to check inclusion of a set of complex zonotopes (obtained by applying a sequence of transformations) inside the original complex zonotope, based on the Definition 2 of invariance. In this regard, we define a set of complex zonotopes whose templates lie in the neighborhood of a given template as follows. Let us consider a real matrix with positive entries $\Upsilon \in \mathbb{R}_{\ge 0}^{\mathrm{rows}(Q)\times\mathrm{cols}(Q)}$ where Q is a complex matrix and a real vector $\rho \in \mathbb{R}^{\mathrm{cols}(e)}$ where e is a real vector.

$$\mathcal{Z}(Q \bigtriangledown \Upsilon,\ e \bigtriangledown \rho,\ r) = \left\{\mathcal{Z}\left(Q+\widehat{Q},\ e+u,\ r\right) \;\middle|\; |Q| \le \Upsilon,\ |u| \le \rho\right\}$$

The following relation is a sufficient condition for checking the required inclusion which is proved later in Lemma 11.

▶ **Definition 10** (Relation for checking inclusion)**.** Let P be a complex matrix such that P^TP is a square invertible matrix. Let $\Upsilon > 0$ be a real matrix with only positive elements. We define the relation

$$\mathcal{Z}(Q \bigtriangledown \Upsilon,\ e \bigtriangledown \rho,\ r) \sqsubseteq \mathcal{Z}(P,\ c,\ s)$$

if all of the following conditions are verified:

$$\exists X, \Delta \in \mathbb{C}^{\mathrm{cols}(P)\times\mathrm{cols}(Q)}, y \in \mathbb{C}^{\mathrm{cols}(P)}:$$
$$PX = Q\mathcal{D}(r),\quad \Delta = \left|P^\dagger\right|\Upsilon\mathcal{D}(r),\quad (e-c) = Py,\quad \delta = \left|P^\dagger\right|\rho \tag{10}$$
$$\forall i \in \{1,\ldots,\mathrm{rows}(X)\}\quad |y_i| + \delta_i + \sum_{j=1}^{\mathrm{cols}(X)} |X_{ij}| + \Delta_{ij} \le s_i \tag{11}$$

▶ **Lemma 11** (Checking inclusion)**.** *If* $\mathcal{Z}(Q \bigtriangledown \Upsilon,\ e \bigtriangledown \rho,\ r) \sqsubseteq \mathcal{Z}(P,\ c,\ s)$ *is true for* $\Upsilon, \rho > 0$*, then the subset inclusion* $\mathcal{Z}(Q \bigtriangledown \Upsilon,\ e \bigtriangledown \rho,\ r) \subseteq \mathcal{Z}(P,\ c,\ s)$ *is true.*

Proof. Let us assume that $\mathcal{Z}\left(Q \bigtriangledown \Upsilon,\ e \bigtriangledown \rho,\ r\right) \sqsubseteq \mathcal{Z}\left(P,\ c,\ s\right)$ is true. Hence, there exist matrices X, Δ and complex vectors y, ρ such that all the equations in (11) are true. Let us consider any $x \in \mathcal{Z}\left(Q \bigtriangledown \Upsilon,\ e,\ R\right)$. Based on the definition of a complex zonotope, there exists $\zeta \in \mathbb{C}^{\text{cols}(P)}$ such that $|\zeta| \leq s$ and $x = \left(Q + \widehat{Q}\right)\zeta + e + u$, $\left|\widehat{Q}\right| \leq \Upsilon$ and $|u| \leq \rho$. We now have to show that $x \in \mathcal{Z}\left(P,\ c,\ s\right)$.

Let us consider a vector $\alpha \in \mathbb{C}^{\text{cols}(\zeta)}$ such that for any $i \in \{1, \ldots, \text{cols}\left(\zeta\right)\}$, the following is true:

$$\alpha_i = \zeta_i / r_i \ \text{ if } \ r_i > 0, \qquad \alpha_i = 0 \ \text{ if } \ r_i = 0 \tag{12}$$

Since $|\zeta| \leq r$, it follows from the above definition that $|\alpha| \leq 1$ and $\zeta = \mathcal{D}\left(r\right)\alpha$. Then we derive the following.

$$\begin{aligned} x &= \left(Q + \widehat{Q}\right)\zeta + e + u = \left(Q + \widehat{Q}\right)\mathcal{D}\left(r\right)\alpha + e + u \\ &= \left(Q + \widehat{Q}\right)\mathcal{D}\left(r\right)\alpha + (e - c) + c + u \end{aligned}$$

and from (10)

$$x = P\left(X\alpha + P^{\dagger}\widehat{Q}\mathcal{D}\left(r\right)\alpha + y + P^{\dagger}u\right) + c \tag{13}$$

We derive the following for any $i \in \{1, \ldots, \text{rows}\left(X\right)\}$

$$\begin{aligned} &\left|X\alpha + P^{\dagger}\widehat{Q}\mathcal{D}\left(r\right)\alpha + y + P^{\dagger}u\right|_i \\ &\leq |y|_i + \left|P^{\dagger}\right||u| + \sum_{j=1}^{\text{cols}(X)}\left(|X|_{ij} + \left(\left|P^{\dagger}\right|\left|\widehat{Q}\right|\mathcal{D}\left(r\right)\right)_{ij}\right)|\alpha|_j \\ &\leq |y|_i + \delta_i + \sum_{j=1}^{\text{cols}(X)}|X|_{ij} + \Delta_{ij} \\ &\leq s_i. \end{aligned} \tag{14}$$

The above second inequality is true because $|\alpha| \leq 1$, $\left|\widehat{Q}\right| \leq \Upsilon$, $\Delta = \left|P^{\dagger}\right|\Upsilon\mathcal{D}\left(r\right)$, $|u| \leq \rho$ and $P^{\dagger}\rho = \delta$, and the last inequality is deduced from (11).

From (13) and (14), we get that $x \in \mathcal{Z}\left(P,\ c,\ s\right)$. As this is true for any $x \in \mathcal{Z}\left(Q,\ e,\ r\right)$, the inclusion $\mathcal{Z}\left(Q,\ e,\ r\right) \subseteq \mathcal{Z}\left(P,\ c,\ s\right)$ is true. ◀

We can algebraically compute bounds on the real projection of a complex zonotope along any direction, that is, the support function, as follows.

▶ **Lemma 12** (Computing support function). *Let us consider a complex zonotope $\mathcal{Z}\left(P,\ c,\ s\right)$ and a vector $w \in \mathbb{R}^{\text{cols}(c)}$. We have the following equality:*

$$\max_{x \in \mathcal{Z}(P,\ c,\ s)} \text{Re}\left(w^T x\right) = w^T c + \left|w^T P\right| s \tag{15}$$

Proof. First we prove that

$$\max_{x \in \mathcal{Z}(P,\ c,\ s)} w^T x \leq w^T c + \left|w^T P\right| s \tag{16}$$

Let us consider $x \in \mathcal{Z}(P,\ c,\ s)$. So, there exists $\zeta \in \mathbb{C}^{\text{cols}(P)}$ such that $|\zeta| \leq s$ and $x = P\zeta + c$. We derive the following:

$$\begin{aligned} \text{Re}\left(w^T x\right) &= w^T c + \text{Re}\left(w^T P \zeta\right) \\ &\leq w^T c + \left|w^T P\right| s \end{aligned} \tag{17}$$

The inequality in the above formula is deduced from the fact that $|\zeta| \leq s$. As the above is true for all $x \in \mathcal{Z}(P,\ c,\ s)$, it proves (16).

Next we prove the following:

$$\max_{x \in \mathcal{Z}(P,\ c,\ s)} w^T x \geq w^T c + \left|w^T P\right| s \tag{18}$$

Let us consider $x = P\zeta + c$ where $\zeta \in \mathbb{C}^{\text{cols}(P)}$ is defined as follows.

$$\zeta_i = s_i \text{ if } \text{Re}\left(w^T P_i\right) \geq 0, \qquad \zeta_i = -s_i \text{ otherwise} \tag{19}$$

Then we get the following:

$$\begin{aligned} \text{Re}\left(w^T x\right) &= \text{Re}\left(w^T P \zeta\right) + w^T c \\ &= \left|w^T P\right| s + w^T c \end{aligned}$$

The second equality in the above is obtained by using (19). This proves (18). From the inequalities (16) and (18), we get (15). ◀

The relation for checking inclusion ((10) and (11)) consists of a set of convex constraints on the variables X, Δ, s, r, e and c when the templates Q and P are fixed (constants). In fact, they constitute a class of convex constraints called *second order conic constraints* (SOCC) [14]. The SOCC constraints can be solved efficiently up to high numerical precision using convex optimization techniques and many solvers are available for the same [22].

5 Using complex zonotopes for verification

In this section, we describe an algorithm based on operations on complex zonotopes to verify safety of NCS. Our algorithm finds a complex zonotope which is a sampling time invariant and satisfies the other required condition (6) for safety. The algorithm has two parts.

1. We find a complex zonotope that is a sampling time invariant, *i.e.*, is invariant with respect to the transformation at all sampling times $t \in [\tau_{\min}, \tau_{\max}]$.
2. Next we verify that the the reachable set of the complex zonotope by continuous evolution, without reset, within the interval $[0, \tau_{\max}]$ remains within the safe set (6).

The detailed procedure is explained in Algorithm 1 and its correctness is proved in Theorem 17. A safe set is specified by linear constraints and an open input set, disturbance input set and initial set bounded by complex zonotopes, as follows.

$$\mathcal{S} = \left\{x \in \mathbb{R}^{n+m} \mid Hx \leq d\right\}, \quad V \subseteq \mathcal{Z}(Q_V,\ c_V,\ s_V), \quad W \subseteq \mathcal{Z}(Q_W,\ c_W,\ s_W)$$
$$\Omega \subseteq \mathcal{Z}(Q_{init},\ c_{init},\ s_{init})$$

Algorithm 1 Verifying safety of NCS.

$k \leftarrow 3$;
$P \leftarrow \begin{bmatrix} E_k & \mathcal{I}_{n+m} & R_{\tau_{\max}} & J_1 Q_V & J_2 Q_W \end{bmatrix}$;
Remove repeated columns and zero columns of P;
% Find sampling time invariant candidate complex zonotope:
$N \leftarrow 10$;
Feasible $\leftarrow$ *False*;
while *Feasible* = *False* **do**
  Feasible $\leftarrow$ *True*;
  for $t \in \mathcal{T}_N$ **do**
    for $M \in \Theta\left(t, \mathcal{E}^t_{\frac{\tau_{\max}-\tau_{\min}}{N}}\right)$ **do**
      if *Feasible* = *True* **then**
        Solve for s, c by convex optimization satisfying (24)–(25);
        if *(24)–(25) is feasible* **then**
          Feasible $\leftarrow$ *True*;
          else
            Feasible $\leftarrow$ *False*;
            $N \leftarrow 2 * N$;
          end
        end
      end
    end
  end
end

% Verify Safety:
Verified $\leftarrow$ *False*;
while *Verified* = *False* **do**
  Choose small $\epsilon > 0$;
  if *(26) is true* **then**
    Verified $\leftarrow$ *True*;
    Return: "NCS is safe"
  end
  else
    $\epsilon \leftarrow \epsilon/2$;
  end
end

5.1 Finding sampling time invariant

We first fix the template of a complex zonotope based on the eigenvectors of the dynamics and then the templates of the input sets. We can also add arbitrary vectors to the template to increase precision. Next we derive a set of convex constraints on center, scale factor and other auxiliary variables of a complex zonotope such that the complex zonotope is a sampling time invariant. We solve for the scale factor and center using convex optimization.

Let us choose a positive integer $k > 0$ and denote by E_k the matrix containing all independent unit eigenvectors of the matrices R_t for all $t \in \mathcal{T}_k$ where

$$\mathcal{T}_k = \left\{ t = \tau_{\min} + i\frac{\tau_{\max} - \tau_{\min}}{k} \;\middle|\; t \in [\tau_{\min}, \tau_{\max}] \right\}$$

The template P of the complex zonotope consists of these eigenvectors, the templates of inputs sets $J_1 Q_V, J_2 Q_W$, their transformations $R_{\tau_{\max}} J_1 Q_V$, $R_{\tau_{\max}} J_2 Q_W$.

$$P = \begin{bmatrix} E_k & \mathcal{I}_{n+m} & R_{\tau_{\max}} & J_1 Q_V & J_2 Q_W \end{bmatrix} \tag{20}$$

If P contains repeated columns or zero columns, they are removed.

▶ Remark (Choice of template). Firstly, we add the eigenvectors E_K of the transformation operators at various sampling times. This heuristic is based on Theorem 7 which says that incorporating eigenvectors in the complex zonotope is closely related to finding sampling time invariants. We also include the identity matrix inside the zonotope to capture bounds along the coordinate directions. Next, we get from Lemma 9 that the resulting complex zonotopes from applying the linear transformation R_t at time t is $R_t \mathcal{I}_{n+m}$ and $R_t E_k$. Since E_k contains eigenvectors of the dynamics at various sampling times, the directions of $R_t E_k$ may not be very different from E_k. So, we do not include $R_t E_k$ for any t. However, we include the other template $R_t \mathcal{I}_{n+m} = R_t$ at maximum sampling time $t_{\max}$ obtained by transforming the identity template. Next, by summing the additive disturbance and open input sets, we get the template $\begin{bmatrix} J_1 Q_V & J_2 Q_W \end{bmatrix}$ in a transformed complex zonotope after the switching at sampling time, by Lemma 9 . Therefore, this template is also concatenated to P_k. Furthermore, adding any arbitrary generator will only increase the accuracy because our optimization will adjust the scaling factors corresponding to the generators. So, we can increase the value of K to increase the accuracy of verification.

▶ **Example 13.** Let us consider the NCS in Example 1. We consider an lower bound $0.1s$ and upper bound $0.3s$ on the sampling time. We consider the safe set $\mathcal{S} = \left\{ z \in \mathbb{R}^3 \mid [100] z \leq 1 \right\}$, initial set $\Omega = [0.85, 0.85] \times [0, 0] \times [0, 0]$, open input set $V = [-0.2, 0.2]$ and bounds on disturbance input set $W = [-0.2, 0.2]$. For $k = 3$, we compute E_k as the concatenation of eigenvectors of $R_{0.1} = A_r \exp(0.1 A_c)$, $R_{0.2} = A_r \exp(0.2 A_c)$ and $R_{0.3} = A_r \exp(0.3 A_c)$ where A_r and A_c are given in Example 1.

Next we have to find the center and scale factor, for the given template P_k such that we get a sampling time invariant. Before we describe the algorithm for this, we derive the prerequisite mathematical results.

We can expand a transformation operator $R_{t+\delta}$ with $\delta > 0$ in a neighborhood of sampling time t using the Taylor expansion as follows.

$$R_{t+\delta} = A_r \exp(A_c t) \exp(A_c \delta) = R_t + R_t A_c \delta + R_t A_c \delta^2/2 + \Lambda_\delta \tag{21}$$
$$\text{where } |\Lambda_\delta| \leq \mathcal{E}_\delta^t = |R_t| \, |\exp(A_c t)| \, |Ac| \delta^3/3.$$

We use the following lemma, which is proved in [24] to bound all matrices

$$\left\{ R_t + R_t A_c \delta + R_t A_c \delta^2/2 \mid \delta \leq \epsilon \right\}$$

for any $\epsilon > 0$ by the convex hull of a finite set of matrices.

▶ **Lemma 14** ([25]). *Let $L_0, L_1, ..., L_r$ be a finite sequence of real matrices and $U_j(\delta) = \sum_{i=1}^{j} L_i \delta^i$. If $0 \leq \delta < \epsilon$, then $U_r(\delta) \in \mathrm{Conv}(U_0(\epsilon), \ldots, U_r(\epsilon))$.*

Proof. This lemma is proved in [24]. ◀

Let us denote the set of finite matrices $\Theta(t,\epsilon) = \left\{\sum_{i=1}^{j} R_t A_c \epsilon^i \mid j = 0,1,2\right\}$. Then using Lemma 14, we have the following set inclusion:

$$\left\{R_t + R_t A_c \delta + R_t A_c \delta^2/2 \mid \delta \leq \epsilon\right\} \subseteq \text{Conv}\left(\Theta(t,\epsilon)\right) \tag{22}$$

Therefore, using the expansion (21) and the above result (22), we get that for any $\epsilon > 0$ and $0 \leq \delta \leq \epsilon$,

$$R_{t+\delta} \in \text{Conv}\left(\left\{M \bigtriangledown \mathcal{E}_\epsilon^t \mid M \in \Theta(t,\epsilon)\right\}\right) \ . \tag{23}$$

In order to verify sampling time invariance of $\mathcal{Z}(P,\ c,\ s)$, it is sufficient to divide the time interval $[\tau_{\min}, \tau_{\max}]$ into small intervals of a chosen size and verify invariance within each of the time intervals. This is explained in the following lemma.

▶ **Lemma 15.** *If there exists $N > 0$ such that $\forall t \in \mathcal{T}_N$, $\forall M \in \Theta\left(t, \frac{\tau_{\max} - \tau_{\min}}{N}\right)$ all of the following is true, then $\mathcal{Z}(P,\ c,\ s)$ is sampling time invariant.*

$$\exists \rho \in \mathbb{R}_{\geq 0}^{\text{cols}(c)} : \mathcal{E}^t_{\frac{\tau_{\max}-\tau_{\min}}{N}} |c| \leq \rho \tag{24}$$

$$\mathcal{Z}\left(\begin{bmatrix} MP & J_1 & J_2 \end{bmatrix} \bigtriangledown \begin{bmatrix} \mathcal{E}^t_{\frac{\tau_{\max}-\tau_{\min}}{N}} |P| & 0 & 0 \end{bmatrix},\ (Mc + J_1 c_v + J_2 c_w) \bigtriangledown \rho,\ \begin{bmatrix} s \\ c_v \\ c_w \end{bmatrix}\right)$$
$$\sqsubseteq \mathcal{Z}(P,\ c,\ s) \tag{25}$$

Proof. Let us consider any $\tau \in [\tau_{\min}, \tau_{\max}]$. There exists $t \in \mathcal{T}_N$, $\delta \in \left[0, \frac{\tau_{\max} - \tau_{\min}}{N}\right]$ such that $\tau = t + \delta$. Then we derive the following.

$$R_\tau \mathcal{Z}(P,\ c,\ s) \oplus J_1 \mathcal{Z}(Q_V,\ c_V,\ s_V) \oplus J_2 \mathcal{Z}(Q_W,\ c_W,\ s_W)$$
$$= \mathcal{Z}\left(\begin{bmatrix} R_\tau P & J_1 Q_V & J_2 Q_W \end{bmatrix},\ R_\tau c + J_1 c_v + J_2 c_W,\ \begin{bmatrix} s & s_v & s_W \end{bmatrix}^T\right)$$
$$\%\ \textit{By (23)},\ \exists M \in \Theta\left(t, \frac{\tau_{\max} - \tau_{\min}}{N}\right),\ \exists \widehat{M} \in 0 \bigtriangledown \mathcal{E}^t_{\frac{\tau_{\max}-\tau_{\min}}{N}} : R_\tau = M + \widehat{M}$$
$$= \mathcal{Z}\left(\begin{bmatrix} MP & J_1 & J_2 \end{bmatrix} + \begin{bmatrix} \widehat{M} & 0 & 0 \end{bmatrix},\ Mc + \widehat{M}c + J_1 c_v + J_2 c_w,\ \begin{bmatrix} s & s_v & s_W \end{bmatrix}^T\right)$$
$$\%\ \textit{As } \widehat{M} \in 0 \bigtriangledown \mathcal{E}^t_{\frac{\tau_{\max}-\tau_{\min}}{N}} \text{ and } \mathcal{E}^t_{\frac{\tau_{\max}-\tau_{\min}}{N}} |c| \leq \rho \text{ where } \mathcal{E}^t_{\frac{\tau_{\max}-\tau_{\min}}{N}} \geq 0$$
$$\subseteq \mathcal{Z}\left(\begin{bmatrix} MP & J_1 & J_2 \end{bmatrix} \bigtriangledown \begin{bmatrix} \mathcal{E}^t_{\frac{\tau_{\max}-\tau_{\min}}{N}} |P| & 0 & 0 \end{bmatrix},\ (Mc + J_1 c_v + J_2 c_w) \bigtriangledown \rho,\ \begin{bmatrix} s \\ c_v \\ c_w \end{bmatrix}\right)$$
$$\%\ \textit{By (25) and Lemma 11 for inclusion checking}$$
$$\subseteq \mathcal{Z}(P,\ c,\ s)\,.$$

Since the above is true for any $\tau \in [\tau_{\min}, \tau_{\max}]$, we get that $\mathcal{Z}(P,\ c,\ s)$ is a sampling time invariant set. ◀

5.2 Safety verification

After finding a sampling time invariant containing the initial set, we have to verify that the condition (6) that the reachable set of $\mathcal{Z}(P,\ c,\ s)$ by continuous evolution within $[0, \tau_{\max}]$ is contained within the safe set. This can be verified by checking a sequence of linear inequalities as described in the following.

▶ **Lemma 16.** *For all $t \in [0, \tau_{\max}]$, we get $\exp(A_c t)\, \mathcal{Z}(P,\ c,\ s) \subseteq S$ if there exists $\epsilon > 0$ such that the following are true:*

$$\forall k \in \mathbb{Z}_{\geq 0} :\ k \leq \frac{\tau_{\max}}{\epsilon},\ \forall i \in \{1, \ldots, \text{rows}(H)\},\ \beta_i(P, c, s) \leq d \text{ where}$$
$$\beta_i(P, c, s) = |H_i \exp(A_c k\epsilon)\, P|\, s + H_i \exp(A_c k\epsilon)\, c + \|H_i\| \exp(\|A_c\|\, \epsilon)\, \|\exp(A_c k\epsilon)\|\, \epsilon\, (\|c\| + \|P\|\, \|s\|) \tag{26}$$

Proof. Let us consider any $t \in [0, \tau_{\max}]$. There exists some $k \in \mathbb{Z}$, $t = k\epsilon + \rho$, $\rho \leq \epsilon$. We can write for $\rho \in [0, \epsilon]$,

$$\exp(A_c \rho) = \sum_{i=0}^{n} \frac{A_c^i \rho^i}{i} = \mathcal{I}_{n+m} + A_c \rho M :$$
$$M \in \mathbb{R}^{(n+m)\times(n+m)},\ \|M\| \leq \exp(\|A_c\|\, \epsilon) \tag{27}$$

We derive the following:

$$\begin{aligned}
H_i \exp(A_c t)\, \mathcal{Z}(P,\ c,\ s) &= H_i \mathcal{Z}(\exp(A_c t)\, P,\ \exp(A_c t)\, c,\ s) \\
&\quad \text{\% By Lemma 12} \\
&\leq |H_i \exp(A_c t)\, P|\, s + H_i \exp(A_c t)\, c \\
&\quad \text{\% By (27)} \\
&= |H_i \exp(A_c k\epsilon)\, P|\, s + H_i \exp(A_c k\epsilon)\, c + |H_i M P|\, s + Mc \\
&\quad \text{\% Substituting the bound from (27)} \\
&\leq |H_i \exp(A_c k\epsilon)\, P|\, s + H_i \exp(A_c k\epsilon)\, c \\
&\quad + \|H_i\| \exp(\|A_c\|\, \epsilon)\, \|\exp(A_c k\epsilon)\|\, \|A_c\|\, \epsilon\, (\|c\| + \|P\|\, \|s\|) \leq d_i
\end{aligned}$$

Therefore, $\exp(A_c t)\, \mathcal{Z}(P,\ c,\ s) \subseteq \{x \in \mathbb{R}^{n+m} \mid Hx \leq d\} = S$, which proves the lemma. ◀

The following theorem summarizes the overall sufficient condition for verifying safety based on complex zonotopes, which can be checked by the procedure described in Algorithm 1.

▶ **Theorem 17.** *We have $\Gamma \subseteq \mathcal{S}$ if there exist $c \in \mathbb{R}^n$, $s \in \mathbb{R}_{\geq 0}^{\text{cols}(P)}$, $\epsilon > 0$ and $N \in \mathbb{Z}_{\geq 0}$ such that all of the following conditions are true.*

1. $\mathcal{Z}(Q_{init},\ c_{init},\ s_{init}) \sqsubseteq \mathcal{Z}(P,\ c,\ s)$.
2. (24)–(25) *are true* $\forall t \in \mathcal{T}_N,\ \forall M \in \Theta(t, \epsilon)$.
3. (26) *is true.*

Proof. By ((24)-(25)) and Lemma 15, we prove that $\Psi = \mathcal{Z}(P,\ c,\ s)$ is a sampling time invariant. By the first condition $\mathcal{Z}(Q_{init},\ c_{init},\ s_{init}) \sqsubseteq \mathcal{Z}(P,\ c,\ s)$, we get that the initial set Ω is contained inside the sampling time invariant Ψ. By (26) and Lemma 16, we prove that for all $t \in [0, \tau_{\max}], \exp(A_c t)\, \mathcal{Z}(P,\ c,\ s) \subseteq \mathcal{S}$. Then based on Theorem 3, we get $\Gamma \subseteq \mathcal{S}$. ◀

Based on Theorem 17, we propose a semi-decision procedure in Algorithm 1 to verify safety. The algorithm is a semi-decision procedure because if it returns that the NCS is safe, then the NCS is indeed safe as proved in Theorem 17. However, the algorithm is not guaranteed to terminate. So, the user can choose to terminate the algorithm in any threshold number of iterations possibly with inconclusive result. In this context, we note NCS is a hybrid system and it is known that verification of reachability of very simple classes of hybrid systems is undecidable [13]. So, it is unlikely that we can not come up with a sure shot decision procedure to verify safety of NCS.

▶ **Example 18.** Let us the NCS in Example 13. We took $K = 3$ and ran our algorithm using the template matrix containing complex eigenvectors as shown in Equation 20, without any random matrix, i.e., $\Pi = 0$. Then we could verify that the x_1 bounds on the unbounded time reachable set is $|x_1| \leq 2.22$.

Comparison with real zonotope. Next, we concatenated the real and imaginary parts of P_k into a real matrix G where repeated column vectors were removed. We ran our algorithm to find a real zonotope invariant containing the real template G, but our algorithm failed. We also considered 5 uniformly randomly generated real valued templates to run our algorithm, but the search for sampling time invariant failed. Thus, complex zonotope based on eigenstructure is shown to be the better choice for verification on this example.

6 Experimental results

We implemented our algorithms and tested them on benchmark examples of NCS. We drew comparison with simple zonotope, i.e., having real valued generators and also the state-of-the-art tool SpaceEx [19]. For comparison with simple zonotope, we took the real template as the concatenation of real and imaginary parts of our complex template without repeated columns, and ran the same algorithm. In SpaceEx [19], the verification is performed by step-by-step forward reachability computation. In SpaceEx, we modeled NCS with uncertain sampling time as a hybrid system with linear guards and linear transitions. For convex optimization, we used CVX (version 2.2) with MOSEK solver (version 7.1) and Matlab 2020a on a computer with 1.4 GHz Intel Core i5 processor and 4 GB 1600 MHz DDR3. The precision of the solver is set to the default precision of CVX.

6.1 Networked platoon of vehicles

This example is adapted from a model of a networked cooperative platoon of vehicles, which is presented as a benchmark in the ARCH workshop [29]. The platoon consists of three follower vehicles M_1, M_2 and M_3 along with a leader board ahead M_4. Each of the vehicles receives feedback input added to the acceleration, which depends on the communication of their relative distances, velocities and accelerations over a WLAN. The distance between a vehicle M_i and its next vehicle M_{i+1}, relative to a reference distances d_i^{ref}, is denoted by e_i. The acceleration of the leader vehicle is a_L which ranges between $[-9, 1](m/s)$. The state of the system is denoted by a 9-dimensional vector $x = [e_1, \dot{e}_1, \ddot{e}_1, e_2, \dot{e}_2, \ddot{e}_2, e_3, \dot{e}_3, \ddot{e}_3]$, which is the reference distance minus relative distances, the relative velocities and relative accelerations of the vehicles.

The disturbance input is the acceleration $a_L \in [-9, 1](m/s^2)$ of the leader board. In our adaptation, we consider that there can be uncertainty in the sampling period of the feedback input. The matrices of the NCS model are given below.

$$A = \begin{bmatrix} 0 & 1.0000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1.0000 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1.6050 & 4.8680 & -3.5754 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0000 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.0000 & 0 & 0 & -1.0000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.1936 & 3.6258 & -3.2396 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0000 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0000 & 0 & 0 & -1.0000 \\ 0.7132 & 3.5730 & -0.0964 & 0.8472 & 3.2568 & -0.0876 & 1.2726 & 3.0720 & -3.1356 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad C = \mathcal{I}_9 \qquad D = 0$$

$$F = \begin{bmatrix} 0 & 0 & 0 & -0.8198 & 0.4270 & -0.0450 & -0.1942 & 0.3626v - 0.0946 & \\ 0.8718 & 3.8140 & -0.0754 & 0 & 0 & 0 & -0.5950 & 0.1294 & -0.0796 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$V = \left\{ \begin{bmatrix} 0 \\ 0 \\ x \end{bmatrix} \mid x \in [-9, 1] \right\}, i.e., \operatorname{Re}\left(\mathcal{Z}\left(\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T, \begin{bmatrix} 0 & 0 & -4 \end{bmatrix}^T, 5 \right) \right)$$

$$W = 0$$

Verification problem

For a given uncertainty of sampling period, the goal is to find the minimum possible reference distances $d^{ref} \in \mathbb{R}^3$ such that the vehicles should not collide, that is, $-e_i \leq d_i^{ref}\ \forall i \in \{1, 2, 3\}$. For our experiment, we fix the lower bound on sampling period as $0.01(s)$ and verify the minimum possible bounds for various values of $\tau_{\max} \in \{0.012, 0.014, 0.016, 0.018s, 0.02\}\, s$. We used our Algorithm 1 to verify the bounds taking $K = 3$. We then used repeated the same algorithm with simple real valued zonotope containing concatenation of real and imaginary parts of the complex template. We also used SpaceEx to verify the bounds and draw a comparison based on the smallest value of the bound verified.

Results

For the different values of $\tau_{\max} \in \{0.012, 0.014, 0.016, 0.018, 0.02\}\, s$, we could verify finite values for $\left[d_1^{ref}, d_2^{ref}, d_3^{ref}\right]$ such that $-e \leq d^{ref}$ using our complex algorithm with complex zonotopes. The verified bounds are given in Table 1. The computation time is less than 310s for every bound and sampling time interval below. On the other hand, the simple zonotope version of our algorithm was unsuccessful in finding finite bounds. SpaceEx terminated unsuccessfully without being able to find any bounds on the reachable set.

Effect of increasing number of sampling times for eigenvectors. We increased K to 5 and got the following smaller bounds for $\tau_{\max} = 0.2s$: $d_1^{ref} = 40$m, $d_2^{ref} = 29m$ and $d_3^{ref} = 19m$. But the computation time increased to $656s$.

Remarks

In a continuous feedback model without switching based on digital sampling, the SpaceEx verification found the bounds $d^{ref} = (30, 30, 16)$ as reported in [29]. However, when there is intermittent switching of feedback input with uncertainty in the sampling time, SpaceEx could not find bounds in our experiment. Possibly the inductive step-by-step reachability algorithm

Table 1 Verified bounds for vehicle platoon.

$\tau_{\min} = 0.01s$

$\tau_{\max}$	d_1^{ref}	d_2^{ref}	d_3^{ref}
$0.012s$	$35m$	$26m$	$17m$
$0.014s$	$38m$	$27m$	$18m$
$0.016s$	$40m$	$29m$	$19m$
$0.018s$	40	$29m$	$19m$
$0.02s$	$44m$	$33m$	$22m$

used in SpaceEx could not overapproximate the reachable set resulting from switching of states in sampling time interval. Also, the simple zonotope version of our algorithm could not find a sampling time invariant. On the other hand, our complex zonotope containing eigenvectors in its template is able to represent a finite invariant for various levels of uncertainty in sampling time. Furthermore, the fact that use of simple zonotope failed to compute invariant but complex zonotope containing complex eigenvectors is successful shows that use of complex eigenvectors increases the chance of finding invariant in the presence of complex eigenstructure.

6.2 Self-balancing two wheeled robot

This example concerns a verification problem for the model of a self-balancing two wheeled robot called NXTway-GS1[1] by Yorihisa Yamamoto, which was presented in the ARCH workshop [23]. We consider the linearized networked control system model from the paper. The state of the plant is represented by a 6-dimensional vector $x_p = (\dot{\theta}, \theta, \dot{\rho}, \rho, \dot{\phi}, \phi)^T$, where θ is the average angle of the left and right wheel, ρ is the body pitch angle, ϕ is the body yaw angle, and the rest coordinates are their respective angular velocities. The output of the plant is represented by a 3-dimensional vector $(\dot{\rho}_{\text{out}}, \theta_{m_1}, \theta_{m_r})^T$ such that $y_p = C_p x_p$. The input to the plant u_p is a 2dimensional vector. The plant gets feedback input from a controller whose state is a 5-dimensional vector $x_c = \left(\theta_{err}, \theta_{ref}, \dot{\theta}_{ref\_lpf}, \rho, \theta_{lpf}\right)$. The variable θ_{err} is integration of error between $\dot{\theta}$ and θ_{ref}, and integration of $\dot{\theta}_{ref}$ is θ_{ref}. The low pass filter applied to $\dot{\theta}_{ref}$ is $\dot{\theta}_{ref\_lpf}$, body pitch angle is ρ, and the low pass filter applied to average valued of left/right motor angle is θ_{lpf}. The output of the controller is a 2-dimensional vector which is used to compute the feedback input. There is also a 2-dimensional unknown disturbance input.

In the benchmark paper [23], the output of the controller is sampled at 4ms to update the feedback input. In our experiment, we also consider the case where there is uncertainty in sampling period a possible disturbance in estimated output due to inaccurate sensors. The original model has an eleven dimensional continuous state of the plant (6) and controller (5) and 5-dimensional input. The trajectories of the system were unbounded along a 3-dimensional subspace of the system which can be found by diagonalization. We decoupled these unbounded directions and performed model reduction to obtain a lower dimensional system. For performing this decoupling, we used linear transformation of the state space based on block diagonalization. The transformed system has an eight dimensional continuous state, 2-dimensional controller output and four dimensional

[1] `http://www.mathworks.com/matlabcentral/fileexchange/19147-nxtway-gs-self-balancing-two-wheeled-robot-controller-design`

input. The matrices of the transformed NCS, disturbance input and output error sets are given below.

$$A = \begin{bmatrix} 0.0000 & 0.0000 & 0.1241 & 0.5638 & 0.5774 & -0.0000 & 0.0066 & 0.5774 \\ -0.0000 & -92.4135 & 0.0000 & 0.0000 & 0.0000 & -0.0000 & -0.0000 & 0.0000 \\ -363.0957 & 0.0000 & -127.3488 & 112.2606 & 85.8204 & -0.0714 & -2.2691 & 89.4013 \\ 256.3623 & 0.0000 & 64.4048 & -77.3066 & -45.5575 & -0.3242 & -17.1890 & -29.2897 \\ 172.3365 & 0.0000 & 34.5588 & -51.1441 & -26.0550 & 0.6640 & -17.2627 & -9.3837 \\ 0 & 0 & 0 & 0 & 0 & -1.0000 & 0 & 0 \\ 2.9915 & 0.0000 & 0.8811 & -0.3305 & 0.0320 & -0.0038 & -0.1829 & 0.2342 \\ 122.3365 & 0.0000 & 45.3074 & -2.8986 & -26.0550 & -0.3320 & 33.3025 & -59.3837 \end{bmatrix}$$

$$B = \begin{bmatrix} -0.0000 & -0.0000 & 0 & 0 \\ -51.3265 & 51.3265 & 0 & 0 \\ 144.4698 & 144.4698 & 0 & 0 \\ -76.6947 & -76.6947 & 0 & 0 \\ -43.8551 & -43.8551 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -0.8985 & -0.8985 & 0 & 0 \\ -43.8551 & -43.8551 & 0 & 0 \end{bmatrix} \qquad D = 0$$

$$C = 1000 \times \begin{bmatrix} -0.6895 & 0.0000 & 0.1537 & 0.6846 & 0.0151 & -0.0135 & 1.1696 & -0.6538 \\ -0.6895 & 0.0000 & 0.1537 & 0.6846 & 0.0151 & -0.0135 & 1.1696 & -0.6538 \end{bmatrix}$$

$$F = \begin{bmatrix} 0.0809 & 0 \\ 0 & 0.0809 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad V = [-100, 100] \times [-100, 100] \qquad W = [-1, 1]$$

Verification problem

The safety requirement is that the *body pitch angle* ρ of the robot should be bounded within $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$. The verification problem for NCS we consider in this paper is to find the largest value of upper bound $\tau_{\max}$ on sampling time such that the system is safe, given a lower bound $\tau_{\min} = 0.1(ms)$ and the safe set $\rho \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. We used Algorithm 1 to verify safety bounds and subsequently find results for the above challenges based on complex zonotope. The same algorithm is repeated with a simple zonotope containing the concatenation of real and imaginary parts of our complex template. Concerning the experiment with SpaceEx using support functions, we tested with the octagon template and a template with 400 uniformly sampled support vectors distributed uniformly.

Results

Our algorithm based on complex zonotope could verify safety for $\tau_{\max} = 2$ms, given $\tau_{\min} = 0.1ms$. But SpaceEx could not find finite bounds on the reachable set for any value of uncertainty in sampling time. The simple zonotope based algorithm found a sampling time invariant, but its bounds were far over the threshold of safety, i.e., $\rho \in [-5.1\pi/2, 5.1\pi/2]$. The computation time for complex zonotope based algorithm is $153s$. We increased K to 7, but could not find any larger verified sampling time interval.

Remarks

The step-by-step reachability algorithm in SpaceEx possibly could not overapproximate resulting set of states from switching over the sampling time interval, due to which it failed to find bounds on reachable set. On the other hand, our complex zonotope containing eigenvectors in its template found bounded invariant which is within the limits of the safe set. Regarding the simple zonotope, although a sampling time invariant was found, the bounds were far over the threshold of safety. The reason complex zonotope is far more accurate than simple zonotope on this example is possibly that complex zonotope is geometrically more expressive, being able to encode nonlinear boundaries of invariants.

We saw that increasing K did not allow verification in a larger time interval. So, the strategy of sampling more time stamps for eigenvector template may not be the best strategy for improving the accuracy of verification. The problem remains open how to select sub-matrices for concatenation to the template to improve the accuracy of verification.

7 Conclusion

Given the pervasiveness of networked control systems with safety-critical applications, it is essential to develop verification algorithms for such systems in the presence of various possible inaccuracies in their execution. In this paper, we developed an algorithm to verify unbounded time safety of NCS with uncertain feedback sampling period, inaccurate output sensing and disturbance input. Our algorithm uses a novel set representation called complex zonotope that can capture convergence of forward reachable sets along eigenvectors and represent invariants. Complex zonotope is essentially an extension of simple zonotopes to the complex domain so as to efficiently compute invariants required by safety verification. Geometrically, their real projections represent a wider class of sets including some non-polytopic sets, while they retain the advantage of usual zonotope that the Minkowski sum and linear transformation can be computed efficiently. The practicality of our algorithm is demonstrated by successfully verifying benchmark examples with high dimensions (≥ 12 state+controller input variables), which the simple zonotope and another state-of-the-art tool failed to verify. An important direction for extension of this research is verifying NCS with non-linear differential equations and feedback. In this context, we need to find complex zonotope approximation of non-linear transformations and also conditions for checking invariance under non-linear transformation.

References

1 Arvind Adimoolam and Thao Dang. Augmented complex zonotopes for computing invariants of affine hybrid systems. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 97–115. Springer, 2017.

2 Arvind Adimoolam and Thao Dang. Template complex zonotopes for stability and invariant verification. In *2017 American Control Conference (ACC)*, pages 2544–2549. IEEE, 2017.

3 Arvind S Adimoolam and Thao Dang. Template complex zonotopes: a new set representation for verification of hybrid systems. In *2016 International Workshop on Symbolic and Numerical Methods for Reachability Analysis (SNR)*, pages 1–2. IEEE, 2016.

4 Arvind S Adimoolam and Thao Dang. Using complex zonotopes for stability verification. In *2016 American Control Conference (ACC)*, pages 4269–4274. IEEE, 2016.

5 Santosh Arvind Adimoolam. *A Calculus of Complex Zonotopes for Invariance and Stability Verification of Hybrid Systems*. PhD thesis, Université Grenoble Alpes, 2018.

6 Assalé Adjé, Pierre-Loïc Garoche, and Alexis Werey. Quadratic zonotopes. In *Asian Symposium on Programming Languages and Systems*, pages 127–145. Springer, 2015.

7 Assalé Adjé, Stéphane Gaubert, and Eric Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In *European Symposium on Programming*, pages 23–42. Springer, 2010.

8 Mohammad Al Khatib, Antoine Girard, and Thao Dang. Stability verification of nearly periodic impulsive linear systems using reachability analysis. *IFAC-PapersOnLine*, 48(27):358–363, 2015.

9 Xavier Allamigeon, Stéphane Gaubert, and Eric Goubault. Inferring min and max invariants using max-plus polyhedra. In *International Static Analysis Symposium*, pages 189–204. Springer, 2008.

10 Xavier Allamigeon, Stéphane Gaubert, Eric Goubault, Sylvie Putot, and Nikolas Stott. A fast method to compute disjunctive quadratic invariants of numerical programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

11 Xavier Allamigeon, Stéphane Gaubert, Nikolas Stott, Éric Goubault, and Sylvie Putot. A scalable algebraic method to infer quadratic invariants of switched systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(4):1–20, 2016.

12 Matthias Althoff. Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pages 173–182, 2013.

13 Rajeev Alur. Formal verification of hybrid systems. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 273–278, 2011.

14 Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

15 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

16 Jamal Daafouz, Pierre Riedinger, and Claude Iung. Stability analysis and control synthesis for switched systems: a switched lyapunov function approach. *IEEE transactions on automatic control*, 47(11):1883–1887, 2002.

17 Lei Ding, Qing-Long Han, Eyad Sindi, et al. Distributed cooperative optimal control of dc microgrids with communication delays. *IEEE Transactions on Industrial Informatics*, 14(9):3924–3935, 2018.

18 Mirko Fiacchini and Irinel-Constantin Morărescu. Set theory conditions for stability of linear impulsive systems. In *53rd IEEE Conference on Decision and Control*, pages 1527–1532. IEEE, 2014.

19 Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

20 Antoine Girard. Reachability of uncertain linear systems using zonotopes. In *International Workshop on Hybrid Systems: Computation and Control*, pages 291–305. Springer, 2005.

21 Antoine Girard and Colas Le Guernic. Zonotope/hyperplane intersection for hybrid systems reachability analysis. In *International Workshop on Hybrid Systems: Computation and Control*, pages 215–228. Springer, 2008.

22 Michael Grant, Stephen Boyd, and Yinyu Ye. Cvx: Matlab software for disciplined convex programming, 2009.

23 Thomas Heinz, Jens Oehlerking, and Matthias Woehrle. Benchmark: Reachability on a model with holes. In *ARCH@ CPSWeek*, pages 31–36, 2014.

24 Laurentiu Hetel, Jamal Daafouz, and Claude Iung. Lmi control design for a class of exponential uncertain systems with application to network controlled switched systems. In *2007 American Control Conference*, pages 1401–1406. IEEE, 2007.

25 Laurentiu Hetel, Jamal Daafouz, Sophie Tarbouriech, and Christophe Prieur. Stabilization of linear impulsive systems through a nearly-periodic reset. *Nonlinear Analysis: Hybrid Systems*, 7(1):4–15, 2013.

26 Kyoung-Dae Kim and Panganamala R Kumar. Cyber–physical systems: A perspective at the centennial. *Proceedings of the IEEE*, 100(Special Centennial Issue):1287–1308, 2012.

27 Alexander B Kurzhanski and Pravin Varaiya. Ellipsoidal techniques for reachability analysis. In *International Workshop on Hybrid Systems: Computation and Control*, pages 202–214. Springer, 2000.

28 Michal Kvasnica. Minkowski addition of convex polytopes, 2005.

29 Ibtissem Ben Makhlouf and Stefan Kowalewski. Networked cooperative platoon of vehicles for testing methods and verification tools. In *ARCH@ CPSWeek*, pages 37–42, 2014.

30 Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

31 Stefan Pettersson and Bengt Lennartson. Hybrid system stability and robustness verification using linear matrix inequalities. *International Journal of Control*, 75(16-17):1335–1355, 2002.

32 Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In *International Workshop on Hybrid Systems: Computation and Control*, pages 477–492. Springer, 2004.

33 Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007.

34 Henrik Sandberg, Saurabh Amin, and Karl Henrik Johansson. Cyberphysical security in networked control systems: An introduction to the issue. *IEEE Control Systems Magazine*, 35(1):20–23, 2015.

35 Sriram Sankaranarayanan, Thao Dang, and Franjo Ivančić. Symbolic model checking of hybrid systems using template polyhedra. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 188–202. Springer, 2008.

36 Ashish Tiwari. Generating box invariants. In *International Workshop on Hybrid Systems: Computation and Control*, pages 658–661. Springer, 2008.

37 Yu-Long Wang and Qing-Long Han. Network-based modelling and dynamic output feedback control for unmanned marine vehicles in network environments. *Automatica*, 91:43–53, 2018.

38 Xian-Ming Zhang and Qing-Long Han. Network-based h∞ filtering using a logic jumping-like trigger. *Automatica*, 49(5):1428–1435, 2013.

Swarms of Mobile Robots: Towards Versatility with Safety

Pierre Courtieu
Conservatoire des arts et métiers, Cédric EA 4629, Paris, France

Lionel Rieg
VERIMAG, Grenoble INP – UGA, CNRS UMR 5104, Université Grenoble-Alpes, Saint Martin d'Hères, France

Sébastien Tixeuil
Sorbonne University, CNRS, LIP6, Paris, France

Xavier Urbain
Université de Lyon, Université Claude Bernard Lyon 1, CNRS, INSA Lyon, LIRIS, UMR 5205, F-69622 Villeurbanne, France

Abstract

We present Pactole, a formal framework to design and prove the correctness of protocols (or the impossibility of their existence) that target mobile robotic swarms. Unlike previous approaches, our methodology unifies in a single formalism the execution model, the problem specification, the protocol, and its proof of correctness. The Pactole framework makes use of the Coq proof assistant, and is specially targeted at protocol designers and problem specifiers, so that a common unambiguous language is used from the very early stages of protocol development. We stress the underlying framework design principles to enable high expressivity and modularity, and provide concrete examples about how the Pactole framework can be used to tackle actual problems, some previously addressed by the Distributed Computing community, but also new problems, while being certified correct.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Self-organization; Theory of computation → Program reasoning; Theory of computation → Logic; Software and its engineering → Formal methods

Keywords and Phrases distributed algorithm, mobile autonomous robots, formal proof

Digital Object Identifier 10.4230/LITES.8.2.2

Supplementary Material *Software (Coq Formalization)*: `https://pactole.liris.cnrs.fr`

Funding This work was partially supported by CNRS peps DiDaSCaL and ANR project SAPPORO 2019-CE25-0005.

Acknowledgements The authors would like to thank the referees whose comments and suggestions helped improve the presentation of this work.

Received 2020-07-09 **Accepted** 2022-01-28 **Published** 2022-12-07

Editor Alessandro Abate, Uli Fahrenberg, and Martin Fränzle

Special Issue Special Issue on Distributed Hybrid Systems

1 Introduction: low cost and high expectations

Swarm Robotics envisions groups of mobile robots self-organizing and cooperating toward the resolution of common objectives, such as patrolling, exploring and mapping disaster areas, constructing ad hoc mobile communication infrastructures to enable communication with rescue teams, etc. In many cases, such groups of robots are deployed in adverse environments (e.g. space, deep sea, disaster areas). Thus, a group must be able to self-organize in the absence of any prior infrastructure and ensure dynamic coordination in spite of the presence of faulty robots as well as environmental changes. A faulty robot can stop its execution (crash) or start to behave in an

arbitrary way either due to some external factors (e.g. electromagnetic fields, attacks) or to some inaccurate information received by its own sensors.

Sending a single complex expensive robot for example to explore a dangerous area is often not the best solution, as some environments are likely to destroy robots in a matter of hours.[1] Instead, using a large number of cheap simple robots one can afford to lose, but that nonetheless are able to coordinate to *globally* solve a given task, is the underlying principle of swarm robotics.

The possibility to have a robot fail being almost certain, the operator has to use cheap robots to form the swarm, implying that very weak individual sensing and actuating capacities are to be privileged. Lowering the individual robots' abilities raises an important theoretical question: which *individual* capacities are necessary (and sufficient) to *collectively* solve a given task? Obviously, robots with more abilities than necessary to solve a task can still solve it. Conversely if a given task solvability requires a particular capacity, and that capacity is unavailable to the robots forming the swarm, no genius algorithm or protocol can come to the rescue.

Another consequence of failure likeliness is that no robot in the swarm should have a *particular* role to assume (e.g. a leader from which other robots wait for orders). Indeed, if the particular robot ceases functioning or starts behaving arbitrarily, the entire swarm fails. Instead, all robots are to be given the same role, and self-organization is to be used to take collective decisions regardless of the current situation.

On the other hand, proving task solvability requires to envision all situations, even the most unlikely ones. A classical setup only considers the system behaviour from a given well-formed initial state. Proofs written in the context of swarm robotics must consider all possible failure occurrences, be they at the individual robot level or induced by a catastrophic change in the environment.

Actual deployment of mobile robotic swarms mandates preliminary theoretical assessments, to ensure the swarm behaves according to its specification, and to assess its practical feasibility with respect to expected completion time and used resources. While the latter is typically quantified through simulations, the former requires a sound mathematical proof of correctness. Most of the literature makes use, for this purpose, of handwritten proofs. As recent findings show [1, 14, 32, 33], handwritten proofs are error-prone, and sometimes erroneous, which may compromise the safety and the correctness of actual deployments. Hence, a recent trend deals with computer-aided proving of important properties for mobile robotic swarms.

Following this trend, our focus in this paper is to propose a unified formal approach that permits us to express both the execution model and its variants, and the property specifications and their proof, relating all that we formally state to the usual model in the Distributed Computing Community. In more detail, our starting point is the model by Suzuki and Yamashita [65] (who was recently awarded the *Prize for Innovation in Distributed Computing*), extended by the many variants the Distributed Computing developed throughout the years [37], unified in a modular formal framework developed in Coq.

Our framework is meant to answer legitimate questions that arise when developing protocols for mobile robotic swarms:

1. Is algorithm A a solution to problem X in model variant Y?
2. Is problem X solvable using model variant Y?
3. Which problems are solvable using model variant Y?
4. What is the weakest model variant that permit to solve problem X?
5. Does the proof for algorithm A remain valid if we switch model variant Y for model variant Z?
6. etc.

[1] https://www.theguardian.com/world/2017/mar/09/fukushima-nuclear-cleanup-falters-six-years-after-tsunami

The remainder of the paper is organized as follows. Section 2 reviews previous work related to the use of formal methods in the context of Distributed Computing, as well as an introductory example. The PACTOLE library for the COQ proof assistant is presented in Section 3. In Section 4, we review the structure of model variants used by the Distributed Computing community for mobile robotic swarms, while its formalization in our framework is presented in Section 5. Several case studies in Section 6 demonstrate the simplicity and universality of our approach. Finally, concluding remarks are presented in Section 8.

2 Formal approaches and their complementary uses

Motivations

Models of distributed computations are traditionally presented in natural language. But the algorithm, even when presented as pseudo-code [49], cannot be understood without the *precise setting* in which it is executed. *Implicit assumptions* [50] are known as folklore but as properties of distributed algorithms rely on very subtle hypotheses, in most cases a small barely noticeable shift in the statement of these assumptions may induce dramatic changes in an algorithm behaviour. Such shifts can still be found in the literature, and have led to erroneous results being published.

Formal methods tackle this difficulty, most notably with the use of tools providing a mathematical language that is non-ambiguous [62]. Given how challenging the task of establishing correct results in the area of distributed computing is, *the simple fact of using a common non-ambiguous format for definitions constitutes a significant asset.*

So far the most popular formal method in the Distributed Computing community has been *model checking*. Formal methods however encompass a wide variety of other methods. *Formal proof* for instance has little to do with model checking, and in most aspects it can be considered as its dual: formal provers can be applied to almost any domain of mathematics [55, 42, 41, 2, 67] but are poorly automated when dealing with higher order properties, whereas model checkers only apply on decidable (hence less expressive) logics but are highly automated. Model checkers produce counter-examples whereas theorem provers do not (at least systematically), etc.

Model Checkers

The power and elegance of model checking lie in building an *abstraction* of the property to prove, tailored so that its validity can be checked *exhaustively* by an automated tool. The correctness of the abstraction being in general proved on paper.

Model checking has been used with impressive success for distributed protocols, both in proving [14, 34, 35, 54, 51, 43, 28], and disproving [32, 33, 14] their correctness. In some cases, it was possible to go as far as program synthesis [16, 59, 36, 31] (that is, generating algorithms that are correct by design using a computer program). It may however be subject to combinatorial explosion, or become undecidable [4].

Consequently, model checking often deals with *instances* of a problem rather than with its full generality. Parameterized model checking sometimes allows for model checking all instances where an (infinite) parameter varies. For example, Sangnier et al. [64] makes use of Presburger formulae to express mobile robotic swarms operating on a discrete space (a ring of size n, meant to be arbitrary, and a parameter of the model). However, a key result of Sangnier et al. [64] in this context is that non-trivial properties (namely, liveness properties) are undecidable. Those recent findings command studying complementary techniques like formal proofs.

Formal Proofs

The formal proof approach consists in writing mathematical proofs in a fully explicit way, leaving absolutely no reasoning detail hidden or implicit. This is (obviously) a very tedious task, and it cannot really be applied without the help of mechanical *tools*, called *proof assistants*, that provide

(1) a language for mathematical definitions and properties;

(2) an interactive system assisting the user in writing all details of the proofs, thus ensuring their correctness *by construction*.

Since the proof system is not bound to be fully automatic, very expressive (undecidable) logics are allowed, making it possible to write virtually any mathematical definition, as witnessed by the wide range of mathematical results that have been proven using these tools [55, 42, 41, 2, 67, 48].

A drawback of being very expressive is an induced lack of automation. Despite the help of a variety of decision procedures for decidable sub-logics of the system, developing proofs in a proof assistant still requires a lot of expertise.

Given its characteristics, one can expect formal proof to be successfully applied in distributed computing, but in a way that is complementary to the model checking approach. Figure 1 gives hints on where a proof assistant and model checking are potentially usable in the everyday life of a researcher in distributed algorithms.

Primarily, it can be used as the *underlying non-ambiguous language* for all *definitions* involved in Distributed Computing: from high-level model specifications to low-level algorithms and all their properties. This is a specific complementary benefit of proof assistants. For instance in our setting it is possible to state and prove properties explicitly quantified over continuous spaces like $\mathbb{R}^2$ or the type of all protocols (functions over functions on $\mathbb{R}^2$):

```
∀ r:(robot → R²) → R², ...
```

or over types populated with infinite objects like demons (infinite streams):

```
∀ d: Stream demonic_action, ...
```

This makes possible to state for instance that a given task is *impossible* to achieve [6, 23, 12] in some model, i.e. that *for all protocols* (even those that cannot be computed with usual operations) there exists an adversary (demon) that will make the protocol fail.

Even if this part is *much* more intricate and needs dedicated expertise, the proof assistant can also be used to *prove* these properties. It is notably more tedious than writing a pen and paper proof because of the required level of details, and it requires expertise in the assistant involved. Expecting an expert in any domain of computer science to become also an expert with a proof assistant it thus somewhat unrealistic at this time. It also misses the point as the fundamental first step consists in providing formal definitions, and not proofs.

Finally the architecture of proof assistants allows for building large libraries of shared definitions for models, problems, protocols, and theorems, ensuring their mutual consistency, and *reusability*. In the long run this makes possible new and increasingly intricate but sound results.

To this goal, collaborations between experts of distributed algorithms and formal proof are needed and the Pactole library is an example of such collaboration, among others [3, 19].

3 The Pactole library for the Coq proof assistant

3.1 The Coq proof assistant

The proof assistant used for this work is Coq [5, 15]. It is based on type theory and its language for definitions and properties is a very rich typed λ-calculus: the calculus of inductive constructions [22, 15].

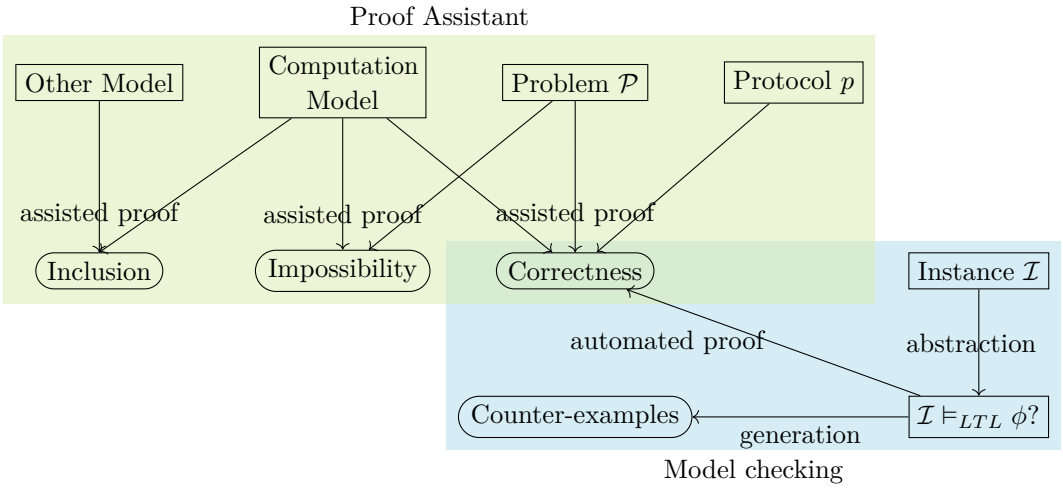


Figure 1 Potential uses of a proof assistant: Definitions, statements of results and their proofs.

A particularity of proof assistants based on type theory is that the *definition* language is also used to express *proofs*. More precisely it makes use of the Curry-Howard correspondence: types are considered as properties, and a term of a given type is actually a *proof* of this property. Checking correctness of proofs therefore amounts to type-checking λ-terms. Since this is a rather simple task, a small kernel can be written, and proofs accepted by the proof assistant have strong guarantees of correctness.

The syntax of the Coq language is similar to that of a functional programming language *à la* ML. Function types are written in the usual Curryfied way: `A → B → C` denotes the type of functions that take a parameter of type `A` and return a function from `B` to `C`, which can also be seen as the type of functions that take an `A` and a `B`, and return a `C` but can be partially applied to their first argument. Applying a function `f` to an argument `a` is simply denoted by juxtaposition: `f a`.

Function definitions and type synonyms are introduced with **`Definition`**. New data types may be defined either by their exhaustive list of constructors with **`Inductive`** (and the pattern matching of such a type is done by the **`match`** ... **`with`** ... **`end`** construction) or as a **`Record`** whose values are of the form `{| fld1 := val1; fld2 := val2 ... |}`, and fields can be accessed by the usual dot notation. For example: let `x` be the record `{| f1 :=` a `; f2 :=` b `|}`, the expression `x.(f1)` has value a.

A particular construction allows for defining (sub-)types by intention: `{ x : T | P x }` represents the type of any element `x` of type `T` *paired with* a proof that `P` holds on `x`.

Coinductive types (mainly infinite streams in our context) can be easily defined in Coq and coinductive values are introduced by **`cofix`**.

3.2 Pactole

Developed with and for the Coq proof assistant, Pactole is a library gathering definitions and proofs on a variety of models of robot swarms. It implements the generic seminal model by Suzuki and Yamashita [65] presented in details in Section 4.1.

Formally proven results are correct by construction and can therefore be highly trusted and reused. It is worth noting that it is still the responsibility of the experts of a certain domain to check *what* those results are the proof of. It is indeed critically important that the definitions

are scrutinized and validated by the community. The *proofs* themselves, while sometimes worth sketching, need not to be human-checked.[2] In that respect, a focus in Pactole is on *the ease to write and read specifications*.

Designed for robots and in particular agents that are *mobile*, Pactole provides a wide range of definitions and proofs, from very high level notions to concrete protocols and their properties:

- Definitions of models, proofs of relations between models (inclusion, equivalence, etc.);
- problem definitions (gathering, exploration, etc.);
- protocol definitions and proofs of correctness;
- proofs of impossibility.

All these notions are inter-dependent. One of the benefits of proposing the widest range of notions of the domain is that they share the same underlying definitions. They are therefore consistent with each other *by construction*.

For example, when dealing with impossibility results for some problem P, and protocols solving P under particular assumptions, sharing the definition of P ensure that these results are correctly linked together.

As another example, when two models m_1 and m_2 are proven equivalent (any execution possible in one is also possible in the other), then any proof made using the definition of m_1 can be transferred to m_2 *without any risk* of a shift in the definitions.

In this article

All the results we present in this paper have been fully formalized and proved in Coq and Pactole. For the sake of clarity a few definitions given in the following have been slightly stripped of some technical details. The actual formal development is publicly available at `https://pactole.liris.cnrs.fr`.

3.3 A tour of formal proof for robotic swarms

A formal semantics of a dynamic system (processor, virtual machine, robot swarm, physical system, etc) is a mathematical object that mimics perfectly the aspects of the behaviour of the system under consideration. All possible behaviours of the system must be possible in the model, and any impossible behaviour of the system must also be impossible in the model. Some things may be left out of scope of the model, usually on the basis of being irrelevant to the particular problem under study. For instance, we may ignore thermal radiation from the sun slowly heating up robots, as in most cases this does not result in any noticeable behavioural change. Or, while modelling a processor, we may choose not to model its performance counters and their associated instructions.

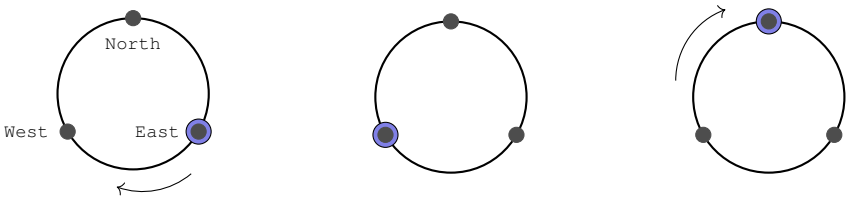
The mathematical object modelling the system is generally a function[3] taking as input the state of the environment of the system and returning as output the *evolution* (the new state) of the system in this environment. In the following we give a series of examples of increasing complexity showing how we model different systems in robotic swarms.

3.3.1 A simple example

Suppose we want to model a single robot evolving on a ring with only three nodes in the following way: when reaching a node in the ring, the robot selects the next node clockwise and starts moving toward this new target.

---

[2] The actual work of reviewing in the context of formal proof is discussed by Bauer and Mahboubi [13, 56].
[3] It may be a relation instead, for instance if the system is non-deterministic.



If we suppose that the robot cannot be interrupted during its move from one node to another, it is sufficient to model the topology with three positions `North`, `East` and `West`. A configuration is then given by a function returning the position of the robot.

```
Inductive Position : Type := North | East | West.
Definition Configuration := robot → Position.
```

It may seem silly to model a configuration with a single robot as a function but this representation generalizes to an arbitrary number of robots, so we choose to use it from the start.

The evolution of the system can then be formalized as a function `round` that takes the configuration and returns the new configuration after one round.

```
Definition round (c : Configuration) : Configuration :=
  fun id:robot ⇒
    match c id with      (* take position of id and compute the next one. *)
    | North ⇒ East
    | East ⇒ West
    | West ⇒ North
    end.
```

We can now define an execution as an infinite stream of configurations obtained by successive applications of `round`,[4] and it is quite straightforward to prove for instance that in any such execution starting from a valid configuration (one position occupied by a robot) from any moment every node is occupied infinitely many times.

```
Lemma all_pos_occupied_eventually : ∀ (c : Configuration) (p : Position),
    Stream.eventually (fun str ⇒ (hd str) Robot1 = p) (execute c).
Proof.
  intros c p.
  (* by cases on positions and configurations *)
  destruct p;destruct (c Robot1) eqn:heq;
    try (constructor 1;now auto);
    try (constructor 2; constructor 1;
         unfold execute, round; simpl;
         rewrite heq; reflexivity);
    try (constructor 2; constructor 2; constructor 1;
         unfold execute, round; simpl;
         rewrite heq; reflexivity).
Qed.


Lemma all_pos_occupied_forever : ∀ (c : Configuration) (p : Position),
    Stream.forever (Stream.eventually (fun str ⇒ (hd str) Robot1 = p))
                   (execute c).
```

[4] See Figure 5 for the coinductive definition of `execute` in Coq.

```
Proof.
  cofix HI.
  constructor.
  - apply all_pos_occupied_eventually.
  - simpl.
    apply HI.
Qed.
```

The important statement here is that *what is true for this function is also true for the computation model it represents.*[5] In other words we have reduced the problem of proving properties of the model to the problem of proving properties of a well defined function; a task theorem provers are perfectly suited for.[6]

3.3.2 The local computations

There is however a problem with this formalization. In the `round` function above, something important is left implicit: we formalized the decision of the robot *without defining the actual embedded algorithm operation*. Instead, we have only formalized a *centralized* protocol. This is a serious gap between the model we want to represent (autonomous robots) and our formalization. Distributed algorithms have very subtle behaviours, in particular because the code is executed on different devices "viewing" the global system from different perspectives. Any attempt to model distributed systems that jumps directly to a centralized vision like this would miss the important, and most difficult part, of distributed systems. In order to represent faithfully the distributed nature of our model we need to separate the computations done locally by each robot from the global behaviour of the system they yield.

To illustrate this, we need to define what the perception of the robot is. In this section we suppose that the robot sees the ring but cannot detect the "real" identity of a node. The robot sees the whole ring and knows on which node it stands, though it has no knowledge of whether it is actually `North`, `East` or `West`. The robot enjoys *chirality*: it can distinguish the node on its left (clockwise) from its right (counterclockwise). Let us rework our example to represent the distributed protocol. In the following we distinguish between two notions:

- the *global* configuration: what is really happening in the system, and where; this viewpoint is called the *global frame of reference* or the *demon's frame of reference*;
- the (local) observation: the global configuration *as seen by the robot*. In our case the observation is composed of three nodes `Me`, `Left` and `Right` named after their positions relative to the observing robot. An observation is a function giving the position of all robots relative to the observing robot. This viewpoint is called the *local frame of reference* or the *robots's frame of reference*.

```
Inductive RelativePosition : Type := Me | Left | Right.
Definition Observation : Type := robot → RelativePosition.
```

The protocol takes as input an observation and produces a *decision*: where to go next, expressed in its own frame of reference.

We reformulate the `round` function with an explicit call to the protocol on the observation by the robot. More precisely `round`

[5] The fact that the represented model itself is the model accepted by the community needs a validation by experts.

[6] Actually our claim is that humans also take benefit in agreeing that the function is accepted as *the actual true definition* of the model, once approved by the community.

(1) establishes the robot's observation,
(2) passes it as a parameter to the protocol and takes back the returned decision (expressed in the robot's own frame of reference), and then
(3) determines where the robot moves in the global frame of reference.

Note that the protocol we consider in our example is very simple: since the robot sees itself at `Me`, its target is always the `Left` node in its own frame of reference.

Denoting by `relative_config` the "localized" version of the configuration where nodes are described relatively to an observing robot (eg. in the first figure of Section 3.3.1 where the robot is at `East`, `East` becomes `Me`, `West` becomes `Left`, `North` becomes `Right`) and `globalise_pos` the converse function mapping local names to global ones, we obtain the simple code:

```
(* Always go left, since this is clockwise. *)
Definition protocol (o : Observation) : RelativePosition := Left.
Definition round (c : Configuration) :=
 fun id : robot ⇒
  let pos_id := c id in                                    (* Where is id? *)
  let obs := relative_config c pos_id in(*id sees c rotated with id on Me*)
  let destination := protocol obs in      (* Call protocol on observation *)
  globalise_pos pos_r destination.     (* Rotate back to global reference *)
```

It is now *provable* that this protocol behaves globally like the centralized version described in Section 3.3.1.

```
Lemma equiv_centralized :
  ∀ c : Configuration, execute c ≡ Centralized.execute c.
Proof.
  cofix HI.
  intros c ; constructor; [ simpl;reflexivity | apply HI].
Qed.
```

This kind of proof may be quite difficult on realistic protocols. It generally (although not in this case) relies on the fact that the algorithm consists in operations that are invariant relative to the frame of reference.

Note that we can consider another distributed algorithm in the same model just by changing the protocol operations.

3.3.3 Weakening the sensing capabilities of robots

In this section, we change the computation model and the `round` function to account for more realistic sensors. In this new model, the robot's compass may be subject to arbitrary recalibration at each round and change its *chirality*, i.e. its `Left` node may correspond to its clockwise or counterclockwise neighbour. Clearly the previous algorithm in this model behaves completely differently and does not satisfy the same properties, because when choosing always `Left` the robot may actually go counterclockwise.

The chirality reversal of the robot is not controllable and may change at each round, each leading to a different possible execution step. To account for this variability of execution we reformulate the model: `round` now takes a new parameter: a `flip` function that selects the chirality of the robot at the current round. The protocol is now called on the possibly flipped observation to simulate the new calibration of the sensors.

More generally, the uncontrollable part of the environment, that is the *alea* the protocol must be robust to, should be defined as a parameter of `round`, so that we can reason *for all* possible values.

Denoting by `mirror_pos/_obs` the reversing of `Left` and `Right` we obtain the code:

```
(* flip id = false → robot id has clockwise orientation,
             true  → counterclockwise orientation *)
Definition round (c : Configuration) (flip : robot → bool) :=
 fun id : robot ⇒
  let pos_id := c id in
  let c_local := relative_config c pos_id in
  let obs := if flip id then mirror_obs c_local else c_local in (* flip? *)
  let dest := protocol obs in            (* Call protocol on observation *)
  let dest_swap := if flip id then mirror_pos dest else dest in (*unflip?*)
  globalise_pos pos_id dest_swap.     (* Rotate back to global reference *)
```

With this version of the formal model it is now possible to prove that, for example, there exist executions that never reach, say, node `East`. It suffices to use an infinite sequence of `flip` functions alternating the chirality of the robot, so that it would go alternatively to `North` and `West`.

```
Lemma exist_never_reaching_East: ∃ c (d : Stream.t (robot → bool)),
    Stream.forever (fun strm ⇒ hd strm Robot1 ≠ East) (execute d c) .
Proof.
  ∃ (fun id ⇒ match id with _ ⇒ North end).        (* initial position *)
  ∃ (alternate (fun x ⇒ true) (fun x ⇒ false)).   (* alternating demon *)
  cofix HI. constructor.
  - simpl. discriminate.
  - simpl. constructor.
    + simpl. discriminate.
    + apply HI.
Qed.
```

3.3.4 Modeling Concurrency

Until now, our example involved one robot only. To model several distributed agents acting at the same time we need to determine the level of synchronicity of the agents. In the version above (Sections 3.3.2 and 3.3.3) we can see that `round` always applies the protocol at each round: the output position of *any* robot is obtained by calling the protocol on its observation. In other words, if multiple robots are present it *activates* all robots at each round. This model where all robots are always active at the same rate is called *fully synchronous* and is presented in more details together with others in Section 4.4.

We can relax this constraint to obtain a more loosely synchronized model: for some reason robots may not be all activated at each round. Similarly to chirality flips in the previous section, the subset of robots activated at each round is not controllable. Thus, in the same fashion that the `flip` parameter allows for quantifying on the uncontrollable variability of the sensors, a new parameter is added to account for the variability of scheduling. To avoid multiplying parameters we group uncontrollable parameters into a single record argument. This argument is called *demonic action* in the following in reference to the view of the environment as an adversary trying to make the protocol's task fail.

```
Record Demonic_action : Type := {
  activate : robot → bool; (* activated at his round? *)
  chirality : robot → bool; (* inverted chirality at this round? *)
}.
Definition round (da : Demonic_action)(c : Configuration)  :=
```

```
fun id : robot ⇒
 if negb (da.(activate) id) then c id   (* if not activated, don't move *)
 else                    (* else move according to protocol and chirality *)
  let pos_id := c id in
  let c_local := relative_config c pos_id in
  let obs := if da.(chirality) id then mirror_obs c_local else c_local in
  let dest := protocol obs in
  let dest_swap := if da.(chirality) id then mirror_pos dest else dest in
  globalise_pos pos_id dest_swap.
```

In this version of the model, most provable properties depend upon a hypothesis on the *fairness* of the scheduler, i.e. constraints on the successive values of `activate` in the demonic action. See Section 4.4.

3.3.5 Other refinements

In the PACTOLE library, we model all the variants of the model described above and a few others.

The demonic action encompasses all external effects altering the operation of our robots. For example, the ground may be slippery and although robots try to reach a different node, some of them may simply not move. In this case, instead of reaching their target, the new locations of the robots depend on another Boolean function provided by the demonic action, say `reach`, expressing whether robots stay on their current location or reach their targets. Another example: a form of asynchronicity can be modelled by considering that robots may be activated while other have not yet reached their destination, etc.

One does not have to keep extending the definitions of demonic action and `round` for each additional environmental effect: there is a general version that can encompass a wide range of external forces and is explained in Section 5.1. In order to get to this point, we must first present the seminal model by Suzuki and Yamashita [65] and the corresponding lattice of models.

4 A lattice of models

Research in Distributed Computing has traditionally considered three complementary approaches:

complexity-driven when a particular problem can be solved in a particular model, it becomes interesting to reduce the complexity of the solutions. Various metrics can be considered, such as memory, time, number and size of exchanged messages, size of a causal chain of events, etc.

model-driven when a particular model is designed for distributed computations (usually mimicking actual networks or systems), it becomes interesting to characterize the set of problems that are solvable in this model.

problem-driven when a particular problem is considered important in a general setting, it becomes interesting to characterize the models that enable solutions of the problem, and the models that make the problem impossible to solve.

The domain of mobile robotic swarms is mostly problem-driven. The focus of past efforts have thus consisted in characterizing which hypotheses are necessary and sufficient to solve a particular problem. Since the various hypotheses considered are sometimes unrelated, it becomes difficult to compare two different models with different sets of hypotheses. However, some particular hypotheses can be ordered, inducing a partial ordering among models. This partial ordering is important for two reasons: *(i)* if a model X is "weaker" than another model Y (in the sense that fewer computations are possible in model X than in model Y), then a solution to a problem considering model Y is also a solution to the problem considering model X, and *(ii)* if a given

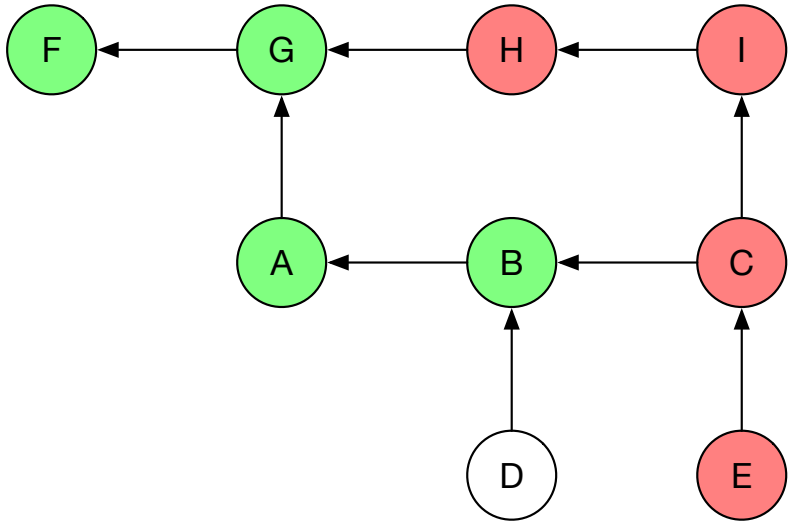


Figure 2 A lattice of models, with proofs on models B and H carrying to other models in the lattice.

problem admits no solution in model X, then it also admits no solution in model Y. Since the ordering between models is only partial, it is possible that two distinct models are both necessary and sufficient for solving a particular problem, albeit being unrelated with respect to the partial order.

Figure 2 depicts a possible partial order of models, where $X \leftarrow Y$ denotes the fact that X is a weaker model than Y. For a given problem, one was able to prove that a solution exists assuming model B, but that no solution exists assuming model H. Hence, from the partial hierarchy of models, it is possible to deduce that the problem is also solvable in models A, G, and F (that are weaker than B), and impossible to solve in models I, C, and E (that are stronger than H). From the current results, it remains unknown whether the problem is solvable assuming model D.

Hypotheses about the model span across various dimensions. The main ones are:

synchronization relates to the fact that mobile robots have independent control flow, and may thus execute their protocol at different paces;

memory relates to the fact that robots may make use of persistent memory, and may want to store various kinds of data (*e.g.* bits or Euclidean positions);

sensors relates to the fact that robots may have limited sensing capabilities, or limited ability to receive messages from other robots;

actuators relates to the fact that robots may have unreliable motion actuators;

faults relates to the fact that robot may follow their prescribed protocol or deviate from it [29].

The rest of the section presents the main relevant hypotheses that have been considered since the paper of Suzuki and Yamashita, and their induced order.

4.1 The Suzuki and Yamashita model

The seminal paper for studying robotic swarms from a Distributed Computing perspective is due to Suzuki and Yamashita [65]. They introduce, in this paper, a mathematical model for studying geometric pattern formation by swarms of possibly oblivious robots. The motivation for studying oblivious robots (that is, robots that do not retain history of past actions) is resilience to faults. For example, if a robot crashes, after rebooting it should not trust the content of its memory, either because it might be corrupted, or because it refers to an outdated view of the system. Ignoring past actions forces the design of algorithms that are simply more versatile.

In the Suzuki and Yamashita initial model, robots are represented as dimensionless points evolving in a bidimensional Euclidean space (that is, $\mathbb{R}^2$), and can accumulate on the same location. They operate in *Look-Compute-Move cycles*. In each cycle, a robot "Looks" at its surroundings and obtains (in its own coordinate system) a snapshot containing some information about the locations of all robots. Based on this visual information, the robot "Computes" a destination location (still in its own coordinate system), and then "Moves" towards the computed location. When the robots are oblivious, the computed destination in each cycle only depends on the snapshot obtained in the current cycle (and not on the past history of actions). The visual snapshots obtained by the robots are not necessarily consistently oriented in any manner. Then, an *execution* of a distributed algorithm by a robotic swarm consists in having every robot repeatedly execute its Look-Compute-Move cycle. In general, executions are *infinite* (even if robots do not move after a while, they still look, compute and decide not to move) and *fair* (every robot executes an infinite number of Look-Compute-Move cycles).

Although this mathematical model is perfectly precise, it allows a great number of variants (developed over a period of 20 years by different research teams [37]), according to the various dimensions described above, namely: sensors, memory, actuators, synchronization, and faults, which we investigate now. This flurry of subtly different models makes reasoning very error-prone as it can easily happen that one designs a protocol in a model, and derives its proof in a slightly different one, without noticing the difference. A summary of all model variants explored in this section in depicted on Figure 4. In order to remain readable, we present all the dimensions separately, so that the overall lattice must be understood as the Cartesian product of all these smaller lattices.

4.2 Sensors

Robots perceive their surroundings through sensors, whose abilities have strong impact on task solvability. The most commonly considered types of sensors vary along several capabilities, described below. Obviously, one can think of other kinds of sensors not described here. For instance, in a completely opaque environment, one may imagine that the only available information is by direct contact through bumpers.

4.2.1 Range

The most obvious parameter of sensors is their range, that denotes how far a robot can sense another robot's location:

full visibility robots are able to sense every other robot's location, regardless of distance;

limited visibility there exists $\lambda > 0$ such that robots are able to sense every other robot's location if their distance to the observing robot is less than λ, and are unable to sense the locations of other robots [38];

k-random there exists $\lambda > 0$ such that robots are able to sense every other robot's location if their distance to the observing robot is less than λ, and up to k robots at distance more than λ, chosen uniformly at random, cannot be sensed (the other "distant" robots can be sensed) [45];

k-enemy there exists $\lambda > 0$ such that robots are able to sense every other robot's location if their distance to the observing robot is less than λ, and up to k robots at distance more than λ, chosen by an adversary, cannot be sensed (the other "distant" robots can be sensed) [45].

Note that in general, robots are not aware of λ. Obviously, a protocol assuming limited visibility is strictly more powerful than one that requires full visibility.

4.2.2 Multiplicity detection

Multiplicity refers to the ability of robots to distinguish (to some extent) the number of robots sharing a given location. There are three variants about the accuracy, ordered by decreasing strength:

no multiplicity detection sensors can only distinguish occupied and unoccupied location, but any estimation about the number of robots present remains unknown;

weak multiplicity detection sensors can distinguish between a single robot or more than one robots at a location, but not their precise number [47];

strong multiplicity detection sensors can accurately count the number of robot at a location.

Another axis for variants is related to the range of the multiplicity detection:

local multiplicity detection indicates that weak or strong multiplicity information is only observable for the position of the observing robot (that is, a robot can only obtain multiplicity information about its own location) [46];

global multiplicity detection indicates that weak or strong multiplicity information can be obtained for all observed positions (that is, a robot can obtain multiplicity information about all locations in its viewing range).

Overall, we thus have five variants for multiplicity: no multiplicity, weak local multiplicity, weak global multiplicity, strong local multiplicity, and strong global multiplicity. Obviously, an algorithm assuming no multiplicity detection is more powerful than one requiring any of the other assumptions. However, some assumptions are uncomparable, *e.g.* weak global multiplicity and strong local multiplicity.

4.2.3 Orientation

This refers to the ability of robots to share some common notion of direction or orientation. Again, there are many variants:

common direction the robots have the same North-South and/or East-West axes, but the direction along these axes may be inverted (this is also called two-axes direction) [40];

common orientation in addition to having the same two axes direction, robots may also share orientation on either one axis (e.g. North only) or two axes (e.g. North and West);

common chirality robots have the same notion of left and right [39].

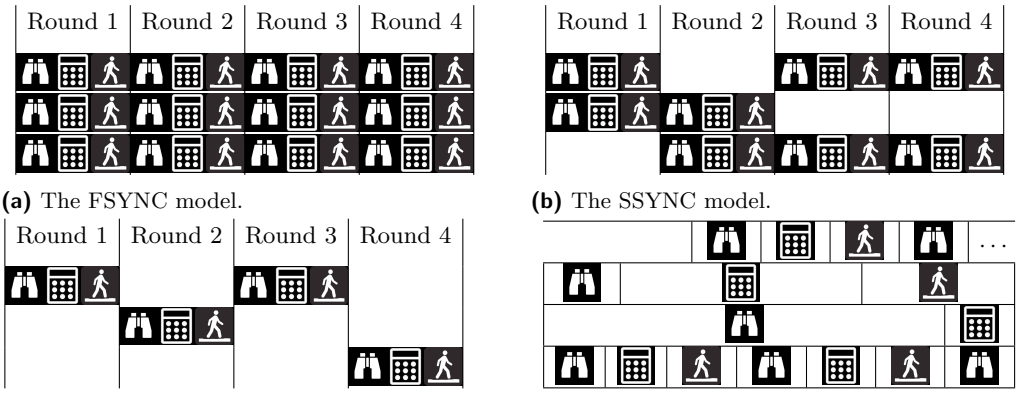
Notice that it is entirely possible to have common chirality without sharing a common direction. When having common direction, orientation on one axis, and chirality, robots are said to have *full compass*. Note that orientation on one axis and chirality amounts to having orientation on two axes.

4.3 Memory/Communication

The benefit of using oblivious robots is that they easily recover from crashes and memory corruption. Nevertheless, several extensions with memory have been proposed, ordered from strongest to weakest:

oblivious only volatile memory is available. Memory is reset at the beginning of each Look-Compute-Move cycle. Robots thus have no memory of past actions. Practically, robots only use their current snapshot in the compute phase.

finite memory robots may have persistent memory between Look-Compute-Move cycles. Several variants of this model called the *luminous* model [26, 66, 44, 27, 25] have been investigated (see below).



(a) The FSYNC model. **(b)** The SSYNC model.

(c) The centralized model. **(d)** The ASYNC model.

Figure 3 Synchronization hypotheses in models.

infinite memory robots may make use of an infinite amount of memory. It allows robots to remember a full observation, as the position of robots may have to be encoded as actual real numbers (in the ego-centred observation).

Since robots are assumed to be anonymous, there is no way of performing point-to-point communication with a particular neighbour. Therefore, communication is handled through broadcast, which is described as the robots having *lights* whose color may be adjusted during the compute phase. In addition to the number of available colours for a robot light (hence the amount of states of information transmitted), there are three kinds of lights:

internal lights are only visible by the emitting robot itself, thus actually represent finite memory (the robot communicates with itself);

external lights are only visible by other robots but not the emitting root, thus they represent communication without memory [60];

full lights combine internal and external lights: they are visible by all robots.

4.4 Synchronicity and fairness

The considered model is based on discrete logical time, that is, on a sequence of events, an event being any change in the state of any robot.

The possible interleavings of those events define the synchronicity level of an execution.

If the Look-Compute-Move cycles are considered atomic, that is no event can occur during a cycle, the model is said to be *semi-synchronous* (SSYNC): a subset of the robots enter (and finish) their cycle and each phase within it simultaneously while the others are idle, hence the notion of *round*. In the constrained version of SSYNC where no robot is idle, that is where *all* robots are activated simultaneously, the execution is said to be *fully-synchronous* (FSYNC). In the case where the cycles are not atomic and may overlap, the execution is said to be *asynchronous* (ASYNC) [40]. Clearly, ASYNC is the strongest model and FSYNC is the weakest.

A fourth synchronicity model exists: the *centralized* one, where only a single robot moves every round. It is a particular case of the SSYNC model (thus it is weaker) but it is incomparable to the FSYNC one.

Figure 3 illustrates these synchronicity hypotheses.

These synchronicity hypotheses between the Look-Compute-Move cycles of robots are of paramount importance for proofs. Many proofs made in weak synchronization models were claimed to hold also under stronger synchronization models but turned out to be incorrect. This is actually the main source of errors in the literature.

In the FSYNC, SSYNC, and centralized models, the actual duration of each phase does not matter since no observation occurs while a robot is moving, which justifies using discrete logical time. On the opposite, the ASYNC model represents the complete lack of synchronization between robots, and duration is important here, as a robot may observe others while they are moving.

Fairness

In all models except FSYNC where all robots are active at all times, the subset of active robots is chosen by the environment. In all generality, nothing prevents the environment, a.k.a the *demon*, from starving some or all robots. Obviously, most tasks are infeasible if some robots never get opportunities to act. Thus, there are fairness constraints on demons: a demon is said *fair* if every robot gets activated infinitely often. This is equivalent to saying that at any point of the execution, every robot is eventually activated.

Although fairness is usually enough for most protocols, it does not give any guarantee on the relative rates of robot activations: a robot may be activated arbitrarily more often than another. To remedy this situation, one can use the stronger fairness condition of *k-fairness*:[7] every robot is activated at least once for every k activations of any other robot.

4.5 Rigid/Flexible Movement

The atomicity of cycles does *not* imply that the computed destination is actually reached by a robot before the start of its new cycle: the robot may be interrupted during its move by the environment.

An execution where all robots always reach the destination returned by the protocol is said to be *rigid*. Conversely, if robots can start a new cycle before they completed their scheduled journey, the execution is *flexible*. In such a case, so as to avoid Zeno-like counter-examples, it is assumed that robots travel at least some minimal non-null distance δ *towards the expected destination* before being subject to restart. In particular, a journey shorter than δ is always completed. This minimal uninterruptable distance is unknown to robots;[8] they may however take into account that such a minimum exists.

4.6 Faults

In an adversarial environment, faults must be considered, either because malicious agents are present or because one of our own agent has been corrupted.

The most common fault hypotheses made in models are (from strongest to weakest):

byzantine faults some robots do not follow the protocol and are controlled by an adversary;

crash some robots may crash and stop acting forever;

no fault correct robots follow the protocol forever.

Notice that the faults described here are permanent, that is, they span the entire execution once they occur.

---

[7] Early literature introduced *k-bounded*: between two activations of a robot, there are at most k activations of any other robot. This is not equivalent to k-fairness since it is vacuously satisfied if a robot is never activated: there are no two activations we should count activations of other robots between. Furthermore, one can prove that k-bounded and fair is equivalent to k-fair, making k-fairness the useful notion.

[8] It would however make little sense to base an algorithm on such an absolute δ as it is possible that robots do not share frames of reference.

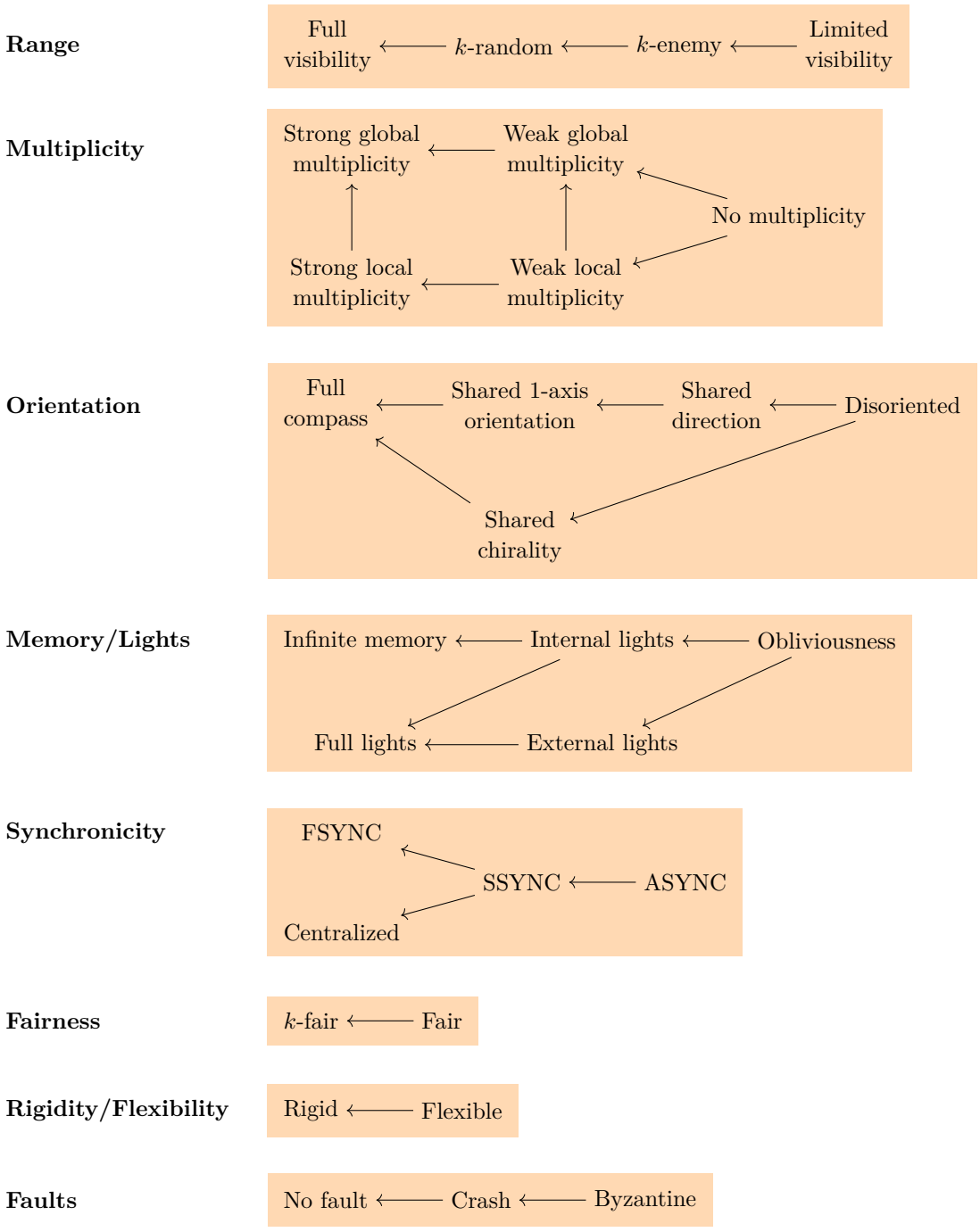


Figure 4 The model lattice for robot swarms, as the Cartesian product of smaller lattices.

5 The formalization of the Suzuki, Yamashita model

The formalization of a computation model in a proof assistant consists of a body of mathematical definitions linked together. The main statement defines the set of *correct computations* in this model. This definition must be inspected very carefully to ensure it complies exactly with what specialists have in mind. It is however not rare that on the occasion of a formalization one realizes that different specialists have slightly different models in mind. Formalization is thus the occasion of clarifying things, either by proving equivalence of models or by establishing more subtle correspondences between them.

The computational model introduced by Suzuki and Yamashita states basically that *robots move in space according to their observation of the environment*. In order to complete a formal description of this model, we hence have to provide a Coq encoding of the relevant space, and of course a way to characterize robots and their sensors (that is the way the environment is perceived).

Implementing robot capabilities and instantiating the universe the robots move in must be generic, abstract, and (relatively) user-friendly. This is crucial, as this is where *the developer* clarifies assumptions and removes ambiguities and *the expert* validates definitions. It is a *sine qua non* for a broad acceptance of a formal framework.

We must also formalize the Look-Compute-Move cycles and their possible interleavings: i.e. the core of the model and its synchronicity level. While the latter is still a parameter chosen by the user, the core itself is agnostic to assumptions about the environment. It is modelled in the framework by a function, `round`, that the developer never has to look at, except maybe for reassurance that it actually encodes Suzuki and Yamashita's robotic swarms model.

We hence shall describe first how `round` simulates the evolution of the system, with completely abstract parameters. Then we explore how various flavours of Suzuki and Yamashita's robots can or cannot solve fundamental tasks, and how to instantiate those variants within our formal framework.

5.1 Structure of the model, abstractions

So as to keep the core of the model as generic as possible, we provide the description of the environment as parameters. This way, those specifications and assumptions are kept abstract in the core, that is the actual description of how the system evolves.

The environment may be defined with several straightforward settings: the *space* where robots are moving, the level of *synchronicity*, the characteristics and capabilities of *robots and their sensors* (for example their accuracy or range), etc. Figure 6 and Section 5.4 show in details the structure of all parameters that must be instantiated.

Following Figure 6 we consider being given:

- a topology with its usual operations,
- the definition of a robot's *state* that includes its location (the position in space it is at), and a way of accessing it for all robots,
- a notion of *observation* that only considers what is allowed by the relevant variant, and finally
- an embedded algorithm : the protocol.

Usual operations regarding the space and its topology typically include a decidable equality on locations and change of frame operators.

A *state* describes, typically as a record, the internal state of robots, including in certain cases of synchronicity (namely ASYNC) their computed destination. Its access is ensured through the *configuration*: a function that takes a robot identifier as argument, and returns its state.

The configuration cannot generally be used as an observation of the robots as it may include private information about internal states, and thus may display *what should not be observed* by other robots. It is the case for example if local sensors have a limited range, or cannot get robots' ids, or even cannot detect multiplicity (i.e., the exact number of robots inhabiting a location in space), etc. Forbidden/private information is thus pruned from the configuration to get an *observation*. That observation is the one and only allowed input for a robot to compute its next destination; examples of its instantiation are given in section 5.3.2.

The protocol, that is the embedded algorithm that returns a path to a destination [9], based on an observation, is shared by all robots. It consists of the actual function mapping observations to (path-containing) states, and some properties (though irrelevant to `round`) ensuring, for example, that two equivalent observations produce equivalent paths and destinations, or on a graph that there is an actual arc towards the targeted node.

While the Pactole library can express all of the synchronicity hypotheses, namely ASYNC, SSYNC, centralized, and FSYNC, we shall focus in the following on the description of SSYNC executions (of which FSYNC executions are a particular case).

Modelling flexible executions simply amounts to allowing for a restart of any robot at any ratio of its trajectory, providing the effectively travelled distance is at least the minimal one δ. The new state is returned accordingly.

5.2 The function `round`

We describe the evolution of the system, following Suzuki and Yamashita's model, with a function: `round`.

5.2.1 Inputs

We want to design a function that, from a configuration, and given an embedded algorithm, returns the next configuration. Two obvious parameters for `round` are thus:

1. the configuration, and
2. the shared algorithm driving the robots: the protocol.

However, what happens in a step of execution depends also on some choice made by the demon, akin to Maxwell's: which robots are activated, what the new frames are, what distances are effectively travelled... We consider those choices as the results of a *demonic action*, which is given as an extra argument to `round`:

3. a demonic action.

We may find in a demonic action:

- the indication that a robot of a given id is activated, that is a function `activate` returning a Boolean when given an id as argument, in the special case of an FSYNC execution, its result is always `true`.
- a function for conversion of frames of references, say `change_frame`,[10]
- the actual function returning relevant choices on the entry of any robot's id. That function is a parameter of the model, it is hereafter referred to as `choose_update`.

[9] Observe that simply assuming robots move toward the destination along a straight line precludes the use of our framework for e.g. proving the correctness of existing algorithms that make use of non-linear paths, such as parametric paths used by Defago et al. [30]. Hence, to preserve generality of the framework, we assume protocols return a path.

[10] As it is constrained by the robot under consideration (recall that frames are self-centred) this function takes also a robot as argument.

Depending on assumptions, robots may also undergo Byzantine failures. As the movements of Byzantine robots is, by definition, not determined by the algorithm, demonic actions must in that case include:

- a function that chooses the next destination of each Byzantine robot, hereafter referred to as `relocate_byz`.
- Finally, one needs a set of properties ensuring that the choices are coherent, for example that robots do not go past their computed destination, do not follow non-existing paths, etc.

The infinite sequence of demonic actions characterizes the choices for the whole execution, and constitutes the *demon* of the execution.

Finally, we have to be able to express flexible executions. Recall that in those, robots may be interrupted/restarted *before* they reach the end of their planned journey, but *after* they travelled at least an absolute distance δ. While the value of δ is unknown to robots, it is used in enforcing that the execution fulfils the aforementioned constraints. It has thus to be provided to `round`:

4. a minimal travel length δ.

It is worth noticing that the formal development allows for comparison of demons; it provides in particular proofs that demons with rigid movements are equivalent to demons with flexible movements and movement ratio 1, in the sense that any execution for one can be also obtained for the other.

We shall now describe the evolution of the system, following Suzuki and Yamashita's model, by devising our function `round`.

5.2.2 Operation

For the sake of simplicity, we focus on SSYNC flexible executions, of which FSYNC flexible executions are the particular case where *all* robots are selected to be activated by every demonic action.

In the remainder of this section, we shall denote the four formal parameters for `round` as follows :

- δ obviously represents δ,
- `r` represents the protocol,
- `da` represents the used demonic action,
- `c` represents the configuration.

The result of `round` is a configuration, that is a *function*. The body of `round` δ `r da c` is hence a functional object taking an identifier, say `id`, as unique parameter, and returning its (new) state. We just have to describe this new state.

The first step is to figure out if the robot is activated, that is susceptible to undergo any change. This is a decision of the demon, and as so is stated in the demonic action. In an SSYNC context, `da.(activate) id` can either return `false`,[11] in which case the new state is the previous one: `c id`; or return `true` in which case choices and changes may apply for robot `id`: usually some change of frame function together with a travel ratio.

Note that this information is irrelevant if `id` is Byzantine. Should it be the case, its new state would be arbitrarily chosen by the demon, using `da.(relocate_byz)`.

Otherwise, if `id` refers to a correct robot, the new state depends on the protocol that requires an observation for `id`.

[11] Recall that this is never the case in an FSYNC execution.

1. The configuration is thus expressed from `id`'s point of view using its new frame of reference provided/chosen by the demonic action `da`. One obtains `id`'s *observation* by pruning the now translated configuration from illegitimate information.
2. **The protocol may now be applied on that observation.**
   The results contains in particular a path to a destination location, but is this *local* target reached before a new cycle starts? That depends of a choice of the demon, and is constrained by the ratio provided in `da.(choose_update) id`.
   Any destination location, in the local targeted state, closer than δ is reached, otherwise the location attained along the path is the one determined by the chosen ratio, the *chosen* target.
3. The demon-chosen target state is computed from the local target state by applying in particular the travel ratio.
4. As the distance between the current location of the robot and the chosen target is to be compared to the *absolute* δ, coordinates have to be translated back to the demon's frame of reference. The actual update with the new state can now be obtained from the result of this comparison: either the new location corresponds to the target locally computed or to the (demon-) chosen one.

5.3 Model specialization

As the core of the model is set once and for all, we may consider the many variants of Suzuki and Yamashita's robots. A slight modification of the robots, in their sensor capabilities for example, can dramatically change the feasibility of any given task. We shall review some of those, and describe how to instantiate the formal model accordingly.

5.3.1 Space

In this paper, we mainly consider the Euclidean plane $\mathbb{R}^2$ [24, 11]. Nevertheless, the formal model is not tied to this choice and we can consider any other space such as the real 3D space ($\mathbb{R}^3$), the real line ($\mathbb{R}$) [6, 23], discrete ones such as a ring ($\mathbb{Z}/n\mathbb{Z}$) or an arbitrary graph [12], possibly with robots moving continuously on edges [7], or even more exotic ones.

Providing a space in the formal framework amounts to define a type of *locations* where the robots are, and the necessary operations to compute: distances, the actual operations of the protocol, and *changes of frames of references*.

The computations by the protocol usually involve basic arithmetics; this is for example the case in Courtieu et al. [24] or Balabonski et al. [11] where all the necessary machinery to compute barycentres is provided with the instantiation of $\mathbb{R}^2$.

Conversions into different frames of references can be as simple as rotations and homothetic transformations in a Euclidean space. They are however subtler when the considered space is a graph, and may then involve permutations of the node names.

To allow for a comfortable use of graphs in Pactole, Balabonski et al. [12] define a (light-weight[12]) template to be instantiated as desired by the user. The relevant interface is designed to link up with the general signature for spaces; it provides also a specialised version for rings.

[12] This template is not intended to be as powerful as a specialised development on graphs, like for instance the LoCo library for local computation on graphs [19].

```
Definition round δ r da c :=
  (** for a given robot, we compute the new configuration *)
  fun id ⇒
    let state := c id in          (*  state: id's state as seen by demon *)
    if da.(activate) id                      (**  Is the robot activated? *)
    then match id with                       (**  Byzantine or correct?   *)
        | Byz b ⇒ da.(relocate_byz) b        (* Demon relocates Byzantine *)
        | Good g ⇒      (* Config. expressed in the frame of g: PHASE 1 *)
          let frame_conv := da.(change_frame) c id in
          let local_config := map_config frame_conv c in
          let obs := obs_from_config local_config id in
                                    (* APPLY r ON OBSERVATION: PHASE 2 *)
          let local_target_state := r obs in
                     (* Demon chooses a point on the path to the target *)
          let chosen_target_state := da.(choose_update)       (* PHASE 3 *)
                                     id
                                     local_config
                                     local_target_state in
          (update                           (* Actual update: PHASE 4 *)
              δ                             (* (including comparison) *)
              local_config
              id
              frame_conv ^-1
              loc_target_state
              chosen_target_state)
        end
    else state.                        (** Inactive robots stay unchanged *)

(** [execute r d config] returns an (infinite stream) execution from an
   initial global configuration config, a demon d and a protocol r
   running on each good robot. Each configuration being the result of
   round applied to the previous configuration (and the corresponding
   demonic_action). *)
Definition execute r : demon → configuration → execution :=
  cofix execute d c :=
    Stream.cons c (execute (Stream.tl d) (round δ r (Stream.hd d) c)).
```

Figure 5 The `round` function, core of the formal simulator, and the `execute` function that produces an infinite execution from it.

5.3.2 Sensors

One of the key concepts in Suzuki and Yamashita's model is the one of observation, that is the way to get some snapshot about the environment. The sensor capabilities can indeed change dramatically what is achievable in any given model. A very simple example of that is perception with a *limited* range: if robots are far enough apart to the point of not being aware of the others, there is no hope of cooperation of any kind!

Even seemingly minor differences may change the impossibility frontier: Gathering is impossible in general [23], but becomes possible with either a common compass [40] or detection of multiplicity [24] (that is, when robots can count the number of robots on a location instead of just detecting that the location is inhabited).[13]

Sensor capabilities are modelled through the way the configuration is perceived, that is transformed into an actual *observation*. The fact that states of robots may include some internal states, and thus may display information that should not be observed by other robots, prevents them to be used directly as an observation.

Depending on the variant one is interested in, assumptions may indeed require that local sensors cannot tell robots apart (anonymity), or detect whether they are correct or Byzantine, or are endorsed with multiplicity detection, etc.

These restrictions can be defined and encapsulated in the notion of *observation*, which characterizes what a robot's sensors can perceive of the global system.

To obtain the observation, that is the only input to the protocol, all forbidden information must be pruned from the configuration. To that goal, a function `obs_from_config` is devised to return an observation when given the actual configuration. This function takes also the internal state of the observer, as the actual perception may depend on characteristics of *its own sensors*.[14]

Sensor capabilities are thus characterized through:

- the datatype for observation, and
- the operation of `obs_from_config`.

5.3.2.1 Anonymity, multiplicity

Modelling capabilities through the type definition is straightforward as the type has to describe only the public information.

Let us say one wants to model anonymous robots, equipped with sensors that can "see" the whole universe, and detect the number of robots inhabiting any location (strong multiplicity). A convenient datatype for the observation in that case may be simply a *multiset* of the inhabited locations, the detection of multiplicity (number of robots in a place) being directly expressed by multiplicities of elements (number of times a location appears as the location of a robot).

This is not tied to the underlying space: in the case of a discrete graph where no node naming or origin is shared between robots, one would have a multiset of nodes, with one node marked as the location of the observing robot.

In the case the anonymous robots cannot distinguish between different inhabitants, detecting only that the location is inhabited, the observation datatype may then be just a *set* of inhabited locations. Again, this would work just as well on a graph rather than the Euclidean plane.

For non-anonymous robots, observations may be sets of pairs consisting of a robot identifier and the location where this robot is.

[13] Constraints on the starting configuration may be necessary, like the forbidding of a bivalent configuration.
[14] Of course, that does implies presence of (parts of) the internal state in the result of `obs_from_config`, which depends on what is allowed by the variant.

5.3.2.2 Accuracy, range limits

It is possible to represent bounded accuracy of sensors, or limited vision, in the sense that the whole space is not perceivable by a single robot. Those are variants that are taken care of in the function rather than in the datatype definition directly.

Bounded accuracy can be obtained by rounding values in the configuration to the adequate level before entering them in the observation. It is for example possible to introduce noise, or even map actual locations to an underlying discrete version of the space, where robots perform their computations.

Limited perception is obtained by deleting from a "total" observation the robots that are out of range for the observer under consideration. Note the use of the internal state, given as a parameter, in that case where location and range of detection are at play to determine the observable ball.

5.3.2.3 Colours, memory

In the *robots with lights* variant, robots are equipped with coloured lights that they can turn on and off. Colours can have a dramatic impact on possibility results, as they introduce a form of communication and, when self-perceived, a certain kind of memory.

Let us consider that the observation for a variant without colours is, say, a multiset of n-tuples (location, id, etc.). Adding colours to that variant basically consists in adding the colour information to that observation, that is considering a multiset of $n+1$-tuples including the colour. An additional tuple that represents the observing robot itself must be added if it can detect its own colour. It must be absent from the observation otherwise, as it introduces some memorized information about the robot's previous state.

Should robots be endowed with unbounded memory, being able to remember all previous observations for example, adding the list of previous observations as an internal state in the definition of the robots suffices. The relevant information from this list would then be included in the observation by `obs_from_config`.

5.4 Formal Parameters of the model

Figure 6 shows all the parameters needed to instantiate a particular model. The main parameters are:

- the location and more generally the **`State`** of the robot;
- the **`observation`** of the robot;
- the characterization of a **`demonic_action`**;
- the way the robots move (**`update_function`**), given the decisions of the robot and of the demon.

These parameters are themselves parameterized by types (`Names`, etc).

6 Examples

Examples provided in this section contain only a small subset of certified results obtained with the Pactole framework. For completeness, we summarize published results based on Pactole in Table 1.

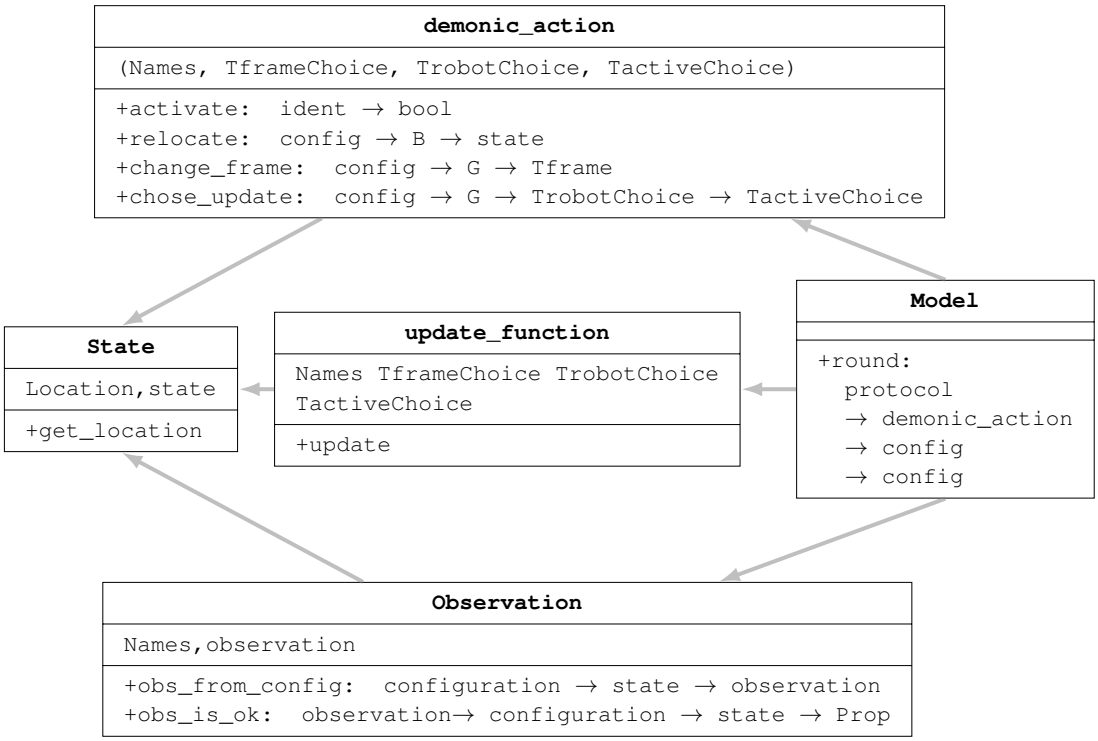


Figure 6 Parameters needed for a model.

6.1 Gathering

Robotic swarms are mostly a problem-driven domain, and as such focus on a few paradigmatic problems, some being fundamental, as for instance *Gathering*. This problem has been extensively studied, in particular by Principe [63], and in a formal setting by Balabonski et al. [10].

In its commonly shared definition, solving *Gathering* consists in having, within finite time, all correct (non-Byzantine) robots to stand on the same location, unknown beforehand, and to stay there indefinitely.

To describe Gathering formally, one has to define static (depending only on the configuration) and dynamic (depending on the demon) properties characterizing:

1. a configuration with all correct robots located at the same position, say p,
2. an execution with all correct robots staying at the same position indefinitely,
3. an execution consisting of a finite number of evolution steps (the actual gathering movements), followed by what is an execution fulfilling the description of item 2.

The first item is easily modelled by a definition `gathered_at` stating that, given a configuration c and a location p, for any robot identifier, if that robot is correct, then its location is p in c.

```
Definition gathered_at (p : location) (c : configuration) : Prop :=
                                  ∀ id, good id → get_location (c id) = p.
```

The second item characterizes an execution e with a location p: at each step in e, that is for each configuration c in e,[15] `gathered_at` p c holds. Let us call this property of p and e: `Gather` p e.

[15] Recall that executions are streams of configurations, so that a property P on the head of a stream e must be projected by `Stream.instant` P e.

Table 1 Certified results based on PACTOLE

Problem	Type of result	Setting	References	LoC
Framework	Core	All		1 000
	Spaces	$\mathbb{R}$, $\mathbb{R}^2$, rings, grids, graphs		5 176
	Observation	multiset, sets		626
Gathering	Impossibility	$\mathbb{R}$, $\mathbb{R}^2$, SSYNC	[23]	1 109
	Correctness	$\mathbb{R}$, $\mathbb{R}^2$, not bivalent, SSYNC	[24]	2 307
	Correctness	$\mathbb{R}^2$, FSYNC, flexible	[11]	609
Convergence	Impossibility	$\mathbb{R}$, 1/3 Byzantine	[6]	578
	Correctness	$\mathbb{R}$, FSYNC		170
Exploration with stop	Impossibility	Ring, FSYNC, $n \vert k$	[12]	474
	Necessary condition	Ring, FSYNC		203
Life-line	Correctness	$\mathbb{R}^2$, FSYNC	[8, 9]	1592
	Model equivalence	Graph, ASYNC	[7]	1187
	Model equivalence	ASYNC, flexible/rigid		275

```
Definition Gather (p : location) (e : execution) : Prop :=
                          Stream.forever (Stream.instant (gathered_at p)) e.
```

The third item states a location p *exists* such that the `Gather` p status is reached in finite time for an execution e. `WillGather` e is directly an inductive property over streams, that is holding from a point in the stream accessible/reachable in finite time.

```
Definition WillGather (e : execution) : Prop :=
                           Stream.eventually (fun ex ⇒ ∃ p, Gather p ex) e.
```

A protocol r achieves Gathering for a demon d if from *any* starting configuration c, all correct robots are eventually gathered forever in the execution obtained from r and d, that is :

```
Definition FullSolGathering r d : Prop :=
                                              ∀ c, WillGather (execute r d c).
```

Expressing now that a given protocol r is a solution to Gathering is simply stating that *for every* demon d, `FullSolGathering` r d holds.[16]

Conversely, expressing that Gathering is unsolvable (under certain assumptions) is simply stating that for any protocol r, it does *not* hold that r solves Gathering *for every* demon.

6.1.1 A model where gathering is proven impossible

In this section we give an example of a model where gathering is proved impossible. This is a well known fact [65, 63], of which a generalized version has been formally proved [23].

Instantiating the model

Let us have an arbitrary even number of robots, say n, of which none is Byzantine, moving on the Euclidean plane (note the use of **`Variable`** and **`Hypothesis`** for parameters left abstract).

[16] This can be constrained to demons *fulfilling the assumptions of the considered variant*: synchronicity, fairness, etc.).

```
Variable n : nat.
Hypothesis even_nG : Nat.Even n.
Definition MyRobots : Names := Robots n 0.
Definition Loc : Location := make_Location R2.
```

Movements are rigid (that is: robots always reach their destination before the next round). No demon interference is applied on robot's choice, and all operations deal with locations only:

```
Definition rchoice : Trobotchoice := location.
Definition state : State location := OnlyLocation.
Definition dchoice : Tactivechoice := unit.
Instance UpdFun : update_function := RigidUpdate.
```

Robots have multiplicity detection but cannot distinguish one robot from another. To model this limitation in sensing capabilities, we define the observation of a robot as a multiset of locations: the multiplicity of a location p gives the number of robots present at p, but robots are not identifiable. We give here the instantiation of the `Observation` model parameter. It is a record containing:

(1) the logical definition of what the observation of a configuration must be: `obs_is_ok`, which is what is used in future proofs;

(2) the definition of the function `obs_from_config` that computes the observation, from the configuration, and that is used in `round` (see figure 5)

(3) the proof of correctness of `obs_from_config` with reference to the characteristic property `obs_is_ok`.

```
Definition multiset_observation : Observation := {|
    observation := multiset location;
    (* Characteristic property of the observation of a config. *)
    obs_is_ok obs (c:config) st :=
         ∀ loc, multiplicity obs loc
                = countA_occ loc (map c MyRobots);
    (* Function computing the observation from config *)
    obs_from_config c st := make_multiset (map c MyRobots) ;
    (* Proof that obs_from_config satisfies characteristic property *)
    obs_from_config_spec c st: obs_is_ok (obs_from_config c st) c st :=
      ...
|}.
```

Note that in this example, the assumptions on the sensors are completely global, unrelated to the internal state of the observer. Thus, `obs_from_config` does not use this state, even though it receives it as a parameter.

Stating the result

We call a position `invalid` if all robots are on two towers of the same height, that is: evenly distributed on exactly two distinct locations. Such a position is known as *bivalent*.

```
Definition invalid (config : configuration) :=
  ∃ pt1 pt2 : location, pt1 =/= pt2
    ∧ multiplicity pt1 (obs_from_config config origin) = nG / 2
    ∧ multiplicity pt2 (obs_from_config config origin) = nG / 2.
```

The final lemma states that for any protocol, if one starts in an `invalid` configuration then there exists a demon that makes the protocol fail, i.e. that prevents the system to reach a gathered position.

```
Theorem noGathering : ∀ r : protocol,  ∀ c : configuration,
 invalid c →
    ∃ d, SSYNC d  ∧  Fair d  ∧  ¬ WillGather (execute r d c).
```

A remark on the dual property

Adding as a condition on the initial configuration that it is *not* invalid (not bivalent) suffices to get a universal algorithm solving Gathering. Developed in PACTOLE, the solution given by Courtieu et al. [24] uses the *exact same specifications* of the model and the environment, thus eliminating any risk of shift, and closing the problem under those assumptions: Fair-SSYNC Gathering of oblivious rigid anonymous robots in $\mathbb{R}^2$ is impossible, unless the initial configuration is *not* bivalent, in which case a protocol is proven correct.

6.1.2 A model where gathering is proven possible

It is however possible to achieve gathering when different capabilities for sensors are assumed. For the sake of the example, we assume now, as in the works of Balabonski et al. [11], that sensors cannot detect multiplicity. It should be stressed here that the formal definition of the problem is *strictly* the same as before, preventing any shift or bias in its definition.

Instantiating the model

Let us have an arbitrary number n (more than 1) of robots (0 Byzantine) moving on the Euclidean plane.

```
Variable n : nat.
Hypothesis H_nG : n >= 2.
Definition MyRobots : Names := Robots n 0.
Definition Loc : Location := make_Location R2.
```

Movements are flexible (i.e. robots may not reach their destination before the next round). Let us have an arbitrary δ representing the minimal distance (in the global reference) a robot moves between two rounds, unless its destination is attained. The only choice made by the demon after a robot's decision is the *ratio* of the path toward its target the robot actually reaches.

```
Variable δ : R.
Definition ratio : Tactivechoice := {x : R | 0 ≤ x ≤ 1}.
Definition rchoice : Trobotchoice := path location.
Definition FlexChoice : update_choice := Flexible.OnlyFlexible.
Instance UpdFun : update_function rchoice location ratio := FlexibleUpdate δ.
```

As robots cannot detect multiplicity, observations may be reduced to a *set* of (inhabited) positions, as remarked in section 5.3.2.1.

```
Definition set_observation : Observation := {|
  observation := set location;
  obs_is_ok s c pt := ∀ l, In l s ↔ In l (map c MyRobots);
  obs_from_config c pt := make_set (map c MyRobots);
  obs_from_config_spec c st : obs_is_ok (obs_from_config c st) c st :=
      ...
 |}.
```

Stating the result

The main theorem of Balabonski et al. [11] states that the protocol `ffgatherR2`, given below, solves the gathering for any fully synchronous demon and any starting configuration.

```
Definition ffgatherR2_pgm (s : observation) : path :=
  paths_in_R2 (isobarycenter (elements s)).

Theorem FSGathering_in_R2 :
  ∀ d, δ > 0 → FSYNC d → FullSolGathering ffgatherR2 d.
```

And with strong detection of multiplicity?

The framework is generic enough to provide also a formal certification for a result by Cohen and Peleg [20, 21] when robot sensors are this time endowed with detection of multiplicity. The only noticeable difference in the two approaches is the definition of the observation: a multiset for Cohen and Peleg's, and a set for Balabonski et al.'s. The proof argument is similar, and only a few technical steps to take into account the new type for observation are required [11].

6.2 Exploration

An interesting benchmarking problem when dealing with robots on graphs is the one of *exploration*, in particular *with stop*. Exploration with stop (also known as Terminating Exploration) requires to ensure that:

1. all nodes are visited by a robot at some point during the execution, and
2. all robots eventually stop moving once all nodes have been visited.

There are thus 2 properties to formalize: for the space to be explored, and for the system to stop evolving.

For a node, say v, being *eventually* visited (inhabited) by a robot is simply an inductive (that is finitely reachable) property on the execution, say e: at some accessible point along e, the configuration returns a state displaying v as the current location.[17] This is a basic property about streams.

```
Definition Will_be_visited v e :=
  Stream.eventually (Stream.instant (is_visited v)) e.
```

The second property, the halting, is built in three steps:

- firstly, one defines what is it for an execution to have two consecutive identical configurations, namely that it *stalls*. It is simply a call to the equivalence relation on configurations

  ```
  Definition Stall e :=
    Config.eq (Stream.hd e) (Stream.hd (Stream.tl e)).
  ```

- secondly, the stall has to hold indefinitely

  ```
  Definition Stopped e := Stream.forever Stall e.
  ```

- and finally the point where the execution is `Stopped` is reached within a finite number of steps; this is property `Will_stop`.

[17] Recall that, an execution being a stream of configurations, a property P on the head of a stream e is projected by `Stream.instant` P e.

```
Definition Will_stop e := Stream.eventually Stopped e.
```

Let `execute` r d c be the execution obtained by running a protocol r with a demon d from an initial configuration c. Protocol r achieves Exploration with stop for a demon d if from *every* initial configuration c, `Will_be_visited` v (`execute` r d c) holds for *every* node v (the exploration part), AND this execution stops, that is `Will_stop` (`execute` r d c) holds.

```
Definition FullSolExplorationStop r d :=
  ∀ c, (∀ v, Will_be_visited v (execute r d c))
       ∧ Will_stop (execute r d c).
```

Similarly to what has been done for Gathering, expressing that a given protocol r is a solution to Exploration with stop is simply stating that for *every* demon d, `FullSolExplorationStop` r d holds.[18]

Conversely, expressing that the problem is unsolvable (under certain assumptions) is simply stating that for any protocol r, it does *not* hold that r is a solution.

6.2.1 A model where Exploration with stop is proven impossible

We want to prove that Exploration with stop on a ring is not possible if the number of robots divides the size of the ring. We proceed along the lines of Balabonski et al. [12].

Instantiating the model

A ring is a special case of finite graph, already defined by the function `Ring` taking as input the size, which must be an integer greater than 1.

```
Variable ring_size : nat.
Hypothesis ring_size_spec : 1 < ring_size.
Instance Ring_space : FiniteGraph := Ring ring_size ring_size_spec.
```

Let us have an arbitrary number kG of robots (of which none is Byzantine) moving on our ring. We assume that kG divides the size `ring_size` of the ring and remove two corner cases: $kG = 1$ and $kG =$ `ring_size`.

```
Variable kG : nat.
Instance Robots : Names := Robots kG 0.
Hypothesis kdn : ring_size mod kG = 0.
Hypothesis k_bounds : 1 < kG < ring_size.
```

As in the first example, robots contain no more information than their locations, robots' observations are the multiset of locations, and the demon does not interfere with the robot's choice, that is, movements are rigid.

```
Definition state : State location := OnlyLocation.
Instance RobotObs : Observation := multiset_observation.
Definition dchoice : Tactivechoice := unit.
```

[18] That is where constraints on the demon should appear, of the form *for all demons verifying such given property*, `FullSolExplorationStop` r d holds, as detailed in Section 6.2.1.

The local frame is the one described in the introductory examples of Section 3.3, which amounts to a translation along the ring (relative locations) and potentially a symmetry (for chirality change).

The difference with these earlier examples is that robots do not choose a new node of the ring to move to, they only pick a direction. The update function is still rigid, and it simply applies the function `move_along` that returns the node reached by following the chosen direction from the robot's current location.

```
Inductive direction := Forward | Backward | SelfLoop.
Definition rchoice : Trobotchoice := direction.
Instance UpdFun : update_function := move_along.
```

Stating the result

The final theorem states that for any protocol r, there is a FSYNC demon d against which r does not solve exploration with stop, that is, there exists a configuration c such that the execution following r and d starting from c either does not terminate or does not explore the ring.

```
Theorem no_exploration :
  ∀ r : protocol, ∃ d : demon, FSYNC d ∧ ¬ FullSolExplorationStop r d.
```

7 Related work

Numerous formalizations and verification tools have been designed and used to account for correctness in distributed computing. The formal treatment of mobile robotic swarms nevertheless requires specific tooling (w.r.t. "classical" distributed computing), as there is no direct transformation of "classical" shared memory or message passing models into models suitable to study mobile robotic entities. A naive transformation from the shared memory model would be to consider that the "values" shared by the robots are their positions, that observing other robots's positions is similar to reading a shared variable, and that moving to a new position is similar to writing a new value in a shared variable. However, some aspects prevent formal solutions for the shared memory model to also apply to the mobile robots model.

First, the notion of "local observation" is quite specific to mobile robots: a robot has only access to a degraded view of its neighbourhood (according to its visual sensors), and the view is obtained in its local (i.e. ego centred) coordinate system. So, the same robot may appear at two different positions at the exact same time in the execution (by two different robots that have different coordinate systems), and, in the same execution, the same robot may appear or not appear at a given observing robot (depending on the location and sensing abilities of the observing robot). Second, in the more interesting ASYNC model, a robot can observe other robots while they move, resulting in getting any intermediate position the observed robot may reach during its movement. This is a strictly weaker setting than the classical "atomic" and "regular" shared memory registers popularized by Lamport [52, 53] (where reads may only return the "before-write" or the "after-write" value), and yet strictly stronger than the classical "safe" register [52, 53] (where *any* value in the domain could be returned by a read) if we make the reasonable assumption that robots cannot cover an infinite surface in a single move. Last: the position of a robot may belong to a continuous (dense) set, while a shared register is typically a discrete value.

This last concern makes model checking mobile robot algorithms quite difficult. If one wants a direct translation of the Suzuki and Yamashita model, one has to consider discrete (i.e. graphs) locations [14, 33, 35]. While this approach is suitable for checking problem instances, it cannot scale as when the number of locations becomes a parameter, interesting properties become

undecidable [64], even if the graph representing locations is as simple as a ring. Hence, the current hope on the model-checking side is to consider more abstract models, in the spirit of the recent approach by Defago et al. [28]. However, more abstract models yield two significant issues. First, models are likely to target a single problem rather than being generic, with no hint earned as how to handle other problems in the same setting. Second, one has to prove that properties obtained mechanically by model-checking for the abstract model echo in interesting properties for the original model (in the approach of Defago et al. [28], this part is handwritten, hindering a fully mechanized checkability of the approach).

From the methodology perspective, Pactole is close to Ivy [61, 58] and LoE [57]: they are based on a master model designed in a proof assistant and a methodology is designed to prove specific protocols in these model.

8 Conclusion

The mathematical description of mobile robotic swarms by Suzuki and Yamashita proved to be a fertile ground[19] of research, with many model variants developed throughout the years, and many applicative domains envisioned by research teams all over the world (see the book edited by Flocchini et al. [37] for a recent survey).

When robotic swarm protocols are developed for critical applications, with lives at stake, reasoning about the model requires the foundations of a formal framework and methodology aimed at mobile robot protocol designers, to enable the certification of tentative robot protocols for any property related to their spatio-temporal behaviour that is useful in practice, or to demonstrate the impossibility of such designs.

A first step in that direction has been proposed with the Pactole framework, which allows so far working formally with:

- *Euclidean spaces* with geometrical constructs (barycentres, smallest enclosing circles, etc., together with their relevant properties) ; *Graphs*, either discrete or with continuous movement along the edges,
- Rigid/flexible moves, SSYNC/FSYNC/ASYNC, various properties on demons (flavours of fairness), equivalences between demons, and means for the theoretical study of their lattice,
- Common notions of observation depending on the capabilities of robots,

including many cases studies, e.g. exploration, gathering, convergence...

Pactole is publicly available to the community at `https://pactole.liris.cnrs.fr`.

Three main axes of development are worth considering.

The first one addresses the issue of probabilistic behaviours. Probabilistic arguments are indeed commonly used in recent results regarding mobile robotic swarms, *e.g.* to break symmetries in configurations [18, 17, 68]. Probabilistic behaviours may also occur outside protocols, in the environment. Including probabilistic behaviours on the autonomous robot side (that is, robots are able to take actions based on some probabilistic source) and on the environment side (that is, the scheduling decisions are based on some probability distribution) in the Pactole framework is thus an important task.

The second one is to ensure that certified developments remain accessible to a non-specialist of formal proof, and to ease the proof burden as much as possible. This amounts to building manageable libraries with clear independent modules, definitions that can be shared and proof steps that can be reused. Fundamental results must be gathered into a formal library, with each relevant notion properly specified in the formal model, and each reusable property receiving a corresponding formal proof.

[19] Masafumi Yamashita was awarded the 2016 Prize for Innovation in Distributed Computing for this seminal work.

Automation is expected wherever possible in this building process. Certifying properties of an algorithm in a proof assistant amounts to developing mechanical techniques matching the proof structures brought to the fore. Developing automation in that context is two-fold. On the one hand it involves high-level tactics to relieve the user from tedious and repetitive proof steps. On the other hand it must help in exhibiting certain properties and obtaining as a result a formal proof of it. In doing this, inputs from model checking approaches are precious due to their capacity to exhibit counter-examples, thus helping the developer, and to discharge automatically some base cases prior to induction steps.

Finally, it is fundamental to provide the prototype of an environment for proof and development of trustworthy distributed protocols for mobile robots, by linking together the results obtained with reference to the formal model, libraries, proof automation and management techniques, etc. Such an environment should allow the user to specify the algorithm, and to certify its properties, using generated verification conditions and proof constructs with the help of decision procedures. To reach this goal, one key challenge is to devise an annotation language rich enough to specify properties in the scope of our studies, but still convenient to use by designers of robot protocols. To ensure protocol validation, it must also integrate proof mechanisms allowing both assisted or automated certification, thus defining a complete certification chain that should be easy enough to use, even for a non specialist.

References

1 Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating and optimizing stabilizing dining philosophers. In *11th European Dependable Computing Conference, EDCC 2015, Paris, France, September 7-11, 2015*, pages 233–244. IEEE, 2015.

2 José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full Proof Cryptography: Verifiable Compilation of Efficient Zero-Knowledge Protocols. In *ACM Conference on Computer and Communications Security*, pages 488–500, 2012.

3 Karine Altisen, Pierre Corbineau, and Stéphane Devismes. A framework for certified self-stabilization. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 36–51. Springer-Verlag, 2016.

4 Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

5 The Coq Proof Assistant. `https://coq.inria.fr/`.

6 Cédric Auger, Zohir Bouzid, Pierre Courtieu, Sébastien Tixeuil, and Xavier Urbain. Certified Impossibility Results for Byzantine-Tolerant Mobile Robots. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium (SSS 2013)*, volume 8255 of *Lecture Notes in Computer Science*, pages 178–186, Osaka, Japan, November 2013. Springer-Verlag.

7 Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Continuous vs. discrete asynchronous moves: A certified approach for mobile robots. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems - 7th International Conference, (NETYS 2019), Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*, pages 93–109, Marrakech, Morocco, June 2019. Springer-Verlag.

8 Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Brief Announcement: Computer Aided Formal Design of Swarm Robotics Algorithms. In Colette Johnen and Stefan Schmid, editors, *Stabilization, Safety, and Security of Distributed Systems - 23th International Symposium, (SSS 2021)*, Virtual conference, November 2021.

9 Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Computer aided formal design of swarm robotics algorithms. *CoRR*, abs/2101.06966, 2021.

10 Thibaut Balabonski, Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Certified gathering of oblivious mobile robots: Survey of recent results and open problems. In Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti, editors, *Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, (FMICS-AVoCS 2017)*, volume 10471 of *Lecture Notes in Computer Science*, pages 165–181, Turin, Italy, September 2017. Springer-Verlag.

11 Thibaut Balabonski, Amélie Delga, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Synchronous gathering without multiplicity detection: A

certified algorithm. *Theory of Computing Systems*, pages 200–218, 2019. https://doi.org/10.1007/s00224-017-9828-z.

12 Thibaut Balabonski, Robin Pelle, Lionel Rieg, and Sébastien Tixeuil. A foundational framework for certified impossibility results with mobile robots on graphs. In Paolo Bellavista and Vijay K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 5:1–5:10. ACM, 2018.

13 Andrej Bauer. How to review formalized mathematics. http://math.andrej.com/2013/08/19/how-to-review-formalized-mathematics/, August 2013.

14 Béatrice Bérard, Pascal Lafourcade, Laure Millet, Maria Potop-Butucaru, Yann Thierry-Mieg, and Sébastien Tixeuil. Formal verification of mobile robot protocols. *Distributed Computing*, 29(6):459–487, 2016.

15 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

16 François Bonnet, Xavier Défago, Franck Petit, Maria Potop-Butucaru, and Sébastien Tixeuil. Discovering and assessing fine-grained metrics in robot networks protocols. In *33rd IEEE International Symposium on Reliable Distributed Systems Workshops, SRDS Workshops 2014, Nara, Japan, October 6-9, 2014*, pages 50–59. IEEE, 2014.

17 Quentin Bramas and Sébastien Tixeuil. Brief Announcement: Probabilistic Asynchronous Arbitrary Pattern Formation. In George Giakkoupis, editor, *35th ACM Symposium on Principles of Distributed Computing (PODC 2016)*, pages 443–445, Chicago, IL, USA, July 2016. ACM.

18 Quentin Bramas and Sébastien Tixeuil. The Random Bit Complexity of Mobile Robots Scattering. *International Journal of Foundations of Computer Science*, 28(2):111–134, 2017.

19 Pierre Castéran, Vincent Filou, and Mohamed Mosbah. Certifying Distributed Algorithms by Embedding Local Computation Systems in the Coq Proof Assistant. In Adel Bouhoula and Tetsuo Ida, editors, *Symbolic Computation in Software Science (SCSS'09)*, 2009.

20 Reuven Cohen and David Peleg. Robot Convergence via Center-of-Gravity Algorithms. In Rastislav Kralovic and Ondrej Sýkora, editors, *Structural Information and Communication Complexity - 11th International Colloquium (SIROCCO 2004)*, volume 3104 of *Lecture Notes in Computer Science*, pages 79–88, Smolenice Castle, Slowakia, June 2004. Springer-Verlag.

21 Reuven Cohen and David Peleg. Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems. *SIAM Journal of Computing*, 34(6):1516–1528, 2005.

22 Thierry Coquand and Christine Paulin-Mohring. Inductively Defined Types. In Per Martin-Löf and Grigori Mints, editors, *International Conference on Computer Logic (Colog'88)*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

23 Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Impossibility of Gathering, a Certification. *Information Processing Letters*, 115:447–452, 2015.

24 Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Certified universal gathering algorithm in $\mathbb{R}^2$ for oblivious mobile robots. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, (DISC 2016)*, volume 9888 of *Lecture Notes in Computer Science*, pages 187–200, Paris, France, September 2016. Springer-Verlag.

25 Shantanu Das, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Forming sequences of patterns with luminous robots. *IEEE Access*, 8:90577–90597, 2020.

26 Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. Autonomous mobile robots with lights. *Theor. Comput. Sci.*, 609:171–184, 2016.

27 Xavier Défago, Adam Heriban, Sébastien Tixeuil, and Koichi Wada. Brief announcement: Model checking rendezvous algorithms for robots with lights in euclidean space. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 41:1–41:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

28 Xavier Défago, Adam Heriban, Sébastien Tixeuil, and Koichi Wada. Using model checking to formally verify rendezvous algorithms for robots with lights in euclidean space. In *International Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020*, pages 113–122. IEEE, 2020.

29 Xavier Défago, Maria Potop-Butucaru, and Sébastien Tixeuil. Fault-tolerant mobile robots. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 234–251. Springer, 2019.

30 Xavier Défago and Samia Souissi. Non-uniform circle formation algorithm for oblivious mobile robots with convergence toward uniformity. *Theor. Comput. Sci.*, 396(1-3):97–112, 2008.

31 Carole Delporte-Gallet, Hugues Fauconnier, Yan Jurski, François Laroussinie, and Arnaud Sangnier. Towards synthesis of distributed algorithms with SMT solvers. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2019.

32 Ha Thi Thu Doan, François Bonnet, and Kazuhiro Ogata. Model checking of a mobile robots perpetual exploration algorithm. In Shaoying Liu, Zhenhua Duan, Cong Tian, and Fumiko Nagoya, editors, *Structured Object-Oriented Formal Language and Method - 6th International Workshop, SOFL+MSVL 2016, Tokyo, Japan, November 15, 2016, Revised Selected Papers*, volume 10189 of *Lecture Notes in Computer Science*, pages 201–219, 2016.

33 Ha Thi Thu Doan, François Bonnet, and Kazuhiro Ogata. Model checking of robot gathering. In James Aspnes and Pascal Felber, editors, *Principles of Distributed Systems - 21th International Conference (OPODIS 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), Lisbon, Portugal, December 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

34 Ha Thi Thu Doan, Kazuhiro Ogata, and François Bonnet. Specifying a distributed snapshot algorithm as a meta-program and model checking it at meta-level. In Kisung Lee and Ling Liu, editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 1586–1596. IEEE Computer Society, 2017.

35 Ha Thi Thu Doan, Adrián Riesco, and Kazuhiro Ogata. An environment for specifying and model checking mobile ring robot algorithms. In Mohsen Ghaffari, Mikhail Nesterenko, Sébastien Tixeuil, Sara Tucci, and Yukiko Yamauchi, editors, *Stabilization, Safety, and Security of Distributed Systems - 21st International Symposium, SSS 2019, Pisa, Italy, October 22-25, 2019, Proceedings*, volume 11914 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2019.

36 Fathiyeh Faghih, Borzoo Bonakdarpour, Sébastien Tixeuil, and Sandeep S. Kulkarni. Automated synthesis of distributed self-stabilizing protocols. *Logical Methods in Computer Science*, 14(1), 2018.

37 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities*, volume 11340 of *Lecture Notes in Computer Science, Theoretical Computer Science and General Issues*. Springer Nature, 2019.

38 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Gathering of asynchronous oblivious robots with limited visibility. In Afonso Ferreira and Horst Reichel, editors, *STACS 2001, 18th Annual Symposium on Theoretical Aspects of Computer Science, Dresden, Germany, February 15-17, 2001, Proceedings*, volume 2010 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2001.

39 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Pattern formation by anonymous robots without chirality. In Francesc Comellas, Josep Fàbrega, and Pierre Fraigniaud, editors, *SIROCCO 8, Proceedings of the 8th International Colloquium on Structural Information and Communication Complexity, Vall de Núria, Girona-Barcelona, Catalonia, Spain, 27-29 June, 2001*, volume 8 of *Proceedings in Informatics*, pages 147–162. Carleton Scientific, 2001.

40 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theoretical Computer Science*, 407(1-3):412–447, 2008.

41 Georges Gonthier. Formal Proof—The Four-Color Theorem. *Notices of the AMS*, 55(11):1382–1393, December 2008.

42 Georges Gonthier. Engineering Mathematics: the Odd Order Theorem Proof. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 1–2. ACM, 2013.

43 Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 372–382. ACM, 2008.

44 Adam Heriban, Xavier Défago, and Sébastien Tixeuil. Optimally gathering two robots. In Paolo Bellavista and Vijay K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 3:1–3:10. ACM, 2018.

45 Adam Heriban and Sébastien Tixeuil. Mobile robots with uncertain visibility sensors. In Keren Censor-Hillel and Michele Flammini, editors, *Structural Information and Communication Complexity - 26th International Colloquium, (SIROCCO 2019)*, volume 11639 of *Lecture Notes in Computer Science*, pages 349–352, L'Aquila, Italy, July 2019. Springer-Verlag.

46 Taisuke Izumi, Tomoko Izumi, Sayaka Kamei, and Fukuhito Ooshita. Randomized gathering of mobile robots with local-multiplicity detection. In Rachid Guerraoui and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings*, volume 5873 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 2009.

47 Tomoko Izumi, Taisuke Izumi, Sayaka Kamei, and Fukuhito Ooshita. Mobile robots gathering algorithm with local weak multiplicity in rings. In Boaz Patt-Shamir and Tínaz Ekim, editors, *Structural Information and Communication Complexity, 17th International Colloquium, SIROCCO 2010, Sirince, Turkey, June 7-11, 2010. Proceedings*, volume 6058 of *Lecture Notes in Computer Science*, pages 101–113. Springer, 2010.

48 Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

49 Igor Konnov, Helmut Veith, and Josef Widder. Who is afraid of model checking distributed algorithms? Unpublished to: CAV Workshop $(EC)^2$, July 2012.

50 Philipp Küfner, Uwe Nestmann, and Christina Rickmann. Formal Verification of Distributed Algorithms - From Pseudo Code to Checked Proofs. In Jos C. M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *IFIP TCS*, volume 7604 of *Lecture Notes in Computer Science*, pages 209–224, Amsterdam, The Netherlands, September 2012. Springer-Verlag.

51 Sandeep S. Kulkarni, Borzoo Bonakdarpour, and Ali Ebnenasir. Mechanical verification of automatic synthesis of fault-tolerant programs. In Sandro Etalle, editor, *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers*, volume 3573 of *Lecture*

Notes in Computer Science, pages 36–52. Springer, 2004.

52 Leslie Lamport. On interprocess communication. part I: basic formalism. *Distributed Comput.*, 1(2):77–85, 1986.

53 Leslie Lamport. On interprocess communication. part II: algorithms. *Distributed Comput.*, 1(2):86–101, 1986.

54 Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, 1994.

55 Xavier Leroy. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

56 Assia Mahboubi. Checking machine-checked proofs. `https://project.inria.fr/coqexchange/checking-machine-checked-proofs/`, July 2017.

57 Vincent Rahli Mark Bickford, Robert L. Constable. Logic of events, a framework to reason about distributed systems. In *2012 Languages for Distributed Algorithms Workshop*, Philadelphia, PA, 2012.

58 Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 190–202. Springer, 2020.

59 Laure Millet, Maria Potop-Butucaru, Nathalie Sznajder, and Sébastien Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In Pascal Felber and Vijay K. Garg, editors, *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, (SSS 2014)*, volume 8756 of *Lecture Notes in Computer Science*, pages 237–251, Paderborn, Germany, September 2014. Springer-Verlag.

60 Takashi Okumura, Koichi Wada, and Xavier Défago. Optimal rendezvous l-algorithms for asynchronous mobile robots with external-lights. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, volume 125 of *LIPIcs*, pages 24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

61 Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630. ACM, 2016.

62 Maria Potop-Butucaru, Nathalie Sznajder, Sébastien Tixeuil, and Xavier Urbain. Formal methods for mobile robots. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 278–313. Springer, 2019.

63 Giuseppe Prencipe. Impossibility of gathering by a set of autonomous mobile robots. *Theoretical Computer Science*, 384(2-3):222–231, 2007.

64 Arnaud Sangnier, Nathalie Sznajder, Maria Potop-Butucaru, and Sébastien Tixeuil. Parameterized verification of algorithms for oblivious robots on a ring. *Formal Methods Syst. Des.*, 56(1):55–89, 2020.

65 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM Journal of Computing*, 28(4):1347–1363, 1999.

66 Giovanni Viglietta. Rendezvous of two robots with visible bits. In Paola Flocchini, Jie Gao, Evangelos Kranakis, and Friedhelm Meyer auf der Heide, editors, *Algorithms for Sensor Systems - 9th International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, ALGOSENSORS 2013, Sophia Antipolis, France, September 5-6, 2013, Revised Selected Papers*, volume 8243 of *Lecture Notes in Computer Science*, pages 291–306. Springer-Verlag, 2013.

67 Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1278–1294, 2015.

68 Yukiko Yamauchi and Masafumi Yamashita. Randomized Pattern Formation Algorithm for Asynchronous Oblivious Mobile Robots. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, (DISC 2014)*, volume 8784 of *Lecture Notes in Computer Science*, pages 137–151, Austin, USA, October 2014. Springer-Verlag.

Higher-Dimensional Timed and Hybrid Automata

Uli Fahrenberg
EPITA Research and Development Laboratory (LRDE), France

Abstract

We introduce a new formalism of higher-dimensional timed automata, based on Pratt and van Glabbeek's higher-dimensional automata and Alur and Dill's timed automata. We prove that their reachability is PSPACE-complete and can be decided using zone-based algorithms. We also extend the setting to higher-dimensional hybrid automata.

The interest of our formalism is in modeling systems which exhibit both real-time behavior and concurrency. Other existing formalisms for real-time modeling identify concurrency and interleaving, which, as we shall argue, is problematic.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Timed and hybrid models

Keywords and Phrases concurrency, real time, higher-dimensional automaton, timed automaton

Digital Object Identifier 10.4230/LITES.8.2.3

Acknowledgements The author acknowledges the support of the *Chaire ISC : Engineering Complex Systems* and École polytechnique where most of this work was carried out. He is most grateful to Kim G. Larsen and Eric Goubault for numerous interesting discussions on the subject of this paper.

Received 2020-08-04 **Accepted** 2022-01-28 **Published** 2022-12-07

Editor Alessandro Abate, Uli Fahrenberg, and Martin Fränzle

Special Issue Special Issue on Distributed Hybrid Systems

1 Introduction

In approaches to non-interleaving concurrency, more than one event may happen at the same time. There is a multitude of formalisms for modeling and analyzing such concurrent systems, e.g., Petri nets [47], event structures [46], configuration structures [58, 57], asynchronous transition systems [8, 50], or more recent variations such as dynamic event structures [6] and Unravel nets [16]. They all share the convention of differentiating between concurrent and interleaving executions; using CCS notation [44], $a|b \neq a.b + b.a$.

For modeling and analyzing embedded or cyber-physical systems, formalisms which use real time are available. These include timed automata [5], time Petri nets [43], timed-arc Petri nets [38], or various classes of hybrid automata [3]. Common for them all is that they identify concurrent and interleaving executions; here, $a|b = a.b + b.a$.

We are interested in formalisms for real-time non-interleaving concurrency. Hence we would like to differentiate between concurrent and interleaving executions *and* be able to model and analyze real-time properties. Few such formalisms seem to be available in the literature. The situation is perhaps best epitomized by the fact that there is a natural non-interleaving semantics for Petri nets [34] which is also used in practice [22, 23], but almost all work on real-time extensions of Petri nets [43, 38, 51, 53], including the popular tool TAPAAL,[1] use an interleaving semantics. (A notable exception here are the time Petri nets of [43] which do have a non-interleaving real-time semantics [18, 17, 7, 32] which has also been used for networks of timed automata [15].)

Also Uppaal,[2] the successful tool for modeling and analyzing networks of timed automata, uses an interleaving semantics for such networks. This leads to great trouble with state-space explosion (see also Sect. 7 of this paper) which can be avoided with a non-interleaving semantics such as we

[1] `http://www.tapaal.net/`
[2] `http://www.uppaal.org/`

propose here. Intuitively, interleaving composition of networks adds $n!$ different interleavings for every concurrent composition of n independent events, whereas in the non-interleaving semantics, only one (n-dimensional) object is added to the system.

We introduce higher-dimensional timed automata (HDTA), a formalism based on the (non-interleaving) higher-dimensional automata of Pratt and van Glabbeek [48, 54] (see also [56]) and the timed automata of Alur and Dill [5, 4]. We show that HDTA can model interesting phenomena which cannot be captured by neither of the formalisms on which they are based, but that their analysis remains just as accessible as the one of timed automata. That is, reachability for HDTA is PSPACE-complete and can be decided using zone-based algorithms.

In the above-mentioned interleaving real-time formalisms, continuous flows and discrete actions are orthogonal in the sense that executions alternate between real-time delays and discrete actions which are immediate, i.e., take no time. (In the hybrid setting, these are usually called flows and mode changes, respectively.) Already Sifakis and Yovine [52] notice that this significantly simplifies the semantics of such systems and hints that this is a main reason for the success of these formalisms (see the more recent [53] for a similar statement).

In the (untimed) non-interleaving setting, on the other hand, events have a (logical; unspecified) duration. This can be seen, for example, in the ST-traces of [55] where actions have a start (a^+) and a termination (a^-) and are (implicitly) running between their start and termination, or in the representation of concurrent systems as Chu spaces over $\mathbf{3} = \{0, \frac{1}{2}, 1\}$, where 0 is interpreted as "before", $\frac{1}{2}$ as "during", and 1 as "after", see [49]. Intuitively, only if events have duration can one make statements such as "while a is running, b starts, and then while b is running, a terminates".

In our non-interleaving real-time setting, we hence abandon the assumption that actions are immediate. Instead, we take the view that actions start and then run during some *specific* time before terminating. While this runs counter to the standard assumption in most of real-time and hybrid modeling, a similar view can be found, for example, in Cardelli's [14].

Given that we abandon the orthogonality between continuous flows and discrete actions, we find it remarkable to see that the standard techniques used for timed automata transfer to our non-interleaving setting. Equally remarkable is, perhaps, the fact that even though *"[t]he timed-automata model is at the very border of decidability, in the sense that even small additions to the formalism [...] will soon lead to the undecidability of reachability questions"* [1], our extension to higher dimensions and non-interleaving concurrency is completely free of such trouble.

We also show that our HDTA model naturally extends to a formalism of higher-dimensional *hybrid* automata (HDHA), which can be used to model cyber-physical systems which exhibit concurrency. We introduce tensor products both for HDTA and HDHA which can be used for a concurrent composition of systems which avoids state-space explosion.

The contributions of this paper are, thus, (1) the introduction of a new formalism of HDTA, a natural extension of higher-dimensional automata and timed automata, in Sect. 3; (2) the proof that reachability for HDTA is PSPACE-complete and decidable using zone-based algorithms, in Sects. 5 and 6; (3) the introduction of a tensor product for HDTA which can be used for parallel composition, in Sect. 7; and (4) the extension of the definition to higher-dimensional hybrid automata together with a non-trivial example of two independently bouncing balls, in Sect. 8.

This paper is based on the conference contribution [27], which has been presented at the 6th IFAC Conference on Analysis and Design of Hybrid Systems in Oxford, UK. Compared to this previous paper, we have included proofs of all statements, improved the presentation and examples, and added a precise definition of tensor product of higher-dimensional hybrid automata.

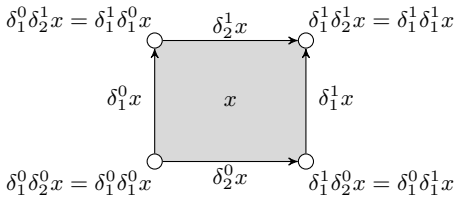


Figure 1 A 2-cube x with its four faces $\delta_1^0 x$, $\delta_1^1 x$, $\delta_2^0 x$, $\delta_2^1 x$ and four corners.

2 Preliminaries

We recall a few facts about higher-dimensional automata and timed automata.

2.1 Higher-Dimensional Automata

Higher-dimensional automata are a generalization of finite automata which permit the specification of independence of actions through higher-dimensional elements. That is, they consist of states and transitions, but also squares which signify that two events are independent, cubes which denote independence of three events, etc. To introduce them properly, we need to start with precubical sets.

A *precubical set* is a graded set $X = \bigcup_{n\in\mathbb{N}} X_n$, with $X_n \cap X_m = \emptyset$ for $n \neq m$, together with mappings $\delta_{k,n}^\nu : X_n \to X_{n-1}$, $k = 1,\dots,n$, $\nu = 0,1$, satisfying the *precubical identity*

$$\delta_{k,n-1}^\nu \delta_{\ell,n}^\mu = \delta_{\ell-1,n-1}^\mu \delta_{k,n}^\nu \qquad (k < \ell)\,.$$

Elements of X_n are called *n-cubes*, and for $x \in X_n$, $n = \dim x$ is its *dimension*. The mappings $\delta_{k,n}^\nu$ are called *face maps*, and we will usually omit the extra subscript n and write δ_k^ν instead of $\delta_{k,n}^\nu$. Intuitively, each n-cube $x \in X_n$ has n *lower faces* $\delta_1^0 x,\dots,\delta_n^0 x$ and n *upper faces* $\delta_1^1 x,\dots,\delta_n^1 x$, and the precubical identity expresses the fact that non-parallel $(n-1)$-faces of an n-cube meet in common $(n-2)$-faces; see Figure 1 for an example.

A precubical set X is *finite* if X is finite as a set. This means that X_n is finite for each $n \in \mathbb{N}$ and that X is *finite-dimensional*: there exists $N \in \mathbb{N}$ such that $X_n = \emptyset$ for all $n \geq N$.

Let Σ be a finite set of *actions* and recall that a *multiset* over Σ is a mapping $\Sigma \to \mathbb{N}$; we denote the set of such by $\mathbb{N}^\Sigma$. The *cardinality* of $S \in \mathbb{N}^\Sigma$ is $|S| = \sum_{a\in\Sigma} S(a)$.

▶ **Definition 1.** A *higher-dimensional automaton* (HDA) is a structure (X, x^0, X^f, λ), where X is a finite precubical set with initial state $x^0 \in X_0$ and accepting states $X^f \subseteq X_0$, and $\lambda : X \to \mathbb{N}^\Sigma$ is a labeling function such that for every $x \in X$,

- $|\lambda(x)| = \dim x$,
- $\lambda(\delta_k^0 x) = \lambda(\delta_k^1 x)$ for all $k \leq \dim x$, and
- $\lambda(x) \setminus \lambda(\delta_k^0 x)$ is a singleton for all $k \leq \dim x$.

The conditions on the labeling ensure that the label of an n-cube is an extension, by one event, of the label of any of its faces. The computational intuition is that when passing from a lower face $\delta_k^0 x$ of $x \in X$ to x itself, the (unique) event in $\lambda(x) \setminus \lambda(\delta_k^0 x)$ is started, and when passing from x to an upper face $\delta_\ell^1 x$, the event in $\lambda(x) \setminus \lambda(\delta_\ell^1 x)$ is terminated.

HDA can indeed model higher-order concurrency of actions. As an example, the hollow cube on the left of Figure 2, consisting of all six faces of a cube but not of its interior, models the situation where the actions a, b and c are mutually independent, but cannot be executed all three concurrently. The full cube on the right of Figure 2, on the other hand, has a, b and c independent

as a set. The left HDA might model a system of three users connected to two printers, so that every two of the users can print concurrently but not all three, whereas the right HDA models a system of three users connected to (at least) three printers.

▶ Remark. Instead of using multisets as we do here, labeling of precubical sets is commonly introduced by defining a precubical set $!\Sigma$ induced by Σ and then letting the labeling be a precubical morphism, see for example [24, 36]. This has the advantage that HDA can be posed as a slice category, but we will not need this here.

There is a rich literature on the geometric and topological analysis of HDA, starting with their *geometric realization* as directed topological spaces. The interested reader is referred to [37, 30, 31, 28, 24, 25].

2.2 Timed Automata

Timed automata extend finite automata with clock variables and invariants which permit the modeling of real-time properties. Let C be a finite set of *clocks*. $\Phi(C)$ denotes the set of *clock constraints* defined as

$$\Phi(C) \ni \phi_1, \phi_2 ::= c \bowtie k \mid \phi_1 \wedge \phi_2 \qquad (c \in C, k \in \mathbb{N}, \bowtie \in \{<, \leq, \geq, >\})\,.$$

Hence a clock constraint is a conjunction of comparisons of clocks to integers.

A *clock valuation* is a mapping $v : C \to \mathbb{R}_{\geq 0}$, where $\mathbb{R}_{\geq 0}$ denotes the set of non-negative real numbers. The *initial* clock valuation is $v^0 : C \to \mathbb{R}_{\geq 0}$ given by $v^0(c) = 0$ for all $c \in C$. For $v \in \mathbb{R}^C_{\geq 0}$, $d \in \mathbb{R}_{\geq 0}$, and $C' \subseteq C$, the clock valuations $v + d$ and $v[C' \leftarrow 0]$ are defined by

$$(v+d)(c) = v(c) + d\,; \quad v[C' \leftarrow 0](c) = \begin{cases} 0 & \text{if } c \in C'\,, \\ v(c) & \text{if } c \notin C'\,. \end{cases}$$

For $v \in \mathbb{R}^C_{\geq 0}$ and $\phi \in \Phi(C)$, we write $v \models \phi$ if v satisfies ϕ and $[\![\phi]\!] = \{v : C \to \mathbb{R}_{\geq 0} \mid v \models \phi\}$.

▶ **Definition 2.** A *timed automaton* is a structure (Q, q^0, Q^f, I, E), where Q is a finite set of locations with initial location $q^0 \in Q$ and accepting locations $Q^f \subseteq Q$, $I : Q \to \Phi(C)$ assigns invariants to states, and $E \subseteq Q \times \Phi(C) \times \Sigma \times 2^C \times Q$ is a set of guarded transitions.

The *semantics* of a timed automaton $A = (Q, q^0, Q^f, I, E)$ is a (usually uncountably infinite) transition system $[\![A]\!] = (S, s^0, S^f, \rightsquigarrow)$, with $\rightsquigarrow \subseteq S \times S$, given as follows:

$$\begin{aligned} S &= \{(q, v) \in Q \times \mathbb{R}^C_{\geq 0} \mid v \models I(q)\} \\ s^0 &= (q^0, v^0) \quad S^f = S \cap Q^f \times \mathbb{R}^C_{\geq 0} \\ \rightsquigarrow &= \{((q, v), (q, v + d)) \mid \forall 0 \leq d' \leq d : v + d' \models I(q)\} \\ &\quad \cup \{((q, v), (q', v')) \mid \exists (q, \phi, a, C', q') \in E : v \models \phi, v' = v[C' \leftarrow 0]\} \end{aligned}$$

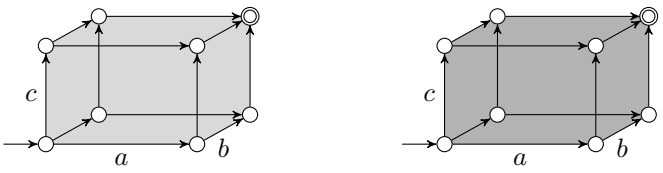


Figure 2 Two example HDA. Left, the hollow cube; right, the full cube.

Note that we are ignoring the labels of transitions here, as we will be concerned with reachability for now. As usual, we say that A is *reachable* iff there exists a finite path $s^0 \rightsquigarrow \cdots \rightsquigarrow s$ in $[\![A]\!]$ for which $s \in S^f$.

The definition of $\rightsquigarrow$ ensures that actions are immediate: whenever $(q, \phi, a, C', q') \in E$, then A passes from (q, v) to (q', v') without any delay. Time progresses only during delays $(q, v) \rightsquigarrow (q, v + d)$ in locations.

▶ Remark. Timed automata have a long and successful history in the modeling and verification of real-time computing systems. Several tools exist which are routinely applied in industry, such as Uppaal [42, 9], RED [59] or Kronos [13]. The interested reader is referred to [12, 41, 1].

3 Higher-Dimensional Timed Automata

Unlike timed automata, higher-dimensional automata make no formal distinction between states (0-cubes), transitions (1-cubes), and higher-dimensional cubes. We transfer this intuition to higher-dimensional timed automata, so that each n-cube has an invariant which specifies when it is enabled and an exit condition giving the clocks to be reset when leaving:

▶ **Definition 3.** A *higher-dimensional timed automaton* (HDTA) is a structure $(L, l^0, L^f, \lambda, \mathsf{inv}, \mathsf{exit})$, where (L, l^0, L^f, λ) is a finite higher-dimensional automaton and $\mathsf{inv} : L \to \Phi(C)$, $\mathsf{exit} : L \to 2^C$ assign *invariant* and *exit* conditions to each n-cube.

The *semantics* of a HDTA $A = (L, l^0, L^f, \lambda, \mathsf{inv}, \mathsf{exit})$ is a (usually uncountably infinite) transition system $[\![A]\!] = (S, s^0, S^f, \rightsquigarrow)$, with $\rightsquigarrow \subseteq S \times S$, given as follows:

$$
\begin{aligned}
S &= \{(l, v) \in L \times \mathbb{R}^C_{\geq 0} \mid v \models \mathsf{inv}(l)\} \\
s^0 &= (l^0, v^0) \quad S^f = S \cap L^f \times \mathbb{R}^C_{\geq 0} \\
\rightsquigarrow &= \{((l, v), (l, v + d)) \mid \forall 0 \leq d' \leq d : v + d' \models \mathsf{inv}(l)\} \\
&\quad \cup \{((\delta^0_k l, v), (l, v')) \mid k \in \{1, \dots, \dim l\}, v' = v[\mathsf{exit}(\delta^0_k l) \leftarrow 0] \models \mathsf{inv}(l)\} \\
&\quad \cup \{((l, v), (\delta^1_k l, v')) \mid k \in \{1, \dots, \dim l\}, v' = v[\mathsf{exit}(l) \leftarrow 0] \models \mathsf{inv}(\delta^1_k l)\}
\end{aligned}
$$

We omit labels from the semantics, as we will be concerned only with *reachability* for now: Given a HDTA A, does there exist a finite path $s^0 \rightsquigarrow \cdots \rightsquigarrow s$ in $[\![A]\!]$ such that $s \in S^f$?

Note that in the definition of $\rightsquigarrow$ above, we allow time to evolve in any n-cube in L. Hence transitions (i.e., 1-cubes) are not immediate. The second line in the definition of $\rightsquigarrow$ defines the passing from an $(n-1)$-cube to an n-cube, i.e., the start of a new concurrent event, and the third line describes what happens when finishing a concurrent event. Exit conditions specify which clocks to reset when leaving a cube.

▶ **Example 4.** We give a few examples of two-dimensional timed automata. The first, in Figure 3, models two actions, a and b, which can be performed concurrently. It consists of four states (0-cubes) l^0, l_1, l_2, l^f, four transitions (1-cubes) e_1 through e_4, and one ab-labeled square (2-cube) u. This HDTA models that performing a takes between two and four time units, whereas performing b takes between one and three time units. To this end, we use two clocks x and y which are reset when the respective actions are started and then keep track of how long they are running.

The clocks are reset by the condition $\mathsf{exit}(l^0) = \{x, y\}$, and the invariants $x \leq 4$ at the a-labeled transitions e_1, e_4 and at the square u ensure that a takes at most four time units. The invariants $x \geq 2$ at l_1, e_3 and l^f take care that a cannot finish before two time units have passed. Note that x is also reset when exiting e_2 and l_2, ensuring that regardless when a is started, whether before b, while b is running, or after b is terminated, it must take between two and four time units.

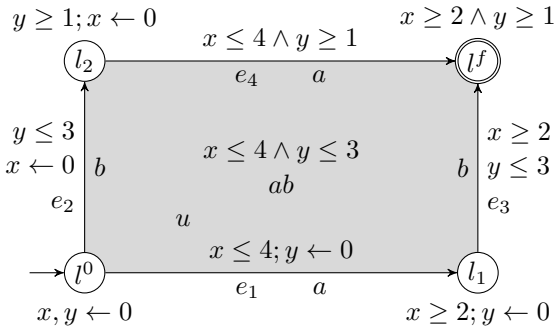


Figure 3 The HDTA of Example 4.

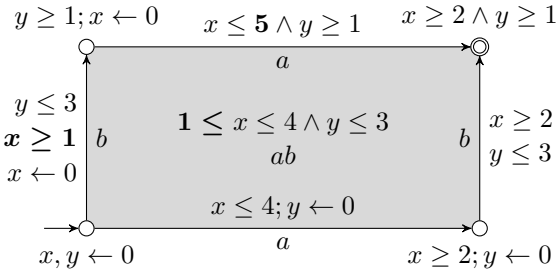


Figure 4 The HDTA of Example 5.

▶ **Example 5.** In the HDTA shown in Figure 4 (where we have omitted the names of states etc. for clarity and show changes to Figure 3 in bold), invariants have been modified so that b can only start after a has been running for one time unit, and if b finishes before a, then a may run one time unit longer. Hence an invariant $x \geq 1$ is added to the two b-labeled transitions and to the ab-square (at the right-most b-transition $x \geq 1$ is already implied), and the condition on x at the top a-transition is changed to $x \leq 5$. Note that the left edge e_2 is now permanently disabled: before entering it, x is reset to zero, but its edge invariant is $x \geq 1$. This is as expected, as b should not be able to start before a.

▶ **Example 6.** The HDTA in Figure 5, where we again show changes to Figure 4 in bold, models the additional constraint that b also *finish* one time unit before a. To this end, an extra clock z is introduced which is reset when b terminates and must be at least 1 when a is terminating. After these changes, the right b-labeled edge is deadlocked: when leaving it, z is reset to zero but needs to be at least one when entering the accepting state. Again, this is expected, as a should not terminate before b.

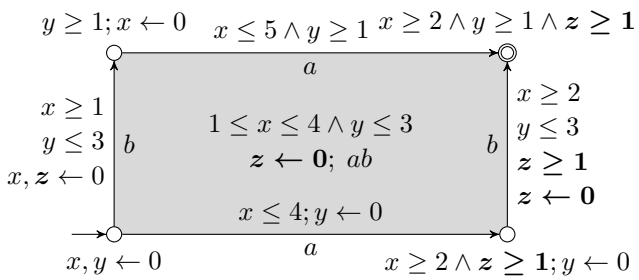


Figure 5 The HDTA of Example 6.

As both vertical edges are now permanently disabled, the accepting state can only be reached through the square. This shows that reachability for HDTA cannot be reduced to one-dimensional reachability along transitions and relates them to the partial HDA of [29, 20].

▶ Remark. In our model of HDTA, the exit conditions of a cube are the same regardless of *how* the cube is exited. One could imagine an extension of the model where exit conditions may be different depending on whether an action is terminated or a new one is started, so that $\mathsf{exit} : L \times L \rightharpoonup 2^C$ would be a partial function from *pairs* of cubes, one a face of the other. Our results still hold for this extension of the model, but we have not seen use for it in examples, and for the sake of simplicity, we do not pursue it here.

4 One-Dimensional Timed Automata

We work out the relation between one-dimensional HDTA (i.e., 1DTA) and standard timed automata. Note that this is not trivial, as in timed automata, clocks can only be reset at transitions, and, semantically, transitions take no time. In contrast, in our 1DTA, resets can occur in states and transitions may take time.

▶ **Proposition 7.** *There is a linear-time algorithm which, given any timed automaton A, constructs a 1DTA A', with one extra clock, so that A is reachable iff A' is.*

Proof. Let $A = (Q, q^0, Q^f, I, E)$ be a timed automaton. It is clear that $L = Q \cup E$ forms a one-dimensional precubical set, with $L_0 = Q$, $L_1 = E$, $\delta_1^0(q, \phi, a, C', q') = q$, and $\delta_1^1(q, \phi, a, C', q') = q'$. Let $l^0 = q^0$ and $L^f = Q^f$. In order to make transitions immediate, we introduce a fresh clock $c \notin C$. For $q \in Q$, let $\lambda(q) = \emptyset$, $\mathsf{inv}(q) = I(q)$, and $\mathsf{exit}(q) = \{c\}$. For $e = (q, \phi, a, C', q') \in E$, put $\lambda(e) = \{a\}$, $\mathsf{inv}(e) = \phi \wedge (c \leq 0)$, and $\mathsf{exit}(e) = C'$. We have defined a 1DTA $A' = (L, l^0, L^f, \lambda, \mathsf{inv}, \mathsf{exit})$ (over clocks $C \cup \{c\}$). As c is reset whenever exiting a state, and every transition has $c \leq 0$ as part of its invariant, it is clear that transitions in A' take no time, and the claim follows. ◀

▶ **Proposition 8.** *There is a linear-time algorithm which, given any 1DTA A, constructs a timed automaton A' over the same clocks such that A is reachable iff A' is.*

Proof. Let $A = (L, l^0, L^f, \lambda, \mathsf{inv}, \mathsf{exit})$ be a 1DTA, we construct a timed automaton $A' = (Q, q^0, Q^f, I, E)$. Because transitions in A may take time, we cannot simply let $Q = L_0$, but need to add extra states corresponding to the edges in L_1. Let, thus, $Q = L$, $I = \mathsf{inv}$, and

$$E = \{(\delta_1^0 x, \mathbf{tt}, \tau, \mathsf{exit}(\delta_1^0 x), x), (x, \mathbf{tt}, \lambda(x), \mathsf{exit}(x), \delta_1^1 x) \mid x \in L_1\},$$

where $\tau \notin \Sigma$ is a fresh (silent) action; see Figure 6 for an illustration.

Semantically, this converts the 0-cube $\delta_1^0 x$ to the location $\delta_1^0 x$; but its exit condition C_1 is moved to the new outgoing τ-edge. Thus, the location $\delta_1^0 x$ is enabled precisely when the 0-cube $\delta_1^0 x$ is enabled, and when it is exited by immediate execution of the τ-edge, its exit condition is applied. Similar considerations apply to the conversion of the 1-cube x to the location x; thus, A' is reachable iff A is. ◀

Note that even though silent transitions in timed automata are a delicate matter [11], the fact that we add them in the last proof is unimportant as we are only concerned with reachability. PSPACE-completeness of reachability for timed automata [2] and Proposition 7 now imply the following:

▶ **Corollary 9.** *Reachability for HDTA is PSPACE-hard.*

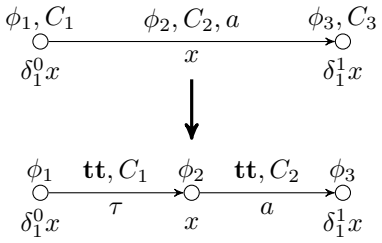


Figure 6 Conversion of 1DTA edge to timed automaton.

5 Reachability for HDTA is PSPACE-Complete

We now turn to extend the notion of *regions* to HDTA, in order to show that reachability for HDTA is decidable in PSPACE.

▶ **Definition 10.** Let $(L, l^0, L^f, \lambda, \mathsf{inv}, \mathsf{exit})$ be a HDTA and $R \subseteq L \times \mathbb{R}^C_{\geq 0} \times L \times \mathbb{R}^C_{\geq 0}$. Then R is an *untimed bisimulation* if $((l^0, v^0), (l^0, v^0)) \in R$ and, for all $((l_1, v_1), (l_2, v_2)) \in R$,

- $l_1 \in L^f$ iff $l_2 \in L^f$;
- for all $(l_1, v_1) \rightsquigarrow (l_1', v_1')$, also $(l_2, v_2) \rightsquigarrow (l_2', v_2')$ for some $((l_1', v_1'), (l_2', v_2')) \in R$;
- for all $(l_2, v_2) \rightsquigarrow (l_2', v_2')$, also $(l_1, v_1) \rightsquigarrow (l_1', v_1')$ for some $((l_1', v_1'), (l_2', v_2')) \in R$.

For a HDTA A, let M_A denote the maximal constant appearing in any $\mathsf{inv}(l)$ for $l \in L$, and let $\cong_{M_A}$ denote the standard region equivalence [5] on $\mathbb{R}^C_{\geq 0}$ defined as follows. For $d \in \mathbb{R}_{\geq 0}$, write $\lfloor d \rfloor$ and $\langle d \rangle$ for the integral, respectively fractional, parts of d, and then for $v, v' : C \to \mathbb{R}_{\geq 0}$, $v \cong_{M_A} v'$ iff

- $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ or $v(x), v'(x) > M_A$, for all $x \in C$,
- $\langle v(x) \rangle = 0$ iff $\langle v'(x) \rangle = 0$, for all $x \in C$ with $v(x) \leq M_A$, and
- $\langle v(x) \rangle \leq \langle v(y) \rangle$ iff $\langle v'(x) \rangle \leq \langle v'(y) \rangle$ for all $x, y \in C$ with $v(x) \leq M_A$ and $v(y) \leq M_A$.

Extend $\cong_{M_A}$ to $[\![A]\!]$ by defining $(l, v) \cong_{M_A} (l', v')$ iff $l = l'$ and $v \cong_{M_A} v'$.

▶ **Lemma 11.** *$\cong_{M_A}$ is an untimed bisimulation.*

Proof. This follows from standard properties of region equivalence [5]. First, $(l^0, v^0) \cong_{M_A} (l^0, v^0)$, and for all $(l_1, v_1) \cong_{M_A} (l_2, v_2)$, $l_1 \in L^f \Leftrightarrow l_2 \in L^f$ because $l_1 = l_2$.

Let $(l, v_1) \cong_{M_A} (l, v_2)$ and $(l, v_1) \rightsquigarrow (l', v_1')$; we show that there is v_2' such that $(l, v_2) \rightsquigarrow (l', v_2')$ and $(l', v_1') \cong_{M_A} (l', v_2')$. The symmetric case is analogous.

Assume $v_1' = v_1 + d$ and $l' = l$, then we have d' such that $v_2' := v_2 + d' \cong_{M_A} v_1 + d$, but then also $(l, v_2) \rightsquigarrow (l', v_2')$ and $(l, v_1') \cong_{M_A} (l, v_2')$.

Assume $l = \delta_k^0 l'$ for some k and $v_1' = v_1[\mathsf{exit}(l) \leftarrow 0] \models \mathsf{inv}(l')$. Let $v_2' = v_2[\mathsf{exit}(l) \leftarrow 0]$, then $v_2' \cong_{M_A} v_1'$ and $v_2' \models \mathsf{inv}(l')$, hence $(l, v_2) \rightsquigarrow (l', v_2')$ and $(l, v_1') \cong_{M_A} (l, v_2')$.

Assume $l' = \delta_k^1 l$ for some k and $v_1' = v_1[\mathsf{exit}(l) \leftarrow 0] \models \mathsf{inv}(l')$. Let $v_2' = v_2[\mathsf{exit}(l) \leftarrow 0]$, then $v_2' \cong_{M_A} v_1'$ and $v_2' \models \mathsf{inv}(l')$, hence $(l, v_2) \rightsquigarrow (l', v_2')$ and $(l, v_1') \cong_{M_A} (l, v_2')$. ◀

For any HDTA A, the *quotient* of $[\![A]\!] = (S, s^0, S^f, \rightsquigarrow)$ under an untimed bisimulation R is defined, as usual, as $[\![A]\!]/R = (S/R, [s^0]_R, S^f/R, \tilde{\rightsquigarrow})$, where S/R is the set of equivalence classes, $[s^0]_R$ is the equivalence class which contains s^0, and $\tilde{\rightsquigarrow} \subseteq S/R \times S/R$ is defined by $\tilde{s} \tilde{\rightsquigarrow} \tilde{s}'$ iff $\exists s \in \tilde{s}, s' \in \tilde{s}' : s \rightsquigarrow s'$.

▶ **Lemma 12.** *Let A be a HDTA and R an untimed bisimulation on A. Then A is reachable iff $[\![A]\!]/R$ is.*

Proof. By definition, an accepting location is reachable in A iff an accepting state is reachable in $[\![A]\!]$. On $[\![A]\!]$, R is a standard bisimulation, hence the claim follows. ◀

▶ **Lemma 13.** *For any HDTA A, the quotient $[\![A]\!]/{\cong_{M_A}}$ is finite.*

Proof. This follows immediately from the standard fact that the set of clock regions, i.e., $\mathbb{R}^C_{\geq 0}/{\cong_{M_A}}$, is finite [5]. ◀

The size of $[\![A]\!]/{\cong_{M_A}}$ is exponential in the size of A, but reachability in $[\![A]\!]/{\cong_{M_A}}$ can be decided in PSPACE, see [5]. Together with Corollary 9, we conclude:

▶ **Theorem 14.** *Reachability for HDTA is PSPACE-complete.*

6 Zone-Based Reachability

We show that the standard zone-based algorithm for checking reachability in timed automata also applies in our HDTA setting. This is important, as zone-based reachability checking is at the basis of the success of tools such as Uppaal, see [42].

Recall that the set $\Phi^+(C)$ of *extended clock constraints* over C is defined by the grammar

$$\Phi^+(C) \ni \phi_1, \phi_2 ::= c \bowtie k \mid c_1 - c_2 \bowtie k \mid \phi_1 \wedge \phi_2 \qquad (c, c_1, c_2 \in C, k \in \mathbb{Z}, \bowtie \in \{<, \leq, \geq, >\}),$$

and that a *zone* over C is a subset $Z \subseteq \mathbb{R}^C_{\geq 0}$ which can be represented by an extended clock constraint ϕ, i.e., such that $Z = [\![\phi]\!]$. Let $\mathcal{Z}(C)$ denote the set of zones over C.

For a zone $Z \in \mathcal{Z}(C)$ and $C' \subseteq C$, the *delay* and *reset* of Z are given by $Z^\uparrow = \{v + d \mid v \in Z\}$ and $Z[C' \leftarrow 0] = \{v[C' \leftarrow 0] \mid v \in Z\}$; these are again zones, and their representation by an extended clock constraint can be efficiently computed [10]. Also zone inclusion $Z' \subseteq Z$ can be efficiently decided.

The *zone graph* of a HDTA $A = (L, l^0, L^f, \lambda, \mathsf{inv}, \mathsf{exit})$ is a (usually infinite) transition system $Z(A) = (S, s^0, S^f, \rightsquigarrow)$, with $\rightsquigarrow \subseteq S \times S$, given as follows:

$$\begin{aligned}
S &= \{(l, Z) \in L \times \mathcal{Z}(C) \mid Z \subseteq [\![\mathsf{inv}(l)]\!]\} \\
s^0 &= (l^0, [\![v^0]\!]^\uparrow \cap [\![\mathsf{inv}(l^0)]\!]) \qquad S^f = S \cap L^f \times \mathcal{Z}(C) \\
\rightsquigarrow &= \{((\delta^0_k l, Z), (l, Z')) \mid k \in \{1, \dots, \dim l\}, Z' = Z[\mathsf{exit}(\delta^0_k l) \leftarrow 0]^\uparrow \cap [\![\mathsf{inv}(l)]\!]\} \\
&\quad \cup \{((l, Z), (\delta^1_k l, Z')) \mid k \in \{1, \dots, \dim l\}, Z' = Z[\mathsf{exit}(l) \leftarrow 0]^\uparrow \cap [\![\mathsf{inv}(\delta^1_k l)]\!]\}
\end{aligned}$$

As an example, Figure 7 shows the zone graph of the HDTA in Figure 3 (Example 4), with zones displayed graphically using x as the horizontal axis and y as the vertical. (We have taken the liberty to simplify by computing unions of zones at the locations u, e_3 and e_4 before proceeding.)

▶ **Lemma 15.** *For any HDTA A, an accepting location is reachable in A iff an accepting state is reachable in $Z(A)$.*

Proof. This follows from standard arguments as to the soundness and completeness of the zone abstraction, see [5]. ◀

Any standard *normalization* technique [10] may now be used to ensure that only a finite portion of the zone graph $Z(A)$ is visited, and then the standard zone algorithms can be employed to efficiently decide reachability in HDTA.

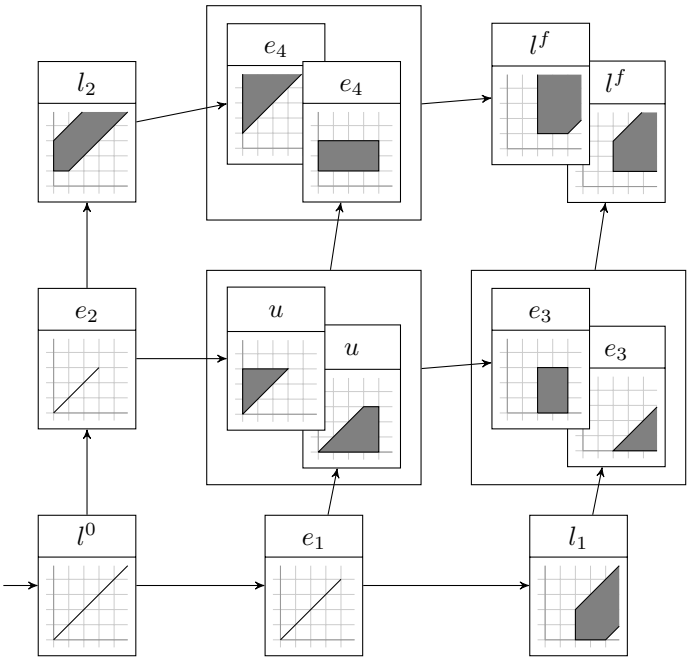


Figure 7 Zone graph of the HDTA in Figure 3.

Figure 8 The two 1DTA of Example 17.

7 Parallel Composition of HDTA

There is a *tensor product* on precubical sets which extends to HDTA and can be used for parallel composition (below we use $\sqcup$ for disjoint unions):

▶ **Definition 16.** Let $A_i = (L^i, l^{i,0}, L^{i,f}, \lambda^i, \mathsf{inv}^i, \mathsf{exit}^i)$, for $i = 1, 2$, be HDTA. The *tensor product* of A^1 and A^2 is $A^1 \otimes A^2 = (L, l^0, L^f, \lambda, \mathsf{inv}, \mathsf{exit})$ given as follows:

$$L_n = \bigsqcup_{p+q=n} L^1_p \times L^2_q \qquad l^0 = (l^{1,0}, l^{2,0}) \qquad L^f = L^{1,f} \times L^{2,f}$$

$$\delta^\nu_i(l^1, l^2) = \begin{cases} (\delta^\nu_i l^1, l^2) & \text{if } i \leq \dim l^1 \\ (l^1, \delta^\nu_{i-\dim l^1} l^2) & \text{if } i > \dim l^1 \end{cases}$$

$$\lambda(l^1, l^2) = \lambda(l^1) \sqcup \lambda(l^2) \qquad \mathsf{inv}(l^1, l^2) = \mathsf{inv}(l^1) \wedge \mathsf{inv}(l^2)$$

$$\mathsf{exit}(l^1, l^2) = \mathsf{exit}(l^1) \sqcup \mathsf{exit}(l^2)$$

Intuitively, tensor product is asynchronous parallel composition, or independent product. In combination with relabeling and restriction, any parallel composition operator can be obtained, see [60] or [24] for the special case of HDA.

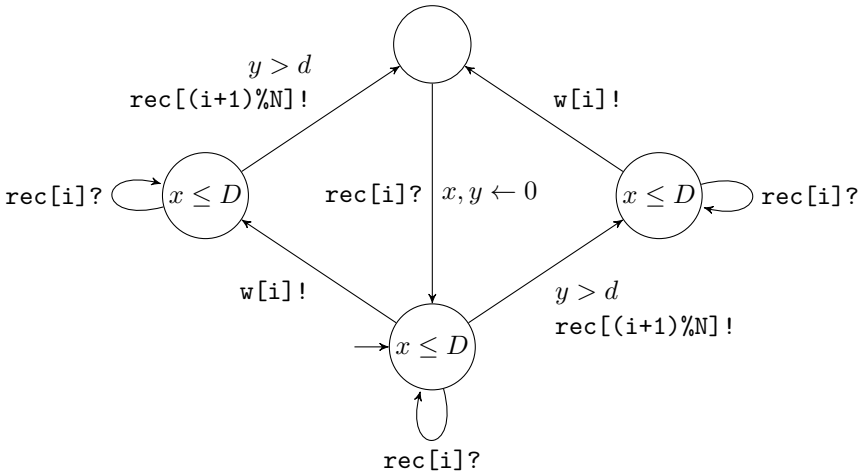


Figure 9 A single node in Milner's scheduler from [19].

▶ **Example 17.** Of the two 1DTA in Figure 8, the first models the constraint that performing the action a takes between two and four time units, and the second, that performing b takes between one and three time units. (In the notation of [14], these are the processes $a[2]{:}a(2){:}0$ and $b[1]{:}b(2){:}0$.) Their tensor product is precisely the HDTA of Example 4.

▶ **Example 18.** Using tensor product for parallel composition, one can avoid introducing spurious interleavings and thus combat state-space explosion. As an example, we recall the real-time version of Milner's scheduler from [19], a real-time round-robin scheduler in which the nodes are simple timed automata. Figure 9 shows one node in the scheduler, with two transitions from the initial to the topmost state, one that outputs w[i] ("work") and another that passes on the token (rec[(i+1)%N]!). These transitions are independent, but because of the limitations of the timed-automata formalism, they have to be modeled as an interleaving diamond. Apart from the number N of nodes the model has two other parameters, real numbers $d < D$ that specify the time interval within which the tokens have to be passed on.

When a larger number of nodes ($\mathtt{N} = 30$, say) are composed into a scheduler, a high amount of interleaving is generated: but most of it is spurious, owing to constraints of the modeling language rather than properties of the system at hand. That is, most of the interleaving in the composed model is an artefact of the modeling formalism and denotes, so to say, higher-dimensional concurrent structures which have been forgotten. The authors of [19] show that especially when d is much smaller than D (for example, $d = 4$ and $D = 30$), verification of the scheduler becomes impossible already for $\mathtt{N} = 6$ nodes.

One can use methods from partial order reduction [33] to *detect* spurious interleavings. Aside from the fact that this has proven to be largely impractical for timed automata [39], we also argue that by using HDTA as a modeling language, partial order reduction is, so to speak, *built into* the model. Spurious interleavings are taken care of during the modeling phase, instead of having to be detected during the verification phase.

8 Higher-Dimensional Hybrid Automata

We show that our definition of HDTA extends to one for higher-dimensional hybrid automata. Let X be a finite set of variables, $\dot{X} = \{\dot{x} \mid x \in X\}$, $X' = \{x' \mid x \in X\}$, and $\mathrm{Pred}(Y)$ the set of (arithmetic) predicates on free variables in Y.

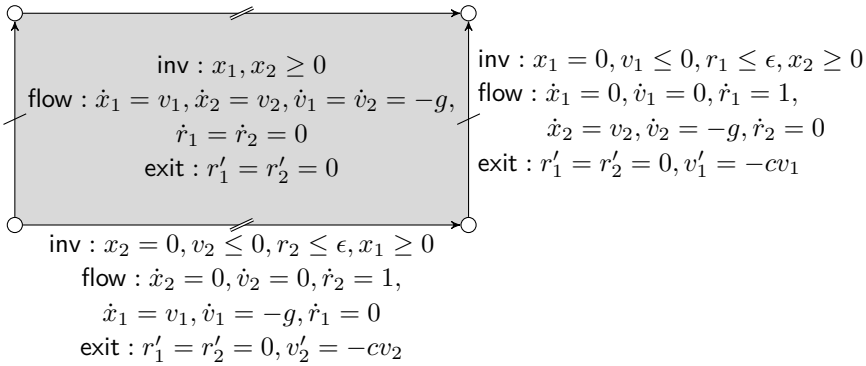


Figure 10 Two independently bouncing balls (opposite edges identified).

▶ **Definition 19.** A *higher-dimensional hybrid automaton* (HDHA) is a structure $(L, \lambda, \mathsf{init}, \mathsf{inv}, \mathsf{flow}, \mathsf{exit})$, where (L, λ) is a finite higher-dimensional automaton and $\mathsf{init}, \mathsf{inv} : L \to \mathrm{Pred}(X)$, $\mathsf{flow} : L \to \mathrm{Pred}(X \cup \dot{X})$, and $\mathsf{exit} : L \to \mathrm{Pred}(X \cup X')$ assign *initial*, *invariant*, *flow*, and *exit* conditions to each n-cube.

Note that we have removed initial and final locations from the definition; this is standard for hybrid automata. Also, remark how this continues our mantra that there is no conceptual difference between states and transitions; everything is an n-cube, and what are transition *guards* in hybrid automata are now invariants.

The *semantics* of a HDHA $A = (L, \lambda, \mathsf{init}, \mathsf{inv}, \mathsf{flow}, \mathsf{exit})$ is a (usually uncountably infinite) transition system $[\![A]\!] = (S, S^0, \rightsquigarrow)$, with $\rightsquigarrow \subseteq S \times S$, given as follows:

$$
\begin{aligned}
S &= \{(l, v) \in L \times \mathbb{R}^X_{\geq 0} \mid v \models \mathsf{inv}(l)\} \\
S^0 &= \{(l, v) \in S \mid v \models \mathsf{init}(l)\} \\
\rightsquigarrow &= \{((l, v), (l, v')) \mid \exists d \geq 0, f \in \mathcal{D}([0, d], \mathbb{R}^X) : f(0) = v, f(d) = v', \\
&\qquad\qquad \forall t \in]0, d[: f(t) \models \mathsf{inv}(q), (f(t), \dot{f}(t)) \models \mathsf{flow}(q)\} \\
&\quad \cup \{((\delta^0_k l, v), (l, v')) \mid k \in \{1, \ldots, \dim l\}, (v, v') \models \mathsf{exit}(\delta^0_k l)\} \\
&\quad \cup \{((l, v), (\delta^1_k l, v')) \mid k \in \{1, \ldots, \dim l\}, (v, v') \models \mathsf{exit}(l)\}
\end{aligned}
$$

Here $\mathcal{D}(D_1, D_2)$ denotes the set of differentiable functions $D_1 \to D_2$, and we write $(v, \dot{v}) \models \mathsf{flow}(q)$ to mean that the predicate $\mathsf{flow}(q)$ on $X \cup \dot{X}$ evaluates to true when the variables are replaced by their values in v and $\dot{v}$; similarly for $(v, v') \models \mathsf{exit}(l)$.

▶ **Example 20.** As a non-trivial example, we show a 2DHA which models two independently bouncing balls, following the *temporal regularization* from [40], in Figure 10. Here, the 2-cube models the state in which both balls are in the air. Its left and right edges are identified, as are its lower and upper edges, so that topologically, this model is a torus.

Its left / right edge is the state in which the second ball is in the air, whereas the first ball is in its ϵ-regularized transition ($\epsilon > 0$) from falling to raising ($v_1' = -cv$, for some $c \in]0, 1[$). Similarly, its lower / upper edge is the state in which the first ball is in the air, while the second ball is ϵ-transitioning.

Due to the identifications, there is only one 0-cube, which models the state in which both balls are ϵ-transitioning; its inv, flow and exit conditions can be inferred from the ones given.

We can extend the tensor product of HDTA to HDHA:

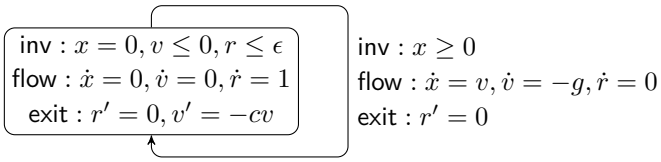


Figure 11 HDHA for a bouncing ball.

▶ **Definition 21.** Let $A_i = (L^i, \lambda^i, \mathsf{init}^i, \mathsf{inv}^i, \mathsf{flow}^i, \mathsf{exit}^i)$, for $i = 1, 2$, be HDHA. The *tensor product* of A^1 and A^2 is $A^1 \otimes A^2 = (L, \lambda, \mathsf{inv}, \mathsf{flow}, \mathsf{exit})$ given as follows:

$$L_n = \bigsqcup_{p+q=n} L^1_p \times L^2_q \qquad \delta^\nu_i(l^1, l^2) = \begin{cases} (\delta^\nu_i l^1, l^2) & \text{if } i \leq \dim l^1 \\ (l^1, \delta^\nu_{i-\dim l^1} l^2) & \text{if } i > \dim l^1 \end{cases}$$

$$\lambda(l^1, l^2) = \lambda(l^1) \sqcup \lambda(l^2)$$

$$\mathsf{init}(l^1, l^2) = \mathsf{init}(l^1) \wedge \mathsf{init}(l^2) \qquad \mathsf{inv}(l^1, l^2) = \mathsf{inv}(l^1) \wedge \mathsf{inv}(l^2)$$

$$\mathsf{flow}(l^1, l^2) = \mathsf{flow}(l^1) \wedge \mathsf{flow}(l^2) \qquad \mathsf{exit}(l^1, l^2) = \mathsf{exit}(l^1) \wedge \mathsf{exit}(l^2)$$

▶ **Example 22.** Figure 11 shows the HDHA for a bouncing ball, with one 1-cube in which the ball is in the air and one 0-cube for its ϵ-regularized transition from falling to raising. Denoting the model as $A(x, v, r)$, we can now construct models for arbitrarily many independently bouncing balls as

$$\bigotimes_{i=1}^{k} A(x_i, v_i, r_i)\,;$$

note that for $k = 2$ we obtain the HDHA from Figure 10. We emphasize that such a simple construction for systems of bounccing balls is not available in the standard interleaving formalisms for hybrid automata [3].

9 Conclusion and Further Work

We have seen that our new formalism of higher-dimensional timed automata (HDTA) is useful for modeling interesting properties of non-interleaving real-time systems, and that reachability for HDTA is PSPACE-complete, but can be decided using zone-based algorithms.

We have also shown how tensor product of HDTA can be used for parallel composition, and that HDTA can easily be generalized to higher-dimensional hybrid automata. We believe that altogether, this defines a powerful modeling formalism for non-interleaving cyber-physical systems.

We have argued that in a non-interleaving real-time setting, events should have a time duration. Note that this differs from [15, 18, 17], where, going back to [7], processes are partial orders of events which are fired punctually. Chatain and Jard in [18] notice that *"[t]ime and causality [do] not necessarily blend well in [...] Petri nets"* and propose to (locally) let time run backwards to get nicer semantics. We should like to argue that our proposal of letting events have duration appears more natural.

We have not paid any attention to categorical notions or results here. Higher-dimensional automata have a natural categorical semantics [24], and also for timed automata, works on categorical semantics are available [26, 45, 21], so this should be a natural extension. We would have liked the semantics of HDTA to be precubical in some sense, but this does not seem easy. A coalgebraic formulation of higher-dimensional automata would help here, but also this is not available.

We have mentioned that techniques from geometry and (directed) topology are used to analyze higher-dimensional automata. We have not used any such techniques here, but it appears only natural to try to extend them to work for HDTA. A starting point could be the *timed higher-dimensional automata* of Goubault's [35], which are essentially connected complexes of singular cubes in a locally compact Hausdorff space. We believe that HDTA can be given semantics as sets of such Goubault automata.

Finally, we should mention that this paper is part of a long-term effort to develop useful theory and tools for the analysis of *distributed hybrid systems*. Such systems consist of cyber-physical components which are distributed in the sense that no central clock synchronization mechanism is available, and the current state-of-the-art in distributed and hybrid systems analysis does not allow for the modeling and analysis of such systems. We believe that through convergence and interaction of methods and tools from concurrency theory, hybrid systems, control theory, and distributed systems, significant advances can be obtained in this area.

References

1 Luca Aceto, Anna Ingólfsdóttir, Kim G. Larsen, and Jiří Srba. *Reactive Systems*. Cambridge University Press, 2007.

2 Luca Aceto and François Laroussinie. Is your model checker on time? On the complexity of model checking for timed modal logics. *Journal of Logic and Algebraic Methods in Programming*, 52-53:7–51, 2002.

3 Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

4 Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990.

5 Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

6 Youssef Arbach, David Karcher, Kirstin Peters, and Uwe Nestmann. Dynamic causality in event structures. In Susanne Graf and Mahesh Viswanathan, editors, *FORTE*, volume 9039 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2015.

7 Tuomas Aura and Johan Lilius. Time processes for time Petri-nets. In Pierre Azéma and Gianfranco Balbo, editors, *ICATPN*, volume 1248 of *Lecture Notes in Computer Science*, pages 136–155. Springer, 1997.

8 Marek A. Bednarczyk. *Categories of asynchronous systems*. PhD thesis, University of Sussex, UK, 1987.

9 Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *SFM-RT*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

10 Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

11 Béatrice Bérard, Antoine Petit, Volker Diekert, and Paul Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36(2-3):145–182, 1998.

12 Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, Nicolas Markey, Joël Ouaknine, and James Worrell. Model checking real-time systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking.*, pages 1001–1046. Springer, 2018.

13 Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.

14 Luca Cardelli. Real time agents. In Mogens Nielsen and Erik Meineche Schmidt, editors, *ICALP*, volume 140 of *Lecture Notes in Computer Science*, pages 94–106. Springer, 1982.

15 Franck Cassez, Thomas Chatain, and Claude Jard. Symbolic unfoldings for networks of timed automata. In Susanne Graf and Wenhui Zhang, editors, *ATVA*, volume 4218 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2006.

16 Giovanni Casu and G. Michele Pinna. Petri nets and dynamic causality for service-oriented computations. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *SAC*, pages 1326–1333. ACM, 2017.

17 Thomas Chatain and Claude Jard. Complete finite prefixes of symbolic unfoldings of safe time Petri nets. In Susanna Donatelli and P. S. Thiagarajan, editors, *ICATPN*, volume 4024 of *Lecture Notes in Computer Science*, pages 125–145. Springer, 2006.

18 Thomas Chatain and Claude Jard. Back in time Petri nets. In Víctor A. Braberman and Laurent Fribourg, editors, *FORMATS*, volume 8053 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2013.

19 Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, Louis-Marie Traonouez, and Andrzej Wasowski. Real-time specifications.

Int. J. Software Tools for Technology Transfer, 17(1):17–45, 2015.

20 Jérémy Dubut. Trees in partial higher dimensional automata. In Mikołaj Bojańczyk and Alex Simpson, editors, *FOSSACS*, volume 11425 of *Lecture Notes in Computer Science*, pages 224–241. Springer, 2019.

21 Jérémy Dubut, Ichiro Hasuo, Shin-ya Katsumata, and David Sprunger. Quantitative bisimulations using coreflections and open morphisms. *CoRR*, abs/1809.09278, 2018.

22 Javier Esparza. A false history of true concurrency: From Petri to tools (invited talk). In Jaco van de Pol and Michael Weber, editors, *SPIN*, volume 6349 of *Lecture Notes in Computer Science*, pages 180–186. Springer, 2010.

23 Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. Monographs Theor. Comput. Sci. Springer, 2008.

24 Uli Fahrenberg. A category of higher-dimensional automata. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2005.

25 Uli Fahrenberg. *Higher-Dimensional Automata from a Topological Viewpoint*. PhD thesis, Aalborg University, Denmark, 2005.

26 Uli Fahrenberg. How to pull back open maps along semantics functors. In Jochen Pfalzgraf, editor, *ACCAT*, 2008.

27 Uli Fahrenberg. Higher-dimensional timed automata. In Alessandro Abate, Antoine Girard, and Maurice Heemels, editors, *ADHS*, volume 51 of *IFAC-PapersOnLine*, pages 109–114. Elsevier, 2018.

28 Uli Fahrenberg, Christian Johansen, Georg Struth, and Krzysztof Ziemiański. Languages of higher-dimensional automata. *Mathematical Structures in Computer Science*, pages 1–39, 2021.

29 Uli Fahrenberg and Axel Legay. Partial higher-dimensional automata. In Lawrence S. Moss and Pawel Sobocinski, editors, *CALCO*, volume 35 of *LIPIcs*, pages 101–115, 2015.

30 Lisbeth Fajstrup, Eric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raussen. *Directed Algebraic Topology and Concurrency*. Springer, 2016.

31 Lisbeth Fajstrup, Martin Raussen, and Éric Goubault. Algebraic topology and concurrency. *Theoretical Computer Science*, 357(1-3):241–278, 2006.

32 Hans Fleischhack and Christian Stehno. Computing a finite prefix of a time Petri net. In Javier Esparza and Charles Lakos, editors, *ICATPN*, volume 2360 of *Lecture Notes in Computer Science*, pages 163–181. Springer, 2002.

33 Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

34 Ursula Goltz and Wolfgang Reisig. The non-sequential behavior of Petri nets. *Information and Control*, 57(2/3):125–147, 1983.

35 Eric Goubault. Durations for truly-concurrent transitions. In Hanne Riis Nielson, editor, *ESOP*, volume 1058 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 1996.

36 Eric Goubault. Labelled cubical sets and asynchronous transition systems: an adjunction. In *Preliminary Proceedings CMCIM'02*, 2002.

37 Marco Grandis. *Directed algebraic topology: models of non-reversible worlds*. New mathematical monographs. Cambridge University Press, 2009.

38 Hans-Michael Hanisch. Analysis of place/transition nets with timed arcs and its application to batch process control. In Marco Ajmone Marsan, editor, *ATPN*, volume 691 of *Lecture Notes in Computer Science*, pages 282–299. Springer, 1993.

39 Henri Hansen, Shang-Wei Lin, Yang Liu, Truong Khanh Nguyen, and Jun Sun. Partial order reduction for timed automata with abstractions. In Armin Biere and Roderick Bloem, editors, *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 391–406. Springer, 2014.

40 Karl Henrik Johansson, Magnus Egerstedt, John Lygeros, and Shankar Sastry. On the regularization of Zeno hybrid automata. *Systems & Control Letters*, 38(3):141–150, 1999.

41 Kim G. Larsen, Uli Fahrenberg, and Axel Legay. From timed automata to stochastic hybrid games. In *Dependable Software Systems Engineering*, pages 60–103. IOS Press, 2017.

42 Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. J. Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

43 Philip M. Merlin and David J. Farber. Recoverability of communication protocols–implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, 1976.

44 Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

45 Mogens Nielsen and Thomas Hune. Bisimulation and open maps for timed transition systems. *Fundamenta Informaticae*, 38(1-2):61–77, 1999.

46 Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.

47 Carl A. Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

48 Vaughan R. Pratt. Modeling concurrency with geometry. In David S. Wise, editor, *POPL*, pages 311–322. ACM Press, 1991.

49 Vaughan R. Pratt. Higher dimensional automata revisited. *Mathematical Structures in Computer Science*, 10(4):525–548, 2000.

50 Mike W. Shields. Concurrent machines. *The Computer Journal*, 28(5):449–465, 1985.

51 Joseph Sifakis. Use of Petri nets for performance evaluation. In *Measuring, Modelling and Evaluating Computer Systems*, pages 75–93. North-Holland, 1977.

52 Joseph Sifakis and Sergio Yovine. Compositional specification of timed systems. In Claude Puech and Rüdiger Reischuk, editors, *STACS*, volume 1046 of *Lecture Notes in Computer Science*, pages 347–359. Springer, 1996.

53 Jiří Srba. Comparing the expressiveness of timed automata and timed extensions of Petri nets. In Franck Cassez and Claude Jard, editors, *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 2008.

54 Rob J. van Glabbeek. Bisimulations for higher dimensional automata. Email message, June 1991.

55 Rob J. van Glabbeek. On the expressiveness of higher dimensional automata. *Theoretical Computer Science*, 356(3):265–290, 2006.

56 Rob J. van Glabbeek. Erratum to "On the expressiveness of higher dimensional automata". *Theoretical Computer Science*, 368(1-2):168–194, 2006.

57 Rob J. van Glabbeek and Gordon D. Plotkin. Configuration structures. In *LICS*, pages 199–209. IEEE Computer Society, 1995.

58 Rob J. van Glabbeek and Gordon D. Plotkin. Configuration structures, event structures and Petri nets. *Theoretical Computer Science*, 410(41):4111–4159, 2009.

59 Farn Wang, Aloysius K. Mok, and E. Allen Emerson. Symbolic model checking for distributed real-time systems. In Jim Woodcock and Peter Gorm Larsen, editors, *FME*, volume 670 of *Lecture Notes in Computer Science*, pages 632–651. Springer, 1993.

60 Glynn Winskel and Mogens Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science*, volume 4. Clarendon Press, Oxford, 1995.

A Hybrid Programming Language for Formal Modeling and Verification of Hybrid Systems

Eduard Kamburjan
Department of Informatics, University of Oslo, Norway
Department of Computer Science, Technische Universität Darmstadt, Germany

Stefan Mitsch
Computer Science Department, Carnegie Mellon University, USA

Reiner Hähnle
Department of Computer Science, Technische Universität Darmstadt, Germany

Abstract

Designing and modeling complex cyber-physical systems (CPS) faces the double challenge of combined discrete-continuous dynamics and concurrent behavior. Existing formal modeling and verification languages for CPS expose the underlying proof search technology. They lack high-level structuring elements and are not efficiently executable. The ensuing modeling gap renders formal CPS models hard to understand and to validate. We propose a high-level *programming*-based approach to formal modeling and verification of hybrid systems as a hybrid extension of an Active Objects language. Well-structured hybrid active programs and requirements allow automatic, reachability-preserving translation into differential dynamic logic, a logic for hybrid (discrete-continuous) programs. Verification is achieved by discharging the resulting formulas with the theorem prover KeYmaera X. We demonstrate the usability of our approach with case studies.

2012 ACM Subject Classification Computing methodologies → Distributed programming languages; Computing methodologies → Model verification and validation; Theory of computation → Logic and verification; Theory of computation → Timed and hybrid models

Keywords and Phrases Active Objects, Differential Dynamic Logic, Hybrid Systems

Digital Object Identifier 10.4230/LITES.8.2.4

Supplementary Material *Software (HABS Simulator Virtual Machine)*: `https://doi.org/10.5281/zenodo.5973904`

Funding This work is partially supported by the `FormbaR` project, part of AG Signalling/DB RailLab in the Innovation Alliance of Deutsche Bahn AG and TU Darmstadt. This material is based upon work supported by AFOSR grant FA9550-16-1-0288.

Received 2020-06-10 **Accepted** 2022-05-11 **Published** 2022-12-07

Editor Alessandro Abate, Uli Fahrenberg, and Martin Fränzle

Special Issue Special Issue on Distributed Hybrid Systems

1 Introduction

Networked cyber-physical systems (CPS) are a main driving force of innovation in computing, from manufacturing to everyday appliances. But to design and model such systems poses a double challenge: first, their *hybrid* nature, with both continuous physical dynamics and complex computations in discrete time steps. Second, their *concurrent* nature: distributed, active components (sensors, actuators, controllers) execute simultaneously and communicate asynchronously. It is notoriously difficult to get CPS models right. *Formal* modeling languages, including hybrid automata [5], hybrid process algebra [27], and logics for hybrid programs [65], can be used to formally verify properties of CPS. Contrary to simulation frameworks, such as Ptolemy [71] or Simulink, however, these languages were *designed for verification* and are based on concepts of the underlying verification technology: automata, algebras, formulas. Their minimalist

syntax lacks standard structuring elements of programming languages such as types, scopes, methods, complex commands, futures, etc. Thus it is hard to adequately represent concurrently executing, communicating, hybrid components with *symbolic* data structures and computations, for example, servers or cloud applications.

Moreover, "low-level" models are hard to *validate*, i.e. to ensure that a CPS model reflects the designer's intention, because these formalisms are not (efficiently) executable. To bridge the modeling gap we propose a high-level *programming*-based approach to formal modeling and verification of hybrid systems.

The basis of our approach is an *Active Objects* (AO) language [29] called `ABS` [48]. AO languages combine OO programming with strong encapsulation as well as asynchronous, parallel execution. Their concurrency model permits to decompose concurrent execution into sequential execution in a compositional manner. We chose `ABS` for its formal semantics, its open source implementation tool chain, and its demonstrated scaling on massively distributed systems [75], but our approach is applicable to other AO languages. `ABS` is efficiently executable via compilation to Erlang and was used to model complex, real-world systems for cloud processing [3], virtualized services [49], data processing [56], and railway operations [53]. However, it lacks the capability to model hybrid systems. The *first main contribution* of this paper is the design of the *Hybrid `ABS`* (`HABS`) language, a conservative (syntax and semantics preserving) extension of `ABS`, generalizing the Active Objects paradigm to *Hybrid Active Objects* (HAO): AO with continuous dynamics. Obviously, it is necessary to accordingly extend the formal semantics of `ABS` and its runtime environment. This is our *second main contribution*. Our *third main contribution* is the implementation of `HABS` and a formal verification tool for it.

Our approach to formal verification of `HABS` programs is based on reachability-preserving translation into an existing verification formalism for hybrid programs. We choose differential dynamic logic ($d\mathcal{L}$) [66, 68, 69], as implemented in the KeYmaera X system [36], because it is based on an imperative programming language that is a good match for the sequential fragment of `HABS` and verification in $d\mathcal{L}$ has been demonstrated to scale to realistic systems (e.g., [47]). The translation from `HABS` to $d\mathcal{L}$ involves to decompose a given `HABS` verification problem into a set of independent *sequential* $d\mathcal{L}$ problems. This is possible, because we impose an interaction pattern for communication on `HABS` that is less restrictive than available component-based techniques [64], yet is general enough to permit intuitive and concise modeling of relevant case studies. The identification of this pattern, the generation of $d\mathcal{L}$ verification conditions, and a reachability preservation theorem constitute our *fourth main contribution*.

The overall approach is illustrated in Fig. 1: A CPS is modeled as an `HABS` program with the aim to analyze its properties statically. One formulates desired properties as invariants that are formally verified to hold under certain assumptions. Before verification is attempted, the model is *validated* by executing it in the runtime environment to ensure that it behaves as intended. A visualization component helps to analyze behavior over time. Subsequently, the verification claim is automatically decomposed and translated into a set of $d\mathcal{L}$ verification problems discharged in KeYmaera X (optionally, formally verified runtime monitors [63] and formally verified machine code is available from KeYmaera X through VeriPhy [18]). Both, unexpected runtime behavior and failed verification attempts, serve to fix the model and/or the claimed properties.

The paper is structured as follows. Sect. 2 gives an informal example of an `HABS` model with a distributed water tank controller. Sect. 3 formally defines syntax and semantics of `HABS`. Sect. 4 describes modeling patterns. Sect. 5 gives theoretical background on $d\mathcal{L}$, the translation into $d\mathcal{L}$, the decomposition theorem, and tells how to prove correctness. It also contains a distributed controller case study. Finally, Sect. 6 discusses related and future work and concludes.

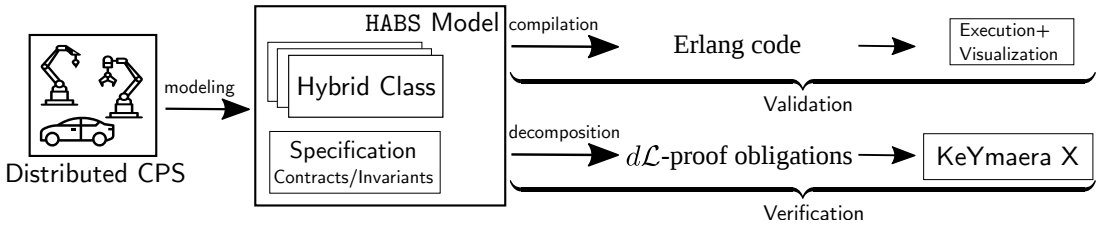


Figure 1 Structure of HABS workflow.

2 Distributed Hybrid Systems by Example

Active Objects [29] are objects that realize actor-based concurrency [44] with futures [28] and cooperative scheduling: Active Objects communicate via asynchronous method calls. On the caller side, each method invocation generates a future as a handle to retrieve the call's result, once it is available. The caller may synchronize on that future, i.e. suspend and wait until it is resolved. At most one process is running on an Active Object at any time. That process suspends when it encounters the synchronization statement **await** on an unresolved future or a false Boolean condition. Once the guard becomes true, the process may be re-scheduled. All fields are strictly object-private.

Running a Hybrid Active Objects (HAO) model of a CPS can be pictured as follows: each object is capable of modeling a physical object, for example, a water tank. It may declare *physical* behavior via ordinary differential equations (ODEs) over "physical" fields, as well as *discrete* behavior via class and method declarations that can be used to control physical behavior. Once an HAO starts executing, the values of the physical fields evolve, governed by their ODEs, even when the controller is idle. This models the intuition that a physical system evolves independently of any observers and controllers.

Object orientation allows natural modeling of hybrid systems: continuous behavior is attached to an *object*, not a process. Processes realize discrete control behavior related to sensors and controllers. Specifically, the controller methods of an object may wait to execute until a certain physical state is reached (event-triggered control, for example, "tank is nearly full"). This "sensing" is modeled with getter methods of physical fields. Obviously, for validation the HABS runtime system must solve the differential equations in the physical model to determine the time point when such a waiting controller can start at the earliest; for verification, ODEs need not be solvable; they are analyzed with invariant-based techniques [67, 70]. Another communication pattern for controllers – time-triggered control – is provided by fixed sampling durations. More complex control patterns can be realized by waiting until the result of a subcomputation, i.e. a future, is ready.

Whenever a control process is activated, it can modify the physical state through actuators (for example, close a valve). In consequence, there are no timed race conditions, but the physical state might be changed by any process at the time it is scheduled. Actuation is modeled with setter methods of physical fields. Execution of control methods is assumed to take no physical time, unless explicitly modeled to do so.

Generally, a CPS can be modeled by several HAOs that communicate with each other via asynchronous method calls, for example, modeling a central controller. Often a controller object has no associated physical behavior; vice versa, an object that models physics, may not contain any control, but only sensor and actuator methods.

We demonstrate HAOs using three variants of water tank models. The first model, **TankMono**, is a single water tank that keeps its water level between two thresholds. It is modeled as a single object that integrates control and physics. The second model, **TankTick**, is also a single water tank,

but it is modeled with two separate objects for tank and controller. The final model, **TankMulti**, is a distributed system of n **TankMono** tanks that, in addition to the local threshold, maintain a global threshold over the sum of all local water levels.

2.1 Base System: TankMono

```
1  interface ISingleTank {
2   /*@ ensures 3 <= outLevel() <= 10 @*/
3   Real outLevel();
4   /*@ ensures −1/2 <= outDrain() <= 1/2 @
    */
5   Real outDrain();
6  }
7  /*@ requires 4 <= inVal <= 9 @*/
8  class CSingleTank(Real inVal)
9         implements ISingleTank {
10 /*@ invariant
11        3 <= level <= 10
12      & −1/2 <= drain <= 1/2
13      & (drain<0−>level>3)
14      & (drain>0−>level<10) @*/
15 physical {
16   Real level = inVal : level' = drain;
17   Real drain = -1/2  : drain' = 0;
18 }
19 Unit run() { this!ctrl(); }
20 Unit ctrl() {
21   await diff (level<=3 & drain<=0) | (level>=10 & drain>=0);
22   if (level <= 3) drain =  1/2;
23   else            drain = -1/2;
24   this.ctrl();
25 }
26 Real outDrain() { return this.drain; }
27 Real outLevel() { return this.level; }
28 }
```

Figure 2 TankMono: A water tank as a single HAO.

Fig. 2 shows an HAO model of a water tank whose **physical** section makes it either fill with $\frac{1}{2}l/sec$ or drain at the same rate, according to the initial values and governing ODEs of the `level` and `drain` fields. Method `ctrl()` realizes a control loop that switches the `drain` field between those states so that the water level stays between $3l$ and $10l$. The controller `ctrl` waits until the water level reaches the upper or lower limit, i.e. until the condition in Fig. 2, Line 21 holds. Depending on the case, it changes the state and calls itself recursively.

The JML style [20] comments in Fig. 2 contain an assumption on the initial state of `inVal` and a conjectured safety invariant and conjectured output guarantees that, in this case, can be proven: if the initial level is between $4l$ and $9l$, then it always stays between $3l$ and $10l$. Note that Lines 13–14 express a safety invariant that must be *shown* to be true, rather than control conditions. Intuitively, Line 13 expresses the property that the tank won't drain below a threshold (`level` > 3) even if water is leaking from it (`drain` < 0). Similarly, Line 14 expresses that the tank won't overflow (`level` < 10) even if water is pumped into the tank (`drain` > 0). Prior to formal verification of this property one typically runs tests to see whether the model behaves as intended. Our implementation allows to simulate and visualize an HAO model. The graph in Fig. 3 shows the behavior of a `CSingleTank` object instantiated with `inVal` $= 5$. In Sect. 5 we show how the class is translated into $d\mathcal{L}$ and how to prove the safety invariant in KeYmaera X for *any* object created with a parameter that satisfies the precondition. The only methods exposed to clients in the interface are `outDrain()` and `outLevel()`.

2.2 Discrete Controller: TankTick

The `ctrl()` method in **TankMono** corresponds to a perfect sensor/controller that physically reacts to the water level and drain. **TankTick** splits controller and sensor into two objects and uses a clock to read the water level at certain intervals. This corresponds to a closed-loop control system with a discrete-time controller that samples the plant behavior.

Fig. 4 shows a water tank realized by a controller `FlowCtrl` and a `Tank` implementation `CTank`. The tank has an in-port (setter) method `inDrain()` and an out-port (getter) method `outLevel()`. It has no active *discrete* behavior on its own (the `run` method is empty), but its state changes nonetheless due to the *continuous* **physical** block. The `FlowCtrl` controller's fields `drain`, `level` are

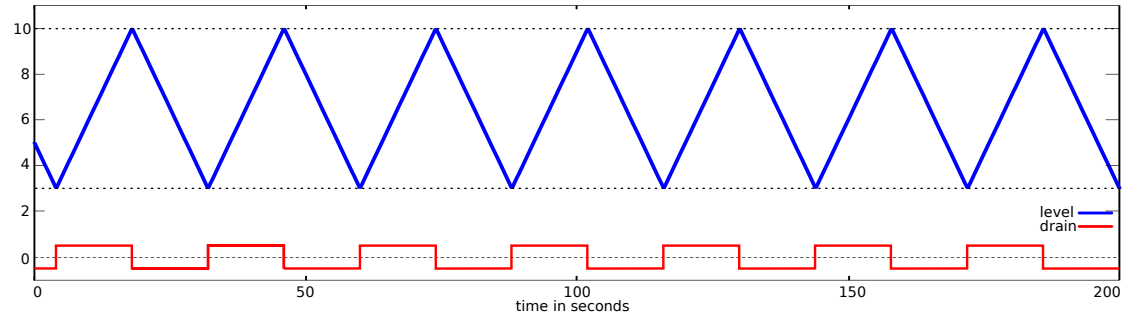


Figure 3 Simulation Output of **TankMono** with `inVal` = 5.

```
1  interface Tank {
2   /* requires -1/2 <= newD <= 1/2; */
3   Unit inDrain(Real newD);
4   /* ensures 3 <= outLevel() <= 10; */
5   Real outLevel();
6  }
7  class CTank(Real inVal) implements Tank {
8   physical {
9     Real level = inVal : level' = drain;
10    Real drain = -1/2 : drain' = 0;
11  }
12  Unit run() { }
13  /* requires newD > 0 -> level <= 9.5  */
14  /* requires newD < 0 -> level >= 3.5 */
15  /* timed_requires inDrain < 1 */
16  Unit inDrain(Real newD) { drain = newD; }
17  Real outLevel() { return level; }
18 }
```

```
19 /* requires 0 < tick < 1 & inVal > 3.5*/
20 class FlowCtrl(Tank t, Real tick, Real inVal) {
21 /* invariant (drain > 0 -> level <= 9.5)
22         & (drain < 0 -> level >= 3.5) */
23   Real drain = -1/2;
24   Real level = inVal;
25
26   Unit run() { this!ctrlFlow(); }
27
28   Unit ctrlFlow() {
29     await duration(tick,tick);
30     level = t.outLevel();
31     if (level <= 3.5) drain =  1/2;
32     if (level >= 9.5) drain = -1/2;
33     t!inDrain(drain);
34     this.ctrlFlow();
35   }
36 }
```

Figure 4 TankTick: A water tank modeled as two HAOs. Invariant and precondition of `CTank` are as in Fig. 2.

its *local copies* of the state of the tank: `CTank.drain`, `CTank.level` are different fields from `FlowCtrl.drain`, `FlowCtrl.level`, respectively, residing in different objects. The `ctrlFlow()` method first updates `level`, decides on the state of `drain`, then pushes the (possibly changed) state of `drain` to the tank. No time passes in the controller, which ensures that the copied fields are synchronized at the end of the round. As the `Tank`'s fields are not directly accessible by the `FlowCtrl` instance, it is not possible to wait on the `Tank`'s `level` with an **await diff** statement. Instead, the controller uses **await duration** to run every `tick` seconds: `tick` is the sampling time of the controller.

The `Tank` interface specification declares an input requirement and a guarantee on returned values. The input requirement of the `inDrain()` specification is a constraint on the input parameter `newD`; specifically, it means that the tank can only be instructed to fill if there is sufficient capacity left (similar for draining). The initial requirement is sufficient to establish the controller's invariant, which in turn ensures that the tank's requirements are met. The `timed_requires` clause stipulates that `inDrain()` is called at least once per second, which suffices for the output guarantee. Fig. 5 shows example output. We stress that all calls to `Tank` methods are *asynchronous*.

2.3 Distributed Tank Control: TankMulti

Consider a system where n water tanks are monitored by a central controller that aims to keep the sum of all water levels between some thresholds. The code in Fig. 6 shows a controller that monitors a list of `ISingleTank` (Fig. 2) instances. Each `tick` seconds the central controller iterates over the list of tanks and if their combined level is almost at the upper threshold, the controller

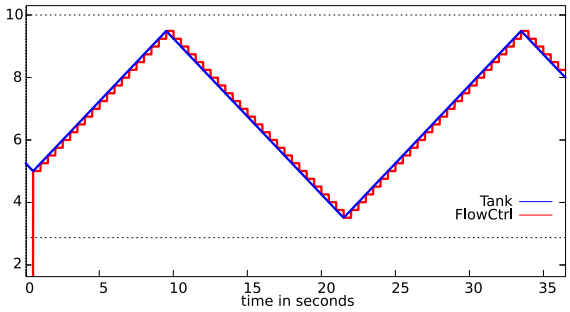


Figure 5 Simulation Output of **TankTick** with `inVal` $= 5$ for $30s$.

drains all water tanks with rising levels (analogously for the lower threshold). Single water tanks still ensure that their local thresholds are observed. To allow the `CControl` instance to manipulate the `ISingleTank` instances, we add the following method to `CSingleTank` (and an analogous method to the interface):

```
1 /* requires newD > 0 -> level < 10  */
2 /* requires newD < 0 -> level >  3  */
3 /* requires -1/2 <= newD <= 1/2 */
4 Unit inDrain(Real newD) { this.drain = newD; }
```

Contrary to the contract in **TankTick**, we do not need to specify how frequently the method is called, because this information is available in the guard of the `ctrl` method of the instances. The recursive call at the end of `ctrl` ensures that there is always one process executing `ctrl` for each instance of `FlowCtrl`.

The graph in Fig.6 shows the simulation output for four water tanks with different initial values. The upper thresholds are managed by the distributed controller and the water tanks cooperatively: Only tanks 1 and 4 reach their local upper thresholds, the others are drained by the distributed controller to maintain the global threshold. The lower local thresholds are managed locally, the lower global threshold is never reached.

2.4 Futures

Future-based communication allows to decouple the call of a method from retrieving its result. For example, consider the code in Fig. 7. Class `Node` can perform some complex and time consuming computations on behalf of class `Client`. To enable load balancing the client has only a reference to an interface `Server`, which relays its request. The `Server` performs basic load balancing by a round-robin scheduling on a list of nodes. It then returns to the issuing client the future of the relayed request *without having to wait* for the computation to finish (Line 17). The client can then retrieve the future (Line 7) to synchronize on it without blocking the interface server (Line 8).

3 Hybrid Active Objects

An informal description of the intended semantics of Hybrid Abstract Objects in the Hybrid Abstract Behavioral Specification (`HABS`) language was provided in Section 2. The present section gives a formal account of its syntax and semantics. `HABS` is an extension of the Active Object language `ABS` [48]. `ABS` itself extends standard OO concepts as follows:

Encapsulation. All fields are strictly object-private.

```
1 class CControl(List<ISingleTank> tanks,
2                Real totalLower,
3                Real totalHigher,
4                Real tick)
5 implements IControl {
6   Unit run() {
7     await duration(tick, tick);
8     Real total = 0;
9     List<ISingleTank> lower  = list[];
10    List<ISingleTank> higher = list[];
11    foreach ( next in tanks ) {
12      Real val = next.outLevel();
13      Real dir = next.outDrain();
14      if (dir < 0 && val > 3)
15        lower  = Cons(next, lower);
16      if (dir > 0 && val < 10)
17        higher = Cons(next, higher);
18      total = total + val;
19    }
20    if (total <= totalLower+1)
21       foreach ( lnext in lower  )
22         lnext!inDrain(1/2);
23    if (total >= totalHigher-1)
24       foreach ( hnext in higher )
25         hnext!inDrain(-1/2);
26    this.run();
27  }
28 }
```

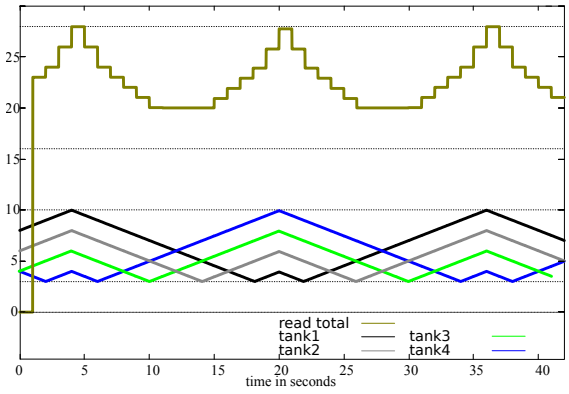


Figure 6 TankMulti: A controller for n **TankMono** instances and an example simulation output. Interface omitted.

Cooperative Scheduling. Active Objects cannot be preempted: a process running in an object may not be interrupted by other processes, unless the active process suspends itself or terminates.

Asynchronous Calls, Futures. All method calls to other objects are asynchronous. Every call not only generates a process on the callee side, but a future that points to that process. A process may pass around a future or synchronize with it to read the return value of the associated process once it has terminated.

As a *Timed* Active Object language, HABS also features:

Simulation Time. HABS allows to manipulate *simulation time* by explicitly advancing (and reading) an internal clock with specific statements. Simulation time is independent of the wall time.

3.1 Syntax

The syntax of HABS is given by the grammar in Fig. 8 and explained in the following section. With `e` we denote standard expressions over fields `f`, variables `v` and operators `|`, `&`, `>=`, `<=`, `+`, `-`, `*`, `/`. Types `T` are all interface names, type-generic futures **Fut**`<T>`, lists `List<T>`, `Real`, `Int`, `Unit` and `Bool`. We also assume the usual functions for lists, etc.

A program contains a main method Main, interfaces $\overline{\text{ID}}$ and classes $\overline{\text{CD}}$. Interfaces are standard, the main method contains a list of object creations. Classes can have parameters $\overline{\text{Tf}}$, these are fields being initialized during object creation. Classes have fields $\overline{\text{FD}}$, methods $\overline{\text{Met}}$, an optional run method Run to start a process, and an optional physical block Phys that declares physical fields. A declaration of a physical field is a field declaration followed by a differential equation. A differential equation is an equation between two differential expressions, which are standard expressions extended with a derivation operator `e'` for $\frac{\text{de}}{\text{dt}}$. HABS supports explicit autonomous differential equations. The differential expressions and the field initialization form an initialized ordinary differential equation, e.g., `Real f = 0: f' = 5-f`. Note that `f = 0` specifies the initial value of `f`, whereas the differential equation `f' = 5-f` is phrased in terms of the time-varying value of `f`, so models logarithmic growth towards `f = 5`.

```
1  class Node {
2     Real compute_internal(Real r1, Real r2, Real r3){ ...   }
3  }
4  class Client(Server s){
5     Unit run(){
6        Fut<Fut<Real>> ffr = s!compute(1,2);
7        Fut<Real> fr = ffr.get;
8        Real r = fr.get;
9        ...
10    }
11 }
12 class Server(Queue<Node> internal, Real param){
13    Fut<Real> compute(Real r1, Real r2){
14       Node n = internal.pop();
15       Fut<Real> fr = n!compute_internal(r1,r2,param);
16       internal.push(n);
17       return fr;
18    }
19 }
```

Figure 7 An example for load balancing using futures. Interfaces omitted.

Prgm ::= $\overline{\text{ID}}$ $\overline{\text{CD}}$ Main	ID ::= **interface** I [**extends** $\overline{\text{I}}$]?{$\overline{\text{MS}}$}	Programs, Interfaces
Main ::= {s?}		Main
CD ::= **class** C [**implements** $\overline{\text{I}}$]? [($\overline{\text{T f}}$)]?{Phys? $\overline{\text{FD}}$ $\overline{\text{Met}}$ Run?}		Classes
Run ::= Unit run() {s}	FD ::= T f = e	Run Method and Fields
Phys ::= **physical** {$\overline{\text{DED}}$}	DED ::= Real f = e : f' = e	Physical Block
MS ::= T m($\overline{\text{T v}}$)	Met ::= MS {s;**return** e;}	Signatures, Methods
s ::= **while** (e) {s} | **if** (e) {s} [**else** {s}]? | s;s		
| **await** g | [T? e]? = rhs		Statements
g ::= **duration**(e,e) | **diff** e | e?		Guards
rhs ::= e | **new** C($\overline{\text{e}}$) | e.**get** | e!m($\overline{\text{e}}$)		RHS Expressions

Figure 8 HABS grammar. T ranges over types, I over interfaces and C over classes. Differential expression de are normal expressions extended with a derivation operator e'.

Methods and statements are mostly standard, we focus on HAO-specific constructs. Methods are called asynchronously with e!m($\overline{\text{e}}$), i.e., after the call, the caller continues execution without waiting for the callee to finish. Instead, the caller generates a *future*. A future identifies the call and can be passed around by the caller. A process interacts in two ways with a future: either by awaiting its result with **await** e? on the guard e?, or by reading its value with e.**get**. Statements e.**get** block the reading *object* – no other process may run on it. In contrast, statements **await** g release the process control over the object while waiting for the guard g to hold. The guard is either a future guard e?, a differential guard **diff** e, or a timed guard **duration**(e1,e2). The future guard e? awaits the result of future e, the differential guard **diff** e suspends the process until the

expression `e` evaluates to true, and the timed guard **duration**`(e1,e2)` suspends the process for at least `e1` time units[1]. The notation `T v = o.m()` is short for **Fut**`<T> f = o!m(); T v = f.`**get**`;` (a call followed by a synchronization).

3.2 Semantics of HABS

HABS extends the structural operational semantics (SOS) for Timed ABS [16] in three aspects:

(i) it includes physical behavior in the object state;
(ii) determines whether a differential guard holds and, if not, when it will at the earliest;
(iii) updates the state whenever time passes.

This affects only expression evaluation and auxiliary functions. *No new SOS rule is needed.* In the following we extend the core of the ABS SOS semantics [16] to hybrid systems.

3.2.1 States

The state of an object has three parts:

(i) a store ρ that maps (physical and non-physical) fields to values, and the variables of the active process[2] to values;
(ii) ODE, the differential equations from its physical block;
(iii) F, the set of current solutions of ODE[3].

A solution f is a function from time to a store which only contains the physical fields. The set F may change, because the ODEs are solved as an initial-value problem with the current state of the physical fields as the initial values. For each $f \in F$ and each physical field `f` the following holds: $f(0)(\texttt{f}) = \rho(\texttt{f})$, i.e., the initial value $f(0)(\texttt{f})$ of physical field `f` is the current value $\rho(\texttt{f})$ in the store ρ. We denote the solutions of ODE with initial values from ρ by $\mathsf{sol}(ODE, \rho)$. We define runtime configurations formally:

$$tcn ::= \mathsf{clock}(\texttt{e})\ cn \qquad cn ::= cn\ cn \mid fut \mid msg \mid ob$$
$$ob ::= (o, \rho, \underbrace{ODE}, F, prc, \overline{prc}) \qquad msg ::= \mathsf{msg}(o, \overline{\texttt{e}}, f)$$
$$prc ::= (\tau, f, \texttt{rs}) \mid \bot \qquad \texttt{rs} ::= \texttt{s} \mid \textbf{suspend}\texttt{;s} \qquad fut ::= \mathsf{fut}(f, \texttt{e})$$

Figure 9 Runtime Syntax of HABS.

▶ **Definition 1** (Runtime Configuration [16]). The runtime syntax of HABS is summarized in Fig. 9: f ranges over future identities, o over object identities, ρ, τ over stores, i.e., assignments from fields or variables to values. A timed configuration has a clock clock with the current time, as an expression of `Real` type and an object configuration cn. An object configuration cn consists of messages msg, futures fut, objects ob, and can be composed $cn\ cn$ (as usual, composition is commutative and associative). A message $\mathsf{msg}(o, \overline{\texttt{e}}, f)$ records callee o, passed parameters $\overline{\texttt{e}}$ and the generated future f. A future configuration $\mathsf{fut}(f, \texttt{e})$ connects the future f with its return value `e`. An object $(o, \rho, F, ODE, prc, \overline{prc})$ has an identifier o, an object store ρ, the current solutions F, an active process prc and a queue of inactive processes. ODE is taken from the class declaration.

[1] The parameter `e2` is used by certain scheduling policies [16], and is not relevant for HABS.
[2] Recall that the active process executes the ABS methods, it does not relate to physical behavior.
[3] The solutions computed relative to the initial values (state) at the last suspension.

A process is either terminated $\bot$ or has the form $(\tau, f, \texttt{rs})$: the process store τ with current state of the local variables, its future f, and the statement `rs` left to execute. The runtime syntax also allows the **suspend** statement, which is used to deschedule a process. Dotted underlined elements are an extension of `HABS` relative to ABS (also in Fig. 10 below).

Given a process store τ and an object store ρ we use $\sigma = \rho \circ \tau$ to denote the state of both fields and local variables. We first define the evaluation of expressions and guards.

3.2.2 Evaluation of Expressions

Expressions `e` are evaluated with a function $[\![\texttt{e}]\!]^{F,t}_{\sigma}$ over a store σ and a set of solutions F at t time units in the future. The semantics of expressions containing physical fields is as follows.

▶ **Definition 2** (Semantics of Expressions)**.** Let F be the set of solutions. Given a store σ, we can check whether F is a model of an expression `e` after t time units. Let $\texttt{f}_p$ be a physical field and $\texttt{f}_d$ a non-physical field of o. The semantics of fields $\texttt{f}_p$, $\texttt{f}_d$, unary operators $\sim \in \{\texttt{!}, \texttt{-}\}$ and binary operators $\oplus \in \{\texttt{|}, \texttt{\&}, \texttt{>=}, \texttt{<=}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$ is defined as follows:

$$[\![\texttt{f}_d]\!]^{F,t}_{\sigma} = \sigma(\texttt{f}_d) \qquad [\![\texttt{f}_p]\!]^{F,t}_{\sigma} = \begin{cases} v & \text{if } \forall f \in F.\ v = f(t)(\texttt{f}_p) \\ \infty & \text{otherwise} \end{cases}$$

$$[\![\sim\texttt{e}]\!]^{F,t}_{\sigma} = \sim[\![\texttt{e}]\!]^{F,t}_{\sigma} \qquad [\![\texttt{e}_1 \oplus \texttt{e}_2]\!]^{F,t}_{\sigma} = [\![\texttt{e}_1]\!]^{F,t}_{\sigma} \oplus [\![\texttt{e}_2]\!]^{F,t}_{\sigma}$$

Outside differential guards, only the evaluation in the current state $[\![\texttt{e}]\!]^{F,0}_{\sigma}$ is needed, which is $\rho(\texttt{f}_p)$ from $f(0)(\texttt{f}_p)$ and this expression is never ∞. We identify $[\![\texttt{e}]\!]^{F}_{\sigma}$ and $[\![\texttt{e}]\!]_{\sigma}$with $[\![\texttt{e}]\!]^{F,0}_{\sigma}$.

3.2.3 Evaluation of Guards

The semantics of an **await** `g` statement is to suspend until the guard holds, i.e. until $[\![\texttt{g}]\!]^{F}_{\sigma}$ evaluates to true. For example, a duration guard **duration**`(e1,e2)` evaluates to true if $[\![\texttt{e1}]\!]^{F}_{\sigma} \leq 0$. Defining the semantics of guards requires two operations: An extension of the *evaluation function* that returns true if the guard holds and the *maximal time elapse* mte^{F}_{σ} returning the time t that may elapse before the guard evaluates to true, or ∞ if it never does.

First we define $mte(\texttt{e})$: the *maximal* time that may elapse without missing an event is the *minimal* time needed by the system to evolve into a state where the guard is guaranteed to hold. This yields also the semantics of the guard itself.

▶ **Definition 3** (Semantics of Differential Guards)**.** Let F be the set of solutions of object o in state σ. Then we define:

$$mte^{F}_{\sigma}(\textbf{diff}\ \texttt{e}) = \operatorname*{\mathbf{argmin}}_{t \geq 0} \left([\![\texttt{e}]\!]^{F,t}_{\sigma} = \text{true} \right)$$

diff `e` is evaluated to true if no time advance is needed:

$$[\![\textbf{diff}\ \texttt{e}]\!]^{F,0}_{\sigma} = \text{true} \iff mte^{F}_{\sigma}(\textbf{diff}\ \texttt{e}) = 0$$

If `e` contains no continuous variable then the differential guard semantics and the evaluation of expressions in Def. 2 coincides with condition synchronization and expression evaluation in the standard `ABS` semantics [48].

$$\textbf{(1)}\quad \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau, f, \textbf{await}\ \texttt{g;s}), q\big) \to \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau, f, \textbf{suspend};\textbf{await}\ \texttt{g;s}), q\big)$$

$$\textbf{(2)}\quad \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau, f, \textbf{suspend};\texttt{s}), q\big) \to \big(o, \rho, \underset{\cdots}{ODE}, \mathsf{sol}(ODE, \rho), \bot, q \circ (\tau, f, \texttt{s})\big)$$

$$\textbf{(3)}\quad \big(o, \rho, \underset{\cdots}{ODE}, F, \bot, q \circ (\tau, f, \textbf{await}\ \texttt{g;s})\big) \to \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau, f, \texttt{s}), q\big) \quad \text{if } [\![\texttt{g}]\!]_{\rho\circ\tau} = \text{true}$$

$$\textbf{(4)}\quad \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau, f, \texttt{v = e;s}), q\big) \to \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau[\texttt{v} \mapsto [\![\texttt{e}]\!]_{\rho\circ\tau}], f, \texttt{s}), q\big) \quad \text{if e contains no call or } \textbf{get}$$

$$\textbf{(5)}\quad \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau, f, \textbf{return}\ \texttt{e;}), q\big) \to \big(o, \rho, \underset{\cdots}{ODE}, \mathsf{sol}(ODE, \rho), \bot, q\big)\ \mathsf{fut}\big(f, [\![\texttt{e}]\!]_{\rho\circ\tau}\big)$$

$$\textbf{(6)}\ \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau, f, \texttt{v = e}_1\texttt{.}\textbf{get}\texttt{;s}), q\big)\ \mathsf{fut}\big(f, \texttt{e}_2\big) \to \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau, f, \texttt{v = e}_2\texttt{;s}), q\big) \quad \text{if } [\![\texttt{e}_1]\!]_{\rho\circ\tau} = f$$

$$\textbf{(7)}\quad \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau, f, \texttt{v = e!m(e}_1, \ldots \texttt{e}_n\texttt{;s}), q\big) \to \big(o, \rho, \underset{\cdots}{ODE}, F, (\tau[\texttt{v} \mapsto \tilde{f}], f, \texttt{s}), q\big)\ \mathsf{msg}\big([\![\texttt{e}]\!]_{\rho\circ\tau}, ([\![\texttt{e}_1]\!]_{\rho\circ\tau}, \ldots, [\![\texttt{e}_n]\!]_{\rho\circ\tau}), \tilde{f}\big) \quad \text{where } \tilde{f} \text{ is fresh}$$

Figure 10 Selected Rules for HABS objects.

3.2.4 Transition System

Fig. 10 gives the most important rules for the semantics of a single object, the omitted rules are given in [16]. Rules **(1)**–**(3)** define the semantics of process suspension. An **await** statement suspends the current process and gives other processes in the queue q a chance to run, even if its guard is evaluated to true. Suspension is modeled in rule **(1)** simply by introducing a **suspend** statement in front of the **await**.[4] Rule **(2)** realizes a **suspend** statement by moving the current process to the object's queue. As explained in Sect. 3.2.3, upon reactivation of a suspended process we must ensure its guard to be true, relative to the solution of *ODE* with *initial values at suspension time*. Therefore, rule **(2)** also recomputes the solutions F. Rule **(3)** can then re-activate a process beginning with an **await** statement, simply by checking whether its guard evaluates to true at current time (advancing time in timed configuration is explained below). An analogous rule (not shown in Fig. 10) activates a process with any other non-**await** statement. Rule **(4)** evaluates an assignment to a local variable. The rule for fields is analogous. Rule **(5)** realizes a termination (with solutions of the ODEs) and **(6)** a future read. Finally, **(7)** is a method call, the rule for transforming a message into a process is straightforward.

For configurations, there are two rules, shown in Fig. 11. Rule **(i)** realizes a step of some object without advancing time, Only if **(i)** is not applicable, i.e. all ABS processes are blocked, rule **(ii)** can be applied. It computes the global maximal time elapse *mte* and advances the time in the clock and all objects. In particular, it decreases syntactically the timed guards.

(i) $\mathsf{clock}(t)\ cn\ cn_1 \to \mathsf{clock}(t)\ cn_2\ cn_1$ with $cn \to cn_2$

(ii) $\mathsf{clock}(t)\ cn \to \mathsf{clock}(t + \tilde{t})\ adv(cn, \tilde{t})$ if **(i)** is not applicable and $mte(cn) = \tilde{t} \neq \infty$

Figure 11 Timed Semantics of HABS configurations.

[4] We follow the original ABS semantics, where suspension is handled with a separate **suspend** statement for reasons of uniformity – in principle, rules **(1)**+**(2)** could be combined.

Fig. 12 shows the auxiliary functions and includes the full definition of *mte*. Note that *mte* is not applied to the currently active process, because, when **(1)** is not applicable, it is currently blocking and, thus, cannot advance time. *The characteristic feature of hybrid objects is that their physical state changes when time advances, even when no process is active.* This is expressed in the semantics by a function $adv(\sigma, t)$ which takes a state σ, a duration t, and advances σ by t time units. For non-hybrid Active Objects $adv(\sigma, t) = \sigma$. There, the function is needed only to modify the process pool of an object for scheduling, not its state, and is used exactly as in [16].

$$
\begin{aligned}
mte(cn_1\ cn_2) &= \mathbf{min}(mte(cn_1), mte(cn_2)) & mte(msg) = mte(fut) &= \infty \\
mte(o, \rho, ODE, F, prc, q) &= [\![\mathbf{min}(mte(q), \infty)]\!]_\rho & mte(\tau, f, \texttt{await g;s}) &= [\![mte(\texttt{g})]\!]_\tau \\
mte(\tau, f, \texttt{s}) &= \infty \text{ if } \texttt{s} \neq \texttt{await g;}\tilde{\texttt{s}} & mte(\texttt{duration(e}_1\texttt{,e}_2\texttt{)}) &= \texttt{e}_1 \\
mte^F_\sigma(\texttt{diff e}) &= \underset{t \geq 0}{\mathbf{argmin}}\ ([\![\texttt{e}]\!]^{F,t}_\sigma = \text{true}) & mte(\texttt{e?}) &= \infty \\
adv(cn_1\ cn_2, t) &= adv(cn_1, t)\ adv(cn_2, t) \\
adv(msg, F, t) &= msg & adv(fut, F, t) &= fut \\
adv((o, \rho, ODE, F, prc, q), F, t) &= (o, adv(\rho, t), ODE, F, adv(prc, F, t), adv(q, F, t)) \\
adv(\bot, F, t) &= \bot \\
adv((\tau, f, \texttt{s}), F, t) &= (\tau, f, \texttt{s}) \text{ if } \texttt{s} \neq \texttt{await duration(e}_1\texttt{,e}_2\texttt{);}\tilde{\texttt{s}} \\
adv((\tau, f, \texttt{await duration(e}_1\texttt{,e}_2\texttt{);s}), F, t) &= (\tau, f, \texttt{await duration(e}_1\texttt{+}t\texttt{,e}_2\texttt{+}t\texttt{);s}) \\
adv(\sigma, t)(\texttt{f}) &= \begin{cases} \sigma(\texttt{f}) & \text{if f is not physical} \\ v & \text{if } \forall f \in F.\ v = f(t)(\texttt{f}) \end{cases}
\end{aligned}
$$

Figure 12 Auxiliary functions. Lifting to lists is not shown.

The *adv* auxiliary function handles uniqueness w.r.t. the solutions of the ODE at the points in time where the solutions are accessed: Note that the solutions are handled as a set F: at time t function *adv* checks that all solutions coincide *at this point in time*. If this is not the case, or if no solution can be found by the implementation, a runtime error is thrown. Also, all solutions are computed without restrictions on the time domain (e.g., for how long they exits) because it is not known for how long the dynamics are followed at this point. Alternatively, one could either impose restrictions on the ODE to enforce uniqueness or non-deterministically choose one of the solutions.

We can now define *traces* of programs and objects.

▶ **Definition 4** (Traces). Given a program Prgm, we denote with clock(0) cn_0 the initial state configuration [16]. A run of Prgm is a (possibly infinite) reduction sequence

$$\mathsf{clock}(0)\ cn_0 \to \mathsf{clock}(t_1)\ cn_1 \to \cdots$$

The trace θ_o of an object o in a run is an assignment from the dense time domain $\mathbb{R}^+$ to states. We say that clock(t_i) cn_i is the final configuration at t_i in a run, if any other timed configuration clock(t_i) $\tilde{cn}_i$ is before it. Fig. 13 gives a formal definition.

For any point in time x, the state of o is taken from the run, if a reduction step was made at x and o was already created. The third case in the definition is illustrated in Fig. 14: At time points y and z, discrete steps are done, but none at x. The state $\theta_o(x)$ is extrapolated from the state $\theta_o(y)$ by following solutions from the last step at point y, if o is created.

$$\theta_o(x) = \begin{cases} \mathit{undefined} & \text{if } o \text{ is not created yet} \\ \rho & \text{if } \mathsf{clock}(x)\ cn \text{ is the final configuration at } x \\ & \text{and } \rho \text{ is the state of } o \text{ in } cn \\ \mathit{adv}(\rho, F, x - y) & \text{if there is no configuration at } \mathsf{clock}(x) \\ & \text{and the last configuration was at } \mathsf{clock}(y) \\ & \text{with state } \rho \text{ and solutions } F \end{cases}$$

Figure 13 Extraction of a trace θ_o for an object o from a given run.

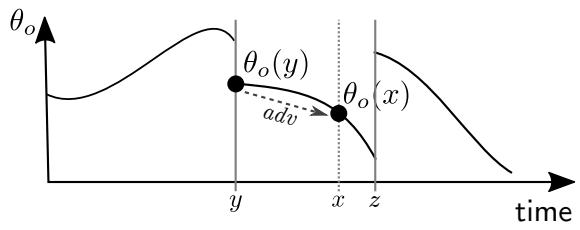


Figure 14 Illustration of the state at time x and two discrete states with $\mathsf{clock}(y)$ and $\mathsf{clock}(z)$.

3.3 The Component Fragment

We define a sublanguage of HABS called *Component HABS* (CHABS) to model component-style architectures with in- and out-ports, as well as dedicated controllers with a read-evaluate-write cycle. Syntactically, a class is a *component* if it can be derived from the syntax in Fig. 8 with the rule for Met replaced by the following:

```
  Met ::= MS [OPort | IPort | Ctrl]
OPort ::= {return this.f;}      IPort ::= {this.f = v; return Unit;}
 Ctrl ::= {sa; si; sc; so; this.m();}
   sa ::= await duration(e,e) | await diff e
   si ::= this.f = e.m() | si;si
   sc ::= while (e) {sc} | if (e) {sc} [else {sc}]? | sc;sc | T? e = e | e!m(ē)
   so ::= e!m(this.f) | so;so
```

Additionally, we demand that the only numerical data types used are `Int`, `Real`. Out-ports return the value of a field and in-ports copy a method parameter into a field. A controller method Ctrl has a timed or differential guard sa, followed by reads si from the out-port methods of other objects (recall that `this.f = e.m()` is a shortcut for an asynchronous call followed by a read, not a synchronous call), computations sc, and writes so to the in-ports of other objects. In the component fragment, we realize a component-based controller with a read-compute-write loop by restricting the `run` method of Fig. 8 to start a controller with an asynchronous call to an object's own controller method Ctrl and each controller ends with a recursive call to itself. The **TankMono** and **TankTick** models are CHABS models, the central controller in **TankMulti** is not. A controller method with a differential guard is an event-triggered controller, a controller with a timed guard a time-triggered controller.

We model instantaneous controllers in CHABS: once controller is scheduled (i.e., after its guards evaluates to true) no time can pass because all calls in Ctrl are to port methods that cannot block the caller and neither suspensions nor future reads are allowed.

3.4 Simulation

The implementation of HABS extends the ABS compiler [81] to compute solutions for differential guards, time elapse, and state advance. To compile differential guards correctly, it needs to compute mte_σ^F(**diff** e) (Def. 3).

The ODEs of a class cannot be changed at runtime and are, therefore, represented as a string in the class table. The simulator uses an external solver to solve initial value problems and minimize/maximize duration between events.

Solutions To compute solutions F, the ODEs and the current state of the physical fields are passed to Maxima [61] as an *initial value problem*. The solution is an equation system or an error. In its default setting, the simulator neither supports non-unique solutions nor non-solvable ODEs. The simulator, however, has the infrastructure to use solvers other than Maxima. This allows us to handle non-linear ODEs: by prefixing the **physical** block with `[1]`, the modeler can select the solver `ic1` (instead of the default `desolve`), which can handle non-linear systems.

Time elapse After solving the initial value problem, Maxima is invoked with a *minimization problem*: it minimizes the time t with the equation system representing F as the constraints (this corresponds to eager mode switching in a hybrid automaton). The result is then handled in the same way as a parameter to a timed guard by the runtime system. Once time has passed and the suspended process is reactivated, the physical fields are updated according to F. This uses the Maxima function `fmin_cobyla`.

State advance To implement the advance function *adv*, if the state of the object changes any physical field, the procedure used to compute time elapse is repeated for every currently suspended differential guard to accumulate the result.

The output files used to visualize a program execution are of the form $t_1, F_1, t_1, F_2, t_2, \ldots, F_n, t_n$. Here t_i are the points in time where the object schedules a process and F_i the function describing its physical behavior in the previous suspended state. Each time a differential guard is reactivated, not only its state is updated, but the solution F_{i+1} and the reactivation time t_{i+1} are written to the output. Each object has its own output file.

A Python script translates output files into a discrete dynamic graph in Maxima format which in turn calls gnuplot that is responsible for creating the graph. The graphs in this work are slightly beautified outputs.

4 Modeling with HABS

We give more examples of HABS models and discuss some design decisions in the language, as well as modeling patterns in HABS for common phenomena in hybrid system control.

4.1 Non-Linear Dynamics

HABS can handle non-linear ODEs and non-linear dynamics to the extent the backends support it. For an example, consider a resistor attached to an alternating current source that produces a sine-formed current. This is described by the class in Fig 15.

We use the non-linear solver of Maxima (by annotating `[1]`). This solver requires the input to satisfy certain syntax constraints, which entail the slightly awkward specification `r' = 0*t`. We must give an explicit ODE for each non-constant variable for KeYmaera X and as HABS requires an autonomous system, we add a clock variable `time` to express sine and cosine.

The example has a `run` method that illustrates validation. We check whether our simple model is in fact a resistor and adheres to the law $R = I/V$: Even before visualization, we can use simple command line output to check I/V by sampling every 1 second. The output for an instance

```
class Resistor(Real init) {
  [1] physical {
    /* format expected by Maxima */
    Real t = 0:    t' = 1;
    Real r = init: r' = 0*t;
    Real i = 0:    i' = cos(t);
    Real v = 0:    v' = r*cos(t);
  }
  Unit run() {
    await duration(1,1);
    println("step: " + toString(now()) +
      " with " + toString(v/i));
    if (timeValue(now()) < 60) this!run();
  }
}
```

```
step: Time(1) with 5
step: Time(2) with 4286450913523623 /
    ↪ 857290182704725
step: Time(3) with 1319812111494398 /
    ↪ 263962422298881
step: Time(4) with 1313376056981147 /
    ↪ 262675211396229
step: Time(5) with 295788950328081 /
    ↪ 59157790065616
step: Time(6) with 723097187038613 /
    ↪ 144619437407721
step: Time(7) with 758118670875062 /
    ↪ 151623734175013
step: Time(8) with 5
step: Time(9) with 5
...
```

Figure 15 A resistor attached to an AC-circuit and its sine-formed current.

`Resistor(5)` is shown in Fig. 15, where `Time(n)` is the symbolic time at the point of time when `now()` is evaluated. In the example this corresponds to seconds. As a next step, we can use the visualization to observe longer trends in Fig. 16, again for a `Resistor(5)`.

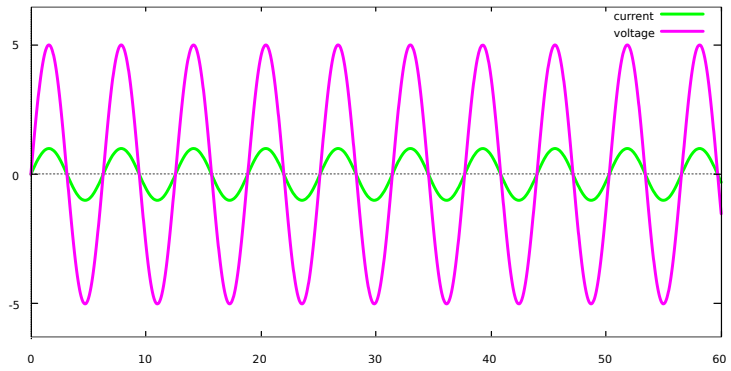


Figure 16 Example simulation output of a `Resistor(5)`.

Finally, we can formally verify the behavior with our translation approach to KeYmaera X by removing the `run` method and, thus, transforming it into a `CHABS` component.

4.2 Delays and Imprecision

Communication is imperfect in realistic models. We demonstrate how to model two such imperfections, delays and imprecision, in `HABS`. We use a simple platooning example, where a follower car wants to follow a lead car at a certain distance. Follower cars are modeled in the `CHABS` class `FollowerCar` in Fig. 17. For simplicity, the minimal (`minDist`) and maximal distance (`maxDist`) to the lead car are independent of the speed and the controller sampling frequency, which means the follower car will not provably stay in the desired distance interval. The time consuming statement **await duration** can be used to model two kinds of delays:

1. Complex computations that take some time to finish.
2. Latency: By adding a time consuming statement as the last statement of a method before the return, one can model delays in a network.

```
class FollowerCar
  (Real inita, Real start,
   Real tick, Real minDist,
   Real maxDist, ICar leadCar)
   implements ICar {
     Real next = start + minDist;
     physical {
       Real a = inita : a' = 0;
       Real v = 0     : v' = a;
       Real x = start : x' = v;
     }
     Unit run() {
       this!ctrlObserve();
     }
     Unit ctrlObserve() {
       await duration(tick, tick);
       next = leadCar.getPosition();
       if(next - x <= minDist) a = a/2;
       if(next - x >= maxDist) a = a*2;
       this.ctrlObserve();
     }

     Real getPosition() {
       return x;
     }
   }
```

Figure 17 Simple platooning example for a follower car following safely behind a lead car

For example, we extend `getPosition()` in `FollowerCar` to model sensing latency as follows:

```
Real getPosition() {
  Real oldVal = x;
  await duration(1/10, 1/10);
  return oldVal;
}
```

Like `ABS`, `HABS` has access to a (uniformly distributed) random number generator. There are functions to generate other statistical distributions. This allows to model imprecision/uncertainty. The following method adapts `getPosition()` to model sensor uncertainty:

```
Real getPosition() {
  Real imp = (random(11) + 95)/100; // number between 0.95 and 1.05
  return this.level * imp;
}
```

4.3 Variability Modeling

One of the main advantages of using a mature programming language as a host for hybrid behavior is that we can use its structuring elements and concepts: `HABS` inherits the module system with import/export clauses[5], as well as the delta-oriented [73], feature-oriented [14] *product line* [8, 74] (DFPL) mechanisms of `ABS` [25] to model variability.

DFPLs define not a single model, but a set of models which are variants of each other. From a given *core* model, so-called code *deltas* define variants based on syntactic operations: removal, modification and addition of classes, methods and fields. A variant is obtained from the core model by applying modifications specified by the deltas to it.

To determine the relevant deltas, each delta has a set of features that activate its application. A feature of a variant corresponds roughly to one implemented feature of the modified model. A set of features is called a *product*. After selecting a product, the corresponding deltas are computed and applied, resulting in an `HABS` model without variability.

[5] Omitted from the language syntax in Sec. 3 for brevity.

```
delta Delay;
modifies class Cars.FollowerCar {
  modifies Real getPosition() {
    Real old = original();
    await duration(1/10,1/10);
    return old;
  }
}
delta Imprecision;
modifies class Cars.FollowerCar {
  modifies Real getPosition() {
    return original()*(random(11)+95)/100;

  }
}
```

```
delta CruiseControl;
modifies class Cars.FollowerCar {
  adds Real ccTick = this.tick*2;
  adds Unit cruise() {
    await duration(ccTick, ccTick);
    if ((v >= 5 || v <= 0) && a != 0)
  {
      a = 0;
    }
    this.cruise();
  }
  modifies Unit run() {
    original();
    this!cruise();
  }
}
```

```
productline PL1;
features FDelay, FImprecision, FCruiseControl;
delta CruiseControl when FCruiseControl;
delta Delay when FDelay;
delta Imprecision after Delay when FImprecision;
```

Figure 18 Product line based on Fig. 17 for variability in position readings and cruise control.

We refrain from introducing the whole variability layer of ABS and refer to [25] for a detailed and formal introduction. Instead, we use the platooning example in Fig. 17 to demonstrate variability modeling in practice. The changes for imprecision and delay, as well as adding a cruise control system can be modeled as a product line. This allows to select the appropriate car product for a concrete system, as summarized in Fig. 18. The product line consists of three deltas (`Delay`, `Imprecision` and `CruiseControl`), three features (`FDelay`, `FImprecision` and `FCruiseControl`) and a knowledge base that defines which features select which delta (**delta** `D` **when** `F`) and in which order deltas are applied if they modify the same method (**delta** `D` **after** `D2`).

The delta `Delay` modifies class `Cars.FollowerCar`[6] and its method `getPosition()`. The modified method first calls the existing variant of the method via **original** and then waits before returning the value. Delta `Imprecision` is similar. Both deltas modify the same method. There are numerous desirable properties, and to make the product line outcome deterministic, we must fix the order in which methods are applied that modify the same method. Here, we demand that `Imprecision` is applied after `Delay`. Delta `CruiseControl` adds a field and method implementing a simple cruise control system. Deltas may also remove methods and fields (not shown here). In our example we represent each delta as a feature, and so any product that refers to a feature invokes its assigned delta. The deltas are applied *syntactically* before type checking. As a result, a standard HABS program is created. For example the product {`FDelay`} results in the code below.

```
class FollowerCar (...) implements ICar {
  ... // as above
  Real getPosition_core() { return x; }
  Real getPosition() { return this.getPosition_core()*(random(11) + 95)/100; }
}
```

[6] `Cars` is the module.

5 Formal Verification of HABS Models

As a prerequisite for formal verification of HABS, we briefly review *differential dynamic logic* ($d\mathcal{L}$) [68, 69] as implemented in the hybrid systems theorem prover KeYmaera X [36]. We then discuss translation from HABS to $d\mathcal{L}$, and sketch formal verification in $d\mathcal{L}$ with sequent proofs.

5.1 Background: Differential Dynamic Logic

Differential dynamic logic expresses the combined discrete and continuous dynamics of hybrid systems in a sequential imperative programming language called *hybrid programs*. Its syntax and informal semantics are in Table 1.

Table 1 Hybrid programs in $d\mathcal{L}$.

Program	Informal semantics
$?\varphi$	Test whether formula φ is true, abort if false
$x := \theta$	Assign value of term θ to variable x
$x := *$	Assign any (real) value to variable x
$\{x' = \theta \ \& \ H\}$	Evolve ODE system $x' = \theta$ for any duration $t \geq 0$ with evolution domain constraint H true throughout
$\alpha; \beta$	Run α followed by β on resulting state(s)
$\alpha \cup \beta$	Run either α or β non-deterministically
α^*	Repeat α n times, for any $n \in \mathbb{N}$

Hybrid programs provide the usual discrete statements: assignment ($x := \theta$), non-deterministic assignment ($x := *$), test ($?\varphi$), non-deterministic choice ($\alpha \cup \beta$), sequential composition ($\alpha; \beta$), and non-deterministic repetition (α^*). A typical modeling pattern combines non-deterministic assignment and test (e.g., "$x := *; ?H$") to choose any value subject to a $d\mathcal{L}$ constraint H. Standard control structures are expressible, for example:

(i) **if** H **then** α **else** $\beta \equiv (?H; \alpha) \cup (?\neg H; \beta)$,
(ii) **if** H **then** $\alpha \equiv (?H; \alpha) \cup (?\neg H)$,
(iii) **while** (H) $\alpha \equiv (?H; \alpha)^*; ?\neg H$.

For continuous dynamics, the notation $\{x' = \theta \,\&\, H\}$ represents an ODE system (derivative x' in time) of the form $x_1' = \theta_1, \ldots, x_n' = \theta_n$. Any behavior described by the ODE stays inside the evolution domain H, i.e. the ODE is followed for a non-deterministic, non-negative period of time, but stops before H becomes false. For example, a basic model of the water level x in a tank draining with flow $-f$ is given by the ODE $\{x' = -f \,\&\, x \geq 0\}$, where the evolution domain constraint $x \geq 0$ means the tank will not drain to negative water levels. With a careful modeling pattern, ODEs can be governed by H so that one can react to events, without restricting or influencing the continuous dynamics modeled in the ODE [72]: The pattern $\{x' = \theta \,\&\, H\} \cup \{x' = \theta \,\&\, \widetilde{H}\}$ permits control intervention to achieve different behavior triggered by an event H. $\widetilde{H}$ is the *weak* complement of H: they share exactly their *boundary* from which both behaviors are possible. For example, $H \equiv x \leq 0$, $\widetilde{H} \equiv x \geq 0$.

The $d\mathcal{L}$-formulas φ, ψ relevant for this paper are propositional logic operators $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\neg\varphi$ and comparison expressions $\theta \sim \eta$, where $\sim \in \{<, \leq, =, \neq, \geq, >\}$ and θ, η are real-valued terms over $\{+, -, \cdot, /\}$. In addition, there is the $d\mathcal{L}$ modal operator $[\alpha]\varphi$. The $d\mathcal{L}$-formula $[\alpha]\varphi$ is true iff φ holds in all states reachable by program α. The formal semantics of $d\mathcal{L}$ [68, 69] is a Kripke semantics in which the states of the Kripke model are the states of the hybrid system. The semantics of a hybrid program α is a relation $[\![\alpha]\!]$ between its initial and final states. Specifically, $\nu \models [\alpha]\varphi$ iff $\omega \models \varphi$ for all states $(\nu, \omega) \in [\![\alpha]\!]$, so all runs of α from ν are safe relative to φ.

Proofs in $d\mathcal{L}$ are sequent calculus proofs on the basis of $d\mathcal{L}$ axioms. For example, validity of the $d\mathcal{L}$ formula $x \geq 0 \rightarrow [x := x + 1 \cup x := 2; \{x' = 3\}]x \geq 1$ over a simple program that either increments the value of x or continuously evolves x with a constant slope $x' = 3$ after setting the initial value of the differential equation with $x := 2$ is shown in the sequent proof below:

$$\dfrac{\dfrac{\dfrac{*}{x \geq 0 \vdash x + 1 \geq 1}\,\text{QE}}{x \geq 0 \vdash [x := x + 1]x \geq 1}\,[:=] \qquad \dfrac{\dfrac{\dfrac{*}{x = 2 \vdash [\{x' = 3\}]x \geq 1}\,\text{dI}}{x \geq 0 \vdash [x := 2][\{x' = 3\}]x \geq 1}\,[:=],\text{hideL}}{x \geq 0 \vdash [x := 2; \{x' = 3\}]x \geq 1}\,[;]}{x \geq 0 \vdash [x := x + 1 \cup x := 2; \{x' = 3\}]x \geq 1}\,[\cup],\wedge_R$$
$$\vdash x \geq 0 \rightarrow [x := x + 1 \cup x := 2; \{x' = 3\}]x \geq 1 \;\rightarrow_R$$

Sequent proofs proceed bottom-up but validity transfers top-down, i.e., from the subgoals above the horizontal bar, the axiom or proof rule annotated to the left of the bar implies the sequent below the horizontal bar. In each step, assumptions are listed to the left of the $\vdash$, and the alternatives to prove to the right of it. The proof starts with step $\rightarrow_R$ to make the left-hand side $x \geq 0$ of the implication available as an assumption. Next, the non-deterministic choice step $[\cup]$ means that both choices must ensure the postcondition $x \geq 1$, so with conjunction splitting $\wedge_R$ we get two subgoals: a left subgoal for the increment program $x := x + 1$ and a right subgoal for the differential equation program. On the increment program branch, we execute the assignment in step $[:=]$ and the result follows by real arithmetic in step QE. On the differential equation branch, step $[;]$ splits the sequential composition into nested box modalities, and then step $[:=], \text{hideL}$ executes the assignment and weakens the now obsolete assumption $x \geq 0$. The branch closes by differential induction dI (intuitively, the dI step expresses that $x \geq 1$ stays true along the flow of the differential equation, see [67]). This concludes the example proof.

5.2 Formal Verification of Components

```
1  interface Tank {
2   /* requires -1/2 <= newD <= 1/2; */
3   Unit inDrain(Real newD);
4   /* ensures 3 <= outLevel <= 10; */
5   Real outLevel();
6  }
7
8  class CTank(Real inVal)
9       implements Tank {
10  /* requires newD > 0 -> level <= 9.5  */
11  /* requires newD < 0 -> level >= 3.5 */
12  /* timed_requires inDrain < 1 */
13  Unit inDrain(Real newD) { ... }
14   ...
15 }
16
17 /* requires 0 < tick < 1 & inVal > 3.5*/
18 class FlowCtrl(Tank t, Real tick,
19                Real inVal) {
20 /* invariant (drain > 0 -> level <= 9.5)
21         & (drain < 0 -> level >= 3.5) */
22 ...
23 }
```

Figure 19 Annotations in the TankTick model, repeated from Fig. 4.

To establish system-wide properties, hybrid active objects must be shown to satisfy their class *invariants*, provided that the constraints expressed in the preconditions are met. We make this precise now. A *class specification* is a tuple $(\mathsf{inv}, \mathsf{pre}, \mathsf{TReq}, \mathsf{Req}, \mathsf{Ens})$, where inv is the class invariant

(annotated */∗ invariant ... ∗/*, see Fig. 19, lines 20–21), a $d\mathcal{L}$ formula over the fields and parameters of the class; pre is the precondition (annotated to class declarations with */∗ requires ... ∗/*, see Fig. 19, line 17), a $d\mathcal{L}$ formula over the initial values of fields and class parameters. TReq is the set of timed input requirements for in-port methods (annotated with */∗ timed_requires ... ∗/*, see Fig. 19, line 12): $d\mathcal{L}$ formulas over a dedicated program variable with the method's name. Req is the set of input requirements for in-port methods (annotated with */∗ requires ... ∗/*, see Fig. 19, lines 2, 10–11): $d\mathcal{L}$ formulas over fields and method parameters. Ens is the set of output guarantees for out-port methods (annotated with */∗ ensures ... ∗/*, see Fig. 19, line 4): $d\mathcal{L}$ formulas over a dedicated program variable with the method's name.

To verify a class C against a class specification, both are translated into $d\mathcal{L}$-formula (1) that expresses safety.

$$\texttt{assumptions}_{\texttt{C}} \rightarrow \left[(\texttt{code}_{\texttt{C}};\ \texttt{plant}_{\texttt{C}})^*\right] \texttt{safety}_{\texttt{C}} \tag{1}$$

The placeholders $\texttt{assumptions}_{\texttt{C}}$, $\texttt{code}_{\texttt{C}}$, $\texttt{plant}_{\texttt{C}}$, and $\texttt{safety}_{\texttt{C}}$ (defined formally in Sect. 5.3 below) encode class C and its specification (inv, pre, TReq, Req, Ens) as follows: The formula $\texttt{assumptions}_{\texttt{C}}$ is the conjunction of pre and conditions on variables that keep track of time. As usual in controller verification, the program repeats a control part $\texttt{code}_{\texttt{C}}$ followed by the continuous behavior $\texttt{plant}_{\texttt{C}}$. The condition $\texttt{safety}_{\texttt{C}}$ must hold after an arbitrary number of iterations. It combines inv with input requirements of in-port methods of referred objects and guarantees of own out-port methods.

Even though formula (1) $\texttt{safety}_{\texttt{C}}$ is a postcondition that must hold only in the final states of the system, we stress that this means at *every real time point* during the continuous dynamics, because ODEs advance for a non-deterministic duration while discrete statements take no time. The modality, therefore, expresses that whenever $\texttt{code}_{\texttt{C}}$ executes completely, the invariant holds. In particular, the invariant holds at the beginning of and throughout the evolution of the continuous dynamics in $\texttt{plant}_{\texttt{C}}$. Thus, validity of formula (1) expresses safety of every correctly created object (with respect to its specification).

The following translation of an HABS class and its specification defines formally how the placeholders are composed. The translation is fully automatic and verification is compositional: only classes whose code changed explicitly need re-verification, not the whole system.

5.3 Translation from CHABS to $d\mathcal{L}$

We use two operations on sets of programs P. Operation $\sum P$ constructs a program that non-deterministically executes one of the elements. Operation $\prod P$ constructs all permutations of sequential element-wise execution. Let $|P| = n$:

$$\sum P = \sum\{p_1, \ldots, p_n\} \ = \ p_1 \cup p_2 \cup \cdots \cup p_n$$
$$\prod P = \{p_1; \ldots; p_n \mid \forall i, j \leq n.\ p_i, p_j \in P \wedge (i \neq j \rightarrow p_i \neq p_j)\}$$

We translate classes C with the following design restrictions:

(1) All controllers update their local caches of other objects before providing information to those objects (for example, read the current water level before instructing the tank to drain or fill); local caches, once updated, are not modified later.

(2) In-port methods with a timed input requirement are only called from timed controllers (for example, a tank that expects to be filled every $5\,s$ is governed by a controller running at a corresponding frequency).

(3) Duration statements are exact (have two identical parameters).

(4) Local variable names are unique.

$$
\left.\begin{array}{r}
\mathsf{trans}(\texttt{f}) \equiv f \text{ , where } f \text{ is a } d\mathcal{L} \text{ variable representing field } \texttt{f} \\
\mathsf{trans}(\texttt{v}) \equiv v \text{ , where } v \text{ is a } d\mathcal{L} \text{ variable representing variable } \texttt{v} \\
\mathsf{trans}(\texttt{e}_1\ op\ \texttt{e}_2) \equiv \mathsf{trans}(\texttt{e}_1)\ op\ \mathsf{trans}(\texttt{e}_2)
\end{array}\right\} \text{expressions } \texttt{e}
$$

$$
\left.\begin{array}{r}
\mathsf{trans}(\textbf{if}(\texttt{e})\{\texttt{s}\}[\textbf{else}\ \{\texttt{s}\}]) \equiv \textbf{if}\ (\mathsf{trans}(\texttt{e}))\ \textbf{then}\ \mathsf{trans}(\texttt{s})[\textbf{else}\ \mathsf{trans}(\texttt{s})] \\
\mathsf{trans}(\textbf{while}(\texttt{e})\{\texttt{s}\}) \equiv \textbf{while}(\mathsf{trans}(\texttt{e}))\mathsf{trans}(\texttt{s}) \qquad \mathsf{trans}(\texttt{s}_1\texttt{;}\texttt{s}_2) = \mathsf{trans}(\texttt{s}_1)\texttt{;}\mathsf{trans}(\texttt{s}_2) \\
\mathsf{trans}(\texttt{[T] v = e}) \equiv \mathsf{trans}(\texttt{v}) := \mathsf{trans}(\texttt{e}) \qquad \mathsf{trans}(\texttt{f = e}) \equiv \mathsf{trans}(\texttt{f}) := \mathsf{trans}(\texttt{e}) \\
\mathsf{trans}(\texttt{e!m()}) \equiv ?\mathit{true} \qquad \mathsf{trans}(\texttt{f = e.m()}) \equiv \mathsf{trans}(\texttt{f}) := *\texttt{;}?\varphi_{\texttt{m}} \\
\text{where } \varphi_{\texttt{m}} \text{ is the postcondition of } \texttt{m}\text{, with the method name replaced by } \mathsf{trans}(\texttt{f})
\end{array}\right\} \text{statements } \texttt{s}
$$

Figure 20 Translation of expressions `e` and statements `s`.

The first two constraints fix the interaction pattern between components, the last two simplify the presentation. For classes following these restrictions, the translation has four phases, each discussed in detail in subsequent paragraphs:

(i) provision of program variables,
(ii) generation of assumptions and safety condition,
(iii) control code generation,
(iv) provision of ODEs and constraints.

5.3.1 Program Variables

For each field, parameter, and local variable in `C` we create a program variable with the same name. For each method `m` we create a time variable $t_{\texttt{m}}$, for each in-port method `m` a tick variable $\mathit{tick}_{\texttt{m}}$, both type **Real**; $\mathit{tick}_{\texttt{m}}$ models the unknown time when an in-port method is *called* next. Time variables are local time for each method and determine when a time-triggered controller or an in-port is *executed* the next time. We denote the set of all tick variables with Tick and the set of all time variables with Time.

5.3.2 Assumptions and Safety Condition

The formula $\texttt{assumptions}_{\texttt{C}}$ (2) is `C`'s precondition pre plus all initializations init plus conditions on the time and tick variables: in the beginning, each time variable starts at zero and the tick variables have an unknown positive value. Each tick variable *tick* has a method $\texttt{m}_{\mathit{tick}}$ that is responsible for its generation. We refer to the timed input requirement of this method with $\psi(\mathit{tick})$, where the method name $\texttt{m}_{\mathit{tick}}$ has been replaced with *tick*. The initial value of the tick variable is also described by the timed input requirement and describes when the method is issued for the first time at the latest.

$$
\texttt{assumptions}_{\texttt{C}} \equiv \mathsf{pre} \wedge \bigwedge_{\varphi \in \mathsf{init}} \varphi \wedge \bigwedge_{t \in \mathrm{Time}} t \doteq 0 \wedge \bigwedge_{\mathit{tick} \in \mathrm{Tick}} \left(0 < \mathit{tick} \wedge \psi(\mathit{tick})\right) \tag{2}
$$

The formula $\texttt{safety}_{\texttt{C}}$ (3) captures the guarantees of class `C`: we need to show that `C`

(i) preserves its own invariant inv;
(ii) provides guarantees Ens about own out-port methods (shows what others can rely on);
(iii) respects timed preconditions TReq^s; and,
(iv) when writing to in-port methods of callees, respects their input requirements Req^s.

If class `C` comes with a time-triggered controller with guard **duration**`(e,e)`, technical constraint 5.3(1) above ensures that at the moment the controller calls an in-port of another object, it has a correct copy of the callee state. Req^s are input requirements of used in-port methods of

other classes than C, where the method parameter is replaced by the field passed to it. Ens are guarantees of all out-port methods of C. Some special care needs to be taken for timed input requirements. With TReq^s, we denote the set of timed input requirements (constructed over *tick*, as above) of all called in-ports where such a clause is given.

$$\mathtt{safety}_\mathtt{C} \equiv \mathsf{inv} \wedge \bigwedge_{\varphi \in \mathsf{Req}^s} \varphi \wedge \bigwedge_{\tau \in \mathsf{TReq}^s} \tau \wedge \bigwedge_{\psi \in \mathsf{Ens}} \psi \tag{3}$$

The safety condition expresses that the controllers of class C respect the input requirements when writing to the in-port methods *of other components* and call in-port methods with a timed input requirement sufficiently open. The structure of controllers in CHABS per Sect. 3.3 enforces that these calls occur last in the controller bodies.

5.3.3 Control Code

The translation of ABS statements to hybrid programs is defined in Fig. 20. We discuss the non-obvious rules: Calls `e!m()` to in-port methods of other objects are mapped to $?true$ (i.e. skip), because there is no effect on the caller object. A read `f=e.m()` from an out-port method is mapped to $\mathsf{trans}(f) := *; ?\varphi_\mathtt{m}$: a non-deterministic assignment, restricted with a subsequent test for the guarantee of the called out-port method.

The translation of ports and control methods has the *general form* **if** (check) **then** {exec; cleanup}. This pattern is instantiated per method type as follows:

- Time-triggered controller `m` with method body **await duration**`(e,e); s; this.m()`: check makes sure the correct duration elapsed and cleanup resets time, so $\mathsf{check} \equiv t_\mathtt{m} \doteq \mathsf{trans}(\mathtt{e})$, $\mathsf{exec} \equiv \mathsf{trans}(\mathtt{s})$, $\mathsf{cleanup} \equiv t_\mathtt{m} := 0$.
- Event-triggered controller `m` with body **await diff** `e; s; this.m()`: check tests the guard, so $\mathsf{check} \equiv \mathsf{trans}(\mathtt{e})$, $\mathsf{exec} \equiv \mathsf{trans}(\mathtt{s})$, $\mathsf{cleanup} \equiv ?true$.
- In-port method `m` with body **this**`.f = v`, input requirement φ and timed input requirement ψ: check ensures the correct duration elapsed, so $\mathsf{check} \equiv t_\mathtt{m} \doteq tick_\mathtt{m}$; exec chooses a value consistent with φ, so $\mathsf{exec} \equiv f := *; ?\varphi$; finally, cleanup does the same for a new duration consistent with ψ (method name replaced by $tick_\mathtt{m}$), so $\mathsf{cleanup} \equiv tick_\mathtt{m} := *; ?tick_\mathtt{m} > 0; ?\psi; t_\mathtt{m} := 0$.
- Out-port methods and the `run` method are not translated. Out-port methods have no effect on object state and their guarantees (included in (1) in $\mathtt{safety}_\mathtt{C}$) must be shown to hold throughout plant execution. The `run` method initializes the system and ensures that every controller can run once before the first plant execution, which is guaranteed in (1) through sequential composition of $\mathtt{code}_\mathtt{C}$; $\mathtt{plant}_\mathtt{C}$.

Let M be the set of all translations of in-port methods and controllers, then:

$$\mathtt{code}_\mathtt{C} \equiv \left(\sum \prod M\right); \left(\sum M\right)^* \tag{4}$$

The controller $\mathtt{code}_\mathtt{C}$ first executes all controllers in a non-deterministically chosen order $(\sum \prod M)$, then allows each controller/in-port to repeat $(\sum M)^*$. The latter replicates eager ABS behavior on satisfied guards: when an event-triggered controller is triggered and its guard still holds after its execution, then in ABS the controller is run again.

Note that $(\sum M)^*$ safely overapproximates all possible orders, including the behavior of the first part $\sum \prod M$. However, including $\sum \prod M$ in $\mathtt{code}_\mathtt{C}$ simplifies practical proofs, because in typical models that disable the check guards at the end of control and in-port method bodies (e.g., a time-triggered controller that resets time in cleanup so that it becomes re-enabled only after some time passes), every method is executed at most once before time advances. The structure of the controller $\mathtt{code}_\mathtt{C}$ mirrors this with the first part $\sum \prod M$ to simplify practical proofs as follows:

(i) the proof obligations of enabled control and in-port methods (i.e., whose check is true) are easier because the outer loop is dropped, and additionally the proof obligations of all the disabled control and in-port methods can be easily disposed of by contradiction with their check guards;

(ii) finding a loop invariant for the second part $(\sum M)^*$ is easy when no method is executed twice before time advances: in that case, the loop invariant for $(\sum M)^*$ must simply imply that none of the check guards holds.

Further note that $\sum\prod M$ does not exclude runs, because the general form **if** (check) **then** {exec; cleanup} of control methods and ports in M ensures that there is progress through the implicit **else** ?*true* even if all controllers and in-ports are disabled.

5.3.4 Plant

The plant of a class C has the form

$$\texttt{plant}_{\texttt{C}} \equiv \sum\{(\texttt{ode},\ \texttt{ode}_t \,\&\, c) \mid c \in \mathcal{C}\} \ , \tag{5}$$

where ode is the ODE from its physical block, $\texttt{ode}_t$ describes the time variables, and the constraints $c \in \mathcal{C}$ partition the domain of the physical fields. The boundaries of the subdomains overlap exactly where the differential guards hold.[7] This models guards as events in $d\mathcal{L}$, following the modeling pattern described in Sect. 5.1. To ensure that no differential guard is omitted, it is necessary that no two differential guards share a program variable. This is not a restriction, as two controllers can be merged with a disjunction: see the guard in Fig. 2.

To define $\mathcal{C}$ let $e_1, \ldots, e_m$ be the translations of differential guards in the class and $\tilde{e}_i$ the weak complement of e_i. Let $t_1, \ldots, t_l$ be all time variables introduced for time-triggered controllers with e_{t_i} the expression in the **duration** statement. Let $pt_1, \ldots, pt_k$ be all time variables introduced for in-port methods and $tick_{pt_i}$ the associated tick variable. We set $\texttt{ode}_t \equiv \{t_1' = 1, \ldots, t_l' = 1, pt_1' = 1, \ldots, pt_k' = 1\}$ and define:

$$\begin{aligned}\mathcal{C} \equiv\ & \big(\{e_1, \tilde{e}_1\} \times \{e_2, \tilde{e}_2\} \times \cdots \times \{e_m, \tilde{e}_m\}\big) \\ & \cup \{t_1 \le e_{t_i}\}_{i \le l} \cup \{t_1 \ge e_{t_i}\}_{i \le l} \cup \{pt_i \le tick_{pt_i}\}_{i \le n} \cup \{pt_i \ge tick_{pt_i}\}_{i \le n}\end{aligned}$$

5.3.5 On the Random Number Generator

We do not translate the `random(i)` expression from HABS to $d\mathcal{L}$, because its semantics is that it returns an *integer* below `i`. However, integer arithmetic is undecidable, which is the reason why $d\mathcal{L}$ opts to embed its modality into a decidable first-order logic over the reals [66]. A straightforward overapproximation with a translation to a variation of `random` that returns a real value is:

$$\texttt{trans}\big(\texttt{f = random(r)}\big) \equiv \texttt{trans(f)} := *; ?\big(0 \le \texttt{trans(f)} < \texttt{trans(r)}\big)$$

5.4 Compositional Verification

We can now state our main theorem: If we can prove safety of all classes, i.e., close all proof obligations, then the whole system is safe, i.e., every class indeed preserves its invariant. Verification is compositional: if we change the code or invariant of one class, only the proof obligation of this class has to be reproven. If we change a method precondition, additionally the proof obligations of all calling classes have to be reproven.

[7] Expressions contain only >=, <=, so weak complement ensures a boundary overlap.

▶ **Theorem 5.** *Let* P *be a set of classes, with each* $\mathtt{C} \in \mathsf{P}$ *associated with* $\varphi_{\mathtt{C}}$ *per formula* (1)*. If all the* $\varphi_{\mathtt{C}}$ *are valid, then for every main block that creates objects satisfying* $\mathsf{pre}_{\mathtt{C}}$ *all reachable states of all objects satisfy* $\mathsf{inv}_{\mathtt{C}}$*.*

Proof Sketch. Recall that the trace of an HAO is an assignment of time to stores (Def. 4). For the proof, each store is indexed by its time and the trace starts with 0 (i.e., the possible offset caused by the delayed object creation is removed):

$$\theta_o(t) = (\rho_t)_{t \in \mathbb{R}^+} = \rho_0 \cdots$$

We are going to use that there are only countably many discrete steps in a run and partition the trace into countably many subtraces. Then we show by induction on these discrete steps that the invariant is always preserved.

Let D be the set of all time points with discrete steps of o in the run that generates θ_o. Note that $0 \in D$ and that $\theta_o(d)$ is the last store defined by the SOS semantics, if several such stores share the same time; further note that this is reflecting the reachability relation of $d\mathcal{L}$.

We define θ_o^d as the subtrace of θ_o starting with d and ending at the next time point of a discrete step. Let $\mathsf{next}(d)$ be the next time point of a discrete step after d, if such a time point exists, and ∞ otherwise:

$$\mathbf{dom}\left(\theta_o^d\right) = \left[d..\mathsf{next}(d)\right] \qquad \text{with} \qquad \theta_o^d(t) = \theta_o(t)$$

We observe that each state in the `HABS` semantics is also a state in the Kripke structure of the semantics if all class parameters are removed. We show that trans preserves reachability: if from a state ρ state ρ' is reachable by an `HABS` statement $\mathtt{s}$ in the `HABS` semantics, then state ρ' is reachable from state ρ by $\mathsf{trans}(\mathtt{s})$. This is justified as follows:

1. The $d\mathcal{L}$ program omits no events, because each event is at a boundary of two evolution domain constraints on a variable and no two events share a variable (each controller has its own time variable).
2. The evolution domain constraints cover all possible states, so no run is rejected for a domain being too small.
3. Each test in $d\mathcal{L}$ formula $\varphi_{\mathtt{C}}$ that discards runs does so using a condition that is provably guaranteed by other objects. For example, the test that discards all runs of an in-port method for inputs not satisfying its input requirements is safe, because on the caller side this condition is part of the safety condition (3).
4. The observation also relies on technical constraint (1) above and the recursive call being at the end of a controller. Together, this guarantees that *at that moment* the caller copy of the callee's state is consistent with the callee's actual state.

Let $D = (d_i)_{i \in \mathbb{N}}$ be an enumeration of the discrete time points and $\hat{\theta}_o^{d_i}$ the union of all subtraces of θ_o up to d_i:

$$\mathbf{dom}\left(\hat{\theta}_o^{d_i}\right) = \bigcup_{j \le i} \mathbf{dom}\left(\theta_o^{d_j}\right) \qquad \text{with} \qquad \hat{\theta}_o^{d_i}(t) = \theta_o(t)$$

We show by induction on i that every state in $\hat{\theta}_o^{d_i}$ is safe, i.e., a model for the invariant $\mathsf{inv}_{\mathtt{C}}$.

Induction Base: $i = 0$**.** It is explicitly checked that $\theta_o^{d_0}$ is safe. By assumption, the object is created in a state $\theta_o^{d_0}$ such that the precondition $\mathsf{pre}_{\mathtt{C}}$ holds. From axiom I of $d\mathcal{L}$ [68] we know that the safety condition must be true in the beginning of the loop, thus validity of $\varphi_{\mathtt{C}}$ implies validity of $\mathsf{pre}_{\mathtt{C}} \to \mathsf{inv}_{\mathtt{C}}$. Since all the formulas $\varphi_{\mathtt{C}}$ are proved in isolated component proofs, we conclude $\mathsf{inv}_{\mathtt{C}}$ holds for all reachable states of all objects as by the correctness argument reachability is preserved.

Induction Step. $i > 0$. This is analogous to the base case, but instead of an explicit check that d_i is safe, we use the induction hypothesis that every state in $\hat{\theta}_o^{d_{i-1}}$ is safe and that the statement for d_i is executed in a state at time $t \in \mathbf{dom}\left(\hat{\theta}_o^{d_{i-1}}\right)$. ◀

▶ Remark. The theorem states soundness of safety properties in $d\mathcal{L}$ proof obligations and does not prove semantic equivalence between the contained $d\mathcal{L}$-program and the HABS class. This approach stands in the tradition of modular deductive verification of object-oriented software, in particular, it follows the structure of systems for distributed object-oriented programs [52]. The main reason to pursue this approach is that the form of proof obligations and the translation of statements cannot be disentangled: the translation of method calls includes the postcondition of the called methods: soundness of the translation relies on the fact that all other proof obligations can be established. This is already the case for discrete, sequential languages [41]. Note that this is *not* circular. As the proof of Theorem 5 shows, we can order all method executions in a run such that we have a well-founded induction on them. The first method execution in every object relies only on the state precondition which is guarenteed at creation. These in turn are guaranteed in the main block, which has no assumptions. Another reason is that each $d\mathcal{L}$ proof obligation corresponds to the (symbolic) execution of one *object* in a class. To model all permissible evolutions of several method executions in a proof, therefore, it is necessary to encode the scheduler. This requires a form of proof obligation that assumes the object invariant (which contains scheduling constraints). This effect is well-known in deductive verification of distributed programs [31, 32, 52].

5.5 Case Study

We illustrate the HABS-to-KeYmaera X translation defined above with the **TankTick** system in Fig. 4. The example, the implementation of the translation and the simulation, as well as the mechanical proofs of the translation are available in the supplementary material.[8] We start with the two-object water tank, whose behavior for an initial level of $5\,l$ is plotted in Fig. 5.

5.5.1 Class CTank

The in-port method `inDrain()` of the `CTank` class gives rise to a time variable $t_{\texttt{inDrain}}$ and a tick variable $tick_{\texttt{inDrain}}$. Following (2), $\mathsf{assumptions}_{\texttt{Tank}}$ is:

$$\begin{aligned}\mathsf{assumptions}_{\texttt{Tank}} \equiv\ & 4 \leq \texttt{inVal} \leq 9 \\ & \wedge\, t_{\texttt{inDrain}} \doteq 0 \wedge 0 < tick_{\texttt{inDrain}} \\ & \wedge\, \texttt{level} \doteq \texttt{inVal} \wedge \texttt{drain} \doteq -1/2\end{aligned} \tag{6}$$

The safety condition says the tank level stays within its limits and that `level` adheres to its contract which happen to be identical. No in-port methods of other classes are used, hence:

$$\mathsf{safety}_{\texttt{Tank}} \equiv 3 \leq \texttt{level} \leq 10\ . \tag{7}$$

The `CTank` class has no controller method, so the `inDrain` method, which has a timed input requirement, per (4) results in $\mathsf{code}_{\texttt{Tank}}$ below

$$\mathsf{code}_{\texttt{Tank}} \equiv \mathsf{p};(\mathsf{p})^* \tag{8}$$

[8] https://doi.org/10.5281/zenodo.5973904

where $\mathsf{p} \equiv \mathsf{trans}(\texttt{inDrain})$ below is translated from Fig. 4 using the translation of Fig. 20:

$$\begin{aligned}\mathsf{p} \equiv \textbf{if } (t_{\texttt{inDrain}} \doteq \mathit{tick}_{\texttt{inDrain}}) \textbf{ then}\\ \quad &\texttt{drain} := *;\\ \quad &?{-1/2} \leq \texttt{drain} \leq 1/2 \land (\texttt{drain} < 0 \to \texttt{level} \geq 3.5)\\ &\qquad\qquad \land (\texttt{drain} > 0 \to \texttt{level} \leq 9.5);\\ \quad &\mathit{tick}_{\texttt{inDrain}} := *;\ ?0 < \mathit{tick}_{\texttt{inDrain}} < 1;\ t_{\texttt{inDrain}} := 0\end{aligned}$$

The plant $\mathsf{plant}_{\texttt{Tank}}$, following shape (5), is based on the physical block and the new clock variable (there are no differential guards), with the evolution domain constraint split along the new time variable $t_{\texttt{inDrain}}$. ODEs of the form $v' = 0$ are default and omitted.

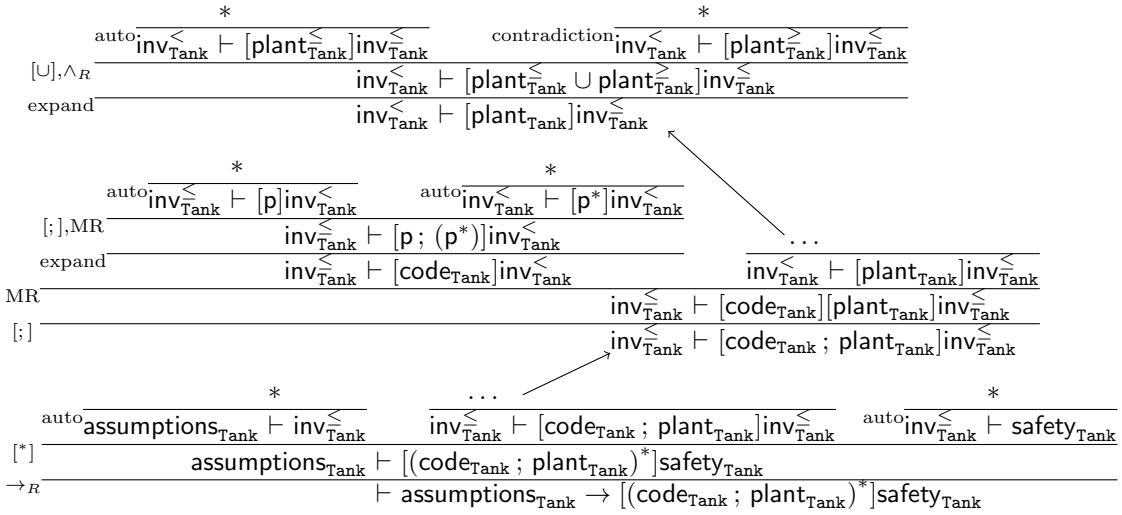
$$\begin{aligned}\mathsf{plant}_{\texttt{Tank}} &\equiv \mathsf{plant}^{\leq}_{\texttt{Tank}} \cup \mathsf{plant}^{\geq}_{\texttt{Tank}}\\ \mathsf{plant}^{\leq}_{\texttt{Tank}} &\equiv \{\texttt{level}' = \texttt{drain}, t'_{\texttt{inDrain}} = 1 \,\&\, t_{\texttt{inDrain}} \leq \mathit{tick}_{\texttt{inDrain}}\}\\ \mathsf{plant}^{\geq}_{\texttt{Tank}} &\equiv \{\texttt{level}' = \texttt{drain}, t'_{\texttt{inDrain}} = 1 \,\&\, t_{\texttt{inDrain}} \geq \mathit{tick}_{\texttt{inDrain}}\}\end{aligned} \tag{9}$$

▶ **Lemma 6.** *Class* `Tank` *is safe, i.e., formula* $\varphi_{\texttt{Tank}}$ *– obtained per* (1) *referring to tank assumptions* $\mathsf{assumptions}_{\texttt{Tank}}$ (6)*, postcondition* $\mathsf{safety}_{\texttt{Tank}}$ (7)*, code* $\mathsf{code}_{\texttt{Tank}}$ (8)*, and plant* $\mathsf{plant}_{\texttt{Tank}}$ (9) *– is valid.*

$$\varphi_{\texttt{Tank}} \equiv \mathsf{assumptions}_{\texttt{Tank}} \to \left[(\mathsf{code}_{\texttt{Tank}};\ \mathsf{plant}_{\texttt{Tank}})^*\right] \mathsf{safety}_{\texttt{Tank}}$$

Proof. See KeYmaera X proofs in the supplementary material. The proof sketch here serves as an illustration of how sequent proofs in KeYmaera X systematically use the invariant annotations in `HABS`. In the proof, we show the inductive loop invariant $\mathsf{inv}^{\leq}_{\texttt{Tank}}$, which expresses that the level always stays within limits and that the next input will be supplied before exceeding the timed input requirement as follows: $3 \leq \texttt{level} \leq 10 \land -1/2 \leq \texttt{drain} \leq 1/2 \land 3 \leq \texttt{level} + \texttt{drain}(\mathsf{tick}_{\texttt{inDrain}} - t_{\texttt{inDrain}}) \leq 10 \land \mathsf{tick}_{\texttt{inDrain}} \leq t_{\texttt{inDrain}}$.

The proof starts in step $\to_R$ to make the left-hand side $\mathsf{assumptions}_{\texttt{Tank}}$ of the implication available as assumptions. Next, $[^*]$ uses the loop invariant $\mathsf{inv}^{\leq}_{\texttt{Tank}}$ for induction: the base case in the left-most subgoal and the use case in the right-most subgoal follow by real arithmetic automation; the induction step in the middle subgoal continues with $[;]$ to split the sequential composition into nested box modalities.



The main insight now is that $\mathsf{code}_{\texttt{Tank}}$ reacts at the latest when $\mathsf{tick}_{\texttt{inDrain}} = t_{\texttt{inDrain}}$ and will reset the timer using $\mathsf{tick}_{\texttt{inDrain}} := 0$, so that the timing requirement $\mathsf{tick}_{\texttt{inDrain}} \leq t_{\texttt{inDrain}}$ can be strengthened to a strict inequality $\mathsf{tick}_{\texttt{inDrain}} < t_{\texttt{inDrain}}$ in the inductive loop invariant. The resulting intermediate condition $\mathsf{inv}^{<}_{\texttt{Tank}}$ is used in step MR to split into two subgoals: in the left subgoal of MR, we show that $\mathsf{code}_{\texttt{Tank}}$ guarantees the intermediate condition $\mathsf{inv}^{<}_{\texttt{Tank}}$. In the right subgoal of MR we show that $\mathsf{plant}_{\texttt{Tank}}$ preserves the loop invariant from that intermediate condition: the plant listens for the event $\mathsf{tick}_{\texttt{inDrain}} = t_{\texttt{inDrain}}$ with a choice between two differential equations, whose evolution domain constraints exactly overlap at the event. On evolution domain $\mathsf{tick}_{\texttt{inDrain}} \leq t_{\texttt{inDrain}}$ in $\mathsf{plant}^{\leq}_{\texttt{Tank}}$, the differential equation preserves the loop invariant, whereas on evolution domain $\mathsf{tick}_{\texttt{inDrain}} \geq t_{\texttt{inDrain}}$ in $\mathsf{plant}^{\geq}_{\texttt{Tank}}$ the contradiction shows that the controller reacts such that the plant can never enter this unsafe behavior. ◀

5.5.2 Time-Triggered Controller `FlowCtrl`

Assumptions $\mathsf{assumptions}_{\texttt{FlowCtrl}}$ of `FlowCtrl` constructed per (2) and plant $\mathsf{plant}_{\texttt{FlowCtrl}}$ constructed per (5) are straightforward. The latter is created for the sake of observing time events, even though no physical block is present:

$$\mathsf{assumptions}_{\texttt{FlowCtrl}} \equiv 0 < \texttt{tick} < 1 \tag{10}$$

$$\begin{aligned}\mathsf{plant}_{\texttt{FlowCtrl}} \equiv\ & \{t'_{\texttt{ctrlFlow}} = 1 \,\&\, t_{\texttt{ctrlFlow}} \geq \texttt{tick}\} \\ & \cup \{t'_{\texttt{ctrlFlow}} = 1 \,\&\, t_{\texttt{ctrlFlow}} \leq \texttt{tick}\}\end{aligned} \tag{11}$$

The safety condition $\mathsf{safety}_{\texttt{FlowCtrl}}$ constructed per (3) is the timed input requirement of the called `inDrain` method and the class invariant (subsumed by the input requirement of `inDrain`):

$$\begin{aligned}\mathsf{safety}_{\texttt{FlowCtrl}} \equiv\ & -1/2 \leq \texttt{drain} \leq 1/2 \land \texttt{tick} < 1 \\ & \land (\texttt{drain} < 0 \rightarrow \texttt{level} \geq 3.5) \\ & \land (\texttt{drain} > 0 \rightarrow \texttt{level} \leq 9.5)\end{aligned} \tag{12}$$

Finally, the code $\mathsf{code}_{\texttt{FlowCtrl}}$ is translated as

$$\mathsf{code}_{\texttt{FlowCtrl}} \equiv \mathsf{q}; (\mathsf{q})^* \tag{13}$$

with

$$\begin{aligned}\mathsf{q} \equiv\ & \textbf{if } (t_{\texttt{ctrlFlow}} \doteq \texttt{tick}) \textbf{ then} \\ & \texttt{level} := *;\ ?3 \leq \texttt{level} \leq 10; \\ & \textbf{if } (\texttt{level} \leq 3.5) \textbf{ then } \{\texttt{drain} := 1/2\}; \\ & \textbf{if } (\texttt{level} \geq 9.5) \textbf{ then } \{\texttt{drain} := -1/2\}; \\ & t_{\texttt{ctrlFlow}} := 0\end{aligned}$$

▶ **Lemma 7.** *Class* `FlowCtrl` *is safe, i.e., formula* $\varphi_{\texttt{FlowCtrl}}$ *– obtained per* (1) *referring to assumptions* $\mathsf{assumptions}_{\texttt{FlowCtrl}}$ (10)*, postcondition* $\mathsf{safety}_{\texttt{FlowCtrl}}$ (12)*, code* $\mathsf{code}_{\texttt{FlowCtrl}}$ (13)*, and plant* $\mathsf{plant}_{\texttt{FlowCtrl}}$ (11) *– is valid.*

$$\varphi_{\texttt{FlowCtrl}} \equiv \mathsf{assumptions}_{\texttt{FlowCtrl}} \rightarrow \left[(\mathsf{code}_{\texttt{FlowCtrl}};\ \mathsf{plant}_{\texttt{FlowCtrl}})^*\right] \mathsf{safety}_{\texttt{FlowCtrl}}$$

Proof. See KeYmaera X-proofs in the supplementary material. ◀

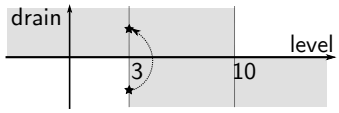


Figure 21 Avoiding Zeno-behavior in **TankMono**.

5.5.3 Event-Triggered Controller `CSingleTank`

Translation of class `CSingleTank` from Fig. 2 illustrates the handling of event-triggered controllers. The plant and code interact. The plant separates the evolution domain into two parts, with the guard of the event-triggered controller (the white areas in Fig. 21) defining their boundary. The gray areas are *larger* than the safe region defined by `3 <= level <= 10`. This is necessary to avoid Zeno behavior in the eager execution semantics of `HABS`: If we used simply the weak complement of the safe region `level <= 3 | level >= 10` as a guard and happen to be in a program state at the boundary (the lower of the states indicated with a star in Fig. 21), then the controller changes the state as shown by the arrow. But if the next state is again *on the boundary*, which is the case when the safe region is too small, then the guard is triggered, the controller loops back to the first state, etc., without physical time being able to advance. The guard in Fig. 2 ensures that after the controller has run, the state is *not* on the boundary anymore. This behavior is exhibited by our implementation, see Fig. 3. The code $\mathsf{code}_{\texttt{CSingleTank}}$ has the form $\mathsf{r};(\mathsf{r})^*$ with r being:

$$\begin{aligned}\mathsf{r} \equiv\ & \textbf{if } (\texttt{level} \leq 3 \wedge \texttt{drain} \leq 0) \vee (\texttt{level} \geq 10 \wedge \texttt{drain} \geq 0) \textbf{ then} \\ & \quad \textbf{if } (\texttt{level} \leq 3) \textbf{ then } \texttt{drain} := 1/2 \textbf{ else } \texttt{drain} := -1/2\end{aligned}$$

The plant of `CSingleTank` with sufficiently large regions is as follows:

$$\begin{aligned}&\mathsf{plant}_{\texttt{CSingleTank}} \equiv \\ &\quad \{\texttt{level}' = \texttt{drain} \,\&\, (\texttt{level} \leq 3 \wedge \texttt{drain} \leq 0) \vee (\texttt{level} \geq 10 \wedge \texttt{drain} \geq 0)\} \\ &\cup \{\texttt{level}' = \texttt{drain} \,\&\, (\texttt{level} \geq 3 \vee \texttt{drain} \geq 0) \wedge (\texttt{level} \leq 10 \vee \texttt{drain} \leq 0)\}\end{aligned}$$

5.6 On Translation into $d\mathcal{L}$

`HABS` programs can be tested and validated, but the programmer needs to avoid writing programs that are

1. inherently difficult to interpret and
2. have a high degree of non-determinism.

Both are good programming and software engineering practices, of course, and the fact that `HABS` is a *programming language* enables one to apply standard techniques for discrete programs.

A back-translation from $d\mathcal{L}$ to `HABS` would provide meaningful validation only for deterministic $d\mathcal{L}$ models. While being possible even in the general case, two traits of $d\mathcal{L}$ programs prohibit easy interpretation and simulation:

Highly Non-Deterministic Structure Additionally to non-deterministic assignment, branching and repetition are both non-deterministic: the – rather non-intuitive – representation of $(\mathtt{s})^*$ in `HABS` is a loop that non-deterministically chooses to break out.

```
1 while(True) {
2     Int i = random(2);
3     if ( i == 1 ) break;
4     s;
5 }
```

This loop may never terminate, while the semantics of $d\mathcal{L}$ loops defines an arbitrary but countable number of repetitions. A similar pattern has to be employed for branching.

Tests The test $?\varphi$ discards a run based on a $d\mathcal{L}$-guard. Translation would require

1. to evaluate $d\mathcal{L}$ formulas, as opposed to Boolean expressions, and
2. a mechanism to abort the program.

This can be emulated by exceptions, but it obfuscates the semantics.

6 Related & Future Work, Conclusion

6.1 Related Work

Previous work on hybrid programming concentrated on purely sequential languages: `HybCore` [39] is a while-language with hybrid behavior and a simulator [40], but lacks formal verification techniques. Its extensional semantics is not able to express the timed properties needed for our distributed controller. $\texttt{While}^{\texttt{dt}}$ [77] is also a while-language and uses infinitesimals instead of ODEs to model continuous dynamics. It has a simple verification system based on Hoare triples [42], but is not executable.

Hybrid Rebeca (HR) [46] proposes to embed hybrid automata directly into the actor language Rebeca. In contrast to `HABS`, no simulation is available and verification is not object-modular: the whole model is translated to a single monolithic hybrid automaton. Because of this, a number of boundedness constraints have to be imposed. The translation is also the semantics: HR has no semantics beyond this translation and is mainly a frontend for Hybrid Automata tools. The verification backend of HR does not support non-linear ODEs (our examples are linear, but `HABS`, KeYmaera X, and Maxima, support non-linear ODEs; `HABS` models with non-linear ODEs are found in the online supplement).

Recent efforts [58, 64] split the verification task in $d\mathcal{L}$ into manageable pieces by modularizing deductive hybrid systems verification with component-based modeling and verification techniques, but impose strict structural requirements on components and communication. The Sphinx modeling tool [62] for $d\mathcal{L}$ represents non-distributed hybrid programs with UML class and activity diagrams, but for verification purposes it translates these model artifacts into a single monolothic hybrid program.

The Architecture Analysis and Design Language (AADL), a language to model hardware and software components in embedded systems, has a hybrid extension [2], which uses the HHL [80] theorem prover as its verification backend [1]. HHL is based on Hoare triples over hybrid CSP programs and duration calculus formulas [57]. Hybrid AADL offers structuring elements for components and their connections on the architecture level. The semantics of hybrid AADL is given as a translation of the *synchronous* fragment of AADL into hybrid CSP, while we extend the semantics of the actor-based programming language `ABS` to combine reasoning about the asynchronous behavior of communicating components in `ABS` with reasoning about the internal combined discrete and continuous component behavior in differential dynamic logic. As a side effect, the extended semantics enables proving the correctness of the translation to differential dynamic logic, as well as translating `HABS` to other formal languages.

A similar approach based on Stateflow/Simulink is implemented in the MARS toolkit [22]. The MARS approach is orthogonal to `HABS`: MARS connects a verification toolkit around a simulation language (which is a daunting task given the missing formal semantics of Stateflow/Simulink), while `HABS` is designed specifically to enable verification and simulation through its languages features. This is reflected in the soundness proof, which is based on a *bi*directional translation.

Another approach based on CSP and the duration calculus combines these formalisms with Object-Z [45]. This enables model-checking for real-time systems (clocks with resets), while we support hybrid systems theorem proving with (non-linear) differential equations. A further integration of Object-Z and (Timed) CSP was investigated by Mahony & Dong [60].

Hybrid Event-B [12, 13] extends Event-B refinement reasoning with continuous behavior between the usual discrete Event-B events. A more lightweight approach [76, 21] models hybrid systems in an abstract way as action systems without differential equations directly in Event-B, and complements analysis in Event-B with simulation in Matlab. Similarly, Dupont et al. [34] use Event-B for a correct-by-construction approach to hybrid systems. They embed the ODEs used for continuous modeling by declaring them as a special theory within Event-B instead of extending the core language itself.

Integrated tools such as Ptolemy [71], Stateflow/Simulink except the aforementioned MARS toolkit, and Modelica, all emphasize simulation, reachability analysis (e.g., CHARON [6, 7], Ariadne [15]), or testing (e.g., [30]). As supporting techniques, they provide modeling notation for timing aspects, signals, and data flow between heterogeneous models. Formal verification of hybrid systems with reachability analysis and model checking tools (SpaceEx [35], CORA [4], Flow* [23]) support modularity [33] based on hybrid I/O automata [59], assume-guarantee reasoning [17, 43], and hybridization [24]. However, they work best for finite-horizon analysis and finite regions (because over-approximations stay tight only for bounded time and from small starting regions). Similar restrictions apply to dReal/dReach [37, 55].

Dynamic I/O automata [9] for modeling dynamic systems introduce a notion of externally visible behavior, the ability to create and destroy automata and change their signature dynamically; those features are all naturally available in our object-oriented approach and do not need special extension like automata-based modeling tools. Our work contrasts with all mentioned simulation and verification approaches by providing a uniform modeling language, validation by simulation, modular infinite-horizon and infinite-region theorem proving through translation from `HABS` to $d\mathcal{L}$.

Translation among hybrid system languages so far centers around hybrid automata as a unifying concept [11, 79]. Others focus on the discrete fragment [38]. Our translation from `HABS` to $d\mathcal{L}$ translates complete hybrid system models written in a *programming language*, including annotations (preconditions, invariants, etc.). It is sound relative to the formal semantics of `HABS` and $d\mathcal{L}$.

Hybrid systems validation through simulation is addressed with translation to Stateflow/Simulink [10]; with a combination of discrete-event and numerical methods [19]; and with co-simulation between control software and dedicated physics simulators [26, 78, 82]. Here, we focus on safety verification, the distributed aspect of `HABS` models, and take a pragmatic first step for simulating continuous models.

In summary, `HABS` is designed for modular deductive verification (unlike simulation-centric tools), infinite-horizon analysis on infinite regions (unlike reachability analysis and model checking tools), without sacrificing high-level programming language features (unlike hybrid systems modularization techniques and assume-guarantee reasoning).

6.2 Future Work

The present work lifts the research on formal semantics of programming languages for hybrid systems from verification-centric minimalistic languages to distributed object-oriented languages. Carrying over techniques, ideas, and analyses from programming language research to hybrid systems programming, presents an intriguing research direction. Our ongoing work on larger case studies with `HABS`, in particular in connection with co-simulation [54], is expected to reveal additional challenges.

We plan to combine the verification of `CHABS` presented here with the more modular approach based on post-regions [51], which does not support timed input requirements yet. Future research avenues include investigating how the static analyses for `ABS`, in particular the deadlock analysis for

boolean guards [50], can be extended for HABS, extending approximate simulation of non-solvable differential equations, experimenting with various computer algebra systems, and supporting guards with non-urgent semantics.

6.3 Conclusion

Distributed hybrid systems are not only difficult to *verify* formally, it is equally hard to *validate* a formal model of them, especially with components using symbolic computations, such as servers. Both activities have conflicting demands, so we propose a translation-based approach: modeling is guided by patterns over hybrid programs and class specifications in HABS, a hybrid extension of the concurrent active-object language ABS. These are automatically decomposed and translated (Thm. 5) into sequential proof obligations of the verification-oriented differential dynamic logic $d\mathcal{L}$ and discharged by the hybrid theorem prover KeYmaera X.

We illustrated the viability of our approach by a case study that features many complications: concurrent behavior, possible non-termination, correctness depending on timing constants, multi-dimensional domain, time lag in sensing, etc.

References

1 Ehsan Ahmad, Yunwei Dong, Shuling Wang, Naijun Zhan, and Liang Zou. Adding formal meanings to AADL with hybrid annex. In Ivan Lanese and Eric Madelaine, editors, *Formal Aspects of Component Software - 11th International Symposium, FACS 2014, Bertinoro, Italy, September 10-12, 2014, Revised Selected Papers*, volume 8997 of *Lecture Notes in Computer Science*, pages 228–247. Springer, 2014. doi:10.1007/978-3-319-15317-9_15.

2 Ehsan Ahmad, Brian R. Larson, Stephen C. Barrett, Naijun Zhan, and Yunwei Dong. Hybrid annex: an AADL extension for continuous behavior and cyber-physical interaction modeling. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 29–38. ACM, 2014. doi:10.1145/2663171.2663178.

3 Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. *Service Oriented Computing and Applications*, 8(4):323–339, 2014.

4 M. Althoff. An introduction to CORA 2015. In *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.

5 Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229, Berlin, Heidelberg, 1993. Springer.

6 Rajeev Alur, Thao Dang, Joel M. Esposito, Rafael B. Fierro, Yerang Hur, Franjo Ivancic, Vijay Kumar, Insup Lee, Pradyumna Mishra, George J. Pappas, and Oleg Sokolsky. Hierarchical hybrid modeling of embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, volume 2211 of *Lecture Notes in Computer Science*, pages 14–31. Springer, 2001. doi:10.1007/3-540-45449-7_2.

7 Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular specification of hybrid systems in CHARON. In Nancy A. Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control, Third International Workshop, HSCC 2000, Pittsburgh, PA, USA, March 23-25, 2000, Proceedings*, volume 1790 of *Lecture Notes in Computer Science*, pages 6–19. Springer, 2000. doi:10.1007/3-540-46430-1_5.

8 Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

9 Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata: A formal and compositional model for dynamic systems. *Inf. Comput.*, 249:28–75, 2016. doi:10.1016/j.ic.2016.03.008.

10 Stanley Bak, Omar Ali Beg, Sergiy Bogomolov, Taylor T. Johnson, Luan Viet Nguyen, and Christian Schilling. Hybrid automata: from verification to implementation. *STTT*, 21(1):87–104, 2019.

11 Stanley Bak, Sergiy Bogomolov, and Taylor T. Johnson. HYST: a source transformation and translation tool for hybrid automaton models. In Antoine Girard and Sriram Sankaranarayanan, editors, *HSCC'15*, pages 128–133. ACM, 2015.

12 Richard Banach, Michael J. Butler, Shengchao Qin, Nitika Verma, and Huibiao Zhu. Core hybrid Event-B I: single hybrid Event-B machines. *Sci. Comput. Program.*, 105:92–123, 2015. doi:10.1016/j.scico.2015.02.003.

13 Richard Banach, Michael J. Butler, Shengchao Qin, and Huibiao Zhu. Core hybrid Event-B II: multiple cooperating hybrid Event-B machines. *Sci. Comput. Program.*, 139:1–35, 2017. doi:10.1016/j.scico.2016.12.003.

14 Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

15 Luca Benvenuti, Davide Bresolin, Pieter Collins, Alberto Ferrari, Luca Geretti, and Tiziano Villa. Assume–guarantee verification of nonlinear hybrid systems with Ariadne. *International Journal of Robust and Nonlinear Control*, 24(4):699–724, 2014. doi:10.1002/rnc.2914.

16 Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

17 Sergiy Bogomolov, Goran Frehse, Marius Greitschus, Radu Grosu, Corina S. Pasareanu, Andreas Podelski, and Thomas Strump. Assume-guarantee abstraction refinement meets hybrid systems. In Eran Yahav, editor, *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, volume 8855 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2014. doi:10.1007/978-3-319-13338-6_10.

18 Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. VeriPhy: Verified controller executables from verified cyber-physical system models. In Dan Grossman, editor, *PLDI*, pages 617–630. ACM, 2018. doi:10.1145/3192366.3192406.

19 Christopher X. Brooks, Edward A. Lee, David Lorenzetti, Thierry S. Nouidui, and Michael Wetter. CyPhySim: a cyber-physical systems simulator. In Antoine Girard and Sriram Sankaranarayanan, editors, *HSCC'15*, pages 301–302. ACM, 2015. doi:10.1145/2728606.2728641.

20 Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

21 Michael J. Butler, Jean-Raymond Abrial, and Richard Banach. Modelling and refining hybrid systems in Event-B and Rodin. In Luigia Petre and Emil Sekerinski, editors, *From Action Systems to Distributed Systems - The Refinement Approach*, pages 29–42. Chapman and Hall/CRC, 2016. doi:10.1201/b20053-5.

22 Mingshuai Chen, Xiao Han, Tao Tang, Shuling Wang, Mengfei Yang, Naijun Zhan, Hengjun Zhao, and Liang Zou. MARS: A toolchain for modelling, analysis and verification of hybrid systems. In Michael G. Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog, editors, *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, pages 39–58. Springer, 2017. doi:10.1007/978-3-319-48628-4_3.

23 Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 2013. doi:10.1007/978-3-642-39799-8_18.

24 Xin Chen and Sriram Sankaranarayanan. Decomposed reachability analysis for nonlinear systems. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*, pages 13–24. IEEE Computer Society, 2016. doi:10.1109/RTSS.2016.011.

25 Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2010.

26 Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. Hybrid co-simulation: it's about time. *Software and Systems Modeling*, 18(3):1655–1679, 2019. doi:10.1007/s10270-017-0633-6.

27 P.J.L. Cuijpers and M.A. Reniers. Hybrid process algebra. *J. of Logic and Algebraic Programming*, 62(2):191–245, 2005.

28 Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.

29 Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Computíng Surveys*, 50(5):1–39, 2017.

30 Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.*, 2(OOPSLA):159:1–159:30, 2018. doi:10.1145/3276529.

31 Crystal Chang Din, Reiner Hähnle, Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. Locally abstract, globally concrete semantics of concurrent programming languages. In Renate A. Schmidt and Cláudia Nalon, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 26th International Conference, TABLEAUX 2017, Brasília, Brazil, September 25-28, 2017, Proceedings*, volume 10501 of *Lecture Notes in Computer Science*, pages 22–43. Springer, 2017. doi:10.1007/978-3-319-66902-1_2.

32 Crystal Chang Din and Olaf Owe. Compositional reasoning about active objects with shared futures. *Formal Asp. Comput.*, 27(3):551–572, 2015. doi:10.1007/s00165-014-0322-y.

33 Alexandre Donzé and Goran Frehse. Modular, hierarchical models of control systems in SpaceEx. In *European Control Conference, ECC 2013, Zurich, Switzerland, July 17-19, 2013*, pages 4244–4251. IEEE, 2013. URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6669815, doi:10.23919/ECC.2013.6669815.

34 Guillaume Dupont, Yamine Aït Ameur, Neeraj Kumar Singh, and Marc Pantel. Event-B hybridation: A proof and refinement-based framework for modelling hybrid systems. *ACM Trans. Embed. Comput. Syst.*, 20(4):35:1–35:37, 2021. doi:10.1145/3448270.

35 Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011. doi:10.1007/978-3-642-22110-1_30.

36 Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In Amy P. Felty and Aart Middeldorp, editors, *CADE'25*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015. doi:10.1007/978-3-319-21401-6_36.

37 Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013. doi:10.1007/978-3-642-38574-2_14.

38 Luis Garcia, Stefan Mitsch, and André Platzer. HyPLC: hybrid programmable logic controller program translation for verification. In *ICCPS'19*, pages 47–56, 2019.

39 Sergey Goncharov and Renato Neves. An adequate while-language for hybrid computation. *CoRR*, abs/1902.07684, 2019.

40 Sergey Goncharov, Renato Neves, and José Proença. Implementing hybrid semantics: From functional to imperative. In Violet Ka I Pun, Volker Stolz, and Adenilso Simão, editors, *Theoretical Aspects of Computing - ICTAC 2020 - 17th International Colloquium, Macau, China, November 30 - December 4, 2020, Proceedings*, volume 12545 of *Lecture Notes in Computer Science*, pages 262–282. Springer, 2020. doi:10.1007/978-3-030-64276-1_14.

41 Daniel Grahl, Richard Bubel, Wojciech Mostowski, Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Modular specification and verification. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors, *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*, pages 289–351. Springer, 2016. doi:10.1007/978-3-319-49812-6_9.

42 Ichiro Hasuo and Kohei Suenaga. Exercises in nonstandard static analysis of hybrid systems. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 462–478. Springer, 2012. doi:10.1007/978-3-642-31424-7_34.

43 Thomas A. Henzinger, Marius Minea, and Vinayak S. Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings*, volume 2034 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2001. doi:10.1007/3-540-45351-2_24.

44 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73*, pages 235–245, 1973.

45 Jochen Hoenicke and Ernst-Rüdiger Olderog. Combining specification techniques for processes, data and time. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, pages 245–266, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

46 Iman Jahandideh, Fatemeh Ghassemi, and Marjan Sirjani. Hybrid Rebeca: modeling and analyzing of cyber-physical systems. *CoRR*, abs/1901.02597, 2019. arXiv:1901.02597.

47 Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Aurora Schmidt, Ryan Gardner, Stefan Mitsch, and André Platzer. A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. *STTT*, 19(6):717–741, 2017. doi:10.1007/s10009-016-0434-1.

48 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2010.

49 Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. A formal model of cloud-deployed software and its application to workflow processing. In Dinko Begusic, Nikola Rozic, Josko Radic, and Matko Saric, editors, *SoftCOM'17*, pages 1–6. IEEE, 2017.

50 Eduard Kamburjan. Detecting deadlocks in formal system models with condition synchronization. *ECEASST*, 76, 2018. doi:10.14279/tuj.eceasst.76.1070.

51 Eduard Kamburjan. From post-conditions to post-region invariants: Deductive verification of hybrid objects. In *HSCC*. ACM, 2021.

52 Eduard Kamburjan, Crystal Chang Din, Reiner Hähnle, and Einar Broch Johnsen. Behavioral contracts for cooperative scheduling. In *20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*, pages 85–121. Springer, 2020.

53 Eduard Kamburjan, Reiner Hähnle, and Sebastian Schön. Formal modeling and analysis of railway operations with Active Objects. *Science of Computer Programming*, 166:167–193, November 2018.

54 Eduard Kamburjan, Rudolf Schlatte, Einar Broch Johnsen, and S. Lizeth Tapia Tarifa. Designing distributed control with hybrid active objects. In *ISoLA*, volume 12479 of *LNCS*. Springer, 2020.

55 Soonho Kong, Sicun Gao, Wei Chen, and Edmund M. Clarke. dReach: δ-reachability analysis for hybrid systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of

Lecture Notes in Computer Science, pages 200–205. Springer, 2015. doi:10.1007/978-3-662-46681-0_15.

56 Jia-Chun Lin, Ingrid Chieh Yu, Einar Broch Johnsen, and Ming-Chang Lee. ABS-YARN: A formal framework for modeling hadoop YARN clusters. In Perdita Stevens and Andrzej Wasowski, editors, *FASE'16*, volume 9633 of *LNCS*, pages 49–65. Springer, 2016.

57 Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. A calculus for hybrid CSP. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010. doi:10.1007/978-3-642-17164-2_1.

58 Simon Lunel, Stefan Mitsch, Benoît Boyer, and Jean-Pierre Talpin. Parallel composition and modular verification of computer controlled systems in differential dynamic logic. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods, The Next 30 Years, Third World Congress, FM, Porto, Portugal*, volume 11800 of *LNCS*, pages 354–370. Springer, 2019.

59 Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Hybrid I/O automata. *Inf. Comput.*, 185(1):105–157, 2003. doi:10.1016/S0890-5401(03)00067-1.

60 Brendan P. Mahony and Jin Song Dong. Deep semantic links of TCSP and Object-Z: TCOZ approach. *Formal Aspects Comput.*, 13(2):142–160, 2002. doi:10.1007/s001650200004.

61 *Maxima Manual*, 5.43.0 edition, 2019. URL: maxima.sourceforge.net.

62 Stefan Mitsch, Grant Olney Passmore, and André Platzer. Collaborative verification-driven engineering of hybrid systems. *Math. Comput. Sci.*, 8(1):71–97, 2014. doi:10.1007/s11786-014-0176-y.

63 Stefan Mitsch and André Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.*, 49(1):33–74, 2016.

64 Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. Tactical contract composition for hybrid system component verification. *STTT*, 20(6):615–643, 2018. Special issue for selected papers from FASE'17.

65 André Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *J. of Logic and Computation*, 20(1):309–352, 2010.

66 André Platzer. The complete proof theory of hybrid systems. In *LICS*, pages 541–550. IEEE, 2012.

67 André Platzer. The structure of differential invariants and differential cut elimination. *Logical Methods in Computer Science*, 8(4):1–38, 2012. doi:10.2168/LMCS-8(4:16)2012.

68 André Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Automated Reasoning*, 59(2):219–265, 2017.

69 André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.

70 André Platzer and Yong Kiam Tan. Differential equation invariance axiomatization. *J. ACM*, 67(1):6:1–6:66, 2020. doi:10.1145/3380825.

71 Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

72 Jan-David Quesel, Stefan Mitsch, Sarah Loos, Nikos Aréchiga, and André Platzer. How to model and prove hybrid systems with KeYmaera: A tutorial on safety. *STTT*, 18(1):67–91, 2016. doi:10.1007/s10009-015-0367-0.

73 Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.

74 Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *STTT*, 14(5):477–495, 2012.

75 Rudolf Schlatte, Einar Broch Johnsen, Jacopo Mauro, Silvia Lizeth Tapia Tarifa, and Ingrid Chieh Yu. Release the beasts: When formal methods meet real world data. In *It's All About Coordination*, volume 10865 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2018.

76 Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Formalizing hybrid systems with Event-B and the Rodin platform. *Sci. Comput. Program.*, 94:164–202, 2014. doi:10.1016/j.scico.2014.04.015.

77 Kohei Suenaga and Ichiro Hasuo. Programming with infinitesimals: A while-language for hybrid system modeling. In *ICALP (2)*, volume 6756 of *Lecture Notes in Computer Science*, pages 392–403. Springer, 2011.

78 Casper Thule, Kenneth Lausdahl, Cláudio Gomes, Gerd Meisl, and Peter Gorm Larsen. Maestro: The INTO-CPS co-simulation framework. *Simulation Modelling Practice and Theory*, 92:45–61, 2019. doi:10.1016/j.simpat.2018.12.005.

79 D. A. van Beek, Michel A. Reniers, Ramon R. H. Schiffelers, and J. E. Rooda. Foundations of a compositional interchange format for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *HSCC'07*, volume 4416 of *LNCS*, pages 587–600. Springer, 2007.

80 Shuling Wang, Naijun Zhan, and Liang Zou. An improved HHL prover: An interactive theorem prover for hybrid systems. In Michael Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering*, pages 382–399, Cham, 2015. Springer International Publishing.

81 Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *STTT*, 14(5):567–588, 2012.

82 Zhenkai Zhang, Emeka Eyisi, Xenofon D. Koutsoukos, Joseph Porter, Gabor Karsai, and Janos Sztipanovits. A co-simulation framework for design of time-triggered automotive cyber physical systems. *Simulation Modelling Practice and Theory*, 43:16–33, 2014.

Bayesian Hybrid Automata: A Formal Model of Justified Belief in Interacting Hybrid Systems Subject to Imprecise Observation

Paul Kröger
Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, Germany

Martin Fränzle
Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, Germany

Abstract

Hybrid discrete-continuous system dynamics arises when discrete actions, e.g. by a decision algorithm, meet continuous behaviour, e.g. due to physical processes and continuous control. A natural domain of such systems are emerging smart technologies which add elements of intelligence, cooperation, and adaptivity to physical entities, enabling them to interact with each other and with humans as systems of (human-)cyber-physical systems or (H)CPSes.

Various flavours of hybrid automata have been suggested as a means to formally analyse CPS dynamics. In a previous article, we demonstrated that all these variants of hybrid automata provide inaccurate, in the sense of either overly pessimistic or overly optimistic, verdicts for engineered systems operating under imprecise observation of their environment due to, e.g., measurement error. We suggested a revised formal model, called Bayesian hybrid automata, that is able to represent state tracking and estimation in hybrid systems and thereby enhances precision of verdicts obtained from the model in comparison to traditional model variants.

In this article, we present an extended definition of Bayesian hybrid automata which incorporates a new class of guard and invariant functions that allow to evaluate traditional guards and invariants over probability distributions. The resulting framework allows to model observers with knowledge about the control strategy of an observed agent but with imprecise estimates of the data on which the control decisions are based.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems

Keywords and Phrases stochastic hybrid systems, Bayesian inference, formal models, cyber-physical systems

Digital Object Identifier 10.4230/LITES.8.2.5

Funding This research was supported by Deutsche Forschungsgemeinschaft under grant number DFG GRK 1765 covering the Research Training Group "SCARE: System Correctness under Adverse Conditions".

Received 2020-10-01 **Accepted** 2021-11-16 **Published** 2022-12-07

Editor Alessandro Abate, Uli Fahrenberg, and Martin Fränzle

Special Issue Special Issue on Distributed Hybrid Systems

1 Introduction

Smart cities, automated transportation systems, smart health, and Industry 4.0 are examples of large-scale applications in which elements of intelligence, cooperation, and adaptivity are added to physical entities, enabling them to interact with each other and with humans as cyber-physical systems or, in the latter case, human-cyber-physical systems (CPSes or HCPSes). Due to the criticality of many of their application domains, such interacting cyber-physical systems call for rigorous analysis of their emergent dynamic behaviour w.r.t. a variety of design goals ranging from safety, stability, and liveness properties over performance measures to human-comprehensibility of their actions and absence of automation surprises. The model of hybrid (discrete-continuous)

automata [2, 28, 19], in its various flavours, has traditionally been suggested as a formal model accurately capturing CPS dynamics and thus facilitating such analysis with mathematical rigour whenever the pertinent requirements can also be formalised, which applies at least for the safety, stability, convergence, and liveness properties.

Hybrid automata (HA) provide a mathematical abstraction of the interaction between decision making, continuous control, and continuous environments. They couple a finite-state control skeleton with a continuous state-space spanned by real-valued variables. The continuous state has its dynamics governed by differential equations selected depending on the current control-skeleton state (often called a discrete mode or a control location), and vice versa state dynamics of the control skeleton is controlled by predicates on the continuous state. Various flavours of HA have been suggested as a means to formally analyse different aspects of hybrid-state dynamical systems, among them deterministic HA facilitating reasoning about their normative behaviour, non-deterministic HA [2, 28] under a demonic interpretation supporting worst-case analysis with respect to disturbances and measurement errors, and stochastic HA enabling quantitative verification [19, 31, 9, 15, 21, 14, 5]. Encoding the dynamics of an actual cyber-physical system into one of the aforementioned modelling frameworks is in general considered a tedious, yet mostly straightforward activity: it is assumed that these frameworks are rich enough to accommodate adequate models of standard components, like sensors measuring physical quantities and actuators modifying such quantities, as well as standard models of physical dynamics, continuous control, and mode-switching control.

In this article, which is an extended version of [17], we demonstrate that despite their embracing expressiveness and contrary to the intuition underlying the above modelling pragmatics, all flavors of hybrid automata fall short of being able to accurately capture the interaction dynamics of systems of well-engineered, rationally acting CPS designs operating under aleatory uncertainty. We show that the corresponding verification verdicts obtained on the best possible approximations of the actual CPS dynamics are across the range of hybrid automata models bound to be either overly optimistic or overly pessimistic, i.e., imprecise.

We identify inaptness to adequately cover rational decision making under uncertain information as the cause of this deficiency of the hybrid-automaton model. As such rational decision making requires manipulation of environmental state estimates to be embedded into the system state itself, necessitating manipulation of state distributions rather than "just" discrete plus real-vector valued state within the CPS and its corresponding formal model, we suggest an appropriate extension of hybrid automata featuring mixture-based probability distributions in some of its state variables. It adopts from metrology the concept of processing noisy measurements by means of filtering and representing the result as a distribution over possible ground truth [20, 29] and incorporates it into HA models. The resulting hybrid models can in general not be reduced to traditional HA featuring a finite-dimensional real-valued state vector, such that verification support remains an open issue that cannot be discharged by appropriate encoding into existing hybrid-automata verification approaches [13].

Organisation of the paper

In the subsequent section, we discuss related work in order to identify a current lack of models for hybrid dynamics being able to directly accommodate inference mechanisms about uncertain state observation. This would, however, not necessarily imply that current models are too weak for producing precise verdicts on system correctness, as an encoding of pertinent methods for fusing measurements could well be possible within existing models. In Sect. 3.2, we therefore demonstrate by means of a running example that traditional hybrid-system models are bound to fail in providing the expected verification verdicts. This in turn motivates us to introduce

filtering and state estimation into a revised model of hybrid automata. Section 3.3 demonstrates that this indeed leads to accurate verdicts adequately reflecting engineering practice, while Sect. 4 shows that an embedding of such environmental state estimation into traditional hybrid automata featuring real-vector state is in general impossible if the state estimation has to deal with states of other autonomous agents. Section 5 provides the formal definition of the suggested extension of hybrid automata before Sect. 6 puts forward ideas on automatic verification support for the resulting rich class of hybrid automata, and Sect. 7 concludes the paper by shedding light on related problems in the field of interacting intelligent systems.

2 Related work

An essential characteristic of cyber-physical systems is their hybrid discrete-continuous state-space, combining a continuous, real-vector state-space with a number of discrete modes determining the dynamics of the continuous evolution. Hybrid automata (HA) [2, 28] have been suggested as a formal model permitting the rigorous analysis of such systems. In their deterministic or demonically nondeterministic form, HA support qualitative reasoning in the sense of exhaustive verification or falsification, over the normative behaviour or the worst-case behaviour of the system. Probabilistic or stochastic extensions of HA, so-called stochastic hybrid automata [21], enable deriving quantitative figures about the satisfaction of a safety target by considering probability distributions over uncertain choices. Several variants of such a quantification have been studied, e.g., HA with discrete [31, 15] or continuous [14] distributions over discrete transitions as well as stochastic differential dynamics within a discrete mode [19].

HA models support the qualitative and quantitative analysis of systems subject to noise, yet lack pertinent means for expressing the effects of state estimation and filtering known to be central to rational strategies in games of incomplete information [25, Chapters 9-11] and thus in optimal control under uncertainty. Formal modelling of systems taking rational decisions based on best estimates of the uncertain and only partially observable state of other agents inherently requires to incorporate two levels of probabilism: first, in the model of system dynamics as probabilistic occurrences of sequences of observations; second, as distributions representing the best estimates the embedded controller can gain about the state of its environment based on these noisy observations. Formal modelling of rational decision making consequently requires the estimations to be explicitly available in the state space of the controller for evaluations of the underlying decisions (e.g., in the evaluation of a transition guard in supervisory control) and secondly correlated observations have to be fused to obtain best estimates, e.g. in form of Bayes filters [3, 22, 24]. Such probabilistic filters are widely used in robotics, e.g. for the estimation of occupancy grids [12, 7], in robust fault detection under noisy environment [6], or for estimating parameters of stochastic processes in biological tissues or molecular structures [30].

Aiming at approximating Maximum Likelihood Estimates for parameters of non-linear systems with non-Gaussian noise, Murphy [26] considers state estimation with switching Kálmán filters in presence of multiple linear dynamic models. In his setting, the time instances for switching to a certain linear dynamics are unknown up to a known stochastic distribution. In combination with stochastic state observations, this gives rise to state estimates in form of joint distributions, approximated by mixtures of Gaussian distributions. However, in addition to limited dynamics, switching between modes is based on Markovian dynamics, i.e., it is not possible to model switching based on probabilistic constraints on state estimates as necessary to model rational decisions about changing a mode as a response to observed states.

This lack of capabilities to model (rational) control decisions including discontinuous updates of the continuous state space is only partially resolved by the models underlying adaptive control theory, which is subject to comprehensive research [23, 18, 27]. In this context, the focus is on the identification of unknown (control) parameters of systems under imperfect observation. However,

these approaches are not sufficient to analyse the behaviour of interacting intelligent systems as they are restricted to identifying the correct choice between a set of (possibly time-variant) dynamical models for the controlled process.

The consequential necessity of applying Bayesian filtering within hybrid systems implementing optimal control was already discovered by Ding et. al. [10]. They present an approach to derive optimal control policies for partially observable discrete time stochastic hybrid systems, where optimality is defined in terms of achieving the maximum probability that the system remains within a set of safe states. In order to be able to apply dynamic programming in search for an optimal solution, Ding et al. replace the partially observable system by an equivalent perfect information system via a sufficient statistics in form of a Bayes filter. This is very close to our approach in mindset, as a sufficient statistics about a Bayesian estimate of the imperfectly known actual system state is at the heart of rational decisions in control under uncertainty. The main difference is that we are trying to formulate a general model facilitating the behavioural analysis of such optimal hybrid control systems, while Ding et al. aim at the construction of such controllers w.r.t. a given safety goal. The latter facilitates a decomposition of the design problem into obtaining a Bayesian filtering process and developing a —then scalar-valued— control skeleton. This renders a direct integration, as pursued in this article, of state distributions and Bayesian inference mechanisms into the state space of an analytical model unnecessary.

In [16], we already suggested a revised formal model, called Bayesian hybrid automata, that is able to represent state tracking and estimation in hybrid systems and integrates probability density functions in its state space thereby enabling modelling of rational decision making under uncertain information. However, this model was not yet capable of covering hybrid-state dynamics of the observed agent when it comes to extrapolation of estimates over time between two measurement instances. As indicated in [17], this requires mixture distributions dealing with all possible decision alternatives (including the case that the decision is pending) in the state space of the model. In this article, we extend our previous work by a formal definition of an extension of Bayesian hybrid automata incorporating mixture distributions and a semantics covering hybrid-state dynamics for state-extrapolation.

3 Inadequacy of traditional hybrid-automata models

Hybrid automata have been conceived in [2, 28] as a formal model seamlessly integrating decision making with control, thus facilitating the modelling and analysis of the joint dynamics of these two layers pertinent to CPSes: discrete decisions, e.g. between the alternative manoeuvres of following a lead car or overtaking it in an autonomous car, do dynamically activate and deactivate continuous control skills, like an automatic distance control implementing the car-following manoeuvre. In HA, the former are described by a finite automaton featuring transitions guarded by (and possibly inducing side effects on) continuous state variables of the control path and the controller, while the latter are governed by differential equations attributed to (and thus changing in synchrony with) the automaton locations and ranging over the continuous state variables and thus describing the state dynamics of both the control path and the controller.

In reality, such CPSes have to operate and draw decisions under a variety of uncertainties stemming from their multi-component nature, as the latter requires mutual state observation between agents. Such sensing of non-local state inevitably induces uncertainties due to, a.o., the measurement inaccuracies inherent to sensor devices. In consequence, the decision making in real CPSes is bound to be rational decision-making under uncertainties. In this section, we demonstrate how existing hybrid-automata models fall short in taking account of such rational decision-making. To showcase the problem, we will in the following exploit a very simple example, mostly taken from [16], of a rational decision-making problem to be solved by a CPS.

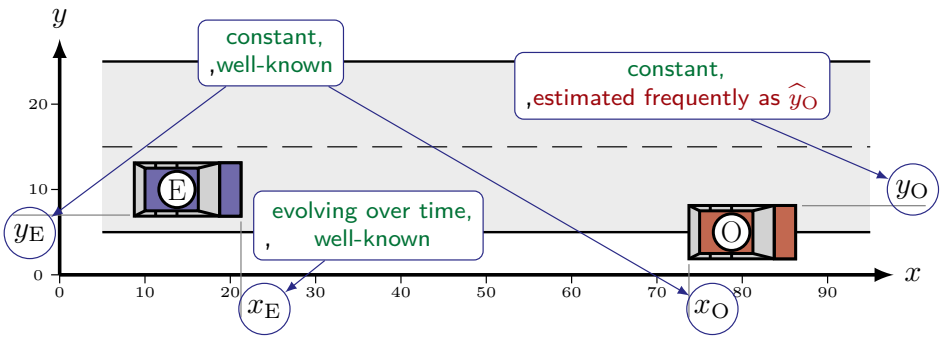


Figure 1 A common traffic situation (taken from [16]): ego vehicle E shall decide between passing the parked obstacle O or halting.

3.1 An example of a control decision problem

Our example deals with a common traffic situation depicted in Fig. 1. Our own autonomous car, called the ego vehicle and denoted by E in the sequel, is driving along a road which features another vehicle O parked further down the road. Despite being parked on the roadside, car O may extend into the lane used by E. E cannot perform a lateral evasive manoeuvre due to dense oncoming traffic. E therefore has to decide between passing the car while keeping its lane and an emergency stop avoiding a collision. It obviously ought to decide for a pass whenever that option is safe due to a small enough intrusion of O into the lane, and it should stop otherwise.

The geometric situation can be described by four real-valued variables: three rigid variables x_O, y_O, and y_E describing the static longitudinal position of O and the static lateral positions of both cars, as well as a flexible, continuously evolving variable x_E representing the momentary longitudinal position of the ego car. For simplicity, we assume that all values except the environmental variable y_O are exactly known to the ego car E. The value of y_O has to be determined by sensing the environment via a possibly inaccurate measurement yielding an estimate $\hat{y}_O$ for y_O. For the sake of providing a concrete instance, we assume a normally distributed measurement error, i.e., $\hat{y}_O \sim \mathcal{N}(y_O, \sigma^2)$, though our findings do not hinge on that particular distribution. As a further simplification we assume that car E either drives with nominal speed ($\dot{x}_E = 1$) or is fully stopped ($\dot{x}_E = 0$) and that it switches between these two dynamics instantaneously.

The design goal is to design an ego car that is both safe and live; the corresponding analysis goal consequently is to prove these two properties. Liveness in this context means that car E eventually passes car O whenever $y_E > y_O$. Safety is defined as the exclusion of the possibility of a collision, i.e., that $x_E < x_O$ stays invariant over time whenever $y_E \leq y_O$. These two properties can be formalised as follows using a straightforward extension of CTL featuring relational atoms over continuous signals akin to Signal Temporal Logic [11]:

$$\textbf{safe} := (y_E \leq y_O) \implies \mathbf{AG}\,(x_E < x_O) \tag{1a}$$

$$\textbf{live} := (y_E > y_O) \implies \mathbf{AF}\,(x_E \geq x_O) \tag{1b}$$

3.2 Hybrid automata models

Dealing with sensory observation of environmental variables and potentially reflecting the pertinent measurement inaccuracies within hybrid-automata models is a classical theme. Figure 2 represents the three standard means of dealing with sensory observation in HA models, exemplified on the example from the previous section: Automaton **Nominal** identifies environmental

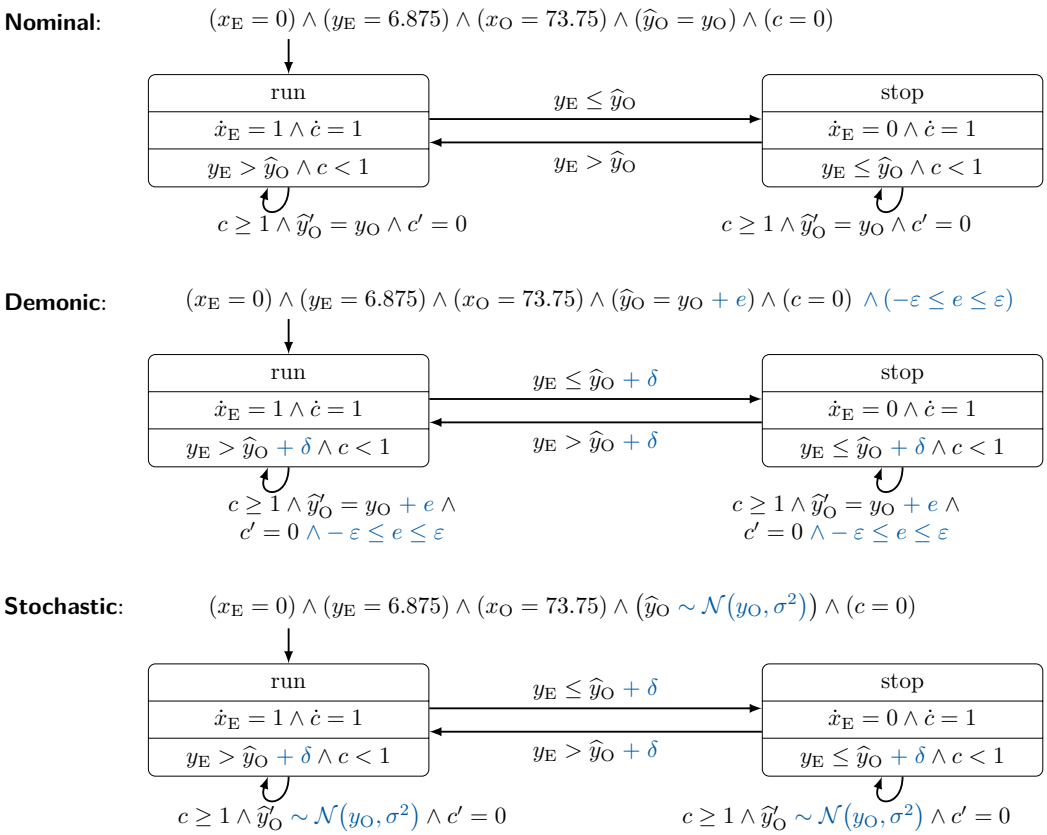


Figure 2 Hybrid automata models for the scenario of Fig. 1 (refinements of [16]). Variable c is a clock variable representing a timer that triggers a measurement every full time unit.

states with their measurements, thereby neglecting measurement error and claiming to draw control decision based on exact environmental entities. **Demonic** models measurement error as a bounded offset e between the actual value y_O and its measurement $\hat{y}_O$, with the offset e nondeterministically chosen afresh upon every take of a measurement. It also employs a safety margin δ within its decision making, passing only when the distance between y_E and $\hat{y}_O$ is larger than the safety margin δ. **Stochastic**, finally, incorporates the faithful model of measurement noise by generating the measurement $\hat{y}_O$ via a normal distribution $\mathcal{N}(y_O, \sigma^2)$ centred around y_O, where σ is the standard deviation of the measurement process.

Case analysis reveals that, depending on the relation between y_E and y_O and the safety margin δ, satisfaction of the two requirements formulae **safe** and **live** by the three models **Nominal**, **Demonic**, and **Stochastic** varies. Satisfaction applies as shown in Table 1.

None of these results seems particularly convincing. The nominal model, ignoring any measurement error in its analysis, optimistically claims its control to be both absolutely safe and live despite its decisions not even catering for adversarial measurement error impacting the non-robust guard $y_E > y_O$. The other two models pessimistically claim that it either is impossible to build any system satisfying any positive safety threshold ($P(\textbf{safe}) \to 0.0$ in **Stochastic**) or to achieve any liveness (**Demonic** $\not\models$ **live**). Given that building such controllers and achieving very high quantitative degrees of, though not absolute, liveness and safety is standard engineering practice, all the above verdicts are disappointing and show inherent deficiencies in our conventional hybrid-state models.

Table 1 Analysis results for the different models. $\to x$ denotes probabilities converging to x in the long-run limit.

(a) Analysis results for automaton **Nominal**.

	safe	**live**
$y_E > y_O$	trivial	sat
$y_E \leq y_O$	sat	trivial

Optimistic verdict, claiming perfect control possible despite the in reality inevitable uncertainty about environmental state.

(b) Automaton **Demonic** (case 3 arises only under insufficient safety margin $\delta < \varepsilon$ where $|e| \leq \varepsilon$).

	safe	**live**
$y_E > y_O + \delta + \max(e)$	trivial	sat
$y_O + \delta + \varepsilon \geq y_E > y_O$	trivial	unsat
$y_E \leq y_O < y_E - \delta + \varepsilon$	unsat	trivial
$y_E - \delta + \varepsilon \leq y_O$	sat	trivial

Pessimistic verdict, rightfully claiming safety at risk whenever an inappropriate safety margin is selected (case 3 in the table), but also claiming liveness perfectly impossible to achieve.

(c) Analysis results for automaton **Stochastic**.

	P(**safe**)	P(**live**)
$y_E > y_O$	1.0	$\to 1.0$
$y_E \leq y_O$	$\to 0.0$	1.0

Pessimistic verdict, claiming achievement of even marginal safety levels impossible over extended periods of time.

3.3 Adding Kálmán filtering

The obvious problem is that the above standard hybrid-automata models neglect the fact that repetition of noisy measurement processes accumulates increasingly better evidence about the true state of the observed entity, albeit always with a remaining uncertainty. While model **Nominal** ignores the impossibility of perfect knowledge, thus yielding inherently optimistic verdicts, models of the shapes **Stochastic** or **Demonic** do not correlate measurements across time series and thus fail to reflect the steady build-up of increasingly precise evidence about the true position y_O of the obstacle. Any form of truly rational decision-making would, however, take advantage of the latter fact; vice versa, any formal model neglecting it provides a coarse overapproximation of actual observational uncertainty resulting in correspondingly pessimistic verification verdicts relative to standard engineering practice employing filtering of measurements.

In the given case of a static obstacle O, as well as in the more general case of a physical process subject to purely linear differential dynamics, standard Kálmán filtering [20] manipulating normal distributions is the method of choice for obtaining best possible estimates of perceived state from independently normally distributed individual measurements. As normal distributions can be represented by a fixed number of parameters, namely their mean value and variance, these can still be incorporated into standard stochastic hybrid-automata models by means of extra variables: Retaining $\hat{y}_O$ as the variable representing the current estimate of the lateral position of O in the scenario from Fig. 1, one has to add a second variable representing the accuracy of the current estimate. This could be the standard deviation or the variance of the estimation error; for simplicity of the update rules it is, however, customary to instead use the precision (i.e. the reciprocal of the variance). Adding a variable ρ representing the precision, the measurement transitions thus change according to the usual Kálmán-filter update rules

$$m \sim \mathcal{N}(y_O, 1/\rho_m) \tag{2a}$$

$$\hat{y}'_O = \frac{\rho \cdot \hat{y}_O + \rho_m \cdot m}{\rho'} \tag{2b}$$

$$\rho' = \rho + \rho_m \tag{2c}$$

where ρ_m is the precision of an individual measurement process and m the recent measurement.

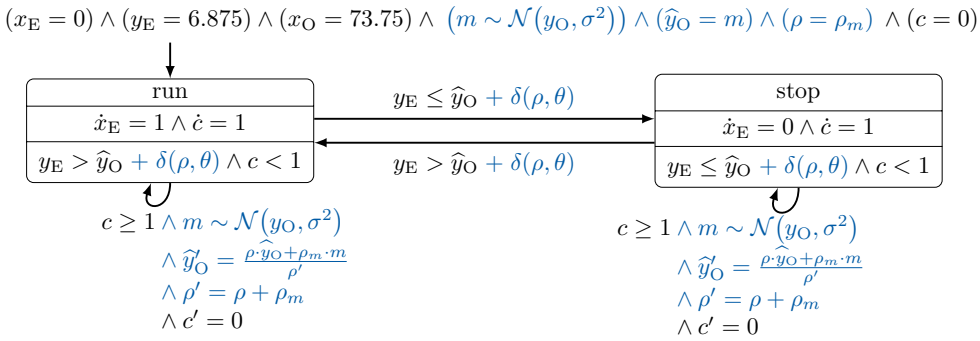


Figure 3 A stochastic hybrid automaton incorporating Kálmán filtering for the measurements of O's position. $\delta(\rho,\theta)$ computes a safety margin yielding confidence θ when the precision is ρ, i.e., is defined by $\int_{-\infty}^{\delta(\rho,\theta)} \mathcal{N}(0, 1/\rho)\mathsf{d}s = \theta$.

The guards governing the decision to move to mode run (as well as the invariant of that mode) change to threshold conditions on the probability mass $P_{x\sim\mathcal{N}(\hat{y}_O, 1/\rho)}(y_E > x) \geq \theta$, checking for sufficient evidence that $y_E > y_O$ holds and thus confining the risk of erroneously moving the car forward when in fact $y_E \leq y_O$ applies to below $1.0 - \theta$. The resulting automaton is depicted in Fig. 3 and reflects the standard engineering practice of Kálmán filtering noisy measurements.

As can be seen from the experimental results reported in Figures 4 and 5, its control performance significantly exceeds all the verdicts for the standard models stated in Table 1. In these experiments, we implemented the automata **Stochastic** and its Kálmán-filtered variant (BHA) both in a safe situation (Figure 4) where car E is ought to pass car O, and in an unsafe situation (Figure 5) where car E should stop since O's sphere overlaps with E's lane. For both situations, the blue solid graph shows the average of switching to or remaining in mode run after a measurement (which is taken at every discrete time instance thereby assuming a step size of 1) for **Stochastic**. The average is constant for both situations. In contrast, the average of driving on converges to 1.0 rapidly for the safe situation while it converges fast to 0.0 for the unsafe situation (illustrated by the green dotted graph). This is where the Bayesian filter's effect manifests itself: all decisions in **Stochastic** are based on the recent (single) measurement thus yielding a constant probability of making a "bad" decision as neither the distance $y_E - y_O$ nor the distribution of the measurement error changes over time. For the BHA, in turn, the integration of all measurement results leads to an estimate in form of a normal distribution of which the mean $\hat{y}_O$ converges to y_O over time while the increasing precision allows for a less conservative safety margin.

The orange dashed-dotted line shows for each discrete time step the probability $P(x_E \geq x_O)$ in **Stochastic**, i.e. the probability that car E has already passed car O in the safe situation and that the cars have already collided in the unsafe situation. The red dashed line shows the same for BHA. As **Stochastic** moves with constant probability, the probability $P(x_E \geq x_O)$ of progressing beyond the other car's position converges to 1.0 in **Stochastic** for both situations. This implies that car E almost surely eventually passes car O in the safe situation, but also almost surely eventually collides in the unsafe situation. These results were already predicted in Table 1. As a consequence of the effect of the filter, the graph for BHA shows for the safe situation that the probability that car E has passed car O increases significantly earlier than for **Stochastic**. Most probably, car E will pass car O quite smoothly after a short while in BHA, while it stutters past O in **Stochastic**. For the unsafe situation, in turn, the red dashed graph shows that the probability of a collision remains very small up to the time horizon.

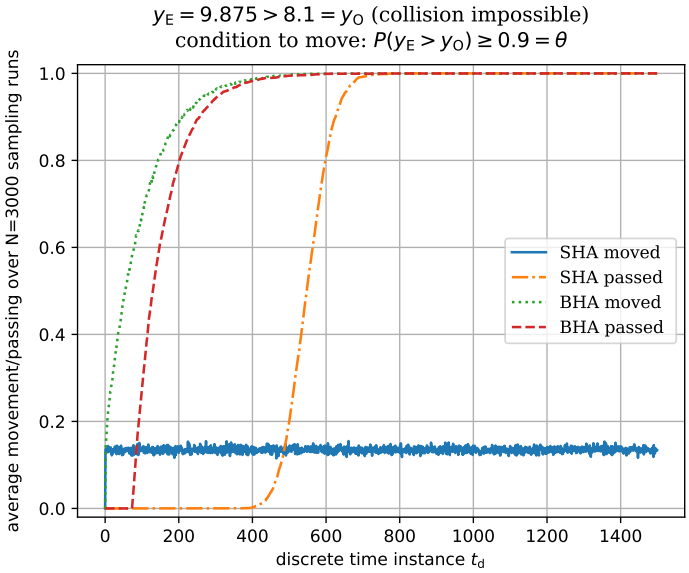


Figure 4 Simulation results for the traffic example (Fig. 1) comparing automaton **Stochastic** (labelled SHA) with its Kálmán-filtered variant (BHA) in a safe situation ($y_E > y_O$). BHA moves steadier (dotted green vs. solid blue line) and passes earlier (red vs. orange).

4 Interacting and cooperating cyber-physical systems

From the above, it might seem that encoding of standard engineering practice into stochastic hybrid automata well is feasible. Issues do, however, get more involved when the perceived objects are subject to more complex dynamics than linear differential equations s.t. normal distributions or other distributions representable by a finite vector of scalar parameters do no longer suffice for encoding optimal state estimates. This applies for example when the observed agent itself is a hybrid or cyber-physical system, as we will show in this next section. The above encoding into a stochastic hybrid automaton with finite-dimensional state becomes infeasible then, instead requiring to embed complex probability distributions directly into the automaton's state space.

To demonstrate this problem induced by the cooperation of smart entities, which hinges on the additional necessity to mutually detect and reason about control decisions of the mutually other agents based on uncertain behavioural observations, we now move on to a slightly more complex scenario involving interaction between cyber-physical systems.

4.1 An example of a cooperative control-decision problem

Imagine two ships approaching each other on a narrow channel permitting opposing traffic only within a designated passing place, as depicted in Fig. 6. The ship reaching the passing place first (ship O) is allowed[1] to draw a decision to which side it turns for mooring while the oncoming ship E enters the passing place. To complicate the issue, we forbid direct communication between the ships. In absence of means of negotiation, the ego ship E has to determine O's ensuing manoeuvre from observing the current lateral position of ship O and decides to move to a certain side as soon as its confidence that O will move to the opposite shore is above a specified threshold.

[1] Please note that this is a toy example ignoring all maritime rules such as COLREGs.

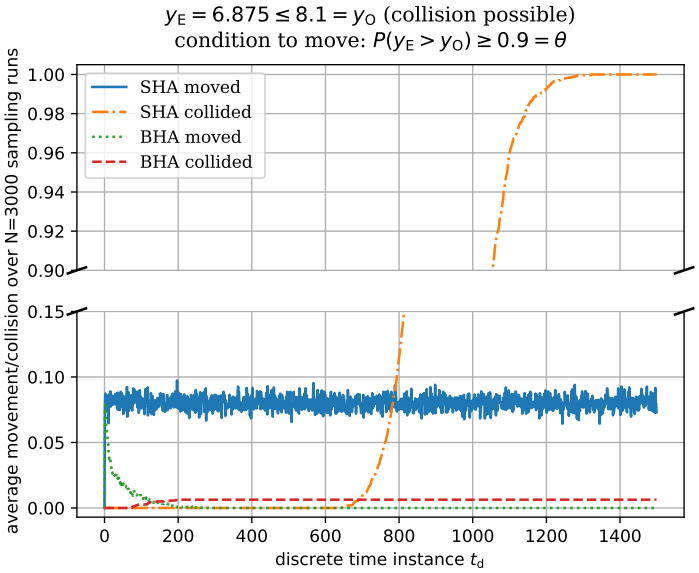


Figure 5 Comparison in an unsafe situation ($y_E < y_O$) of the traffic example (Fig. 1). The Kálmán-filtered BHA enhances safety as it almost surely stops (dotted green line) and its collision probability saturates (dashed red), whereas the latter diverges for the SHA (dash-dotted orange) due to a constant rate of stuttering movement (solid blue).

We assume that ship O has perfect knowledge about its own longitudinal (x_O) and lateral (y_O) position. Ship E, in turn, has perfect knowledge about its own position (x_E and y_E) while it maintains estimates $\hat{x}_O$ and $\hat{y}_O$ of O's position. The problem for E is to determine, based on these estimates, to which side O will evade. Filtering w.r.t. a single known dynamics of O is no longer possible as dynamics depend on O's decision for which, in turn, E has only a probability distribution based on the estimate of O's position. Instead, mixture distributions dealing with all possible decision alternatives (including the case that the decision is pending) have to be dealt with. Each mixture component then covers the part of the state space of y_O that results in a certain decision and is subject to the corresponding dynamics within the filter process.

This obviously requires an extension of the stochastic hybrid automaton setting, as the estimates no longer constitute Gaussians due to the decision process itself, which is reflected by chopping the distributions at the thresholds of guards/invariants. That the underlying dynamics is non-linear only adds to the problem.

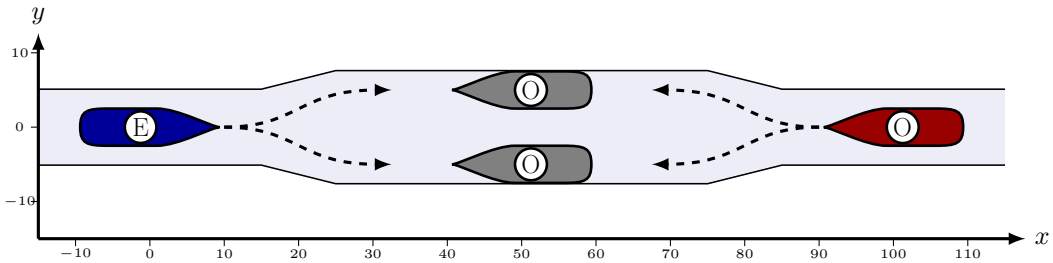


Figure 6 Two ships approaching each other on a channel. The red ship (labelled O) decides to move to the right side of the passing place if its lateral position is larger than 0, and to the left otherwise. The ego ship (blue, labelled E) tries to determine O's manoeuvre and to move to the opposite side.

4.2 Formal modelling of the scenario

Given the complexity of the state estimation and rational decision processes sketched above, a decomposition of the overall problem into a set of interacting automata with dedicated functionalities seems appropriate. Figure 7 shows such a decomposition for the —still simple— case of unilateral observation, i.e., that ship O does not observe ship E and that their control behaviour consequently is not mutually recursive.

The roles of the various automata are as follows:

- *Observed automata (OA)* represent entities that are observed by the ego system. In the example, ship O is modelled by an observed automaton. As ship O has perfect knowledge about its own state (and as its behaviour is independent from E's in the unilateral case), its automaton model is a traditional hybrid automaton of the same shape as **Nominal** in Fig. 2.
- *Estimate automata (EA)* provide estimates of O's current state to E. They cover the perception process, i.e. they reflect the (possibly error-afflicted) environmental perception of the ego system and update quantitative estimates $\hat{x}$ of the observed parameters x. In simple cases, they will regularly at sampling intervals take noisy copies $m \sim \mathcal{N}(x, 1/\rho_m)$ of the observed physical states and incorporate them into estimates $\hat{x}$ for refine estimates $\hat{x}$. In addition, they extrapolate estimates over time between measurement intervals. The steps involved in creating and updating the estimates thus are manifold:
  1. Temporal extrapolation starts with splitting the current estimate, i.e., state distribution for the observed entity O according to O's known mode selection dynamics. In the example, this would imply splitting the $\hat{y}_O$ values of the part of the distribution that is associated to mode 'run' at 0 and associating its negative branch to mode 'left' and the non-negative to mode 'right'.
  2. Reflecting possible sequences of instantaneous discrete jumps of O's control automaton before time elapses, repeat step 1 until a fixed-point is reached which indicates that no further discrete jump is possible.
  3. For each mode, extrapolate these "fragments" of the distribution associated to the mode along the pertinent mode dynamics, which is followed for the duration of a single time step.[2]
  4. Take the resulting extrapolated distribution of O's state, which now reflects the estimate of O's state at the next measurement sampling time[3], and pursue a Bayesian update of each of the individual fragments of the "dissected" distribution with a fresh measurement.

[2] For simplicity, we are assuming a discrete-time model here.
[3] We assume that the size of the discrete time step equals the inter-sample time. Otherwise repeat from step 1 until the inter-sample duration is reached.

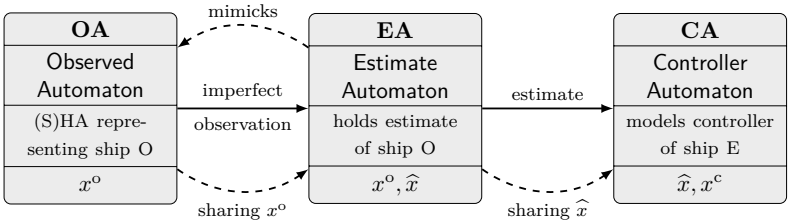
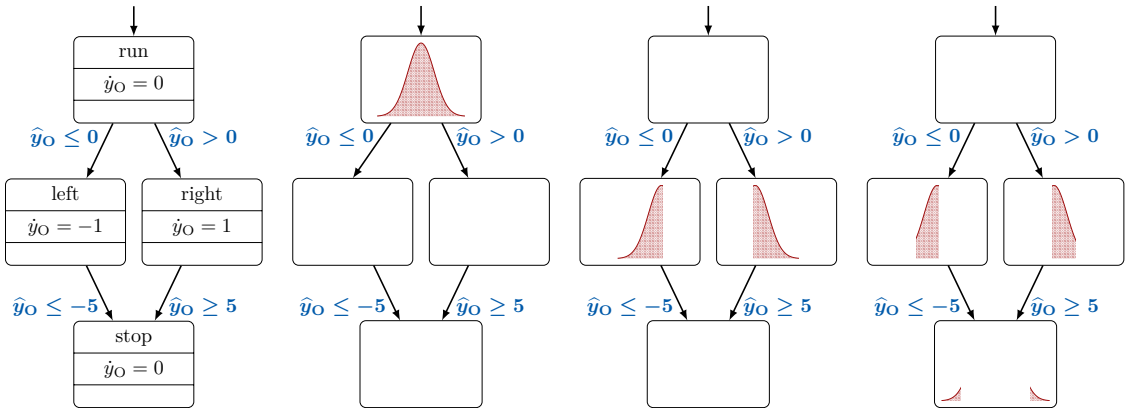


Figure 7 Interplay between different automata modelling unilateral observation. Lowermost section lists types of variables accessed by the automata: x^o for system variables of the observed entity O, $\hat{x}$ for state estimate variables associated to x^o within entity E, and x^c for system variables of the ego entity E.



(a) Control automaton of the observed ship O.

(b) A first noisy position measurement entering the estimation automaton EA.

(c) EA estimating O's next state via a lifting of O's next-state relation.

(d) EA iterating the lifted next-state relation until fixed-point is reached.

Figure 8 Estimating the observed ship's state within an estimation automaton: A noisy position measurement corresponds to a distribution of possible positions for O (b), each of which would drive O's control automaton (a) to a specific state. The EA reflects this by synchronously computing its estimate of O's hybrid state via a lifting of O's next-state relation to estimate distributions (c–d).

5. Build the mixture of the resulting posterior "fragments" paired with their corresponding modes.

For a simple case, where the prior distribution merely stems from a single noisy measurement, steps 1 and 2 of the state extrapolation process are illustrated in Fig. 8b–8d.

- *Controller automata (CA)* represent the controller of the ego system, i.e. of ship E in the example. Such a CA accesses estimate variables provided by EA if control decisions to be drawn involve estimated parameters. The corresponding decisions are "rational" in so far as safety-critical mode switches are based on sufficient confidence that the corresponding guard property is satisfied. Confidence here again relates to the probability that the guard condition gc holds true w.r.t. the estimated distributions: a critical transition is only taken if $P_{x\sim\mu}(gc) \geq \theta$, where θ denotes the required confidence and μ the mixture representing the current estimate of O's state. A safe alternative action (including a stay in the current mode iff its invariant bears sufficient evidence of being satisfied) has to be taken whenever no critical action can be justified with sufficient confidence.

For the sake of a concise presentation, we assume that there is a single measurement process or sensor for each observed parameter which is reflected within the estimate automaton. More differentiated observation processes are possible by, e.g., introducing another class of automata, possibly called *perception automata*, explicitly being responsible for measurements as suggested in [17]. Such an automaton could be a traditional stochastic hybrid automaton being located between OA and EA in Fig. 7 providing a noisy copy the observed physical state to the estimate automaton where the measurement process then depends on the perception automaton's internal state which, in turn, might change w.r.t., i.a., the internal state of the estimate automaton and the controller automaton.

Without digging into further detail of the above automata, it should be obvious that they go well beyond what can be encoded within hybrid automata models with their discrete plus finite-dimensional real-valued state-space:

1. As *controller automata* have to draw inferences about mixtures (representing state estimates) in order to evaluate their guards and invariants, such mixtures must be part of the state-space that controllers can observe.
2. As the state estimation by estimate automata involves active manipulation of such mixtures, these mixtures have to be part of their dynamic state.

State distributions therefore become first class members of the dynamic state themselves. As such state distributions can only rarely be encoded by finite-dimensional vectorial state (e.g., if they are bound to stay within a class of distributions featuring a description by a fixed set of parameters, like with normal distributions), this requires a completely fresh —and much more complex— set-up of the theory of hybrid automata extending beyond finite-dimensional vectorial state towards distributions as states. That this complication is necessary for obtaining accurate verdicts on control performance is witnessed by Figures 4 and 5.

5 Formal definition of the composite model

In the previous section, we presented a concept of formally modelling hybrid automata comprising Bayesian filter techniques for estimates of states of observed entities as well as mixture distributions representing those estimates. In this section, we introduce the formal models of observed automata, estimate automata, controller automata, and their combination into Bayesian hybrid automata. The resulting model is an extension of Bayesian hybrid automata suggested in [16] which were not yet capable of covering hybrid-state dynamics of observed entities within estimates.

5.1 Observed automaton

From an abstract perspective, an observed automaton might be an (almost) arbitrary flavour of traditional hybrid automaton. The assumption of perfect knowledge for observed automata as well as the assumption of unilateral observation allow to restrict the definition of observed automaton to deterministic variants. Aiming at a Gaussian character of estimates, we make two further assumptions which are a result of the fact that arbitrary continuous dynamics would lead to a "deformation" of probability density functions as well as arbitrary updates on discrete transitions would do. We hence assume that

1. the continuous dynamics of OA are constant for each mode, and
2. all updates of the continuous state of OA on a discrete transition is a shift by a constant.

In this article, we distinguish different types of Boolean predicates each of which represents a type of conditions on the continuous state space of a hybrid automaton enabling or disabling discrete control decisions, i.e. a type of transition guards and mode invariants. The first type is the traditional condition which essentially is equivalent to guard and invariant conditions of traditional hybrid automata.

▶ **Definition 1** (Traditional condition)**.** Let $\mathcal{X}$ be a set of n real-valued variables. A *traditional condition* is a predicate $c_\mathrm{t} : \mathbb{R}^n \to \{\mathsf{true}, \mathsf{false}\}$. We denote the set of all traditional conditions by C_t.

We now define observed automata akin to the definition of Kowalewski et al. [21] with the restrictions mentioned above as follows:

▶ **Definition 2** (Syntax of observed automata)**.** An observed automaton is a tuple $OA = (\mathcal{L}^\mathrm{o}, \mathcal{X}^\mathrm{o}, \mathsf{d}^\mathrm{o}, \mathsf{i}^\mathrm{o}, \Delta^\mathrm{o}, \mathsf{g}^\mathrm{o}, \mathsf{u}^\mathrm{o}, \mathcal{I}^\mathrm{o})$ where

- $\mathcal{L}^{\mathrm{o}} = \{\ell_1^{\mathrm{o}}, \cdots, \ell_{l^{\mathrm{o}}}^{\mathrm{o}}\}$ is a finite set of *discrete control modes* a.k.a. control *locations* of the automaton which is the discrete state space of *OA*,
- $\mathcal{X}^{\mathrm{o}} = (x_1^{\mathrm{o}}, \cdots, x_{n^{\mathrm{o}}}^{\mathrm{o}})$ is an ordered finite set of continuous *system variables* conventionally represented as vector $\underline{x}^{\mathrm{o}}$ and spanning the continuous state space of *OA* s.t. a pair $(\ell^{\mathrm{o}}, \mathsf{x}^{\mathrm{o}}) \in \mathcal{L}^{\mathrm{o}} \times \mathbb{R}^{n^{\mathrm{o}}}$ is a state of the automaton with $\mathsf{x}^{\mathrm{o}} : \mathcal{X}^{\mathrm{o}} \to \mathbb{R}$ being a variable valuation which is synonymously used for a concrete vector in $\mathbb{R}^{n^{\mathrm{o}}}$,
- $\mathsf{d}^{\mathrm{o}} : \mathcal{L}^{\mathrm{o}} \to \mathbb{R}^{n^{\mathrm{o}}}$ is a *mode-dependent dynamics* defining the evolution of the continuous system variables $\underline{x}^{\mathrm{o}}$ in relation to the control mode by specifying a differential equation $\dot{\underline{x}}^{\mathrm{o}} = \mathsf{d}(\ell^{\mathrm{o}})$,
- $\mathsf{i}^{\mathrm{o}} : \mathcal{L}^{\mathrm{o}} \to C_{\mathrm{t}}$ is a function describing the *invariants* per control mode, i.e. the part of the continuous state space for which *OA* may remain in the corresponding control mode,
- $\Delta^{\mathrm{o}} \subseteq \mathcal{L}^{\mathrm{o}} \times \mathcal{L}^{\mathrm{o}}$ is a *discrete transition relation* between modes,
- $\mathsf{g}^{\mathrm{o}} : \Delta^{\mathrm{o}} \to C_{\mathrm{t}}$ is a *guard function* decorating each discrete transition with a traditional condition defining the part of the continuous state space for which the corresponding transition is enabled s.t. Δ^{o} is rendered deterministic,
- $\mathsf{u}^{\mathrm{o}} : \Delta^{\mathrm{o}} \to (\mathbb{R}^{n^{\mathrm{o}}} \to \mathbb{R}^{n^{\mathrm{o}}})$ is an *update function* decorating each discrete transition with a function $\mathsf{x} \mapsto \mathsf{x} + \underline{\mathsf{c}}$ with $\underline{\mathsf{c}} \in \mathbb{R}^{n^{\mathrm{o}}}$ updating $\underline{x}^{\mathrm{o}}$ when the transition is taken, and
- $\mathcal{I}^{\mathrm{o}} \in \mathcal{L}^{\mathrm{o}} \times \mathbb{R}^{n^{\mathrm{o}}}$ is the initial state of *OA*.

We denote the set of all states of *OA* by Σ^{o}.

▶ **Definition 3** (Semantics of observed automata). A run of an observed automaton is a sequence $\langle \sigma_0^{\mathrm{o}}, \sigma_1^{\mathrm{o}}, \cdots \rangle$ of states $\sigma_i^{\mathrm{o}} \in \Sigma^{\mathrm{o}}$ with $\sigma_0^{\mathrm{o}} = \mathcal{I}^{\mathrm{o}}$ according to rule InitOA which is defined as

$$\frac{}{\sigma_0 = \mathcal{I}^{\mathrm{o}}}\ \textsc{InitOA}$$

and for all $i \in \mathbb{N}^{>0}$ we have a transition $\sigma_{i-1}^{\mathrm{o}} \longrightarrow_{\textsc{StepOA}} \sigma_i^{\mathrm{o}}$ where the successor state is derived according to rule StepOA which essentially is the concatenation of a discrete transition with a subsequent (discrete) time step. Rule StepOA is defined as follows:

$$\frac{\sigma_{i-1} \qquad \exists \sigma, \sigma_i \in \Sigma^{\mathrm{o}} : \sigma_{i-1} \xrightarrow{\textsc{JumpOA}} \sigma \xrightarrow{\textsc{TimeOA}} \sigma_i}{\sigma_i}\ \textsc{StepOA}$$

Rule JumpOA reflects a sequence of discrete jumps of *OA*. Since *OA* is deterministic and jumps are carried out instantaneously without consuming time, possible jumps enabled after a preceding jump have to be executed before a time step is possible. Hence, rule JumpOA is basically the repeated application of taking a discrete transition, i.e. the repeated application of rule JumpOA*, until a fixed-point is reached:

$$\frac{\begin{array}{c}\sigma \\ \exists \sigma^1, \cdots, \sigma^k \in \Sigma^{\mathrm{o}} : \sigma \xrightarrow{\textsc{JumpOA}^*} \sigma^1 \xrightarrow{\textsc{JumpOA}^*} \cdots \xrightarrow{\textsc{JumpOA}^*} \sigma^{k-1} \xrightarrow{\textsc{JumpOA}^*} \sigma^k \\ \sigma^{k-1} = \sigma^k\end{array}}{\sigma^k}\ \textsc{JumpOA}$$

Rule JumpOA* reflects a single discrete transition of *OA*. If a discrete transition $\left(\ell^{\mathrm{o}}, \ell^{\mathrm{o}\prime}\right)$ is enabled, the update of the continuous state maps x^{o} to $\mathsf{x}^{\mathrm{o}\prime}$, and $\mathsf{x}^{\mathrm{o}\prime}$ satisfies the invariant of $\ell^{\mathrm{o}\prime}$, a jump from $(\ell^{\mathrm{o}}, \mathsf{x}^{\mathrm{o}})$ to $\left(\ell^{\mathrm{o}\prime}, \mathsf{x}^{\mathrm{o}\prime}\right)$ is possible:

$$\frac{\begin{array}{c}(\ell^{\mathrm{o}}, \mathsf{x}^{\mathrm{o}}) \\ \left(\ell^{\mathrm{o}}, \ell^{\mathrm{o}\prime}\right) \in \Delta^{\mathrm{o}} \\ \mathsf{g}^{\mathrm{o}}\left(\left(\ell^{\mathrm{o}}, \ell^{\mathrm{o}\prime}\right)\right)(\mathsf{x}^{\mathrm{o}}) \equiv \mathsf{true} \\ \mathsf{u}^{\mathrm{o}}\left(\left(\ell^{\mathrm{o}}, \ell^{\mathrm{o}\prime}\right)\right)(\mathsf{x}^{\mathrm{o}}) = \mathsf{x}^{\mathrm{o}\prime} \\ \mathsf{i}^{\mathrm{o}}\left(\left(\ell^{\mathrm{o}\prime}\right)\right)(\mathsf{x}^{\mathrm{o}\prime}) \equiv \mathsf{true}\end{array}}{\left(\ell^{\mathrm{o}\prime}, \mathsf{x}^{\mathrm{o}\prime}\right)} \textsc{JumpOA}^*$$

Assume there is a solution $\underline{\mathsf{X}} : [0, \mathsf{t}] \to \mathbb{R}^{n^{\mathrm{o}}}$ to the ordinary differential equation $\mathrm{d}\underline{x}^{\mathrm{o}}/\mathrm{d}t = \mathsf{d}^{\mathrm{o}}(\ell^{\mathrm{o}})$. If $\underline{\mathsf{X}}$ starts in x^{o} and ends in $\mathsf{x}^{\mathrm{o}\prime}$ and all points in the image of $\underline{\mathsf{X}}$ satisfy the invariant of ℓ^{o}, then a time step of length t from $(\ell^{\mathrm{o}}, \mathsf{x}^{\mathrm{o}})$ to $\left(\ell^{\mathrm{o}}, \mathsf{x}^{\mathrm{o}\prime}\right)$ is possible:

$$\frac{(\ell^{\mathrm{o}}, \mathsf{x}^{\mathrm{o}}) \qquad \underline{\mathsf{X}}(0) = \mathsf{x}^{\mathrm{o}} \qquad \underline{\mathsf{X}}(\mathsf{t}) = \mathsf{x}^{\mathrm{o}\prime} \qquad \forall \mathsf{t}' \in [0, \mathsf{t}] : \mathsf{i}^{\mathrm{o}}(\ell^{\mathrm{o}})(\underline{\mathsf{X}}(\mathsf{t}')) \equiv \mathsf{true}}{\left(\ell^{\mathrm{o}}, \mathsf{x}^{\mathrm{o}\prime}\right)} \textsc{TimeOA}$$

We assume $\mathsf{t} \in \mathbb{R}^{>0}$ to be arbitrary but fixed.

The components of the observed automaton for our maritime example from Figure 6 could be defined as follows:

$$\mathcal{L}^{\mathrm{o}} := \{\mathrm{straight}^{\mathrm{o}}, \mathrm{left}^{\mathrm{o}}, \mathrm{right}^{\mathrm{o}}, \mathrm{stop}^{\mathrm{o}}\} \tag{3a}$$

$$\mathcal{X}^{\mathrm{o}} := (x_{\mathrm{O}}, y_{\mathrm{O}}) \tag{3b}$$

$$\mathsf{d}^{\mathrm{o}}(\ell^{\mathrm{o}}) := \begin{cases} [\dot{x}_{\mathrm{O}} = -1, \dot{y}_{\mathrm{O}} = 0]^T & \text{iff } \ell^{\mathrm{o}} = \mathrm{straight}^{\mathrm{o}} \\ [\dot{x}_{\mathrm{O}} = -1, \dot{y}_{\mathrm{O}} = -1]^T & \text{iff } \ell^{\mathrm{o}} = \mathrm{left}^{\mathrm{o}} \\ [\dot{x}_{\mathrm{O}} = -1, \dot{y}_{\mathrm{O}} = 1]^T & \text{iff } \ell^{\mathrm{o}} = \mathrm{right}^{\mathrm{o}} \\ [\dot{x}_{\mathrm{O}} = 0, \dot{y}_{\mathrm{O}} = 0]^T & \text{iff } \ell^{\mathrm{o}} = \mathrm{stop}^{\mathrm{o}} \end{cases} \tag{3c}$$

$$\mathsf{i}^{\mathrm{o}}(\ell^{\mathrm{o}}) := \begin{cases} x_{\mathrm{O}} \geq 85 & \text{iff } \ell^{\mathrm{o}} = \mathrm{straight}^{\mathrm{o}} \\ y_{\mathrm{O}} \geq -5 & \text{iff } \ell^{\mathrm{o}} = \mathrm{left}^{\mathrm{o}} \\ y_{\mathrm{O}} \leq 5 & \text{iff } \ell^{\mathrm{o}} = \mathrm{right}^{\mathrm{o}} \\ \mathsf{true} & \text{iff } \ell^{\mathrm{o}} = \mathrm{stop}^{\mathrm{o}} \end{cases} \tag{3d}$$

$$\Delta^{\mathrm{o}} := \{(\mathrm{straight}^{\mathrm{o}}, \mathrm{left}^{\mathrm{o}}), (\mathrm{straight}^{\mathrm{o}}, \mathrm{right}^{\mathrm{o}}), (\mathrm{left}^{\mathrm{o}}, \mathrm{stop}^{\mathrm{o}}), (\mathrm{right}^{\mathrm{o}}, \mathrm{stop}^{\mathrm{o}})\} \tag{3e}$$

$$\mathsf{g}^{\mathrm{o}}(\delta) := \begin{cases} x_{\mathrm{O}} \leq 85 \wedge y_{\mathrm{O}} < 0 & \text{iff } \delta = (\mathrm{straight}^{\mathrm{o}}, \mathrm{left}^{\mathrm{o}}) \\ x_{\mathrm{O}} \leq 85 \wedge y_{\mathrm{O}} \geq 0 & \text{iff } \delta = (\mathrm{straight}^{\mathrm{o}}, \mathrm{right}^{\mathrm{o}}) \\ y_{\mathrm{O}} \leq -5 & \text{iff } \delta = (\mathrm{left}^{\mathrm{o}}, \mathrm{stop}^{\mathrm{o}}) \\ y_{\mathrm{O}} \geq 5 & \text{iff } \delta = (\mathrm{right}^{\mathrm{o}}, \mathrm{stop}^{\mathrm{o}}) \end{cases} \tag{3f}$$

$$\mathsf{u}^{\mathrm{o}}(\mathsf{x}_{\mathrm{O}}, \mathsf{y}_{\mathrm{O}}) := (\mathsf{x}_{\mathrm{O}}, \mathsf{y}_{\mathrm{O}}) \text{ for all } \delta \in \Delta^{\mathrm{o}} \tag{3g}$$

$$\mathcal{I}^{\mathrm{o}} := (\mathrm{straight}^{\mathrm{o}}, (x_{\mathrm{O}} = 100, y_{\mathrm{O}} = 0)) \tag{3h}$$

The automaton is illustrated in Figure 9.

5.2 Estimate automaton

Estimate automata govern the estimate of the observed system variables which are, essentially, a combination of information from a history of measurement results according to Bayes' theorem [29]. Estimate automata reflect the process of taking possibly error-afflicted measurements and applying the Bayes filter.

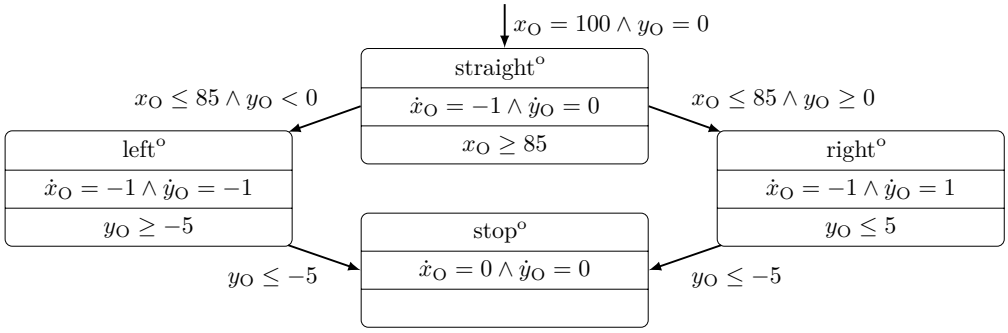


Figure 9 Observed automaton for example from Fig. 6. Ship O switches to mode 'left°' if its lateral position is smaller than zero when entering the passing place, and to mode 'right°' otherwise. It stops when it reaches a shore.

An essential part of such a filter is to extrapolate the estimate along the continuous dynamics between two measurements. An estimate can be considered as a set of states each of which can be regarded to be the true state with some likelihood. Each of these states has to be evolved along the correct dynamics. Unfortunately, in the hybrid automata setting, this is not necessarily the same dynamics for all states as already indicated in Sect. 4.2: assume a set $A \subset \mathbb{R}$ enabling a discrete transition (ℓ, ℓ') while for $B = \mathbb{R} \setminus A$ the transition is not enabled. Then, the observed automaton may switch to ℓ' for all $\mathsf{x} \in A$ while it has to remain in ℓ for all $\mathsf{x} \in B$. Consequently, the dynamics of mode ℓ has to be applied to $\mathsf{x} \in A$ as well as the differential equations of mode ℓ' govern extrapolations of trajectories starting in B.

This setting can be taken into account by interpreting estimates within estimate automata as a list of sets of continuous states annotated by their likelihood to be the true state in form of probability density functions as well as the control mode they are governed by. Such a list is then basically a mixture distribution where each mixture component is annotated by a control mode. In the example above, a discrete jump would lead to two mixture components $\hat{\mathsf{x}}_\ell$ and $\hat{\mathsf{x}}_{\ell'}$ for ℓ and ℓ' where the support of the (re-normalised) estimate $\hat{\mathsf{x}}_\ell$ is restricted to A while the support of $\hat{\mathsf{x}}_{\ell'}$ is restricted to B. We call such an extended mixture distribution a mixture estimate.

We now formally introduce components used by estimate automata including mixture estimates before providing the definition of the estimate automaton itself.

▶ **Definition 4** (Mixture estimate)**.** A *mixture estimate* $\mu = (\mu_1, \cdots, \mu_k)$ is a finite ordered set of mixture components where each component is an n-variate probability density function $\mu_i : \mathbb{R}^n \to \mathbb{R}$ and is labelled by an automata location and the weight of the component, i.e. we have labelling functions $\lambda_\ell : \mathcal{P}(\mathbb{R}^n) \to \mathcal{L}$ and $\lambda_P : \mathcal{P}(\mathbb{R}^n) \to (0, 1]$ s.t. $\sum_{i=0}^{k} \lambda_P(\hat{\mathsf{x}}_i) = 1.0$ where $\mathcal{P}(\mathbb{R}^n)$ is the set of all probability density functions over $\mathbb{R}^n$.

A mixture estimate can be considered as a probability density function which is defined as the weighted sum of its components, i.e.

$$\mu(\mathsf{x}) = \sum_{i=1}^{k} \mu_i(\mathsf{x}) \cdot \lambda_P(\mu_i) \tag{4}$$

where $\sum_{i=1}^{k} \lambda_P(\mu_i) = 1.0$. We denote the set of all mixture estimates over $\mathbb{R}^n$ by $\mathcal{M}(\mathbb{R}^n)$.

Uncertain conditions allow to model traditional conditions in the control strategy of the observed entity from the observer's perspective where a control decision is made with uncertainty since the continuous state satisfying or unsatisfying the corresponding condition is only estimated. However, they are rather a function than a predicate: for a given estimate $p \in \mathcal{P}(\mathbb{R}^n)$ and a traditional condition $c_\mathrm{t} \in C_\mathrm{t}$, an uncertain condition returns a normalised copy p' of p for which the support $\mathrm{supp}(p')$ is restricted to those values satisfying c_t.

▶ **Definition 5** (Uncertain condition)**.** An *uncertain condition* is a function $c_\mathrm{u} : \mathcal{P}(\mathbb{R}^n) \times C_\mathrm{t} \to \mathcal{P}(\mathbb{R}^n)$ with $c_\mathrm{u}(p, c_\mathrm{t}) \mapsto p'$ where the partial estimate over those values $\mathsf{x} \in \mathbb{R}^n$ that satisfy c_t is defined as

$$p''(\mathsf{x}) = \begin{cases} p(\mathsf{x}) & \text{iff } c_\mathrm{t}(\mathsf{x}) \equiv \mathsf{true} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

and lifted to a probability density function

$$p'(\mathsf{x}) = p''(\mathsf{x}) \cdot \frac{1}{\beta} \tag{6}$$

by re-normalisation based on the *branch probability*

$$\beta = \int p''(\mathsf{x}) \, \mathsf{dx} \tag{7}$$

which describes the probability that c_t is satisfied.

An uncertain jump then describes the effect of the discrete transition relation of the observed automaton to the observer's mixture estimate.

▶ **Definition 6** (Uncertain jump)**.** Assume a hybrid automaton with a discrete transition relation Δ, a guard function g, and an update function u. An *uncertain jump* is a function $c_\mathrm{u}^\mathrm{m} : \mathcal{M}(\mathbb{R}^n) \to \mathcal{M}(\mathbb{R}^n)$ that takes a mixture estimate and applies the uncertain condition on all mixture components before the discrete jump of the continuous state space is applied to the resulting components, thereby generating a new mixture estimate:

$$\mu \mapsto \bigcup_{\mu_i \in \mu} \left(\bigcup_{\delta \in \Delta} \underbrace{\mathsf{u}^\mathrm{o}(\delta)(c_\mathrm{u}(\mu_i, \mathsf{g}^\mathrm{o}(\delta)))}_{=\mu_i'} \right) \cup \underbrace{\left\{ c_\mathrm{u}\left(\mu_i, \neg \bigvee_{\delta \in \Delta} \mathsf{g}^\mathrm{o}(\delta)\right) \right\}}_{=\mu_i''} \tag{8}$$

where μ_i' are those components obtained from following a discrete transition, μ_i'' is the component obtained from that part of the continuous state space for which no transition is enabled, δ is an outbound transition of the mode annotated to μ_i (i.e. $\delta = (\ell, \ell') : \lambda_\ell(\mu_i) = \ell$), and $\mathsf{u}^\mathrm{o}(\delta)(\mu_j(\mathsf{x})) = \mu_j(2\mathsf{x} - \mathsf{u}^\mathrm{o}(\delta)(\mathsf{x}))$ is the lifting of the discontinuous update of the continuous state when a discrete transition is taken to probability density functions. The new mixture components are labelled with the target location of the corresponding transition, i.e. $\lambda_\ell(\mu_i') = \ell'$ and $\lambda_\ell(\mu_i'') = \lambda_\ell(\ell)$ for the mixture component representing states remaining the in source location. Furthermore, each mixture component is labelled by a weight which is the probability that the new component is the true estimate, i.e. that the run of the observed automaton follows the sequence of discrete transitions from which the component is obtained. We consequently have $\lambda_P(\hat{\mathsf{x}}') = \lambda_P(\mu_i) \cdot \beta$ (and $\lambda_P(\hat{\mathsf{x}}'') = \lambda_P(\mu_i) \cdot \beta$, respectively) where β is calculated during evaluation of c_u according to Equation 7. In order to avoid a division by zero, components with $\beta = 0.0$ (hence components obtained from impossible discrete transitions) are simply dropped.

Perception is usually via sensors which, in general, is afflicted by errors. We model the process of drawing a measurement by a sensor function.

▶ **Definition 7** (Sensor). Let $e \in \mathbb{R}^n$ be a random measurement error drawn according to an error distribution $\hat{e} \in \mathcal{P}(\mathbb{R}^n)$, denoted by $e \sim \hat{e}$. A *sensor* is a function $s : \mathbb{R}^n \to \mathbb{R}^n$ with $\mathsf{x} \mapsto \mathsf{x} + e$ and $e \sim \hat{e}$.

Bayes filters provide a recursive calculation of estimates combining "knowledge" from a sequence of measurements. In case of purely linear dynamics and a normally distributed measurement error, i.e. $e \sim \mathcal{N}(\mathsf{x}, \sigma^2)$, a Kálmán-filter would be an instance of a Bayes filter yielding best estimates of the observed parameters. Such a dynamics and error model facilitates the implementation of concrete instances. However, as our findings do not hinge on that particular setup, we describe the application of such a filter in a very general manner by applying Bayes' rule:

▶ **Definition 8** (Filter). Let $\mathsf{r}_k \in \mathbb{R}^n$ be the k-th measurement result obtained from a sensor s while $p_k(\mathsf{x}) = p(\mathsf{x} \mid \mathsf{r}_k, \cdots, \mathsf{r}_1)$ is the estimate of $\mathsf{x} \in \mathbb{R}^n$ after k measurements. Furthermore, $\hat{r}$ is the conditional probability distribution $\hat{r}(\mathsf{r} \mid \mathsf{x}) = \hat{e}(\mathsf{r} + \mathsf{x})$ of measurement results given x is the true parameter. By *filter* we denote a function $f : \mathcal{P}(\mathbb{R}^n) \times \mathbb{R}^n \to \mathcal{P}(\mathbb{R}^n)$ with

$$(p_{k-1}, \mathsf{r}_k) \mapsto p_k(\mathsf{x} \mid \mathsf{r}_k, \cdots, \mathsf{r}_1) = \frac{\hat{r}(\mathsf{r}_k \mid \mathsf{x}) \cdot p_{k-1}(\mathsf{x} \mid \mathsf{r}_{k-1}, \cdots, \mathsf{r}_1)}{\hat{r}_{\mathbb{R}}(\mathsf{r}_k)} \tag{9}$$

where $\hat{r}_{\mathbb{R}}(\mathsf{r}_k) = \int \hat{r}(\mathsf{r}_k \mid \mathsf{x}^*) \cdot p_{k-1}(\mathsf{x}^* \mid \mathsf{r}_{k-1}, \cdots, \mathsf{r}_1) \mathsf{d}\mathsf{x}^*$.

Now that we have sensors and filters, we can define a measurement function as the combination of a sensor and a filter.

▶ **Definition 9** (Measurement action). Let s be a sensor while f is a filter function. A *measurement action* is a function $m : \mathcal{M}(\mathbb{R}^n) \times \mathbb{R}^n \to \mathcal{M}(\mathbb{R}^n)$ with $(\mu, \mathsf{x}) \mapsto \{f(\mu_i, \mathsf{r}) \mid \mu_i \in \mu\}$ for a $\mathsf{r} = s(\mathsf{x})$ fixed for all components μ_i of μ.

An estimate automaton is, essentially, a copy of the observed automaton augmented by measurement actions, estimate variables accommodating mixture estimates, and a filter function as well as a semantics in form of sequences of estimates.

▶ **Definition 10** (Syntax of estimate automata). An *estimate automaton* is a tuple $EA = (\mathcal{L}^{\mathrm{e}}, \mathcal{X}^{\mathrm{o}}, \hat{\mathcal{X}}, m, \mathsf{d}^{\mathrm{e}}, \mathsf{i}^{\mathrm{e}}, \Delta^{\mathrm{e}}, \mathsf{g}^{\mathrm{e}}, \mathsf{u}^{\mathrm{e}}, \mathcal{I}^{\mathrm{e}})$ where the elements annotated with e are copies of the corresponding elements of OA. The set $\mathcal{X}^{\mathrm{o}}$ of system variables is read-only shared with OA, i.e. a state change of $\mathcal{X}^{\mathrm{o}}$ in EA is directly passed through to $\mathcal{X}^{\mathrm{o}}$ in EA. Furthermore,

- $\hat{\mathcal{X}} = (\hat{x}_1, \cdots, \hat{x}_{n^{\mathrm{o}}})$ is an ordered finite set of *estimate variables* conventionally represented by a vector $\underline{\hat{x}}$ spanning the stochastic state space of EA s.t. $\hat{\mathsf{x}}$ is a state of EA where $\hat{\mathsf{x}} : \hat{\mathcal{X}} \to \mathcal{M}(\mathbb{R})$ is the corresponding variable valuation specifying the marginal distributions of a mixture estimate in $\mathcal{M}(\mathbb{R}^{n^{\mathrm{o}}})$ for which $\hat{\mathsf{x}}$ is synonymously used and $\hat{x}_i$ is associated to x_i^{o} in the sense that $\hat{x}_i$ accommodates an estimate of x_i^{o} in OA for all $i \in \{1, \cdots, n^{\mathrm{o}}\}$, and
- m is a *measurement action*.

We denote the set of all states of EA by Σ^{e}.

▶ **Definition 11** (Semantics of estimate automata). A run of an estimate automaton is a sequence $\langle \hat{\mathsf{x}}_0, \hat{\mathsf{x}}_1, \cdots \rangle$ of mixture estimates where $\hat{\mathsf{x}}_0$ is deduced via rule InitEA which allows to derive the initial state of $\underline{\hat{x}}$ given the initial state $\sigma_0^{\mathrm{o}} = \mathcal{I}^{\mathrm{e}}$ of OA:

$$\frac{\sigma_0^{\mathrm{o}} = (\ell_0^{\mathrm{o}}, \mathsf{x}_0^{\mathrm{o}}) \qquad \mathsf{r}_0 = s(\mathsf{x}_0^{\mathrm{o}}) \qquad \hat{\mathsf{x}} = \mu(\mathsf{x} \mid \mathsf{r}_0) = \{\mu_1(\mathsf{x} \mid \mathsf{r}_0) = \hat{e}(\mathsf{r}_0 - \mathsf{x}_0^{\mathrm{o}})\}}{\hat{\mathsf{x}}} \text{ InitEA}$$

with $\lambda_\ell(\hat{\mathsf{x}}) = \ell_0^{\mathrm{o}}$ and $\lambda_P(\hat{\mathsf{x}}) = 1.0$ where s is the sensor of m and $\hat{e}$ is the error distribution of s.

For all $i \in \mathbb{N}^{>0}$ we then have a transition $\hat{\mathsf{x}}_{i-1} \longrightarrow_{\textsc{StepEA}} \hat{\mathsf{x}}_i$ where the successor estimate is derived according to rule StepEA which essentially is the concatenation of a discrete transition with a subsequent (discrete) time step:

$$\frac{\hat{\mathsf{x}}_{i-1} \qquad \exists \hat{\mathsf{x}}^*, \hat{\mathsf{x}}_i : \hat{\mathsf{x}}_{i-1} \xrightarrow{\textsc{JumpEA}} \hat{\mathsf{x}}^* \xrightarrow{\textsc{TimeEA}} \hat{\mathsf{x}}_i}{\hat{\mathsf{x}}_i}\ \textsc{StepEA}$$

Rule JumpEA is an abbreviation for two steps:

1. updating the mixture estimate via rule Measure and
2. applying the discrete dynamics via rule TransEA until a fixed-point is reached (akin to rule JumpOA).

$$\frac{\begin{array}{c}\hat{\mathsf{x}} \\ \exists \hat{\mathsf{x}}^0 : \hat{\mathsf{x}} \xrightarrow{\textsc{Measure}} \hat{\mathsf{x}}^0 \\ \exists \hat{\mathsf{x}}^1, \cdots, \hat{\mathsf{x}}^k : \hat{\mathsf{x}}^0 \xrightarrow{\textsc{TransEA}} \hat{\mathsf{x}}^1 \xrightarrow{\textsc{TransEA}} \cdots \xrightarrow{\textsc{TransEA}} \hat{\mathsf{x}}^{k-1} \xrightarrow{\textsc{TransEA}} \hat{\mathsf{x}}^k \\ \hat{\mathsf{x}}^{k-1} = \hat{\mathsf{x}}^k \end{array}}{\hat{\mathsf{x}}^k}\ \textsc{JumpEA}$$

Rule Measure updates the estimate according by applying the measurement action m to each component of the mixture estimate:

$$\frac{\mathsf{x}^{\mathrm{o}} \qquad \hat{\mathsf{x}} \qquad \hat{\mathsf{x}}' = m(\hat{\mathsf{x}}, \mathsf{x}^{\mathrm{o}})}{\hat{\mathsf{x}}'}\ \textsc{Measure}$$

Rule TransEA reflects the effect of *OA*'s control laws on the mixture estimate according to an uncertain jump.

$$\frac{\hat{\mathsf{x}} \qquad \hat{\mathsf{x}}' = c_{\mathrm{u}}^{\mathrm{m}}(\hat{\mathsf{x}})}{\hat{\mathsf{x}}'}\ \textsc{TransEA}$$

The rule TimeEA describes a discrete time step, i.e. the application of the continuous dynamics of OA (and EA, respectively) for t time units to the carrier of each mixture component:

$$\frac{\hat{\mathsf{x}} \qquad \hat{\mathsf{x}}' = \{\mu_i(2\hat{\mathsf{x}} - \hat{\mathsf{x}}^*) \mid \mu_i \in \hat{\mathsf{x}}\}}{\hat{\mathsf{x}}'}\ \textsc{TimeEA}$$

where for $\hat{\mathsf{x}}^*$ there is a solution solution $\underline{\mathsf{X}} : [0, \mathsf{t}] \to \mathbb{R}^{n^{\mathrm{o}}}$ to the ordinary differential equation $\mathsf{dx}/\mathsf{dt} = \mathsf{d}^{\mathrm{o}}(\lambda_\ell(\hat{\mathsf{x}}))$ with $\underline{\mathsf{X}}(0) = \mathsf{x}$ and $\underline{\mathsf{X}}(\mathsf{t}) = \mathsf{x}^*$ and $\mathsf{i}^{\mathrm{o}}(\lambda_\ell(\mathsf{x}))(\underline{\mathsf{X}}(\mathsf{t}')) = \mathsf{true}$ for all $\mathsf{t}' \in [0, \mathsf{t}]$. We assume $\mathsf{t} \in \mathbb{R}^{>0}$ to be arbitrary but fixed.

For our maritime example from Figure 6, the estimate automaton would be a copy of the observed automaton defined in (3) extended by a set $\hat{\mathcal{X}}$ of estimate variables and measurement action m as follows:

$$\hat{\mathcal{X}} := (\hat{y}_{\mathrm{O}}) \text{ with } \hat{y}_{\mathrm{O}} = \{\mu_1, \cdots, \mu_k\} \text{ as defined in Def. 4 and } \mu_i \text{ is a normal distribution over } \mathbb{R} \tag{10i}$$

$$m(\hat{\mathsf{y}}_{\mathrm{O}}, \mathsf{r}) := \{f(\mu_i, \mathsf{r}) \mid \mu_i \in \hat{\mathsf{y}}_{\mathrm{O}}\} \text{ where } f \text{ is a Kálmán filter , and } \mathsf{r} \text{ is a measurement result obtained by a sensor } s \text{ as defined in Def. 7 where the measurement error is normally distributed, i.e. } \hat{e} = \mathcal{N}(0, 1) \tag{10j}$$

Note that the main difference between observed automaton and estimate automaton is its interpretation, i.e. within the semantics (see Def. 11). Its graphical representation consequently is identical to Fig. 9.

5.3 Controller automaton

A controller automaton models the controller of the ego system which accesses the estimates of the estimate automaton in order to make rational decisions in the sense that mode switches based on estimated parameters are executed only in case of sufficient confidence that the corresponding guard property is satisfied. In this context, confidence is the probability that the guard property is satisfied w.r.t. the current mixture estimate, i.e. the probability distribution representing the belief about the state of the observed entity. We denote the class of predicates for such control decisions by "rational conditions".

▶ **Definition 12** (Rational condition)**.** Let $\mathcal{X}$ be a set of n system variables while $\hat{\mathcal{X}}$ is the set of n estimate variables s.t. $\hat{\mathsf{x}}_i$ accommodates an estimate of the value of x_i. Furthermore, let $c_\mathrm{t} \in C_\mathrm{t}$ be a traditional condition. A *rational condition* is a predicate $c_\mathrm{r} : \mathcal{M}(\mathbb{R}^n) \to \{\mathsf{true}, \mathsf{false}\}$ s.t.

$$c_\mathrm{r}(\mu) \equiv \begin{cases} \mathsf{true} & \text{iff } P_{\hat{\mathsf{x}}}(c_\mathrm{t} \equiv \mathsf{true}) \bowtie \varepsilon, \text{ and} \\ \mathsf{false} & \text{otherwise} \end{cases} \tag{11}$$

with $\bowtie \in \{\geq, >\}$, $c_\mathrm{t} \in C_\mathrm{t}$, and $\varepsilon \in [0, 1]$ and

$$P_{\hat{\mathsf{x}}}(c_\mathrm{t} \equiv \mathsf{true}) = \int \mathbf{1}_{c_\mathrm{t}}(\mathsf{x}) \cdot \mu(\mathsf{x})\, \mathsf{dx} \tag{12}$$

where

$$\mathbf{1}_{c_\mathrm{t}}(\mathsf{x}) = \begin{cases} 1 & \text{iff } c_\mathrm{t}(\mathsf{x}) \equiv \mathsf{true}, \text{ and} \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

Hence, a rational condition is satisfied iff the corresponding traditional condition c_t is satisfied with a probability larger than (or equal to) ε w.r.t. mixture estimate μ. We denote the set of all rational conditions by C_r.

As controller automata are not observed in the current setting, we can relax the restrictions we made for observed automata and allow non-determinism and arbitrary continuous dynamics and updates. However, for the sake of a more concise presentation, we perpetuate those restrictions.

▶ **Definition 13** (Syntax of controller automata)**.** A *controller automaton* is a a tuple $CA = (\mathcal{L}^\mathrm{c}, \mathcal{X}^\mathrm{c}, \mathsf{d}^\mathrm{c}, \mathsf{i}^\mathrm{c}, \Delta^\mathrm{c}, \mathsf{g}^\mathrm{c}, \mathsf{u}^\mathrm{c}, \mathcal{I}^\mathrm{c})$ where

- $\mathcal{L}^\mathrm{c} = \{\ell^\mathrm{c}_1, \cdots, \ell^\mathrm{c}_{l^\mathrm{c}}\}$ with $\mathcal{L}^\mathrm{c} \cap \mathcal{L}^\mathrm{o} = \emptyset$ is a finite set of *discrete control modes* a.k.a. control *locations* of the automaton which is the discrete state space of CA,
- $\mathcal{X}^\mathrm{c} = (x^\mathrm{c}_1, \cdots, x^\mathrm{c}_{n^\mathrm{c}})$ with $\mathcal{X}^\mathrm{c} \cap \mathcal{X}^\mathrm{o} = \emptyset$ is an ordered finite set of continuous *system variables* conventionally represented as vector $\underline{x}^\mathrm{c}$ and spanning the continuous state space of CA s.t. a pair $(\ell^\mathrm{c}, \mathsf{x}^\mathrm{c}) \in \mathcal{L}^\mathrm{c} \times \mathbb{R}^{n^\mathrm{c}}$ is a state of the automaton with $\mathsf{x}^\mathrm{c} : \mathcal{X}^\mathrm{c} \to \mathbb{R}$ being a variable valuation which is synonymously used for a concrete vector in $\mathbb{R}^{n^\mathrm{c}}$,
- $\hat{\mathcal{X}} = (\hat{x}_1, \cdots, \hat{x}_{n^\mathrm{o}})$ is an ordered finite set of estimate variables as defined in Def. 10 and read-only shared with EA, i.e. a state change of $\hat{\mathcal{X}}$ in EA is directly passed through to $\hat{\mathcal{X}}$ in CA,
- $\mathsf{d}^\mathrm{c} : \mathcal{L}^\mathrm{c} \to \mathbb{R}^{n^\mathrm{c}}$ is a *mode-dependent dynamics* defining the evolution of the continuous system variables $\underline{x}^\mathrm{c}$ in relation to the control mode by specifying a differential equation $\dot{\underline{x}}^\mathrm{c} = \mathsf{d}(\ell^\mathrm{c})$,
- $\mathsf{i}^\mathrm{c} : \mathcal{L}^\mathrm{c} \to C_\mathrm{t} \times C_\mathrm{r}$ is a function describing the *invariants* per control mode, i.e. the part of the continuous state space including the estimates for which CA may remain in the corresponding control mode,
- $\Delta \subseteq \mathcal{L}^\mathrm{c} \times \mathcal{L}^\mathrm{c}$ is a *discrete transition relation* between modes,

- $\mathsf{g}^{\mathrm{c}} : \Delta^{\mathrm{c}} \to C_{\mathrm{t}} \times C_{\mathrm{r}}$ is a *guard function* decorating each discrete transition with a traditional condition or a rational condition defining the part of the stochastic state space of *EA* as well as the part of the continuous state space of *CA* for which the corresponding transition is enabled s.t. Δ^{c} is rendered deterministic,
- $\mathsf{u}^{\mathrm{c}} : \Delta^{\mathrm{c}} \to (\mathbb{R}^{n^{\mathrm{c}}} \to \mathbb{R}^{n^{\mathrm{c}}})$ is an *update function* decorating each discrete transition with a function $\mathsf{x} \mapsto \mathsf{x} + \underline{\mathsf{c}}$ with $\underline{\mathsf{c}} \in \mathbb{R}^{n^{\mathrm{c}}}$ updating $\underline{x}^{\mathrm{c}}$ when the transition is taken, and
- and $\mathcal{I} \in \mathcal{L}^{\mathrm{c}} \times \mathbb{R}^{n^{\mathrm{c}}}$ is the initial state of *CA*.

We denote the set of all states of *CA* by Σ^{c}.

▶ **Definition 14** (Semantics of controller automata)**.** A run of an estimate automaton is a sequence $\langle \sigma_0^{\mathrm{c}}, \sigma_1^{\mathrm{c}}, \cdots \rangle$ of states $\sigma_i^{\mathrm{c}} \in \Sigma^{\mathrm{c}}$ obeying deduction rules defined analogously to Def. 3 except for rules incorporating guards and invariants. We hence assume rules InitCA, StepCA, and JumpCA to be adopted straightforwardly from the semantics of the observed automaton where JumpCA refers to JumpCA* which respect both traditional and rational conditions for guards and invariants:

$$\frac{\begin{array}{c}(\ell^{\mathrm{c}}, \mathsf{x}^{\mathrm{c}}) \\ \hat{\mathsf{x}} \\ (\ell^{\mathrm{c}}, \ell^{\mathrm{c}\prime}) \in \Delta^{\mathrm{c}} \\ \gamma\big((\ell^{\mathrm{c}}, \ell^{\mathrm{c}\prime})\big)(\mathsf{x}^{\mathrm{c}}, \hat{\mathsf{x}}) \equiv \mathsf{true} \\ \mathsf{u}^{\mathrm{c}}\big((\ell^{\mathrm{c}}, \ell^{\mathrm{c}\prime})\big)(\mathsf{x}^{\mathrm{c}}) = \mathsf{x}^{\mathrm{c}\prime} \\ \iota\big(\ell^{\mathrm{c}\prime}\big)\big(\mathsf{x}^{\mathrm{c}\prime}, \hat{\mathsf{x}}\big) \equiv \mathsf{true}\end{array}}{\big(\ell^{\mathrm{c}\prime}, \mathsf{x}^{\mathrm{c}\prime}\big)}\ \textsc{JumpCA}^*$$

where

$$\gamma(\delta)(\mathsf{x}^{\mathrm{c}}, \hat{\mathsf{x}}) \equiv c_{\mathrm{t}}(\mathsf{x}^{\mathrm{c}}) \wedge c_{\mathrm{r}}(\hat{\mathsf{x}}) \ \text{ with } \mathsf{g}^{\mathrm{c}}(\delta) = (c_{\mathrm{t}}, c_{\mathrm{r}}) \tag{14}$$

and

$$\iota(\ell)(\mathsf{x}^{\mathrm{c}}, \hat{\mathsf{x}}) \equiv c_{\mathrm{t}}(\mathsf{x}^{\mathrm{c}}) \wedge c_{\mathrm{r}}(\hat{\mathsf{x}}) \ \text{ with } \mathsf{i}^{\mathrm{c}}(\delta) = (c_{\mathrm{t}}, c_{\mathrm{r}}) \tag{15}$$

lift guard and invariant conditions to predicates $\mathbb{R}^{n^{\mathrm{c}}} \times \mathcal{M}(\mathbb{R}^{n^{\mathrm{o}}}) \to \{\mathsf{true}, \mathsf{false}\}$.

Analogously, rule TimeCA reflects a (discrete) time step using predicates $\iota(\ell)$:

$$\frac{(\ell^{\mathrm{c}}, \mathsf{x}^{\mathrm{c}}) \quad \hat{\mathsf{x}} \quad \underline{\mathsf{X}}(0) = \mathsf{x}^{\mathrm{c}} \quad \underline{\mathsf{X}}(\mathsf{t}) = \mathsf{x}^{\mathrm{c}\prime} \quad \forall \mathsf{t}' \in [0, \mathsf{t}] : \iota(\ell^{\mathrm{c}})(\underline{\mathsf{X}}(\mathsf{t}'), \hat{\mathsf{x}}) \equiv \mathsf{true}}{\big(\ell^{\mathrm{c}}, \mathsf{x}^{\mathrm{c}\prime}\big)}\ \textsc{TimeCA}$$

for an arbitrary but fixed $\mathsf{t} \in \mathbb{R}^{>0}$.

A controller automaton for ship E in our example from Fig. 6 is shown in Fig. 10. At position $x_{\mathrm{E}} = 15$ it makes its decision to turn to a shore when it has sufficient confidence that for O the opposite shore is closer (and thus that O is turning to that shore). We assume that E performs an emergency stop if no sufficient confidence has been built up at that point. The corresponding automaton components are defined as follows:

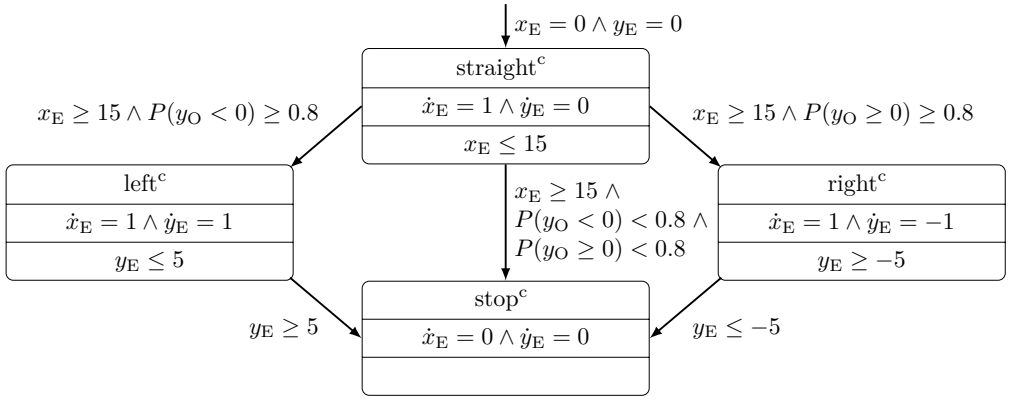


Figure 10 Controller automaton for the maritime example in Fig. 6. Ship E switches to mode 'left^c' when entering the passing place if it has sufficient confidence that ship O turns to the right shore (from E's perspective), and to mode 'right^c' if it has sufficient confidence, that O turns to the left shore. E stops if none of the two options have sufficient confidence. It also stops when reaching a shore.

$$\mathcal{L}^c := \{\text{straight}^c, \text{left}^c, \text{right}^c, \text{stop}^c\} \tag{16a}$$

$$\mathcal{X}^c := (x_E, y_E) \tag{16b}$$

$$\mathsf{d}^c(\ell^c) := \begin{cases} [\dot{x}_E = 1, \dot{y}_E = 0]^T & \text{iff } \ell^e = \text{straight}^c \\ [\dot{x}_E = 1, \dot{y}_E = 1]^T & \text{iff } \ell^c = \text{left}^c \\ [\dot{x}_E = 1, \dot{y}_E = -1]^T & \text{iff } \ell^c = \text{right}^c \\ [\dot{x}_E = 0, \dot{y}_E = 0]^T & \text{iff } \ell^c = \text{stop}^c \end{cases} \tag{16c}$$

$$\mathsf{i}^c(\ell^c) := \begin{cases} (x_E \leq 15, \mathsf{true}) & \text{iff } \ell^c = \text{straight}^c \\ (y_E \leq 5, \mathsf{true}) & \text{iff } \ell^c = \text{left}^c \\ (y_E \geq -5, \mathsf{true}) & \text{iff } \ell^c = \text{right}^c \\ (\mathsf{true}, \mathsf{true}) & \text{iff } \ell^c = \text{stop}^c \end{cases} \tag{16d}$$

$$\Delta^c := \{(\text{straight}^c, \text{left}^c), (\text{straight}^c, \text{right}^c), (\text{left}^c, \text{stop}^c), (\text{right}^c, \text{stop}^c), (\text{straight}^c, \text{stop}^c)\} \tag{16e}$$

$$\mathsf{g}^c(\delta) := \begin{cases} (x_E \geq 15, P(y_O < 0) \geq 0.8) & \text{iff } \delta = (\text{straight}^c, \text{left}^c) \\ (x_E \geq 15, P(y_O \geq 0) \geq 0.8) & \text{iff } \delta = (\text{straight}^c, \text{right}^c) \\ (x_E \geq 15, P(y_O < 0) < 0.8 \wedge P(y_O \geq 0) < 0.8) & \text{iff } \delta = (\text{straight}^c, \text{stop}^c) \\ (y_E \geq 5, \mathsf{true}) & \text{iff } \delta = (\text{left}^c, \text{stop}^c) \\ (y_E \leq -5, \mathsf{true}) & \text{iff } \delta = (\text{right}^c, \text{stop}^c) \end{cases} \tag{16f}$$

$$\mathsf{u}^c(\mathsf{x}_E, \mathsf{y}_E) := (\mathsf{x}_E, \mathsf{y}_E) \text{ for all } \delta \in \Delta^c \tag{16g}$$

$$\mathcal{I}^c := (\text{straight}^c, (x_E = 0, y_E = 0)) \tag{16h}$$

5.4 Bayesian hybrid automaton

▶ **Definition 15** (Bayesian hybrid automaton)**.** A *Bayesian hybrid automaton* is a tuple $BHA = (OA, EA, CA)$ where

- OA is an observed automaton as defined in Def. 2,
- EA is an estimate automaton as defined in Def. 10, and
- CA is a controller automaton as defined in Def. 13.

A state of *BHA* is a tuple $\sigma^{\mathrm{b}} = (\sigma^{\mathrm{o}}, \sigma^{\mathrm{e}}, \sigma^{\mathrm{c}})$ with $\sigma^{\mathrm{o}} \in \Sigma^{\mathrm{o}}$, $\sigma^{\mathrm{e}} \in \Sigma^{\mathrm{e}}$, and $\sigma^{\mathrm{c}} \in \Sigma^{\mathrm{c}}$. By Σ^{b} we denote the set of all states of *BHA*.

▶ **Definition 16** (Semantics of Bayesian hybrid automaton)**.** A run of a Bayesian hybrid automaton is a sequence $\langle \sigma_0^{\mathrm{b}}, \sigma_1^{\mathrm{b}}, \cdots \rangle$ of states which is obtained by a step of the observed automaton followed by a step of the estimate automaton and finally a step of the controller automaton. Hence, the initial state σ_0^{b} is derived by the following rule:

$$\frac{\vdash_{\textsc{InitOA}} \sigma^{\mathrm{o}} \qquad \{\sigma^{\mathrm{o}}\} \vdash_{\textsc{InitEA}} \sigma^{\mathrm{e}} \qquad \vdash_{\textsc{InitCA}} \sigma^{\mathrm{c}}}{(\sigma^{\mathrm{o}}, \sigma^{\mathrm{e}}, \sigma^{\mathrm{c}})}\ \textsc{InitBHA}$$

For all $i \in \mathbb{N}^{>1}$ we have a step according to rule StepBHA:

$$\frac{(\sigma_{i-1}^{\mathrm{o}}, \sigma_{i-1}^{\mathrm{e}}, \sigma_{i-1}^{\mathrm{c}}) \qquad \{\sigma_{i-1}^{\mathrm{o}}\} \vdash_{\textsc{StepOA}} \sigma_i^{\mathrm{o}} \qquad \{\sigma_{i-1}^{\mathrm{e}}, \sigma_i^{\mathrm{o}}\} \vdash_{\textsc{StepEA}} \sigma_i^{\mathrm{e}} \qquad \{\sigma_{i-1}^{\mathrm{c}}, \sigma_i^{\mathrm{e}}\} \vdash_{\textsc{StepCA}} \sigma_i^{\mathrm{c}}}{(\sigma_i^{\mathrm{o}}, \sigma_i^{\mathrm{e}}, \sigma_i^{\mathrm{c}})}\ \textsc{StepBHA}$$

5.5 Reduction of the mixture estimate size

The above semantics of estimate automata is based on, i.a., splitting probability density functions along discrete dynamics of the automaton. Thus the number of mixture components grows exponentially with the number of steps in the worst case. The reason for splitting and generating mixtures is to apply different continuous dynamics on specific parts of the continuous state space during extrapolation. We consequently can merge mixture components μ_i and μ_j by exploiting Eqn. (4) if they are labelled with the same discrete state, i.e., if $\lambda_\ell(\mu_i) = \lambda_\ell(\mu_j)$. In order to integrate such a merge of components, rule TransEA has to be modified s.t. it uses an *uncertain jump with merger* instead of the previously defined uncertain jump. The modified rule obviously requires that there exists a concise representation of the result which not explicitly itemises the individual summands. Such a representation is not inherently available.

▶ **Definition 17** (Uncertain jump with merger)**.** Assume a hybrid automaton with a set of modes $\mathcal{L}$, a discrete transition relation Δ, a guard function g, and an update function u. An *uncertain jump with reduction* is a function $c_{\mathrm{u}}^{\mathrm{M}} : \mathcal{M}(\mathbb{R}^n) \to \mathcal{M}(\mathbb{R}^n)$ that applies an uncertain jump and reduces the number of mixture components according to Eqn. (4), i.e.

$$\mu \mapsto \bigcup_{\ell \in \mathcal{L}} \{\mu_\ell\} \tag{17}$$

where

$$\mu_\ell(\mathsf{x}) = \left(\sum_{\mu_i \in M_\ell} \mu_i(\mathsf{x}) \cdot \lambda_P(\mu_i) \right) \cdot \frac{1}{\lambda_P(\mu_\ell)} \tag{18}$$

with $\lambda_\ell(\mu_\ell) = \ell$ and the weight of the merged mixture component is

$$\lambda_P(\mu_\ell) = \sum_{\mu_i \in M_\ell} \lambda_P(\mu_i) \tag{19}$$

and the set of all mixture components in the result of an uncertain jump labelled by location ℓ is

$$M_\ell = \{\mu_i \mid \mu_i \in c_{\mathrm{u}}^{\mathrm{m}}(\mu) \text{ and } \lambda_\ell(\mu_i) = \ell\}\,. \tag{20}$$

6 Automated verification

While there is a strong body of research concerning state-exploratory methods for hybrid-system verification or falsification (for an overview cf. [13]), all of these methods are currently confined to finite-dimensional, hybrid discrete-continuous state. With complex, mixture-type distributions becoming part of the state-space of the system itself, these methods are no longer applicable: State-exploratory methods for stochastic hybrid automata [31, 15, 1, 14] do, of course, manipulate complex mixtures over hybrid state, but as these stem from the stochastic transition dynamics rather than from the state-space itself, state-exploratory methods covering estimate automata inherently have to add another layer of mixtures (due to the iterated transition dynamics of estimate automata) ranging over state mixtures (themselves being part of the state of estimate automata). To the best of our knowledge, neither a comprehensive tool nor data structures facilitating such an analysis do currently exist.

Simulation faithfully reflecting the arising distributions in a frequentistic sense, however, seems feasible. This would facilitate rigorous statistical model-checking [32]. As mixtures of hybrid states are part of the state-space itself due to the hybrid-state estimation process (see Fig. 8), the underlying simulator, however, has to manipulate the corresponding mixtures as part of its state-space.

If all mixtures arising are linear combinations of interval-restrictions of normal distributions, as in Fig. 8, a length-unbounded list of such interval-restrictions suffices for representing the mixture part of the state space, which then has to be combined with the usual discrete and real-vector-valued states of hybrid automata. Each individual interval restriction of a (scaled) normal distribution can be represented by a five-tuple $(b, e, m, \sigma, \alpha)$ of real numbers encoding the density

$$p_{(b,e,m,\sigma,\alpha)}(x) = \begin{cases} \alpha \cdot \mathcal{N}(m, \sigma^2) & \text{iff } b \leq x \leq e, \\ 0 & \text{otherwise,} \end{cases}$$

thus facilitating a computational representation of the corresponding mixtures as lists of such tuples. Evaluation of simple rational guards $P_{x\sim M}(x \bowtie k) \geq \theta$ over such a mixture M represented as a finite list, where k is a constant and $\bowtie$ denotes an inequality, is also computationally feasible. Hence, a probabilistic simulator could be built provided the distributions arising in the estimate automata remain mixtures of interval-restrictions of normal distributions, which, however, would severely confine the admissible dynamics of the observed processes: non-linear dynamics, deforming the (interval-restrictions of) normal distributions, would have to be excluded, and even affine rotations are cumbersome to deal with.

An obvious alternative is the approximation of the mixtures arising as states in the estimation automata by a set of particles, as in particle-filtering [4]. As the effect of the state extrapolation within the estimation process on each of these particles can be computed whenever the state dynamics of the observed process O can be computed, due to the extrapolation concerning a single particle coinciding exactly with O's state dynamics, simulation using such a particle approximation of the estimation mixtures is feasible even with non-linear dynamics. It should be noted, however, that extrapolation of the state of particles occurs at a different place here than in classical analysis of stochastic systems by particle-based simulation: within each state advancement step of a *single* simulation run of the overall system, we have to advance all particles representing the estimation component of the overall state-space.

Development of this technology for simulation and statistical model-checking is currently commencing in our group.

7 Summary

In the above, we have used two examples to argue that traditional hybrid-automata models, be they deterministic, nondeterministic, or stochastic, are insufficient for obtaining precise verdicts on the safety and liveness, etc., of interacting cyber-physical systems. We identified inaptness to represent rational decision-making under uncertain information as the cause of this deficiency. One may, however, argue that there might be other cures to the problem than the introduction of state estimation into the observing CPS and consequently also into its formal model, the latter necessitating a significant extension and complication of the model of hybrid automata and its related formal analysis techniques.

One such cure could be the introduction of communication within systems of interacting CPSes: the need for state estimation for an observed agent would vanish if all agents would actively communicate the local measurements that their decisions are based on (and optionally also communicate the decisions themselves) rather than having to estimate these from imprecisely observed physical behaviour. This is true, but does not provide a panacea. Not only may the mere cost as well as all kinds of reasons for data privacy and protection render such an approach undesirable, it is also bound to fail when we face socio-technical interaction within human-cyber-physical systems, as humans will neither be able nor willing to communicate a sufficiently complete representation of their perception to the CPS components. HCPSes will inherently have to rely on state estimation permitting technical systems to obtain an image of the humans' states and plans based on behavioural observations, neurophysiological measurements, etc. [8].

Another apparent cure would be the introduction of machine-learned components into the CPS for the sake of estimating (and possibly extrapolating into reasonable future) the state of observed entities. This would, however, not fundamentally change the problems induced to analysis and verification of such systems: the example of deep neural networks used for classification tasks indicates that such machine-learned state estimators manipulate "probabilities"[4] too, necessitating a very similar treatment in models and analysis engines when trying to reason rigorously about the interactive behaviour of mutually coupled systems featuring DNN components within environmental state assessment.

We conclude that the introduction of state estimation processes, with their associated complex state spaces, is a necessary addendum to the hybrid-automata framework in order render them ripe for the modelling and analysis demands of the era of interacting intelligent systems. The development of corresponding automatic verification technology, though constituting a mostly unsolved scientific challenge, is of utmost societal importance.

References

1 Alessandro Abate, Joost-Pieter Katoen, John Lygeros, and Maria Prandini. Approximate model checking of stochastic hybrid systems. *Eur. J. Control*, 16(6):624–641, 2010. doi:10.3166/ejc.16.624-641.

2 Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1992. doi:10.1007/3-540-57318-6_30.

3 David Barber. *Bayesian Reasoning and Machine Learning*. Cambride University Press, 2012. doi:10.1017/CBO9780511804779.

4 Karl Berntorp and Stefano Di Cairano. Particle filtering for automotive: A survey. In *22th International Conference on Information Fusion*, pages 1–8, July 2019. URL: https://www.merl.com/publications/TR2019-069.

5 L.M. Bujorianu and J. Lygeros. Toward a general theory of stochastic hybrid systems. In *Stochastic*

[4] We are adding quotes here, as the "probability" assigned to a given label by a DNN classifier does not constitute a probability in a frequentistic sense or according to other conventional interpretations of probability theory.

Hybrid Systems: Theory and Safety Critical Applications, volume 337 of *LNCIS*, pages 3–30. Springer-Verlag, 2006. doi:10.1007/11587392_1.

6 C. Combastel. Merging Kalman filtering and zonotopic state bounding for robust fault detection under noisy environment. *IFAC-PapersOnLine*, 48(21):289–295, 2015. 9th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS 2015. doi:10.1016/j.ifacol.2015.09.542.

7 Christophe Coué, Cédric Pradalier, Christian Laugier, Thierry Fraichard, and Pierre Bessiere. Bayesian occupancy filtering for multitarget tracking: an automotive application. *International Journal of Robotics Research*, 25(1):19–30, January 2006. doi:10.1177/0278364906061158.

8 Werner Damm, Martin Fränzle, Andreas Lüdtke, Jochem W. Rieger, Alexander Trende, and Anirudh Unni. Integrating neurophysiological sensors and driver models for safe and performant automated vehicle control in mixed traffic. In *2019 IEEE Intelligent Vehicles Symposium*, pages 82–89. IEEE, 2019. URL: https://ieeexplore.ieee.org/xpl/conhome/8792328/proceeding.

9 M.H.A. Davis. *Markov Models and Optimization*. Chapman & Hall, London, 1993.

10 J. Ding, A. Abate, and C. Tomlin. Optimal control of partially observable discrete time stochastic hybrid systems for safety specifications. In *2013 American Control Conference*, pages 6231–6236, June 2013. doi:10.1109/ACC.2013.6580815.

11 Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems - 8th International Conference, FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings*, volume 6246 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2010. doi:10.1007/978-3-642-15297-9_9.

12 Alberto Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, June 1989. doi:10.1109/2.30720.

13 Martin Fränzle, Mingshuai Chen, and Paul Kröger. In memory of Oded Maler: automatic reachability analysis of hybrid-state automata. *SIGLOG News*, 6(1):19–39, 2019. doi:10.1145/3313909.3313913.

14 Martin Fränzle, Ernst Moritz Hahn, Holger Hermanns, Nicolás Wolovick, and Lijun Zhang. Measurability and safety verification for stochastic hybrid systems. In Marco Caccamo, Emilio Frazzoli, and Radu Grosu, editors, *Proceedings of the 14th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2011, Chicago, IL, USA, April 12-14, 2011*, pages 43–52. ACM, 2011. doi:10.1145/1967701.1967710.

15 Martin Fränzle, Holger Hermanns, and Tino Teige. Stochastic satisfiability modulo theory: A novel technique for the analysis of probabilistic hybrid systems. In Magnus Egerstedt and Bud Mishra, editors, *Hybrid Systems: Computation and Control, 11th International Workshop, HSCC 2008, St. Louis, MO, USA, April 22-24, 2008. Proceedings*, volume 4981 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2008. doi:10.1007/978-3-540-78929-1_13.

16 Martin Fränzle and Paul Kröger. The demon, the gambler, and the engineer – reconciling hybrid-system theory with metrology. In Cliff Jones, Ji Wang, and Naijun Zhan, editors, *Symposium on Real-Time and Hybrid Systems*, volume 11180 of *Theoretical Computer Science and General Issues*, pages 165–185, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-01461-2_9.

17 Martin Fränzle and Paul Kröger. Guess what I'm doing! Rendering formal verification methods ripe for the era of interacting intelligent systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 255–272, Cham, 2020. Springer International Publishing.

18 Adrian Gambier. Multivariable adaptive state-space control: A survey. In *2004 5th Asian Control Conference (IEEE Cat. No.04EX904)*, volume 1, pages 185–191 Vol.1, July 2004.

19 J. Hu, J. Lygeros, and S. Sastry. Towards a theory of stochastic hybrid systems. In *Hybrid Systems: Computation and Control*, volume 1790 of *LNCS*, pages 160–173. Springer-Verlag, 2000. doi:10.1007/3-540-46430-1_16.

20 Rudolph Emil Kálmán. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960. doi:10.1115/1.3662552.

21 S. Kowalewski, M. Garavello, H. Guéguen, G. Herberich, R. Langerak, B. Piccoli, J. W. Polderman, and C. Weise. *Hybrid automata*, pages 57–86. Cambridge University Press, 2009. doi:10.1017/CBO9780511807930.004.

22 Helge Langseth, Thomas D. Nielsen, Rafael Rumí, and Antonio Salmerón. Inference in hybrid bayesian networks. *Reliability Engineering & System Safety*, 94(10):1499–1509, 2009. doi:10.1016/j.ress.2009.02.027.

23 Eugene Lavretsky. Robust and adaptive control methods for aerial vehicles. In Kimon P. Valavanis and George J. Vachtsevanos, editors, *Handbook of Unmanned Aerial Vehicles*, pages 675–710, Dordrecht, 2015. Springer Netherlands. doi:10.1007/978-90-481-9707-1_50.

24 R. P. S. Mahler. Multitarget Bayes filtering via first-order multitarget moments. *IEEE Transactions on Aerospace and Electronic Systems*, 39(4):1152–1178, October 2003. doi:10.1109/TAES.2003.1261119.

25 Michael Maschler, Eilon Solan, and Shmuel Zamir. *Game Theory*. Cambridge University Press, 2013. doi:10.1017/cbo9780511794216.

26 Kevin P. Murphy. Switching Kalman filters, 1998.

27 Kumpati S. Narendra and Zhuo Han. Adaptive control using collective information obtained from multiple models. *IFAC Proceedings Volumes*, 44(1):362–367, 2011. 18th IFAC World Congress. doi:10.3182/20110828-6-IT-1002.02237.

28 Anil Nerode and Wolf Kohn. Models for hybrid systems: Automata, topologies, controllability, observability. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors,

Hybrid Systems, volume 736 of *Lecture Notes in Computer Science*, pages 317–356. Springer, 1992. doi:10.1007/3-540-57318-6_35.

29 Simo Särkkä. *Bayesian Filtering and Smoothing*. Cambridge University Press, New York, NY, USA, 2013.

30 C. Sherlock, A. Golightly, and C. S. Gillespie. Bayesian inference for hybrid discrete-continuous stochastic kinetic models. *Inverse Problems*, 30(11):114005, November 2014. doi:10.1088/0266-5611/30/11/114005.

31 Jeremy Sproston. Decidable model checking of probabilistic hybrid automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2000)*, volume 1926 of *LNCS*, pages 31–45. Springer, 2000. doi:10.1007/3-540-45352-0_5.

32 Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 223–235. Springer, 2002. doi:10.1007/3-540-45657-0_17.

From Dissipativity Theory to Compositional Construction of Control Barrier Certificates

Ameneh Nejati
Department of Electrical Engineering, Technical University of Munich, Germany
Department of Computer Science, LMU Munich, Germany

Majid Zamani
Department of Computer Science, University of Colorado Boulder, USA
Department of Computer Science, LMU Munich, Germany

Abstract

This paper proposes a compositional framework based on dissipativity approaches to construct control barrier certificates for networks of continuous-time stochastic hybrid systems. The proposed scheme leverages the structure of the interconnection topology and a notion of so-called *control storage certificates* to construct control barrier certificates compositionally. By utilizing those certificates, one can compositionally synthesize state-feedback controllers for interconnected systems enforcing safety specifications over a finite-time horizon. In particular, we leverage dissipativity-type compositionality conditions to construct *control barrier certificates* for interconnected systems based on corresponding control storage certificates computed for subsystems. Using those constructed control barrier certificates, one can quantify upper bounds on probabilities that interconnected systems reach certain *unsafe* regions in finite-time horizons. We employ a systematic technique based on the sum-of-squares optimization program to search for storage certificates of subsystems together with their corresponding safety controllers. We demonstrate our proposed results by applying them to a temperature regulation in a circular building containing 1000 rooms. To show the applicability of our approaches to dense networks, we also apply our proposed techniques to a *fully-interconnected network*.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Mathematics of computing → Stochastic processes; Theory of computation → Timed and hybrid models

Keywords and Phrases Compositional barrier certificates, Stochastic hybrid systems, Dissipativity theory, Large-scale networks, Formal controller synthesis

Digital Object Identifier 10.4230/LITES.8.2.6

Funding This work was supported in part by the H2020 ERC Starting Grant AutoCPS (grant agreement No. 804639) and the NSF under Grant CNS-2145184.

Received 2020-08-31 **Accepted** 2022-02-11 **Published** 2022-12-07

Editor Alessandro Abate, Uli Fahrenberg, and Martin Fränzle

Special Issue Special Issue on Distributed Hybrid Systems

1 Introduction

Motivations. Formal methods are becoming a promising scheme to design controllers for complex stochastic systems against high-level logic properties, *e.g.,* those expressed as linear temporal logic (LTL) formulae [25]. Since the closed-form characterization of synthesized policies for continuous-time continuous-space stochastic systems is not available in general, formal policy synthesis for those complex systems is naturally very challenging due to their continuous state sets.

To mitigate the encountered computational complexity, one potential direction is to approximate original models by simpler ones with finite state sets (*a.k.a.,* finite Markov decision processes (MDPs)). However, due to discretizing the state and input sets, the finite-abstraction based techniques suffer severely from the curse of dimensionality problem. To alleviate this issue, compositional techniques have been introduced in the past few years to construct finite MDPs of interconnected systems based on constructing finite MDPs of smaller subsystems [11, 12, 13, 14, 15, 16, 19, 20].

Although the proposed compositional frameworks in the setting of finite abstractions can mitigate the effects of the state-explosion problem, the curse of dimensionality may still occur in the level of subsystems given their range of state and input sets. These challenges motivate the need to employ *control barrier certificates* as a discretization-free approach for synthesizing controllers for complex stochastic systems. In this respect, discretization-free techniques based on barrier certificates for stochastic hybrid systems are initially proposed in [26]. Stochastic safety verification using barrier certificates for switched diffusion processes and classes of stochastic hybrid systems is, respectively, proposed in [29] and [8]. Verification of MDPs using barrier certificates is proposed in [1]. Temporal logic synthesis of stochastic systems via control barrier certificates is presented in [9]. Compositional construction of control barrier certificates for *discrete-time* stochastic *control and switched* systems is respectively proposed in [2, 17].

Contributions. This paper proposes a compositional scheme based on dissipativity approaches for the construction of control barrier certificates for a class of continuous-time continuous-space stochastic hybrid systems, namely, jump-diffusion systems. Particularly, we compositionally construct control barrier certificates of interconnected jump-diffusion systems based on so-called *control storage certificates* of subsystems by leveraging dissipativity-type compositionality reasoning. The proposed compositionality condition can enjoy the structure of the interconnection topology and may not require any constraints on the number or even gains of subsystems (cf. Remark 6 and the case study). Using those constructed control barrier certificates, one can quantify upper bounds on probabilities that interconnected systems reach certain unsafe regions in finite-time horizons. We finally utilize a systematic technique based on the sum-of-squares (SOS) optimization program [24] to search for control storage certificates of subsystems. We illustrate the effectiveness of our proposed results by applying them to a temperature regulation in a circular building containing 1000 rooms by compositionally synthesizing safety controllers (together with the corresponding control storage certificates) regulating the temperature of each room in a comfort zone within a bounded-time horizon. We also apply our proposed techniques to a *fully-interconnected network* to show their applicabilities to non-sparse interconnection topologies.

Recent Works. Compositional construction of control barrier certificates for continuous-time stochastic systems is also proposed in [18], but using a different compositionality scheme, namely, based on *small-gain* conditions. Our proposed compositionality approach here is potentially less conservative than the one presented in [18] since the dissipativity-type compositional reasoning, proposed in this work, can enjoy the structure of the interconnection topology and may not require any constraints on the number or gains of the subsystems (cf. Remark 6). Furthermore, the provided results in [18] ask an additional condition (*i.e.,* [18, condition (3)]) which is required for the satisfaction of *small-gain* type compositionality conditions, while we do not need such an extra condition in our setting. Besides, we enlarge the class of systems here to a fragment of continuous-time stochastic *hybrid* ones by adding Poisson processes to the dynamics, whereas the results in [18] only deal with systems described by stochastic differential equations without jumps.

Control barrier functions for stochastic systems in the presence of process and measurement noises are presented in [5]. Although the proposed results in [5] are also for continuous-time stochastic systems, they are only presented in a monolithic framework and dealing only with Brownian motions as sources of the noise. In comparison, we propose here a *compositional* approach for the construction of barrier functions for networks of stochastic systems affected by both Brownian motions and Poisson processes. The results in [5] propose a rather *qualitative* satisfaction of safety specification in which the safety property is either satisfied with the probability 1 or not satisfied. As a result, the proposed approach there is very conservative and the proposed

optimization problem is not going to be feasible for many scenarios depending on different dynamics and safety specifications. In contrast, our work proposes a *quantitative* version of satisfaction in which one can get a lower bound on the probability of satisfaction which is less than one.

2 Continuous-Time Stochastic Hybrid Systems

2.1 Notation and Preliminaries

The following notation is utilized throughout the paper. We denote sets of nonnegative and positive integers by $\mathbb{N}_0 := \{0, 1, 2, \ldots\}$ and $\mathbb{N} := \{1, 2, 3, \ldots\}$, respectively. The symbols $\mathbb{R}$, $\mathbb{R}_{>0}$, and $\mathbb{R}_{\geq 0}$ denote sets of real, positive, and nonnegative real numbers, respectively. We use $\mathbb{R}^n$ to denote an n-dimensional Euclidean space and $\mathbb{R}^{n\times m}$ to denote the space of real matrices with n rows and m columns. We denote by $\mathsf{diag}(a_1, \ldots, a_N)$ and $\mathsf{blkdiag}(a_1, \ldots, a_N)$, respectively, a diagonal matrix in $\mathbb{R}^{N\times N}$ with diagonal scalar and matrix entries $a_1, \ldots, a_N$ starting from the upper left corner. Given a matrix $A \in \mathbb{R}^{n\times m}$, $\mathsf{Tr}(A)$ represents the trace of A which is the sum of all its diagonal elements. We employ $x = [x_1; \ldots; x_N]$ to denote the corresponding vector of a dimension $\sum_i n_i$, given N vectors $x_i \in \mathbb{R}^{n_i}$, $n_i \in \mathbb{N}$, and $i \in \{1, \ldots, N\}$. Given a vector $x \in \mathbb{R}^n$, $\|x\|$ denotes the Euclidean norm of x. Given functions $f_i : X_i \to Y_i$, for any $i \in \{1, \ldots, N\}$, their Cartesian product $\prod_{i=1}^N f_i : \prod_{i=1}^N X_i \to \prod_{i=1}^N Y_i$ is defined as $(\prod_{i=1}^N f_i)(x_1, \ldots, x_N) = [f_1(x_1); \ldots; f_N(x_N)]$. The identity matrix in $\mathbb{R}^{n\times n}$ is denoted by $\mathbb{I}_n$. A function $\gamma : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, is said to be a class $\mathcal{K}$ function if it is continuous, strictly increasing, and $\gamma(0) = 0$. A class $\mathcal{K}$ function $\gamma(\cdot)$ is said to be a class $\mathcal{K}_\infty$ if $\gamma(r) \to \infty$ as $r \to \infty$.

We consider a probability space $(\Omega, \mathcal{F}_\Omega, \mathbb{P}_\Omega)$, where Ω is the sample space, $\mathcal{F}_\Omega$ is a sigma-algebra on Ω comprising subsets of Ω as events, and $\mathbb{P}_\Omega$ is a probability measure that assigns probabilities to events. We assume that triple $(\Omega, \mathcal{F}_\Omega, \mathbb{P}_\Omega)$ is endowed with a filtration $\mathbb{F} = (\mathcal{F}_s)_{s\geq 0}$ satisfying the usual conditions of completeness and right continuity. Let $(\mathbb{W}_s)_{s\geq 0}$ be a b-dimensional $\mathbb{F}$-Brownian motion and $(\mathbb{P}_s)_{s\geq 0}$ be an r-dimensional $\mathbb{F}$-Poisson process. We assume that the Poisson process and Brownian motion are independent of each other. The Poisson process $\mathbb{P}_s = [\mathbb{P}_s^1; \cdots; \mathbb{P}_s^{\mathsf{r}}]$ models r events whose occurrences are assumed to be independent of each other.

2.2 Continuous-Time Stochastic Hybrid Systems

We consider a class of continuous-time stochastic hybrid systems (ct-SHS) as formalized in the following definition.

▶ **Definition 1.** A continuous-time stochastic hybrid system (ct-SHS) in this work is characterized by the tuple

$$\Sigma = (X, U, W, \mathcal{U}, \mathcal{W}, f, \sigma, \rho, Y_1, Y_2, h_1, h_2), \tag{1}$$

where:

- $X \subseteq \mathbb{R}^n$ is the state set of the system;
- $U \subseteq \mathbb{R}^m$ is the *external* input set of the system;
- $W \subseteq \mathbb{R}^p$ is the *internal* input set of the system;
- $\mathcal{U}$ and $\mathcal{W}$ are respectively subsets of sets of all $\mathbb{F}$-progressively measurable processes taking values in $\mathbb{R}^m$ and $\mathbb{R}^p$;
- $f : X \times U \times W \to \mathbb{R}^n$ is the drift term which is globally Lipschitz continuous: there exist constants $\mathscr{L}_x, \mathscr{L}_\nu, \mathscr{L}_w \in \mathbb{R}_{\geq 0}$ such that $\|f(x, \nu, w) - f(x', \nu', w')\| \leq \mathscr{L}_x\|x - x'\| + \mathscr{L}_\nu\|\nu - \nu'\| + \mathscr{L}_w\|w - w'\|$ for all $x, x' \in X$, for all $\nu, \nu' \in U$, and for all $w, w' \in W$;
- $\sigma : \mathbb{R}^n \to \mathbb{R}^{n\times \mathsf{b}}$ is the diffusion term which is globally Lipschitz continuous with the Lipschitz constant $\mathscr{L}_\sigma$;

- $\rho : \mathbb{R}^n \to \mathbb{R}^{n\times \mathsf{r}}$ is the reset term which is globally Lipschitz continuous with the Lipschitz constant $\mathscr{L}_\rho$;
- $Y_1 \subseteq \mathbb{R}^{q_1}$ is the *external* output set of the system;
- $Y_2 \subseteq \mathbb{R}^{q_2}$ is the *internal* output set of the system;
- $h_1 : X \to Y_1$ is the *external* output map;
- $h_2 : X \to Y_2$ is the *internal* output map.

A continuous-time stochastic hybrid system Σ satisfies

$$\Sigma : \begin{cases} \mathsf{d}\xi(t) = f(\xi(t),\nu(t),w(t))\mathsf{d}t + \sigma(\xi(t))\mathsf{d}\mathbb{W}_t + \rho(\xi(t))\mathsf{d}\mathbb{P}_t, \\ \zeta_1(t) = h_1(\xi(t)), \\ \zeta_2(t) = h_2(\xi(t)), \end{cases} \tag{2}$$

$\mathbb{P}$-almost surely ($\mathbb{P}$-a.s.) for any $\nu \in \mathcal{U}$ and any $w \in \mathcal{W}$, where stochastic processes $\xi : \Omega\times\mathbb{R}_{\geq 0} \to X$, $\zeta_1 : \Omega \times \mathbb{R}_{\geq 0} \to Y_1$, and $\zeta_2 : \Omega \times \mathbb{R}_{\geq 0} \to Y_2$ are, respectively, called the *solution process* and the external and internal *output trajectories* of Σ. We also use $\xi_{a\nu w}(t)$ to denote the value of the solution process at the time $t \in \mathbb{R}_{\geq 0}$ under input trajectories ν and w from an initial condition $\xi_{a\nu w}(0) = a$ $\mathbb{P}$-a.s., where a is a random variable that is $\mathcal{F}_0$-measurable. We also denote by $\zeta_{1_{a\nu w}}$ and $\zeta_{2_{a\nu w}}$ the external and internal *output trajectories* corresponding to the *solution process* $\xi_{a\nu w}$. Here, we assume that the Poisson processes $\mathbb{P}^{\hat{z}}_s$, for any $\hat{z} \in \{1,\dots,\mathsf{r}\}$, have rates $\bar{\lambda}_{\hat{z}}$. We emphasize that the postulated assumptions on f, σ, and ρ ensure existence, uniqueness, and strong Markov property of the solution process [22].

Given the ct-SHS in (1), we are interested in Markov policies to control the system as defined in the next definition.

▶ **Definition 2.** A Markov policy for the ct-SHS Σ in (1) is a map $\varrho : \mathbb{B}(U) \times X \times \mathbb{R}_{\geq 0} \to [0,1]$, with $\mathbb{B}(U)$ being the Borel sigma-algebra on the external input space, such that $\varrho(\cdot \,|\, \cdot, t)$ is a universally measurable stochastic kernel for all $t \in \mathbb{R}_{\geq 0}$ [27]. For any state $x \in X$ at time t, the input $\nu(t)$ is chosen according to the probability measure $\varrho(\cdot \,|\, x, t)$.

Although we define continuous-time stochastic hybrid systems with outputs, we assume the full-state information is available for the sake of controller synthesis. The role of outputs are mainly for the sake of interconnecting systems as explained in detail in Section 4.

Given the main contribution of this work which is developing a compositional approach for the construction of control barrier certificates, we are ultimately interested in investigating interconnected systems without having internal signals. In this case, the tuple (1) reduces to $(X, U, \mathcal{U}, f, \sigma, \rho, Y, h)$ with $f : X \times U \to \mathbb{R}^n$, and ct-SHS (2) can be re-written as

$$\Sigma : \begin{cases} \mathsf{d}\xi(t) = f(\xi(t),\nu(t))\,\mathsf{d}t + \sigma(\xi(t))\,\mathsf{d}\mathbb{W}_t + \rho(\xi(t))\,\mathsf{d}\mathbb{P}_t, \\ \zeta(t) = h(\xi(t)). \end{cases}$$

In the next sections, we propose an approach for the compositional construction of control barrier certificates for interconnected ct-SHS. To do so, we define, in the next section, notions of control storage and barrier certificates for ct-SHS and interconnected versions, respectively.

3 Control Storage and Barrier Certificates

In this section, we first introduce a notion of control storage certificates (CSC) for ct-SHS with both internal and external signals. We then define a notion of control barrier certificates (CBC) for ct-SHS with only external signals. We leverage the former notion to compositionally construct the latter one for interconnected systems. We then employ the latter notion to quantify upper bounds on the probability that the interconnected system reaches certain unsafe regions in finite-time horizons.

▶ **Definition 3.** Consider a ct-SHS $\Sigma = (X, U, W, \mathcal{U}, \mathcal{W}, f, \sigma, \rho, Y_1, Y_2, h_1, h_2)$. Let $X_0, X_1 \subseteq X$ be initial and unsafe sets of the system, respectively. A twice differentiable function $\mathcal{B} : X \to \mathbb{R}_{\geq 0}$ is called a control storage certificate (CSC) for Σ if there exist $\kappa \in \mathcal{K}_\infty$, $\gamma, \lambda, \psi \in \mathbb{R}_{\geq 0}$, and a symmetric matrix $\bar{X}$ with conformal block partitions $\bar{X}^{z\bar{z}}$, $z, \bar{z} \in \{1, 2\}$, where $\bar{X}^{22} \preceq 0$, such that

- $\forall x \in X_0$,

$$\mathcal{B}(x) \leq \gamma, \tag{3}$$

- $\forall x \in X_1$,

$$\mathcal{B}(x) \geq \lambda, \tag{4}$$

- and $\forall x \in X$, $\exists \nu \in U$, such that $\forall w \in W$,

$$\mathcal{L}\mathcal{B}(x) \leq -\kappa(\mathcal{B}(x)) + \psi + \begin{bmatrix} w \\ h_2(x) \end{bmatrix}^T \underbrace{\begin{bmatrix} \bar{X}^{11} & \bar{X}^{12} \\ \bar{X}^{21} & \bar{X}^{22} \end{bmatrix}}_{\bar{X} :=} \begin{bmatrix} w \\ h_2(x) \end{bmatrix}, \tag{5}$$

where $\mathcal{L}\mathcal{B}$ is the *infinitesimal generator* of the stochastic process acting on the function $\mathcal{B}$ [21], as defined in the next remark.

▶ Remark 1. Note that the *infinitesimal generator* $\mathcal{L}$ of the process $\xi(t)$ acting on the function $\mathcal{B} : X \to R_{\geq 0}$ is defined as

$$\mathcal{L}\mathcal{B}(x) = \partial_x \mathcal{B}(x) f(x, \nu, w) + \frac{1}{2} \mathsf{Tr}(\sigma(x)\sigma(x)^T \partial_{x,x} \mathcal{B}(x)) + \sum_{j=1}^{\mathsf{r}} \bar{\lambda}_j (\mathcal{B}(x + \rho(x)\mathsf{e}^{\mathsf{r}}_j) - \mathcal{B}(x)), \tag{6}$$

where $\partial_x \mathcal{B}(x) = \left[\frac{\partial \mathcal{B}(x)}{\partial x_i}\right]_i$ is a row vector, $\partial_{x,x} \mathcal{B}(x) = \left[\frac{\partial^2 \mathcal{B}(x)}{\partial x_i \partial x_j}\right]_{i,j}$, $\bar{\lambda}_j$ is the rate of Poisson process, and $\mathsf{e}^{\mathsf{r}}_j$ denotes an r-dimensional vector with 1 on the j-th entry and 0 elsewhere.

▶ Remark 2. Since the control input ν in condition (5) is independent of internal inputs w (*i.e.,* state information of other subsystems), the employed quantifier in (5) implicitly implies that one can synthesize *decentralized* controllers for Σ . However, one can design *distributed* control policies by changing the sequence of the quantifier in (5) to $\forall x \in X, \forall w \in W, \exists \nu \in U$. In this case, the chance of finding control storage certificates gets increased; however, one needs to measure the state information of other subsystems to deploy the synthesized controllers.

▶ Remark 3. Note that a local storage certificate captures the role of w (*i.e.,* the effect of interaction between subsystems in the interconnected topology) using the quadratic term in the right-hand side of (5). This term is interpreted in dissipativity theory as the supply rate of the system [3] which is initially used to show the stability of a network based on stabilities of its subsystems. Here, we choose this function to be quadratic which results in tractable compositional conditions later in the form of linear matrix inequalities (cf. (13)).

Now we modify the above notion for the interconnected ct-SHS without internal signals. This notion will be utilized in Theorem 5 for quantifying upper bounds on the probability that the interconnected system (without internal signals) reaches certain unsafe regions in a finite-time horizon.

▶ **Definition 4.** Consider the (interconnected) system $\Sigma = (X, U, \mathcal{U}, f, \sigma, \rho, Y, h)$, and $X_0, X_1 \subseteq X$ as respectively initial and unsafe sets of the interconnected system. A twice differentiable function $\mathcal{B} : X \to \mathbb{R}_{\geq 0}$ is called a control barrier certificate (CBC) for Σ if

- $\forall x \in X_0$,

$$\mathcal{B}(x) \leq \gamma \tag{7}$$

- $\forall x \in X_1$,

$$\mathcal{B}(x) \geq \lambda \tag{8}$$

- and $\forall x \in X$, $\exists \nu \in U$ such that

$$\mathcal{L}\mathcal{B}(x) \leq -\kappa(\mathcal{B}(x)) + \psi, \tag{9}$$

for some $\kappa \in \mathcal{K}_\infty$, $\gamma, \lambda, \psi \in \mathbb{R}_{\geq 0}$, with $\lambda > \gamma$.

▶ Remark 4. Note that stochastic storage certificates satisfying conditions (3)-(5) are not useful on their own to ensure the safety of the corresponding subsystems and the interconnected system as a whole. Stochastic storage certificates are some appropriate tools used to construct overall control barrier certificates given that some compositionality conditions are satisfied (cf. (13),(14)). The safety of the system can then be verified via Theorem 5 only using the constructed control barrier certificate.

The next theorem shows the usefulness of CBC to quantify upper bounds on the probability that the interconnected system reaches certain unsafe regions in a finite-time horizon.

▶ **Theorem 5.** *Let $\Sigma = (X, U, \mathcal{U}, f, \sigma, \rho, Y, h)$ be an (interconnected) ct-SHS without internal signals. Suppose $\mathcal{B}$ is a CBC for Σ as in Definition 4, and there exists a constant $\hat{\kappa} \in \mathbb{R}_{>0}$ such that the function $\kappa \in \mathcal{K}_\infty$ in (9) satisfies $\kappa(s) \geq \hat{\kappa} s$, $\forall s \in \mathbb{R}_{\geq 0}$. Then the probability that the solution process of Σ starts from any initial state $\xi(0) = x_0 \in X_0$ and reaches X_1 under the policy $\nu(\cdot)$ within a time horizon $[0, T_d] \subseteq \mathbb{R}_{\geq 0}$ is formally quantified as*

$$\mathbb{P}_\nu^{x_0}\Big\{\xi(t) \in X_1 \text{ for some } 0 \leq t \leq T_d \;\Big|\; \xi(0) = x_0\Big\} \leq \begin{cases} 1 - (1 - \frac{\gamma}{\lambda})e^{-\frac{\psi T_d}{\lambda}}, & \text{if } \lambda \geq \frac{\psi}{\hat{\kappa}}, \\ \frac{\hat{\kappa}\gamma + (e^{\hat{\kappa} T_d} - 1)\psi}{\hat{\kappa}\lambda e^{\hat{\kappa} T_d}}, & \text{if } \lambda \leq \frac{\psi}{\hat{\kappa}}. \end{cases} \tag{10}$$

The proof of Theorem 5 is provided in Appendix.

▶ Remark 5. In Section 5, we reformulate conditions of Definition 4 as an optimization problem such that one can minimize values of γ and ψ in order to obtain a better upper bound that is as tight as possible.

In the next section, we analyze networks of stochastic hybrid subsystems and show under which conditions one can construct a CBC of an interconnected system utilizing the corresponding CSC of subsystems.

4 Compositional Construction of CBC

In this section, we analyze networks of stochastic hybrid subsystems, $i \in \{1, \ldots, N\}$,

$$\Sigma_i = (X_i, U_i, W_i, \mathcal{U}_i, \mathcal{W}_i, f_i, \sigma_i, \rho_i, Y_{1_i}, Y_{2_i}, h_{1_i}, h_{2_i}), \tag{11}$$

and discuss how to construct a CBC of the interconnected system based on CSC of subsystems using dissipativity-type compositional conditions. We first formally define the interconnected stochastic hybrid systems.

▶ **Definition 6.** Consider $N \in \mathbb{N}$ stochastic hybrid subsystems $\Sigma_i = (X_i, U_i, W_i, \mathcal{U}_i, \mathcal{W}_i, f_i, \sigma_i, \rho_i, Y_{1_i}, Y_{2_i}, h_{1_i}, h_{2_i})$, $i \in \{1, \dots, N\}$, and a matrix M defining the coupling between these subsystems. We require the condition $M \prod_{i=1}^{N} Y_{2i} \subseteq \prod_{i=1}^{N} W_i$ to have a well-posed interconnection. The interconnection of Σ_i, $\forall i \in \{1, \dots, N\}$, is the ct-SHS $\Sigma = (X, U, \mathcal{U}, f, \sigma, Y, h)$, denoted by $\mathcal{I}(\Sigma_1, \dots, \Sigma_N)$, such that $X := \prod_{i=1}^{N} X_i$, $U := \prod_{i=1}^{N} U_i$, $f := \prod_{i=1}^{N} f_i$, $\sigma := \mathsf{blkdiag}(\sigma_1(x_1), \dots, \sigma_N(x_N))$, $\rho := \mathsf{blkdiag}(\rho_1(x_1), \dots, \rho_N(x_N))$, $Y := \prod_{i=1}^{N} Y_{1i}$, and $h = \prod_{i=1}^{N} h_{1i}$, with the internal inputs constrained according to

$$[w_1; \cdots; w_N] = M[h_{21}(x_1); \cdots; h_{2N}(x_N)].$$

We assume that for hybrid subsystems $\Sigma_i, i \in \{1, \dots, N\}$, there exist CSC $\mathcal{B}_i$ as defined in Definition 3 with the corresponding functions, constant, and matrices denoted by $\kappa_i \in \mathcal{K}_\infty$, $\gamma_i, \lambda_i, \psi_i \in \mathbb{R}_{\geq 0}$, $\bar{X}_i$, $\bar{X}_i^{11}$, $\bar{X}_i^{12}$, $\bar{X}_i^{21}$, and $\bar{X}_i^{22}$. In the next theorem, we compositionally construct a control barrier certificate for the interconnected system Σ as in Definition 4.

▶ **Theorem 7.** *Consider an interconnected stochastic hybrid system $\Sigma = \mathcal{I}(\Sigma_1, \dots, \Sigma_N)$ induced by $N \in \mathbb{N}$ stochastic hybrid subsystems Σ_i and the coupling matrix M. Suppose that each subsystem Σ_i admits a CSC $\mathcal{B}_i$ as defined in Definition 3 with the corresponding initial and unsafe sets X_{0_i} and X_{1_i}, respectively. Then*

$$\mathcal{B}(x) := \sum_{i=1}^{N} \mu_i \mathcal{B}_i(x_i) \tag{12}$$

is a CBC for the interconnected system $\Sigma = \mathcal{I}(\Sigma_1, \dots, \Sigma_N)$ with the corresponding initial and unsafe sets $X_0 := \prod_{i=1}^{N} X_{0_i}$, $X_1 := \prod_{i=1}^{N} X_{1_i}$, respectively, if there exist $\mu_i > 0$, $i \in \{1, \dots, N\}$, such that

$$\begin{bmatrix} M \\ \mathbb{I}_{\tilde{q}} \end{bmatrix}^T \bar{X}_{cmp} \begin{bmatrix} M \\ \mathbb{I}_{\tilde{q}} \end{bmatrix} \preceq 0, \tag{13}$$

$$\sum_{i=1}^{N} \mu_i \lambda_i > \sum_{i=1}^{N} \mu_i \gamma_i, \tag{14}$$

where

$$\bar{X}_{cmp} := \begin{bmatrix} \mu_1 \bar{X}_1^{11} & & & \mu_1 \bar{X}_1^{12} & & \\ & \ddots & & & \ddots & \\ & & \mu_N \bar{X}_N^{11} & & & \mu_N \bar{X}_N^{12} \\ \mu_1 \bar{X}_1^{21} & & & \mu_1 \bar{X}_1^{22} & & \\ & \ddots & & & \ddots & \\ & & \mu_N \bar{X}_N^{21} & & & \mu_N \bar{X}_N^{22} \end{bmatrix}, \tag{15}$$

and $\tilde{q} = \sum_{i=1}^{N} q_{2i}$ with q_{2i} being dimensions of the internal *output of subsystems Σ_i.*

The proof of Theorem 7 is provided in Appendix.

▶ Remark 6. Condition (13) is similar to the LMI appeared in [3] as a compositional stability condition based on the dissipativity theory. It is shown in [3] that this condition holds independently of the number of subsystems in many physical applications with particular interconnection topologies, *e.g.,* skew symmetric.

▶ Remark 7. Note that the condition (14) in general is not very restrictive since constants μ_i in (12) play a significant role in rescaling CSC for subsystems while normalizing the effect of internal gains of other subsystems. One can expect that condition (14) holds in many applications due to this rescaling.

5 Computation of CSC

In this section, we formulate the proposed conditions in Definition 3 as a sum-of-squares (SOS) optimization problem [24] and provide a systematic approach for computing CSC and corresponding control policies for subsystems Σ_i. The SOS optimization technique relies on the fact that a polynomial is non-negative if it can be written as a sum of squares of different polynomials. In order to utilize an SOS optimization, we raise the following assumption.

▶ Assumption 1. Subsystem Σ_i has a continuous state set $X_i \subseteq \mathbb{R}^{n_i}$ and continuous external and internal input sets $U_i \subseteq \mathbb{R}^{m_i}$ and $W_i \subseteq \mathbb{R}^{p_i}$. Moreover, the drift term $f_i : X_i \times U_i \times W_i \to \mathbb{R}^{n_i}$ is a polynomial function of the state x_i and external and internal inputs ν_i, w_i. Furthermore, diffusion and reset terms $\sigma_i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_i \times \mathsf{b}_i}$ and $\rho_i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_i \times \mathsf{r}_i}$ are polynomial functions of the state x_i.

Under Assumption 1, one can reformulate the proposed conditions in Definition 3 as an SOS optimization problem to search for a polynomial CSC $\mathcal{B}_i(\cdot)$, and a polynomial control policy $\nu_i(\cdot)$. The following lemma provides a set of sufficient conditions for the existence of such CSC required in Definition 3, which can be solved now as an SOS optimization problem.

▶ **Lemma 8.** *Suppose Assumption 1 holds and sets $X_{0_i}, X_{1_i}, X_i, U_i, W_i$ can be defined by vectors of polynomial inequalities $X_{0_i} = \{x_i \in \mathbb{R}^{n_i} \mid g_{0_i}(x_i) \geq 0\}$, $X_{1_i} = \{x_i \in \mathbb{R}^{n_i} \mid g_{1_i}(x_i) \geq 0\}$, $X_i = \{x_i \in \mathbb{R}^{n_i} \mid g_i(x_i) \geq 0\}$, $U_i = \{\nu_i \in \mathbb{R}^{m_i} \mid g_{\nu_i}(\nu_i) \geq 0\}$, and $W_i = \{w_i \in \mathbb{R}^{p_i} \mid g_{w_i}(w_i) \geq 0\}$, where the inequalities are defined element-wise. Suppose there exist a sum-of-square polynomial $\mathcal{B}_i(x_i)$, constants $\gamma_i, \lambda_i, \psi_i \in \mathbb{R}_{\geq 0}$, functions $\kappa_i \in \mathcal{K}_\infty$, a symmetric matrix $\bar{X}_i$ with conformal block partitions $\bar{X}_i^{z\bar{z}}$, $z, \bar{z} \in \{1, 2\}$, where $\bar{X}_i^{22} \preceq 0$, polynomials $l_{\nu_{ji}}(x)$ corresponding to the j^{th} input in $\nu_i = [\nu_{1_i}; \nu_{2_i}; \ldots; \nu_{m_i}] \in U_i \subseteq \mathbb{R}^{m_i}$, and vectors of sum-of-squares polynomials $l_{0_i}(x_i)$, $l_{1_i}(x_i)$, $l_i(x_i, \nu_i, w_i)$, $l_{\nu_i}(x_i, \nu_i, w_i)$, and $l_{w_i}(x_i, \nu_i, w_i)$ of appropriate dimensions such that the following expressions are sum-of-squares polynomials:*

$$-\mathcal{B}_i(x_i) - l_{0_i}^T(x_i) g_{0_i}(x_i) + \gamma_i \tag{16}$$

$$\mathcal{B}_i(x_i) - l_{1_i}^T(x_i) g_{1_i}(x_i) - \lambda_i \tag{17}$$

$$\begin{aligned} &-\mathcal{L}\mathcal{B}_i(x_i) - \kappa_i(\mathcal{B}_i(x_i)) + \begin{bmatrix} w_i \\ h_{2_i}(x_i) \end{bmatrix}^T \begin{bmatrix} \bar{X}_i^{11} & \bar{X}_i^{12} \\ \bar{X}_i^{21} & \bar{X}_i^{22} \end{bmatrix} \begin{bmatrix} w_i \\ h_{2_i}(x_i) \end{bmatrix} \\ &+ \psi_i - \sum_{j=1}^{m_i} (\nu_{ji} - l_{\nu_{ji}}(x_i)) - l_i^T(x_i, \nu_i, w_i) g_i(x_i) - l_{\nu_i}^T(x_i, \nu_i, w_i) g_{\nu_i}(\nu_i) - l_{w_i}^T(x_i, \nu_i, w_i) g_{w_i}(w_i). \end{aligned} \tag{18}$$

Then, $\mathcal{B}_i(x_i)$ satisfies conditions (3)-(5) in Definition 3 and $\nu_i = [l_{\nu_{1_i}}(x_i); \ldots; l_{\nu_{m_i}}(x_i)]$, $i \in \{1, \ldots, N\}$, is the corresponding safety controller.

The proof of Lemma 8 is provided in Appendix.

▶ Remark 8. Note that function $\kappa_i(\cdot)$ in (18) can cause nonlinearity on unknown parameters of $\mathcal{B}_i$. A possible way to avoid this issue is to consider a linear function $\kappa_i(s) = \hat{\kappa}_i s, \forall s \in \mathbb{R}_{\geq 0}$, with some given constant $\hat{\kappa}_i \in \mathbb{R}_{>0}$. Then one can employ bisection method to minimize the value of $\hat{\kappa}_i$.

▶ Remark 9. Note that for computing the sum-of-squares polynomial $\mathcal{B}_i(x_i)$ fulfilling reformulated conditions (16)-(18), one can readily employ existing software tools available in the literature such as SOSTOOLS [23] together with a semi-definite programming (SDP) solver such as SeDuMi [28].

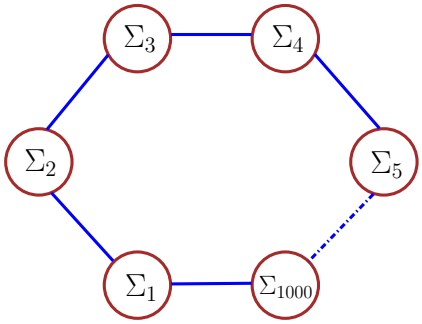


Figure 1 A circular building in a network of 1000 rooms.

6 Case Studies

6.1 Room Temperature Network

To illustrate the effectiveness of the proposed results, we first apply our approaches to a temperature regulation in a network of 1000 rooms, each equipped with a heater and connected circularly as depicted in Figure 1. We compute the CSC of each room while compositionally synthesizing safety controllers to regulate the temperature of each room in a comfort zone for a bounded-time horizon.

The model of this case study is borrowed from [6] by including stochasticity in the model. The evolution of the temperature $T(\cdot)$ can be described by the interconnected jump-diffusion

$$\Sigma : \begin{cases} \mathsf{d}T(t) = (AT(t) + \theta T_h \nu(t) + \beta T_E)\mathsf{d}t + G\mathsf{d}\mathbb{W}_t + R\mathsf{d}\mathbb{P}_t, \\ \zeta(t) = T(t), \end{cases}$$

where A is a matrix with diagonal elements $a_{ii} = -2\eta - \beta - \theta\nu_i(t)$, $i \in \{1, \dots, n\}$, off-diagonal elements $a_{i,i+1} = a_{i+1,i} = a_{1,n} = a_{n,1} = \eta$, $i \in \{1, \dots, n-1\}$, and all other elements are identically zero. Parameters $\eta = 0.005$, $\beta = 0.06$, and $\theta = 0.156$ are conduction factors, respectively, between rooms $i \pm 1$ and i, the external environment and the room i, and the heater and the room i. Moreover, $G = R = 0.1\mathbb{I}_n$, $T_E = [T_{e_1}; \dots; T_{e_n}]$, $T(t) = [T_1(t); \dots; T_n(t)]$, and $\nu(t) = [\nu_1(t); \dots; \nu_n(t)]$. Outside temperatures are the same for all rooms: $T_{ei} = -15\,^\circ\mathsf{C}$, $\forall i \in \{1, \dots, n\}$, and the heater temperature is $T_h = 48\,^\circ\mathsf{C}$. We consider the rates of Poisson processes as $\bar{\lambda}_i = 0.1, \forall i \in \{1, \dots, n\}$. Now by considering the individual rooms as Σ_i described by

$$\Sigma_i : \begin{cases} \mathsf{d}T_i(t) = (a_{ii}T_i(t) + \theta T_h \nu_i(t) + \eta w_i(t) + \beta T_{e_i})\mathsf{d}t + 0.1\mathsf{d}\mathbb{W}_{t_i} + 0.1\mathsf{d}\mathbb{P}_{t_i}, \\ \zeta_{1_i}(t) = T_i(t), \\ \zeta_{2_i}(t) = T_i(t), \end{cases} \tag{19}$$

one can readily verify that $\Sigma = \mathcal{I}(\Sigma_1, \dots, \Sigma_N)$ where the coupling matrix M is defined as $m_{i,i+1} = m_{i+1,i} = m_{1,n} = m_{n,1} = 1$, $i \in \{1, \dots, n-1\}$, and all other elements are identically zero.

The regions of interest in this example are $X_i = [1\ 50]$, $X_{0_i} = [19.5\ 20]$, $X_{1_i} = [1\ 17] \cup [23\ 50]$, $\forall i \in \{1, \dots, n\}$. The main goal is to find a CBC for the interconnected system, using which a safety controller is synthesized for Σ maintaining the temperatures of rooms in the comfort zone $W = [17\ 23]^{1000}$. The idea here is to search for CSC and accordingly design local controllers for subsystems Σ_i. Consequently, the controller for the interconnected system Σ is simply a vector such that its ith component is the controller for subsystem Σ_i. We employ the software tool SOSTOOLS [23] and the SDP solver SeDuMi [28] to compute CSC as described in Section 5. According to Lemma 8, we compute CSC of order 2 as $\mathcal{B}_i(T_i) = 0.3112T_i^2 - 12.3035T_i + 121.59906$

and the corresponding safety controller $\nu_i(T_i) = -0.0120155T_i + 0.7$ for all $i \in \{1, \ldots, n\}$. Moreover, the corresponding constants and functions in Definition 3 satisfying conditions (3)-(5) are quantified as $\gamma_i = 0.08, \lambda_i = 2.7, \kappa_i(s) = \hat{\kappa}_i s, \forall s \in \mathbb{R}_{\geq 0}$ with $\hat{\kappa}_i = 10^{-7}, \psi_i = 5 \times 10^{-3}$, and

$$\bar{X}_i = \begin{bmatrix} \hat{\kappa}_i e^{-4}\eta^2 & 0 \\ 0 & -\hat{\kappa}_i e^{-4}\theta^2 T_h^2 \end{bmatrix}. \tag{20}$$

We now proceed with Theorem 7 to construct a CBC for the interconnected system using CSC of subsystems. By selecting $\mu_i = 1, \forall i \in \{1, \ldots, n\}$, and utilizing $\bar{X}_i$ in (20), the matrix $\bar{X}_{cmp}$ in (15) reduces to

$$\bar{X}_{cmp} = \begin{bmatrix} \hat{\kappa}_i e^{-4}\eta^2 \mathbb{I}_n & 0 \\ 0 & -\hat{\kappa}_i e^{-4}\theta^2 T_h^2 \mathbb{I}_n \end{bmatrix},$$

and condition (13) is reduced to

$$\begin{bmatrix} M \\ \mathbb{I}_n \end{bmatrix}^T \bar{X}_{cmp} \begin{bmatrix} M \\ \mathbb{I}_n \end{bmatrix} = \hat{\kappa}_i e^{-4}\eta^2 M^T M - \hat{\kappa}_i e^{-4}\theta^2 T_h^2 \mathbb{I}_n \preceq 0,$$

without requiring any restrictions on the number or gains of subsystems. We used $M = M^T$, and $4\hat{\kappa}_i e^{-4}\eta^2 - \hat{\kappa}_i e^{-4}\theta^2 T_h^2 \preceq 0$ by employing Gershgorin circle theorem [4] to show the above LMI. Moreover, the compositionality condition (14) is also met since $\lambda_i > \gamma_i, \forall i \in \{1, \ldots, n\}$. Then by employing the results of Theorem 7, one can conclude that $\mathcal{B}(T) = \sum_{i=1}^{1000}(0.3112T_i^2 - 12.3035T_i + 121.59906)$ is a CBC for the interconnected system Σ with $\gamma = 80, \lambda = 2700, \kappa(s) = 10^{-7}s, \forall s \in \mathbb{R}_{\geq 0}$, and $\psi = 5$. Accordingly, $\nu(T) = [-0.0120155T_1 + 0.7; \ldots; -0.0120155T_{1000} + 0.7]$ is the overall safety controller for the interconnected system.

By employing Theorem 5, one can guarantee that the temperature of the interconnected system Σ starting from initial conditions inside $X_0 = [19.5\ 20]^{1000}$ remains in the safe set $[17\ 23]^{1000}$ during the time horizon $T_d = 10$ with the probability of at least 96%, *i.e.,*

$$\mathbb{P}_\nu^{x_0}\Big\{\xi(t) \notin X_1 \mid \xi(0) = x_0,\ \forall t \in [0, 10]\Big\} \geq 0.96\,. \tag{21}$$

Closed-loop state trajectories of a representative room with 10 different noise realizations are illustrated in Figure 2.

With the assumption that all dynamics and barrier certificates are polynomial types, the computational complexity of using SOS in our setting is linear with respect to the number of subsystems. Whereas, if one is interested in solving the problem in a monolithic manner, the complexity will be polynomial in terms of the number of subsystems [30]. In the worst-case scenario, the computational complexity in the monolithic manner will be exponential in terms of the number of subsystems if the underlying dynamics and barrier certificates are not polynomial.

Importance of Compositionality Conditions. In order to demonstrate the importance of the compositionality conditions, we raise the following counter example. Consider a network of *two rooms* each equipped with a heater and connected circularly, as illustrated in Figure 3, with dynamics as in (19) with $T_{e_i} = -100, \forall i \in \{1, 2\}$. One can readily verify that $\Sigma = \mathcal{I}(\Sigma_1, \Sigma_2)$ where the coupling matrix M is defined as $M = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Let regions of interest be the same as before. We compute CSC of order 2 as $\mathcal{B}_i(T_i) = 0.76484T_i^2 - 30.18033T_i + 297.73079$ and its corresponding controller $\nu_i(T_i) = 0.0120155T_i + 0.7$ for all $i \in \{1, 2\}$, with

$$\bar{X}_i = \begin{bmatrix} 4 \times 10^{-4} & 20 \\ 20 & 5 \times 10^{-4} \end{bmatrix}.$$

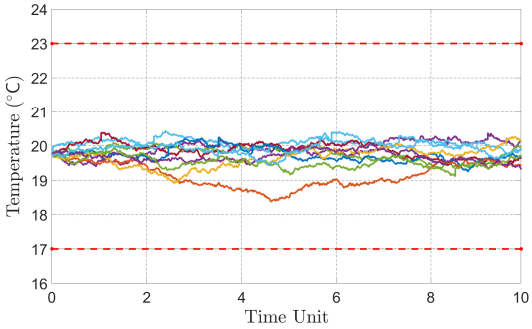


Figure 2 Closed-loop state trajectories of a representative room with 10 noise realizations in a network of 1000 rooms.

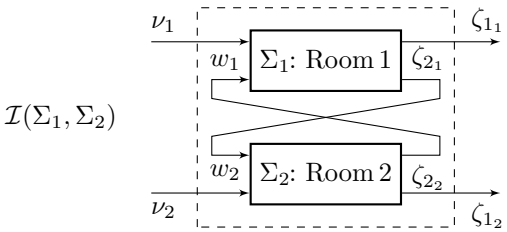


Figure 3 Interconnection of two rooms Σ_1 and Σ_2.

We now select $\mu_i = 1, \forall i \in \{1, 2\}$, and construct the matrix $\bar{X}_{cmp}$ in (15) as

$$\bar{X}_{cmp} = \begin{bmatrix} 4 \times 10^{-4} & 0 & 20 & 0 \\ 0 & 4 \times 10^{-4} & 0 & 20 \\ 20 & 0 & 5 \times 10^{-4} & 0 \\ 0 & 20 & 0 & 5 \times 10^{-4} \end{bmatrix}.$$

Now we check the compositionality condition in (13) as

$$\begin{bmatrix} M \\ \mathbb{I}_n \end{bmatrix}^T \bar{X}_{cmp} \begin{bmatrix} M \\ \mathbb{I}_n \end{bmatrix} \not\succeq 0,$$

with eigenvalues equal to -39.9991 and 40.0009. Since the compositionality condition is violated, one cannot automatically conclude that $\mathcal{B}(T) = \mathcal{B}_1(T_1) + \mathcal{B}_2(T_2)$ is a barrier certificate for the overall system. To show this issue, we employ $\mathcal{B}(T) = 0.76484T_1^2 - 30.18033T_1 + 297.73079 + 0.76484T_2^2 - 30.18033T_2 + 297.73079$ and check the corresponding conditions for the overall barrier certificate (*i.e.,* conditions (7)-(9)) with $\gamma = \gamma_1 + \gamma_2, \lambda = \lambda_1 + \lambda_2, \psi = \psi_1 + \psi_2$. As it can be observed from Figures 4-6, although conditions (7),(8) are satisfied for the overall barrier certificates $\mathcal{B}(T) = \mathcal{B}_1(T_1) + \mathcal{B}_2(T_2)$, condition (9) is violated since it is positive at some ranges of $X_1 \times X_2$.

Then one can readily verify that $\mathcal{B}(T) = \mathcal{B}_1(T_1) + \mathcal{B}_2(T_2)$ is not necessarily a barrier certificate for the overall network ensuring its safety even though all the rooms are the same and storage certificates are input independent.

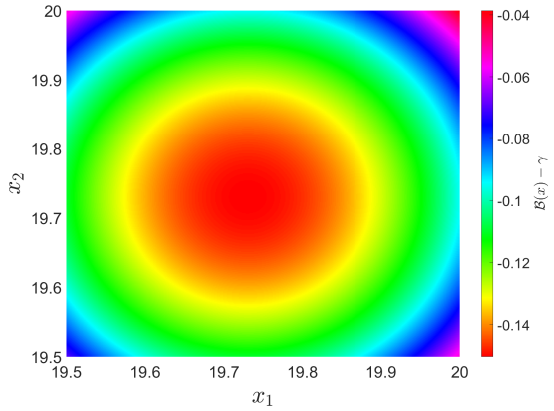


Figure 4 Satisfaction of condition (7). As observed, this condition is negative for all ranges of $x_1 \in X_{0_1}$ and $x_2 \in X_{0_2}$.

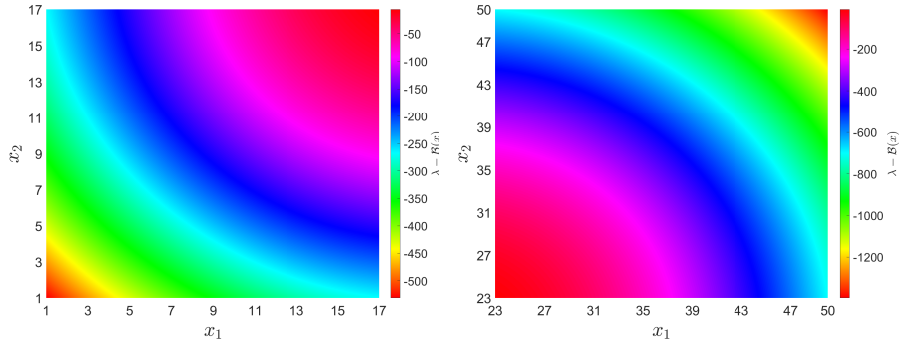


Figure 5 Satisfaction of condition (8). The condition is negative for all ranges of $x_1 \in X_{1_1}$ and $x_2 \in X_{1_2}$.

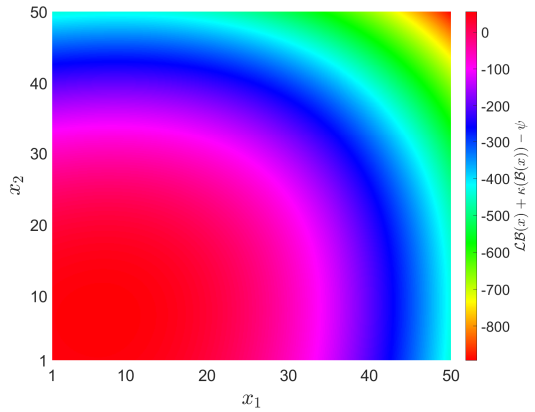


Figure 6 Violation of condition (9). As observed, this condition is positive for some ranges of $x_1 \in X_1$ and $x_2 \in X_2$.

6.2 Fully-Interconnected Network

To show the applicability of our approach to strongly connected networks, we consider interconnected linear ct-SHS

$$\Sigma : \begin{cases} \mathsf{d}\xi(t) = (\bar{G}\xi(t) + B\nu(t))\mathsf{d}t + G\mathsf{d}\mathbb{W}_t + R\mathsf{d}\mathbb{P}_t, \\ \zeta(t) = \xi(t), \end{cases}$$

with matrix $\bar{G} = (-I_n - L) \in \mathbb{R}^{n\times n}$, where L is the Laplacian matrix of a complete graph [7]:

$$L = \begin{bmatrix} n-1 & -1 & \cdots & \cdots & -1 \\ -1 & n-1 & -1 & \cdots & -1 \\ -1 & -1 & n-1 & \cdots & -1 \\ \vdots & & \ddots & \ddots & \vdots \\ -1 & \cdots & \cdots & -1 & n-1 \end{bmatrix}_{n\times n}.$$

We partition $\xi(t) = [\xi_1(t); \ldots; \xi_n(t)]$, and $\nu(t) = [\nu_1(t); \ldots; \nu_n(t)]$. Moreover, $B = 0.15\mathbb{I}_n$ and $G = R = 0.1\mathbb{I}_n$. We also consider rates of Poisson processes as $\bar{\lambda}_i = 0.1, \forall i \in \{1, \ldots, n\}$. Now by considering the individual subsystems as

$$\Sigma_i : \begin{cases} \mathsf{d}\xi_i(t) = (-\xi_i(t) + 0.15\nu_i(t) + w_i(t))\mathsf{d}t + 0.1\mathsf{d}\mathbb{W}_{t_i} + 0.1\mathsf{d}\mathbb{P}_{t_i}, \\ \zeta_{1_i}(t) = \xi_i(t), \\ \zeta_{2_i}(t) = \xi_i(t), \end{cases}$$

one can readily verify that $\Sigma = \mathcal{I}(\Sigma_1, \ldots, \Sigma_N)$ where the coupling matrix M is defined as $M = -L$.

The regions of interest in this example are $X_i = [2\ 6], X_{0_i} = [2\ 4], X_{1_i} = [5\ 6], \forall i \in \{1, \ldots, n\}$. For the sake of simulation, we fix $n = 15$. The main goal is to find a CBC for the interconnected system and design its corresponding safety controller Σ maintaining the state of the interconnected system in the safe set $W = [2\ 5]^{15}$. According to Lemma 8, we compute CSC of order 4 as $\mathcal{B}_i(x_i) = 0.0002x_i^4 - 0.0024x_i^3 + 0.0109x_i^2 - 0.0207x_i + 0.0146$ and the corresponding safety controller $\nu_i(x_i) = -5.1465x_i^2 + 60.3564$ for all $i \in \{1, \ldots, 15\}$. The corresponding constants and functions in Definition 3 satisfying conditions (3)-(5) are computed as $\gamma_i = 10^{-4}, \lambda_i = 2 \times 10^{-3}, \kappa_i(s) = \hat{\kappa}_i s, \forall s \in \mathbb{R}_{\geq 0}$ with $\hat{\kappa}_i = 10^{-7}, \psi_i = 10^{-6}$, and

$$\bar{X}_i = \begin{bmatrix} 10^{-6} & 10^{-2} \\ 10^{-2} & -5 \times 10^{-4} \end{bmatrix}. \tag{22}$$

We now proceed with Theorem 7 to construct a CBC for the interconnected system using CSC of subsystems. By selecting $\mu_i = 1, \forall i \in \{1, \ldots, n\}$, and utilizing $\bar{X}_i$ in (22), the matrix $\bar{X}_{cmp}$ in (15) is reduced to

$$\bar{X}_{cmp} = \begin{bmatrix} 10^{-6}\mathbb{I}_n & 10^{-2}\mathbb{I}_n \\ 10^{-2}\mathbb{I}_n & -5 \times 10^{-4}\mathbb{I}_n \end{bmatrix},$$

and condition (13) is reduced to

$$\begin{bmatrix} -L \\ \mathbb{I}_n \end{bmatrix}^T \bar{X}_{cmp} \begin{bmatrix} -L \\ \mathbb{I}_n \end{bmatrix} = 10^{-6}L^TL - 10^{-2}(L + L^T) - 5 \times 10^{-4}\mathbb{I}_n \preceq 0,$$

which is always satisfied without requiring any restrictions on the number or gains of subsystems. In order to show the above LMI, we used $L = L^T \succeq 0$ which is always true for Laplacian matrices of undirected graphs. Moreover, the compositionality condition (14) is also satisfied since $\lambda_i > \gamma_i, \forall i \in \{1, \ldots, n\}$. Then by employing Theorem 7, one can conclude that $\mathcal{B}(x) =$

$\sum_{i=1}^{15}(0.0002x_i^4 - 0.0024x_i^3 + 0.0109x_i^2 - 0.0207x_i + 0.0146)$ is a CBC for the interconnected system Σ with $\gamma = 0.0015, \lambda = 0.03, \kappa(s) = 10^{-7}s, \forall s \in \mathbb{R}_{\geq 0}$, and $\psi = 1.5 \times 10^{-5}$. Accordingly, $\nu(x) = [-5.1465x_1^2 + 60.3564; \ldots; -5.1465x_{15}^2 + 60.3564]$ is the overall safety controller for the interconnected system.

By leveraging Theorem 5, one can guarantee that the state of the interconnected system Σ starting from initial conditions inside $X_0 = [2\ 4]^{15}$ remains in the safe set $[2\ 5]^{15}$ during the time horizon $T_d = 10$ with the probability of at least 95%, *i.e.,*

$$\mathbb{P}_\nu^{x_0}\Big\{\xi(t) \notin X_1 \mid \xi(0) = x_0,\ \forall t \in [0, 10]\Big\} \geq 0.95\,.$$

Closed-loop state trajectories of a representative subsystem with 10 different noise realizations are illustrated in Figure 7.

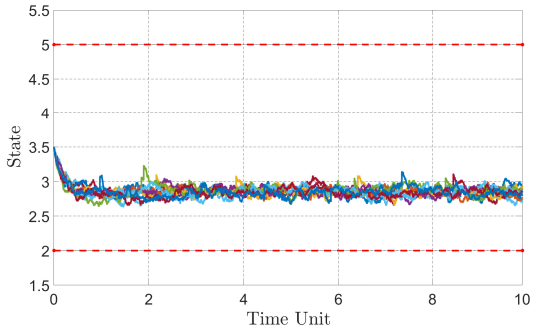


Figure 7 Closed-loop state trajectories of a representative subsystem with 10 noise realizations.

7 Conclusion

In this work, we proposed a compositional scheme based on dissipativity approaches for constructing control barrier certificates of large-scale continuous-time continuous-space stochastic hybrid systems while providing upper bounds on the probability that interconnected systems reach certain unsafe regions in finite-time horizons. The main goal was to synthesize control policies satisfying safety properties for interconnected systems by utilizing control storage certificates of subsystems. We constructed control barrier certificates for interconnected stochastic systems using control storage certificates of subsystems as long as some dissipativity-type compositional conditions hold. We employed a systematic approach based on the sum-of-squares optimization program and computed control storage certificates of subsystems. We illustrated our proposed results on two case studies with circular and fully-interconnected topologies.

References

1 M. Ahmadi, B. Wu, H. Lin, and U. Topcu. Privacy verification in POMDPs via barrier certificates. In *Proceedings of the 57th IEEE Conference on Decision and Control (CDC)*, pages 5610–5615, 2018.

2 M. Anand, A. Lavaei, and M. Zamani. Compositional construction of control barrier certificates for large-scale interconnected stochastic systems. *Proceedings of the 21st IFAC World Congress*, 53(2):1862–1867, 2020.

3 M. Arcak, C. Meissen, and A. Packard. *Networks of dissipative systems*. SpringerBriefs in Electrical and Computer Engineering. Springer, 2016.

4 H. E. Bell. Gershgorin's theorem and the zeros of polynomials. *The American Mathematical Monthly*, 72(3):292–295, 1965.

5 A. Clark. Control barrier functions for complete and incomplete information stochastic systems. In *Proceedings of the American Control Conference (ACC)*, pages 2928–2935, 2019.

6 A. Girard, G. Gössler, and S. Mouelhi. Safety controller synthesis for incrementally stable switched systems using multiscale symbolic models. *IEEE Transactions on Automatic Control*, 61(6):1537–1549, 2016.

7 C. Godsil and G. Royle. *Algebraic graph theory*. Graduate Texts in Mathematics. Springe, 2001.

8 C. Huang, X. Chen, W. Lin, Z. Yang, and X. Li. Probabilistic safety verification of stochastic hybrid systems using barrier certificates. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):186, 2017.

9 P. Jagtap, S. Soudjani, and M. Zamani. Formal synthesis of stochastic systems via control barrier certificates. *IEEE Transactions on Automatic Control*, 66(7):3097–3110, 2020.

10 H. J. Kushner. *Stochastic Stability and Control*. Mathematics in Science and Engineering. Elsevier Science, 1967.

11 A. Lavaei. *Automated Verification and Control of Large-Scale Stochastic Cyber-Physical Systems: Compositional Techniques*. PhD thesis, Technische Universität München, Germany, 2019.

12 A. Lavaei, S. Soudjani, A. Abate, and M. Zamani. Automated verification and synthesis of stochastic hybrid systems: A survey. *Automatica*, 2022.

13 A. Lavaei, S. Soudjani, and M. Zamani. Compositional construction of infinite abstractions for networks of stochastic control systems. *Automatica*, 107:125–137, 2019.

14 A. Lavaei, S. Soudjani, and M. Zamani. Compositional abstraction-based synthesis for networks of stochastic switched systems. *Automatica*, 114, 2020.

15 A. Lavaei, S. Soudjani, and M. Zamani. Compositional abstraction of large-scale stochastic systems: A relaxed dissipativity approach. *Nonlinear Analysis: Hybrid Systems*, 36, 2020.

16 A. Lavaei, S. Soudjani, and M. Zamani. Compositional (in)finite abstractions for large-scale interconnected stochastic systems. *IEEE Transactions on Automatic Control*, 65(12):5280–5295, 2020.

17 A. Nejati, S. Soudjani, and M. Zamani. Compositional construction of control barrier certificates for large-scale stochastic switched systems. *IEEE Control Systems Letters*, 4(4):845–850, 2020.

18 A. Nejati, S. Soudjani, and M. Zamani. Compositional construction of control barrier functions for networks of continuous-time stochastic systems. *Proceedings of the 21st IFAC World Congress*, 53(2):1856–1861, 2020.

19 A. Nejati, S. Soudjani, and M. Zamani. Compositional abstraction-based synthesis for continuous-time stochastic hybrid systems. *European Journal of Control*, 57:82–94, 2021.

20 A. Nejati and M. Zamani. Compositional construction of finite MDPs for continuous-time stochastic systems: A dissipativity approach. *Proceedings of the 21st IFAC World Congress*, 53(2):1962–1967, 2020.

21 B. Oksendal. *Stochastic differential equations: an introduction with applications*. Springer Science & Business Media, 2013.

22 B. K. Øksendal and A. Sulem. *Applied stochastic control of jump diffusions*, volume 498. Springer, 2005.

23 A. Papachristodoulou, J. Anderson, G. Valmorbida, S. Prajna, P. Seiler, and P. Parrilo. SOSTOOLS version 3.00 sum of squares optimization toolbox for MATLAB. *arXiv:1310.4716*, 2013.

24 P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical programming*, 96(2):293–320, 2003.

25 A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

26 S. Prajna, A. Jadbabaie, and G. J. Pappas. A framework for worst-case and stochastic safety verification using barrier certificates. *IEEE Transactions on Automatic Control*, 52(8):1415–1428, 2007.

27 K. Ross. Stochastic control in continuous time. *Lecture Notes on Continuous Time Stochastic Control*, pages P33–P37, 2008.

28 J. F. Sturm. Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones. *Optimization methods and software*, 11(1-4):625–653, 1999.

29 R. Wisniewski and M. L. Bujorianu. Stochastic safety analysis of stochastic hybrid systems. In *Proceedings of the 56th IEEE Conference on Decision and Control*, pages 2390–2395, 2017.

30 T. Wongpiromsarn, U. Topcu, and A. Lamperski. Automata theory meets barrier certificates: Temporal logic verification of nonlinear systems. *IEEE Transactions on Automatic Control*, 61(11):3344–3355, 2015.

8 Appendix

Proof of Theorem 5. Based on condition (8), we have $X_1 \subseteq \{x \in X \mid \mathcal{B}(x) \geq \lambda\}$. Then one has

$$\mathbb{P}_{\nu}^{x_0}\Big\{\xi(t) \in X_1 \text{ for some } 0 \leq t \leq T_d \mid \xi(0) = x_0\Big\} \leq \mathbb{P}_{\nu}^{x_0}\Big\{\sup_{0\leq t\leq T_d} \mathcal{B}(\xi(t)) \geq \lambda \mid \xi(0) = x_0\Big\}. \tag{23}$$

One can acquire the upper bound in (10) by applying [10, Theorem 1, Chapter III] to (23) and respectively utilizing conditions (9) and (7). ◀

Proof of Theorem 7. We first show that conditions (7) and (8) in Definition 4 hold. For any $x := [x_1; \ldots; x_N] \in X_0 = \prod_{i=1}^{N} X_{0_i}$ and from (3)

$$\mathcal{B}(x) = \sum_{i=1}^{N} \mu_i \mathcal{B}_i(x_i) \leq \sum_{i=1}^{N} \mu_i \gamma_i = \gamma,$$

and similarly for any $x := [x_1; \ldots; x_N] \in X_1 = \prod_{i=1}^{N} X_{1_i}$ and from (4)

$$\mathcal{B}(x) = \sum_{i=1}^{N} \mu_i \mathcal{B}_i(x_i) \geq \sum_{i=1}^{N} \mu_i \lambda_i = \lambda,$$

satisfying conditions (7) and (8) with $\gamma = \sum_{i=1}^{N} \mu_i \gamma_i$ and $\lambda = \sum_{i=1}^{N} \mu_i \lambda_i$. Note that $\lambda > \gamma$ according to (14). Now, we show that the condition (9) holds, as well. One can obtain the chain of inequalities in (24) using condition (13) and by defining $\kappa(\cdot), \psi$ as

$$\kappa(s) := \min \Big\{ \sum_{i=1}^{N} \mu_i \kappa_i(s_i) \;\Big|\; s_i \geq 0,\ \sum_{i=1}^{N} \mu_i s_i = s \Big\},$$
$$\psi := \sum_{i=1}^{N} \mu_i \psi_i.$$

Then $\mathcal{B}$ is a CBC for Σ, which completes the proof. ◀

Proof of Lemma 8. Since condition (16) is sum-of-squares, we have $0 \leq \mathcal{B}_i(x_i) - l_{0_i}^T(x_i) g_i(x_i) - \gamma_i$. Since the term $l_{0_i}^T(x_i) g_{0_i}(x_i)$ is non-negative over X_0, the new condition (16) implies the condition (3) in Definition 3. Similarly, one can show that (17) implies condition (4) in Definition 3. Now we show that condition (18) implies (5), as well. By selecting external inputs $\nu_{ji} = l_{\nu_{ji}}(x_i)$ and since terms $l_i^T(x_i, \nu_i, w_i) g_i(x_i), l_{\nu_i}^T(x_i, \nu_i, w_i) g_{\nu_i}(\nu_i), l_{w_i}^T(x_i, \nu_i, w_i) g_{w_i}(w_i)$ are non-negative over the set X, we have

$$\mathcal{L}\mathcal{B}_i(x_i) \leq -\kappa_i(\mathcal{B}_i(x_i)) + \psi_i + \begin{bmatrix} w_i \\ h_{2_i}(x_i) \end{bmatrix}^T \begin{bmatrix} \bar{X}_i^{11} & \bar{X}_i^{12} \\ \bar{X}_i^{21} & \bar{X}_i^{22} \end{bmatrix} \begin{bmatrix} w_i \\ h_{2_i}(x_i) \end{bmatrix},$$

which implies that the function $\mathcal{B}_i(x_i)$ is a CSC and completes the proof. ◀

$$
\begin{aligned}
\mathcal{LB}(x) &= \mathcal{L}\sum_{i=1}^{N}\mu_i\mathcal{B}_i(x_i) = \sum_{i=1}^{N}\mu_i\mathcal{LB}_i(x_i)\\
&\leq \sum_{i=1}^{N}\mu_i\big(-\kappa_i(\mathcal{B}_i(x_i)) + \psi_i + \begin{bmatrix} w_i \\ h_{2_i}(x_i)\end{bmatrix}^T \begin{bmatrix} \bar{X}_i^{11} & \bar{X}_i^{12} \\ \bar{X}_i^{21} & \bar{X}_i^{22}\end{bmatrix}\begin{bmatrix} w_i \\ h_{2_i}(x_i)\end{bmatrix}\big)\\
&= \sum_{i=1}^{N} -\mu_i\kappa_i(\mathcal{B}_i(x_i)) + \sum_{i=1}^{N}\mu_i\psi_i\\
&\quad + \begin{bmatrix} w_1 \\ \vdots \\ w_N \\ h_{2_1}(x_1) \\ \vdots \\ h_{2_N}(x_N)\end{bmatrix}^T \begin{bmatrix} \mu_1\bar{X}_1^{11} & & & \mu_1\bar{X}_1^{12} & & \\ & \ddots & & & \ddots & \\ & & \mu_N\bar{X}_N^{11} & & & \mu_N\bar{X}_N^{12} \\ \mu_1\bar{X}_1^{21} & & & \mu_1\bar{X}_1^{22} & & \\ & \ddots & & & \ddots & \\ & & \mu_N\bar{X}_N^{21} & & & \mu_N\bar{X}_N^{22}\end{bmatrix}\begin{bmatrix} w_1 \\ \vdots \\ w_N \\ h_{2_1}(x_1) \\ \vdots \\ h_{2_N}(x_N)\end{bmatrix}\\
&= \sum_{i=1}^{N} -\mu_i\kappa_i(\mathcal{B}_i(x_i)) + \sum_{i=1}^{N}\mu_i\psi_i\\
&\quad + \begin{bmatrix} M\begin{bmatrix} h_{2_1}(x_1) \\ \vdots \\ h_{2_N}(x_N)\end{bmatrix} \\ h_{2_1}(x_1) \\ \vdots \\ h_{2_N}(x_N)\end{bmatrix}^T \begin{bmatrix} \mu_1\bar{X}_1^{11} & & & \mu_1\bar{X}_1^{12} & & \\ & \ddots & & & \ddots & \\ & & \mu_N\bar{X}_N^{11} & & & \mu_N\bar{X}_N^{12} \\ \mu_1\bar{X}_1^{21} & & & \mu_1\bar{X}_1^{22} & & \\ & \ddots & & & \ddots & \\ & & \mu_N\bar{X}_N^{21} & & & \mu_N\bar{X}_N^{22}\end{bmatrix}\begin{bmatrix} M\begin{bmatrix} h_{2_1}(x_1) \\ \vdots \\ h_{2_N}(x_N)\end{bmatrix} \\ h_{2_1}(x_1) \\ \vdots \\ h_{2_N}(x_N)\end{bmatrix}\\
&= \sum_{i=1}^{N} -\mu_i\kappa_i(\mathcal{B}_i(x_i)) + \sum_{i=1}^{N}\mu_i\psi_i\\
&\quad + \begin{bmatrix} h_{2_1}(x_1) \\ \vdots \\ h_{2_N}(x_N)\end{bmatrix}^T \begin{bmatrix} M \\ \mathbb{I}_{\tilde{q}}\end{bmatrix}^T \begin{bmatrix} \mu_1\bar{X}_1^{11} & & & \mu_1\bar{X}_1^{12} & & \\ & \ddots & & & \ddots & \\ & & \mu_N\bar{X}_N^{11} & & & \mu_N\bar{X}_N^{12} \\ \mu_1\bar{X}_1^{21} & & & \mu_1\bar{X}_1^{22} & & \\ & \ddots & & & \ddots & \\ & & \mu_N\bar{X}_N^{21} & & & \mu_N\bar{X}_N^{22}\end{bmatrix}\begin{bmatrix} M \\ \mathbb{I}_{\tilde{q}}\end{bmatrix}\begin{bmatrix} h_{2_1}(x_1) \\ \vdots \\ h_{2_N}(x_N)\end{bmatrix}\\
&\leq \sum_{i=1}^{N} -\mu_i\kappa_i(\mathcal{B}_i(x_i)) + \sum_{i=1}^{N}\mu_i\psi_i \leq -\kappa(\mathcal{B}(x)) + \psi. \qquad (24)
\end{aligned}
$$

Real-Time Verification for Distributed Cyber-Physical Systems

Hoang-Dung Tran
University of Nebraska-Lincoln, Lincoln, Nebraska, USA

Luan Viet Nguyen
University of Dayton, Dayton, Ohio, USA

Patrick Musau
Vanderbilt University, Nashville, Tennessee, USA

Weiming Xiang
Augusta University, Nashville, Tennessee, USA

Taylor T. Johnson
Vanderbilt University, Nashville, Tennessee, USA

Abstract

Safety-critical distributed cyber-physical systems (CPSs) have been found in a wide range of applications. Notably, they have displayed a great deal of utility in intelligent transportation, where autonomous vehicles communicate and cooperate with each other via a high-speed communication network. Such systems require an ability to identify maneuvers in real-time that cause dangerous circumstances and ensure the implementation always meets safety-critical requirements. In this paper, we propose a real-time decentralized reachability approach for safety verification of a distributed multi-agent CPS with the underlying assumption that all agents are time-synchronized with a low degree of error. In the proposed approach, each agent periodically computes its local reachable set and exchanges this reachable set with the other agents with the goal of verifying the system safety. Our method, implemented in Java, takes advantages of the timing information and the reachable set information that are available in the exchanged messages to reason about the safety of the whole system in a decentralized manner. Any particular agent can also perform local safety verification tasks based on their local clocks by analyzing the messages it receives. We applied the proposed method to verify, in real-time, the safety properties of a group of quadcopters performing a distributed search mission.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies
Keywords and Phrases Verification, Reachability Analysis, Distributed Cyber-Physical Systems
Digital Object Identifier 10.4230/LITES.8.2.7
Related Version *Previous Version*: https://doi.org/10.1007/978-3-030-21759-4_15 [31]
Supplementary Material *Software (Source Code)*: https://github.com/verivital/rtreach
Funding The material presented in this paper is based upon work supported by the Air Force Office of Scientific Research (AFOSR) through contract number FA9550-22-1-0019 and the Defense Advanced Research Projects Agency (DARPA) through contract number FA8750-18-C-0089. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFOSR or DARPA.
Received 2020-10-22 **Accepted** 2022-01-28 **Published** 2022-12-07
Editor Alessandro Abate, Uli Fahrenberg, and Martin Fränzle
Special Issue Special Issue on Distributed Hybrid Systems

1 Introduction

The emergence of 5G technology has inspired a massive wave of the research and development in science and technology in the era of IoT where the communication between computing devices has become significantly faster with lower latency and power consumption. The power of this modern communication technology influences and benefits all aspects of Cyber-Physical Systems (CPSs) such as smart grids, smart homes, intelligent transportation and smart cities. In particular, the study of autonomous vehicles has become an increasingly popular research field in both academic and industrial transportation applications. Automotive crashes pose significant financial and life-threatening risks, and there is an urgent need for advanced and scalable methods that can efficiently verify a distributed system of autonomous vehicles.

Over the last two decades, although many methods have been developed to conduct reachability analysis and safety verification of CPS, such as the approaches proposed in [1, 4, 5, 7, 13, 14, 17, 20, 23, 30, 32], applying these techniques to *real-time distributed* CPS remains a big challenge. This is due to the fact that, 1) all existing techniques have intensive computation costs and are usually too slow to be used in a real-time manner and, 2) these techniques target the safety verification of a *single* CPS, and therefore they naturally cannot be applied efficiently to a *distributed* CPS where clock mismatches and communication between agents (i.e., individual systems) are essential concerns. Since the future autonomous vehicles systems will work distributively involving effective communication between each agent, there is an urgent need for an approach that can provide formal guarantees of the safety of distributed CPS in real-time. More importantly, the safety information should be defined based on the *agents local clocks* to allow these agents to perform "intelligent actions" to escape from the upcoming dangerous circumstances. For example, if an agent A knows based on its local clock that it will collide with an agent B in the next 5 seconds, it should perform an action such as stopping or quickly finding a safe path to avoid the collision.

In this paper[1], we propose a *decentralized real-time reachability* approach for safety verification of a distributed CPS with multiple agents. We are particularly interested in two types of safety properties. The first one is a *local safety property* which specifies the local constraints of the agent operation. For example, each agent is only allowed to move within a specific region, does not hit any obstacles, and its velocity needs to be limited to a specific range. This type of property does not require the information of other agents and can be verified locally at run-time. The second safety property is a *global property* defined on the states of multiple agents. Particularly, we consider a peer-to-peer collision free property and a generalized property where we want to verify if all agents satisfy a set of linear constraints (on the states of all agents) defining the property, e.g., two agents do not go into the same region at the same time.

Our decentralized real-time reachability approach works as follows. Each agent *locally* and *periodically computes* the local reachable set of states from the current local time to the next T seconds, and then *encodes* and *broadcasts* its reachable set information to the others via a communication network. When the agent receives a reachable set message, it immediately *decodes* the message to read the reachable set information of the sender, and then performs *peer-to-peer collision checking* based on its current state and the reachable set of the sender. Verifying a generalized global property involving the states of N agents is done at the time an agent receives all needed reachable sets from other agents. Additionally, the local safety property of the agent is verified simultaneously with the reachable set computation process at run-time. The proposed verification approach is based on an underlying assumption that is, all agents are time-synchronized to some level of accuracy. This assumption is reasonable as it can be achieved by using existing

[1] This paper is an extension of [31].

time synchronization protocols such as the Network Time Protocol (NTP). Our approach has successfully verified in real-time the local safety properties and collision occurrences for a group of quadcopters conducting a search mission.

The rest of the paper is organized as follows. Section 2 presents briefly the distributed CPS modeling and its verification problems. Section 3 gives the detail of real-time reachability for single agent and how to use it for real-time local safety verification. Section 4 addresses the utilization reachable set messages for checking peer-to-peer collision. Section 5 investigates the global safety verification problem. Section 6 presents the implementation and evaluation of our approach via a distributed search application using quadcopters.

2 Problem Formulation

In this paper, we consider a distributed CPS with N agents that can communicate with each other via an asynchronous communication channel.

Communication Model

The communication between agents is implemented by the *actions* of sending and receiving messages over an asynchronous communication channel. We formally model this communication model as a single automaton, Channel, which stores the set of in-flight messages that have been sent, but are yet to be delivered. When an agent sends a message m, it invokes a *send(m)* action. This action adds m to the *in-flight* set. At any arbitrary time, the Channel chooses a message in the in-flight set to either delivers it to its recipient or removes it from the set. All messages are assumed to be unique and each message contains its sender and recipient identities. Let M be the set of all possible messages used in communication between agents. The sending and receiving messages by agent i are denoted by $M_{i,*}$ and $M_{*,i}$, respectively.

Agent Model

The i^{th} agent is modeled as a hybrid automaton [16, 27] defined by the tuple $\langle \mathcal{A}_i = V_i, A_i, \mathcal{D}_i, \mathcal{T}_i \rangle$, where:

a) V_i is a set of variables consisting of the following:
- **i)** a set of continuous variables X_i including a special variable clk_i which records the agent's *local time*, and
- **ii)** a set of discrete variables Y_i including the special variable $msghist_i$ that records all sent and received messages. A valuation $\mathbf{v}_i$ is a function that associates each $v_i \in V_i$ to a value in its type. We write $val(V_i)$ for the set of all possible valuations of V_i. We abuse the notion of $\mathbf{v}_i$ to denote a state of $\mathcal{A}_i$, which is a valuation of all variables in V_i.

The set $Q_i \triangleq val(V_i)$ is called the set of *states*.

b) A_i is a set of *actions* consisting of the following subsets:
- **i)** a set $\{send_i(m) \mid m \in M_{i,*}\}$ of send actions (i.e., output actions),
- **ii)** a set $\{receive_i(m) \mid m \in M_{*,i}\}$ of receive actions (i.e., input actions), and
- **iii)** a set H_i of other, ordinary actions.

c) $\mathcal{D}_i \subseteq val(V_i) \times A_i \times val(V_i)$ is called the set of *transitions*. For a transition $(\mathbf{v}_i, a_i, \mathbf{v}'_i) \in \mathcal{D}_i$, we write $\mathbf{v}_i \xrightarrow{a_i} \mathbf{v}'_i$ in short.
- **i)** If $a_i = send_i(m)$ or $receive_i(m)$, then all the components of $\mathbf{v}_i$ and $\mathbf{v}'_i$ are identical except that m is added to $msghist$ in $\mathbf{v}'_i$. That is, the agent's other states remain the same on message sends and receives. Furthermore, for every state $\mathbf{v}_i$ and every receive action a_i, there must exist a $\mathbf{v}'_i$ such that $\mathbf{v}_i \xrightarrow{a_i} \mathbf{v}'_i$, i.e., the automaton must have well-defined behavior for receiving any message in any state.
- **ii)** If $a_i \in H_i$, then $\mathbf{v}_i.msghist = \mathbf{v}'_i.msghist$.

d) $\mathcal{T}_i$ is a collection of trajectories for X_i. Each trajectory of X_i is a function mapping an interval of time $[0, t], t \geq 0$ to $val(V_i)$, following a flow rate that specifies how a real variable $x_i \in X_i$ evolving over time. We denote the *duration* of a trajectory as τ_{dur}, which is the right end-point of the interval t.

Agent Semantics

The *behavior* of each agent can be defined based on the concept of an *execution* which is a particular run of the agent. Given an initial state $\mathbf{v}_i^0$, an *execution* α_i of an agent A_i is a sequence of states starting from $\mathbf{v}_i^0$, defined as $\alpha_i = \mathbf{v}_i^0, \mathbf{v}_i^1, \ldots$, and for each index j in the sequence, the state update from $\mathbf{v}_i^j$ to $\mathbf{v}_i^{j+1}$ is either a transition or trajectory. A state $\mathbf{v}_i^j$ is *reachable* if there exists an executing that ends in $\mathbf{v}_i^j$. We denote $\mathsf{Reach}(A_i)$ as the reachable set of agent A_i.

System Model

The formal model of the complete system, denoted as System, is a network of hybrid automata that is obtained by parallel composing the agent's models and the communication channel. Formally, we can write, $\mathsf{System} \triangleq \mathcal{A}_1 \| \ldots \mathcal{A}_N \| \mathsf{Channel}$. Informally, the agent $\mathcal{A}_i$ and the communication channel $\mathsf{Channel}$ are synchronized through sending and receiving actions. When the agent A_i sends a message $m \in M_{i,j}$ to the agent A_j, it triggers the $send_i(m)$ action. At the same time, this action is synchronized in the $\mathsf{Channel}$ automaton by putting the message m in the *in-flight* set. After that, the $\mathsf{Channel}$ will trigger (non-deterministically) the $receive_j(m)$ action. This action is synchronized in the agent A_j by putting the message m into the message history $msghist_j$.

In this paper, we investigate three real-time safety verification problems for distributed cyber-physical systems as defined in the following.

▶ Problem 1 (Local safety verification in real-time). The real-time local safety verification problem is to compute online the reachable set $\mathsf{Reach}(A_i)$ of the agent and verify if it violates the local safety property, i.e., checking $\mathsf{Reach}(A_i) \cap \mathcal{U}_i = \emptyset?$, where $\mathcal{U}_i \triangleq \{x_i | C_i x_i \leq d_i, x_i \in X_i\}$ is the unsafe set of the agent.

▶ Problem 2 (Decentralized real-time collision verification). The decentralized real-time collision verification problem is to reason in real-time whether an agent A_i will collide with other agents from its current local time t_c^i to the *computable, safe time instance in the future* T_{safe} based on

i) the *clock mismatches*, and

ii) the *exchanging reachable set messages* between agents.

Formally, we require that $\forall\ t_c^i \leq t \leq T_{safe}, d_{ij}(t) \geq l$, where $d_{ij}(t)$ is the distance between agents A_i and A_j at the time t of the agent A_i local clock, and l is the allowable safe distance between agents.

▶ Problem 3 (Decentralized real-time global safety verification). The decentralized real-time global safety verification problem is to construct online (at each agent) the reachable set of all agents $globalReach$ and verify if it violates the global safety property, i.e., checking $globalReach \cap \mathcal{U} = \emptyset$, where $\mathcal{U} \triangleq Cx \leq d$, $x = [x_1^T, \ldots, x_N^T]^T, x_i \in X_i$, is the unsafe set of the whole system.

3 Real-Time Local Safety Verification

The first important step in our approach is, each agent A_i computes forwardly its reachable set of states from the current local time t^i to the next $(t^i + T)$ seconds which is defined by $\mathcal{R}_i[t^i, t^i + T]$. Since there are many variables used in the agent modeling that are irrelevant in safety verification, we only need to compute the reachable set of state that is related to the agent's physical dynamics

(so called as *motion dynamics*) which is defined by a nonlinear ODE $\dot{x}_i = f(x_i, u_i)$, where $x_i \in \mathbb{R}^n$ is state vector and $u_i \in \mathbb{R}^m$ is the control input vector. The agent can switch from one mode to the another mode via discrete transitions, and in each mode, the control law may be different. When the agent computes its reachable set, the only information it needs are its current set of states $x_i(t^i)$ and the current control input $u_i(t^i)$. It should be clarified that although the control law may be different among modes, the control signal u_i is updated with the same control period T_c^i. Consequently, u_i is a constant vector in each control period.

Assuming that the agent's current time is $t_j^i = j \times T_c$, using its local sensors and GPS, we have the current state of the agent x_i. Note that the local sensors and the provided GPS can only provide the information of interest to some accuracy, therefore the actual state of the agent is in a set $x_i \in I_i$. The control signal u_i is computed based on the state x_i and a reference signal, e.g., a set point denoting where the agent needs to go to, and then computed control signal is applied to the actuator to control the motion of the agent. From the current set of states I_i and the control signal u_i, we can compute the forward reachable set of the agent for the next $t_j^i + T$ seconds. This reachable set computation needs to be completed after an amount of time $T_{runtime}^i < T_c^i$ because if $T_{runtime}^i \geq T_c^i$, a new u_i will be updated. The control period T_c^i is chosen based on the agent's motion dynamics, and thus to control an agent with fast dynamics, the control period T_c^i needs to be sufficiently small. This is the source of the requirement that the allowable run-time for reachable set computation be small.

To compute the reachable set of an agent in real-time, we use the well-known face-lifting method [6, 9] and a *hyper-rectangle* to represent the reachable set. This method is useful for short-time reachability analysis of real-time systems. It allows users to define an allowable run-time $T_{runtime}^i$, and has no dynamic data structures, recursion, and does not depend on complex external libraries as in other reachability analysis methods. More importantly, the accuracy of the reachable set computation can be iteratively improved based on the *remaining allowable run-time*.

Algorithm 3.1 describes the real-time reachability analysis for one agent. The Algorithm works as follows. The time period $[t^i, t^i + T]$ is divided by M steps. The reach time step is defined by $h_i = T/M$. Using the reach time step and the current set I_i, the face-lifting method performs a *single-face-lifting operation*. The results of this step are a new reachable set and a *remaining reach time* $T_{remainReachTime}^i < T$. This step is iteratively called until the reachable set for the whole time period of interest $[t^i, t^i + T]$ is constructed completely, i.e., the remaining reach time is equal to zero. Interestingly, with the reach time step size h_i defined above, the face-lifting algorithm may be finished quickly after an amount of time which is smaller than the allowable run-time $T_{runtime}^i$ specified by user, i.e., there is still an amount of time called remaining run time $T_{remainRunTime}^i < T_{runtime}^i$ that is available for us to recall the face-lifting algorithm with a smaller reach time step size, for example, we can recall the face-lifting algorithm with a new reach time step $h_i/2$. By doing this, the conservativeness of the reachable set can be iteratively improved. The core step of face-lifting method is the single-face-lifting operation. We refer the readers to [6] for further detail. As mentioned earlier, the local safety property of each agent can be verified at run-time simultaneously with the reachable set computation process. Precisely, let $\mathcal{U}_i \triangleq C_i x_i \leq d_i$ be the unsafe region of the i^{th} agent, the agent is said to be safe from t^i to $t^i + t \leq t^i + T$ if $\mathcal{R}_i[t^i, t^i + t] \cap \mathcal{U}_i = \emptyset$. Since the reachable set $\mathcal{R}_i[t^i, t^i + t]$ is given by the face-lifting method at run-time, the local safety verification problem for each agent can be solved at run-time. Since Algorithm 3.1 computes an over-approximation of the reachable set of each agent in a short time interval, it guarantees the soundness of the result as described in the following lemma.

▶ **Lemma 1** ([6, 9]). *The real-time reachability analysis algorithm is sound, i.e., the computed reachable set contains all possible trajectories of agent A_i from t^i to $t^i + T$.*

Algorithm 3.1 Real-time reachability analysis for agent A_i.

Input: I_i, u_i, t^i, T, h_i, $T^i_{runtime}$, $\mathcal{U}_i$
Output: $\mathcal{R}_i[t^i, t^i + T]$, $safe = true$ or $safe = uncertain$

procedure INITIALIZATION
  $step = h_i$ % Reach time step
  $T_1^i = T^i_{runtime}$ % Remaining run-time
procedure REACHABILITY ANALYSIS
  while $(T_1^i > 0)$ **do**
    $\mathcal{CR} = I_i$ % Current reachable set
    $safe = true$
    $T_2^i = T$ % Remaining reach time
    while $T_2^i > 0$ **do**
      % Do Single Face Lifting
      $\mathcal{R}, T' = SFL(\mathcal{CR}, step, T_2^i, u_i)$
      $\mathcal{CR} = \mathcal{R}$ % Update reach set
      $T_2^i = T'$ % Update remaining reach time
      if $(\mathcal{CR} \cap \mathcal{U}_i \neq \emptyset)$ **then**: $safe = uncertain$
      $\mathcal{R}_i[t^i, t^i + T] = \mathcal{CR}$
    % *Update remaining runtime*
    $T_1^i = T_1^i - (A_i.currentTime() - t^i)$
    if $T_1^i \leq 0$ **then** break
    else
      $step = h_i/2$ % Reduce reach time step
  return $\mathcal{R}_i[t^i, t^i + T] = \mathcal{CR}, safe$

4 Decentralized Real-Time Collision Verification

Our collision verification scheme is performed based on the exchanged reachable set messages between agents. For every control period T_c, each agent executes the real-time reachability analysis algorithm to check if it is locally safe and to obtain its current reachable set with respect to its current control input. When the current reachable set is available, the agent encodes the reachable set in a message and then broadcasts this message to its cooperative agents and listens to the upcoming messages sent from these agents. When a reachable set message arrives, the agent immediately decodes the message to construct the current reachable set of the sender and then performs peer-to-peer collision detection. The process of computing, encoding, transferring, decoding of the reachable set along with collision checking is illustrated in Figure 1 based on the agent's local clock.

Let t^i_{rs}, t^i_e, t^i_{tf}, t^i_d, and t^i_c respectively be the instants at which we compute, encode, transfer, decode the reachable set and do collision checking on the agent A_i. Note that these time instants are based on the agent A_i's local clock. The actual run-times are defined as follows.

$\tau^i_{rs} = t^i_e - t^i_{rs}$, % *reachable set computation time*,
$\tau^i_e = t^i_{tf} - t^i_e$, % *encoding time*,
$\tau^i_{tf} \approx t^j_d - t^i_{tf}$, % *transferring time*,
$\tau^i_d = t^i_c - t^i_d$, % *decoding time*.

Note that we do not know the exact transfer time τ^i_{tf} since it depends on two different local time clocks. The above transfer time formula describes its approximate value when neglecting the mismatch between the two local clocks. The actual reachable set computation time is close to the

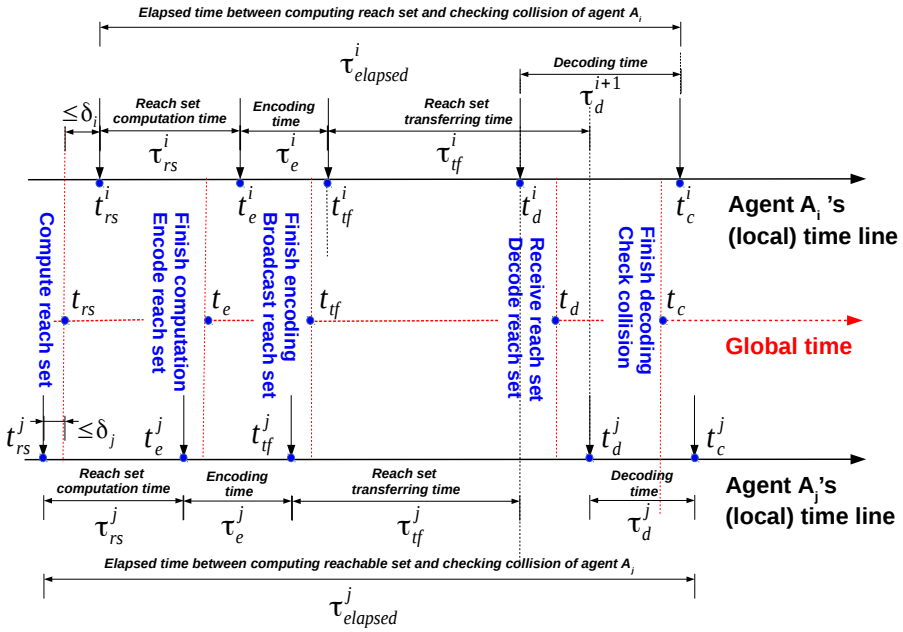


Figure 1 Timeline for reachable set computing, encoding, transferring, decoding and collision checking. The timeline for these core steps in verification is plotted in parallel with the virtual global time under the assumption that the agent's clock is synchronized with the global time within an error between $-\delta_*$ and δ_*.

allowable run-time chosen by user, i.e., $\tau^i_{rs} \approx T^i_{runtime}$. We will see later that the encoding time and decoding time are fairly small in comparison with the transferring time, i.e., $\tau^i_e \approx \tau^i_d \ll \tau^i_{tf}$. All of these run-times provide useful information for selecting an appropriate control period T_c for an agent. However, for collision checking purposes, we only need to consider the time instants that an agent starts computing reachable set t^i_{rs} and collision checking t^i_c.

A reachable set message contains three pieces of information: the reachable set which is a list of intervals, the time period (based on the local clock) in which this reachable set is valid, i.e., the start time t^i_{rs} and the end time $t^i_{rs} + T$ and the time instant that this message is sent. Based on the timing information of the reachable set and the time-synchronization errors, an agent can examine whether or not a received reachable set contains information about the future behavior of the sent agent which is useful for collision checking. The usefulness of the reachable sets used in collision checking is defined as follows.

▶ **Definition 2** (Useful reachable sets)**.** Let δ_i and δ_j respectively be the time-synchronization errors of agent A_i and A_j in comparison with the *virtual global time* t, i.e, $t - \delta_i \le t^i \le t + \delta_i$ and $t - \delta_j \le t^j \le t + \delta_j$, where t^i and t^j are current local times of A_i and A_j respectively. The reachable sets $\mathcal{R}_i[t^i_{rs}, t^i_{rs} + T]$ and $\mathcal{R}_j[t^j_{rs}, t^j_{rs} + T]$ of the agent A_j that are available at the agent A_i at time t^i_c are *useful* for collision checking between A_i and A_j if:

$$\begin{aligned} &t^i_c < t^j_{rs} + T - \delta_i - \delta_j, \\ &t^i_c < t^i_{rs} + T. \end{aligned} \tag{1}$$

Assume that we are at a time instant where the agent A_i checks if a collision occurs. This means that the current local time is t^i_c. Note that agent A_i and A_j are synchronized to the global time with errors δ_i and δ_j respectively. The reachable set $\mathcal{R}_j[t^j_{rs}, t^j_{rs} + T]$ is useful if it contains information about the *future behavior* of agent A_j under the view of the agent A_i based on its

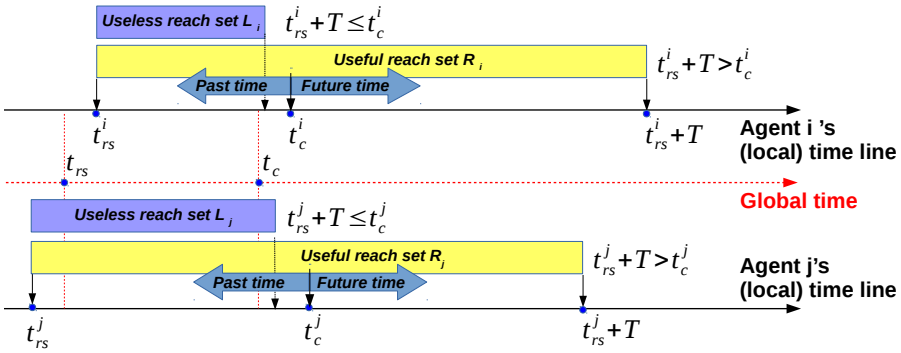


Figure 2 Useful reachable sets. An exchanged reachable set is useful for real-time verification if and only if it contains the estimation of all possible trajectories of an agent in a time period in the future.

Algorithm 4.2 Decentralized Real-Time Collision Verification at Agent A_i.

Input: l, % safe distance between agents
Output: $collision, T_{safe}$ % collision flag and safe time interval in the future

procedure PEER-TO-PEER COLLISION DETECTION
  if new message $\mathcal{R}_j[t^j_{rs}, t^j_{rs}+T]$ arrive **then**
    decode message
    $t^i_c = A_i.current\_time()$ % current time
    $t^i_{rs} = \mathcal{R}_i.t^i_{rs}$ % current reachable set start time
    if $t^i_c < t^j_{rs} + T - \delta_i - \delta_j$ and $t^i_c < t^i_{rs} + T$ **then** % check usefulness
      compute possible minimum distance d_{min} between two agents
      if $d_{min} > l$ **then**
        Collision = false
        $T_{safe} = min(t^j_{rs} + T - \delta_i - \delta_j, t^i_{rs} + T)$
      else
        Collision = uncertain, $T_{safe} = [\,]$
      store the message

local clock. This can be guaranteed if we have: $t^j_{rs} + T \geq t^i_{rs} - \delta_j + T > t^i_c + \delta_i$. Additionally, the current reachablet set of agent A_i contains information about its future behavior if $t^i_c < t^i_{rs} + T$ as depicted in Figure 2. We can see that if $t^i_c > t^j_{rs} + T + \delta_i + \delta_j$, then the reachable set of A_j contains a past information, and thus it is useless for collision checking. One interesting case is when $t^j_{rs} + T - \delta_i - \delta_j < t^i_c < t^j_{rs} + T + \delta_i + \delta_j$. In this case, we do not know whether the received reachable set is useful or not.

▶ Remark. We note that the proposed approach does not rely on the concept of Lamport's happens-before relation [22] to compute the local reachable set of each agent. If the agent could not receive reachable messages from others until a requested time-stamp expires, it still calculates the local reachable set based on its current state and the state information of other agents in the messages it received previously. In other words, our method does not require the reachable set of each agent to be computed corresponding to the ordering of the events (sending or receiving a message) in the system, but only relies on the local clock period and the time-synchronization errors between agents. Such implementation ensures that the computation process can be accomplished in real-time, and is not affected by the message transmission delay.

The peer-to-peer collision checking procedure depicted in Algorithm 4.2 works as follows: when a new reachable set message arrives, the receiving agent decodes the message and checks the usefulness of the received reachable set and its current reachable set. Then, the agent combines

its current reachable set and the received reachable set to compute the minimum possible distance between two agents. If the distance is larger than an allowable threshold l, there is no collision between two agents in some known time interval in the future, i.e., T_{safe}.

▶ **Lemma 3.** *The decentralized real-time collision verification algorithm is sound.*

Proof. From Lemma 1, we know that the received reachable set $\mathcal{R}_j[t^j_{rs}, t^j_{rs} + T]$ contains all possible trajectories of the agent A_j from t^j_{rs} to $t^j_{rs} + T$. Also, the current reachable set of the agent A_i, $\mathcal{R}_i[t^i_{rs}, t^i_{rs} + T]$, contains all possible trajectories of the agent from t^i_{rs} to $t^i_{rs} + T$. If those reachable sets are useful, then they contains all possible trajectories of two agents from t^i_c to sometime $T_{safe} = min(t^j_{rs} + T - \delta_i - \delta_j, t^i_{rs} + T)$ in the future based on the agent A_i clock. Therefore, the minimum distance d_{min} between two agents computed from two reachable sets is the smallest distance among all possible distances in the time interval $[t^i_c, T_{safe}]$. Consequently, the collision free guarantee is sound in the time interval $[t^i_c, T_{safe}]$. ◀

We have studied how to use exchanged reachable sets to do peer-to-peer collision detection. Next, we consider how to verify online the global behavior of a distributed CPS in decentralized manner.

5 Decentralized Real-Time Global Safety Verification

▶ **Definition 4** (Globally useful reachable set.)**.** Consider a distributed CPS with N agents with time synchronization errors $\delta_i, i = 1, 2, \ldots, N$, a globally useful reachable set of the whole system under the view of agent A_i based on its current local time clock t^i_c is defined below:

$$globalReach = \bigwedge_{i=1}^{N} \mathcal{R}_i[t^i_{rs}, t^i_{rs} + T] \wedge \mathcal{T},$$
$$\mathcal{T} \triangleq (t^i_c \leq t \leq T + min\{t^i_{rs} - \delta_i - \delta_j\}, j \neq i, 1 \leq j \leq N). \quad (2)$$

For any time t such that $t^i_c \leq t \leq T + min\{t^i_{rs} - \delta_i - \delta_j\}$ for $\forall\ 1 \leq j \leq N, i \neq j$, we have $\mathcal{R}_i(t) \subseteq \mathcal{R}_i[t^i_{rs}, t^i_{rs} + T], \forall i$. In other words, *globalReach* contains all possible trajectories of all agents from the current local time t^i_c of agent A_i to the future time defined by $T + min\{t^i_{rs} - \delta_i - \delta_j\}, j \neq i, 1 \leq j \leq N$. The globally useful reachable set is a collection of all useful reachable sets (defined in the previous section) received and decoded at an agent A_i under its current local clock t^i_c. The inner intersection determines that at the time that all reachable sets have been received and decoded at the agent A_i, i.e., t^i_c, only a portion of each received reachable set $\mathcal{R}_j[t^j_{rs}, t^j_{rs} + T]$ between $[t^i_c, T + min\{t^i_{rs} - \delta_i - \delta_j]\}$ is useful for checking collision.

It should be noted that to construct a global reachable set, an agent needs to wait for all messages arrive and then decodes all these messages. This process may have an expensive computation cost, especially when the number of agents increases. Since this global reachable set is only valid in an interval of time, the amount of time that is available for verify the global property may be small and not enough for the agent to perform the global safety verification. Having additional hardware for handling in parallel the processes of receiving/decoding messages is a good solution to overcome this challenge.

Using the globally useful reachable set, the global safety verification problem is equivalent to checking whether the globally useful reachable set intersects with the global unsafe region defined by $\mathcal{U} \triangleq Cx \leq d$, where $x = [x_1^T, x_2^T, \cdots, x_N^T]^T$ and x_i is the state vector of agent A_i. The procedure for global safety verification is summarized in Algorithm 5.3.

▶ **Lemma 5.** *The decentralized real-time global safety verification algorithm is sound.*

Algorithm 5.3 Decentralized Real-Time Global Safety Verification for Agent A_i.

Input: $\mathcal{U}$, % global unsafe constraints
Output: $global\_safe, T_{global\_safe}$ % global safe flag and safe time interval in the future

1: **procedure** INITIALIZATION
2: $global\_safe = true$ % global safety flag
3: **procedure** GLOBAL SAFETY VERIFICATION
4: **if** all useful messages are available **then**
5: $t_c^i = A_i.current\_time()$
6: recheck if all messages are still useful
7: construct globally useful reach set $globalReach$
8: **if** $(globalReach \cap \mathcal{U} \neq \emptyset)$ **then**
9: $global\_safe = uncertain$
10: $T_{global\_safe} = [\,]$
11: **else**
12: $global\_safe = true$
13: $T_{global\_safe} = T + min\{t_{rs}^i - \delta_i - \delta_j\}, j \neq i, 1 \leq j \leq N$

Proof. Similar to Lemma 3, the soundness of the verification algorithm is guaranteed because of the soundness of the globally useful reachable set containing all possible trajectories of all agents at any time $t \in \mathcal{T}$, where $\mathcal{T} \triangleq (t_c^i \leq t \leq T + min\{t_{rs}^i - \delta_i - \delta_j\}, j \neq i, 1 \leq j \leq N)$. ◀

6 Case study

The decentralized real-time safety verification for distributed CPS proposed in this paper is implemented in Java as a package called *drreach*. This package is currently integrated as a library in StarL, which is a novel platform-independent framework for programming reliable distributed robotics applications on Android [24]. StarL is specifically suitable for controlling a distributed network of robots over WiFi since it provides many useful functions and sophisticated algorithms for distributed applications. In our approach, we use the reliable communication network of StarL which is assumed to be asynchronous and peer-to-peer. There may be message dropouts and transmission delays; however, every message that an agent tries to send is eventually delivered with some time guarantees. All experimental results of our approach are reproducible and available online at: `http://www.verivital.com/rtreach/`.

6.1 Experiment setup

We evaluate the proposed approach via a distributed search application using quadcopters[2] in which each quadcopter executes its search mission provided by users as a list of way-points depicted in Figure 3. These quadcopters follow the way-points to search for some specific objects. For safety reasons, they are required to work only in a specific region defined by users. In this case study, the quadcopters are controlled to operate at the same constant altitude. It has been shown from the experiments that the proposed approach is promisingly scalable as it works well for a different number of quadcopters. We choose to present in this section the experimental results for the distributed search application with eight quadcopters.

[2] A video recording is available at: `https://youtu.be/YC_7BChsIf0`

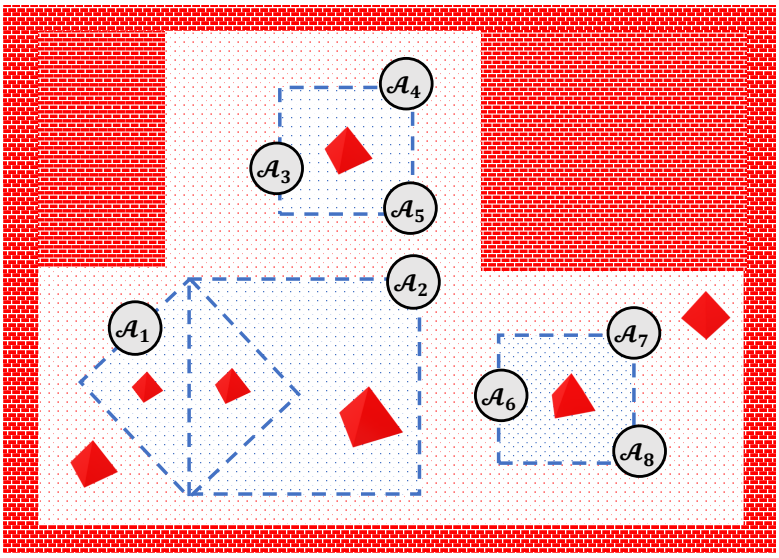


Figure 3 Distributed Search Application Using Quadcopters.

The first step in our approach is locally computing the reachable set of each quadcopter using the face-lifting method [6,9,18]. The quadcopter has nonlinear motion dynamics given in Equation 3 in which θ, ϕ, and ψ are the pitch, roll, and yaw angles, $f = \Sigma_{i=1}^{4} T_i$ is the sum of the propeller forces, m is the mass of the quadcopter and $g = 9.81 m/s^2$ is the gravitational acceleration constant. As the quadcopter is set to operate on a constant altitude, we have $\ddot{z} = 0$ which yields the following constraint: $f = \frac{mg}{\cos(\theta)\cos(\phi)}$. Let v_x and v_y be the velocities of a quadcopter along with x- and y- axes. Using the constraint on the total force, the motion dynamics of the quadcopter can be rewritten as a 4-dimensional nonlinear ODE as depicted in Equation 4.

$$\begin{aligned} \ddot{x} &= \frac{f}{m}(\sin(\psi)\sin(\phi) + \cos(\psi)\sin(\theta)\cos(\phi)), \\ \ddot{y} &= \frac{f}{m}(\sin(\psi)\sin(\theta)\cos(\phi) - \sin(\phi)\cos(\psi)), \\ \ddot{z} &= \frac{f}{m}\cos(\theta)\cos(\phi) - g, \end{aligned} \tag{3}$$

$$\begin{aligned} \dot{x} &= v_x, \\ \dot{v}_x &= g\tan(\theta), \\ \dot{y} &= v_y, \\ \dot{v}_y &= g\frac{\tan(\phi)}{\cos(\theta)}. \end{aligned} \tag{4}$$

A PID controller (a proportional-integral-derivative controller widely used in industrial control systems [2]) is designed to control the quadcopter to move from its current position to desired way-points. Details about the controller parameters can be found in the available source code. The PID controller has a control period of $T_c = 200$ milliseconds. In every control period, the control inputs pitch (θ) and roll (ϕ) are computed based on the current positions of the quadcopter and the current target position (i.e., the current way-point it needs to go). Using the control inputs, the current positions and velocities given from GPS and the motion dynamics of the quadcopter, the real-time reachable set computation algorithm (Algorithm 3.1) is executed *inside* the controller. This algorithm computes the reachable set of a quadcopter from its current local time to the next $T = 2$ seconds. The allowable run-time for this algorithm is $T_{runtime} = 10$ milliseconds. The local safety property is verified by the real-time reachable set computation algorithm at run-time. The computed reachable set is then encoded and sent to another quadcopter. When a reachable set message arrives, the quadcopter decodes the message to reconstruct the current reachable set of the sender. The GPS error is assumed to be 2%. The time-synchronization error between the

quadcopter7 finishes computing reach set and stores the reach set
quadcopter6 finishes computing reach set and stores the reach set
quadcopter5 finishes computing reach set and stores the reach set
quadcopter1 finishes computing reach set and stores the reach set
quadcopter2 encodes its reach set to send out in 0.027134 milliseconds
quadcopter5 encodes its reach set to send out in 0.012657 milliseconds
quadcopter5 broadcasts its reach set to others
quadcopter2 broadcasts its reach set to others
quadcopter7 encodes its reach set to send out in 0.013169 milliseconds
quadcopter7 broadcasts its reach set to others
quadcopter0 encodes its reach set to send out in 0.012709 milliseconds
quadcopter0 broadcasts its reach set to others
quadcopter0 does not violate its local safety property
quadcopter1 encodes its reach set to send out in 0.012707 milliseconds
quadcopter1 broadcasts its reach set to others
quadcopter6 encodes its reach set to send out in 0.011081 milliseconds
quadcopter6 broadcasts its reach set to others
quadcopter1 does not violate its local safety property
quadcopter2 does not violate its local safety property
quadcopter5 does not violate its local safety property
quadcopter7 does not violate its local safety property
quadcopter6 may violates its local safety specification at time 2019-02-17 17:29:51.344
quadcopter7 finishes computing reach set and stores the reach set
quadcopter5 finishes computing reach set and stores the reach set
quadcopter6 finishes computing reach set and stores the reach set
Reach set (hull) of quadcopter0 that is valid from 2019-02-17 17:29:49.075 to 2019-02-17 17:29:51.074 of its local time is:
dim = 0 -> [-263.98, 1034.28]
dim = 1 -> [-329.46, -287.49]
dim = 2 -> [129.36, 301.87]
dim = 3 -> [30.00, 58.48]
Current reach set (hull) of quadcopter1 that is valid from 2019-02-17 17:29:49.386 to 2019-02-17 17:29:51.383 of its local time is:
dim = 0 -> [1959.99, 2040.00]
dim = 1 -> [-0.00, -0.00]
dim = 2 -> [-285.97, 395.76]
dim = 3 -> [-177.55, -149.38]
Current local time of quadcopter1 is 2019-02-17 17:29:49.423
Useful time for checking collision and global safety property for quadcopter1 is 1645 milliseconds
The received reachable set from quadcopter0 is useful
quadcopter1 will not collide with quadcopter0 in the next 1.645 seconds

Figure 4 A sample of events for verifying the local safety property and collision occurrence.

quadcopters is $\delta = 3$ milliseconds. We want to verify in real-time: 1) local safety property for each quadcopter; 2) collision occurrence; and 3) geospatial free property. The local safety property is defined by $v_x \leq 500$, i.e., the maximum allowable velocities along the x-axis of two arbitrary quadcopters are not larger than $500m/s$. The collision is checked using the minimum allowable distance between two arbitrary quadcopters $d_{min} = 100$. The geospatial free property requires that the some quadcopters never go into a specific region at the same time.

6.2 Verifying local safety property and collision occurrence

Figure 4 presents a sample of a sequence of events happening in the distributed search application. One can see that each quadcopter can determine based on its local clocks if there is no collision to some known time in the future. In addition, the local safety property can also be verified at run-time. For example, in the figure, the quadcopter 1 receives a reachable set message from the quadcopter 0 which is valid from $17:29:49.075$ to $17:29:51.074$ of the quadcopter 0's clock. After decoding this message, taking into account the time-synchronization error δ, quadcopter 1 realizes that the received reachable set message is useful for checking collision for the next 1.645 seconds of its clock. After checking collision, quadcopter 1 knows that it will not collide with the quadcopter 0 in the next 1.645 seconds (based on its clock).

It should be noted that we can intuitively verify the collision occurrences by observing the intermediate reachable sets of all quadcopters and their interval hulls. The *intermediate* reachable sets of the quadcopters in every $[0, 2s]$ time interval computed by the real-time reachable set

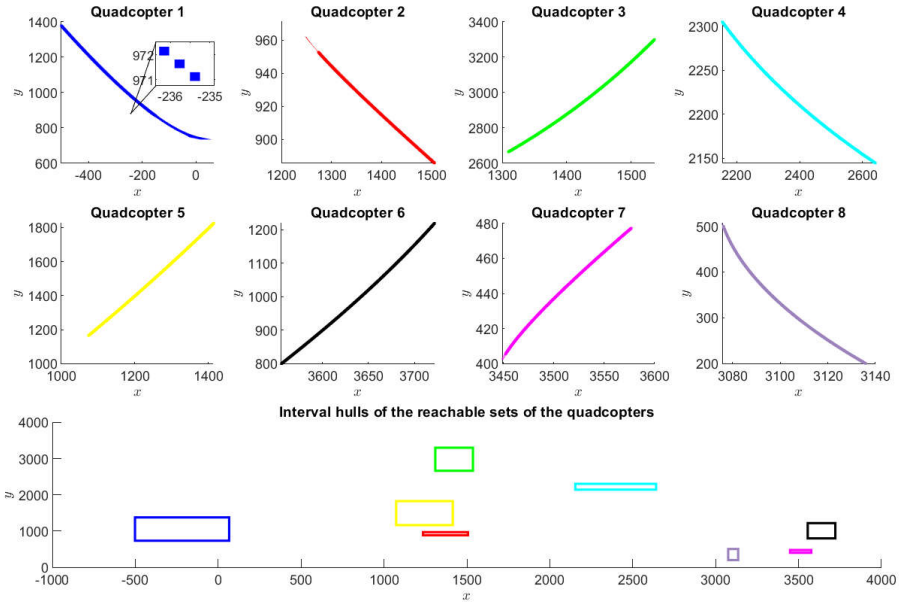


Figure 5 One sample of the reachable sets of eight quadcopters in an $[0, 2s]$ time interval and their interval hulls.

Table 1 The average encoding time τ_e, decoding time τ_d, transferring time τ_{tf}, collision checking time τ_c and total verification time VT of the quadcopters.

Time	Quad. 1	Quad. 2	Quad. 3	Quad. 4	Quad. 5	Quad. 6	Quad. 7	Quad. 8
Encoding Time τ_e (ms)	0.058	0.055	0.0553	0.0525	0.0557	0.0583	0.0584	0.0597
Decoding Time τ_d (ms)	0.0169	0.0193	0.0197	0.019	0.0210	0.0181	0.0177	0.022
Transferring Time τ_{tf} (ms)	2.64	2.48	1.42	1.11	1.12	1.08	1.05	1.13
Collision Checking Time τ_c (ms)	0.04	0.05	0.07	0.05	0.03	0.07	0.07	0.14
Total Verification Time VT (ms)	28.9363	27.9	20.6232	18.3055	18.2527	18.235	18.0223	19.1037

computation algorithm (i.e., Algorithm 3.1) is described in Figure 5. The zoom plot within the figure presents a very short-time interval reachable set of the quadcopters. We note that the intermediate reachable set of a quadcopter is represented as a list of hyper-rectangles and is used for verifying the local safety property at run-time. The reachable set that is sent to another quadcopter is the interval hull of these hyper-rectangles. The intermediate reachable set cannot be transferred via a network since it is very large (i.e., hundreds of hyper-rectangles). The interval hull of all hyper-rectangles contained in the intermediate reachable set covers all possible trajectories of a quadcopter in the time interval of $[0, 2s]$. Therefore, it can be used for safety verification. One may question why we use the interval hull instead of using the convex hull of the reachable set since the former one results in a more conservative result. The reason is that the convex hull of hundreds of hyper-rectangles is a time-consuming operation that cannot be used in a real-time setting. Therefore, in the real-time setting, interval hull operation is a suitable solution. From the figure, we can see that the interval hulls of the reachable set of all quadcopters do not intersect with each other. Therefore, there is no collision occurrence (in the next 2 seconds of global time).

Since we implement the decentralized real-time safety verification algorithm inside the quadcopter's controller, it is important to analyze whether or not the verification procedure affects the control performance of the controller. To reason about this, we measure the average encoding,

decoding, transferring and collision checking times for all quadcopters using 100 samples which are presented in Table 1. We note that the transferring time τ_{tf} is the average time for one message transferred from other quadcopters to the i^{th} quadcopter. It can be seen that the encoding, decoding and collision checking times at each quadcopter constitute a tiny amount of time. The total verification time is the sum of the reachable set computation, encoding, transferring, decoding and collision checking times. Note that the allowable runtime for reachable set computation algorithm is specified by users as $T_{runtime} = 10$ milliseconds. Therefore, the (average) total time for the safety verification procedure on each quadcopter is

$$VT_i = T_{runtime} + \tau_e^i + (N-1) \times (\tau_{tf}^i + \tau_d^i + \tau_c^i), \tag{5}$$

where $i = 1, 2, \ldots, N$, and N is the number of quadcopters. As shown in Table 1, the (average) total verification time for each quadcopter is small (< 30 milliseconds), compared to the control period $T_c = 200$ milliseconds. Besides, from the experiment, we observe that the computation time for the control signal of the PID controller $\tau_{control}^i$ (not presented in the table) is also small, i.e., from 5 to 10 milliseconds. Since $VT_i + \tau_{control}^i < T_c/4 = 50$ milliseconds, we can conclude that the verification procedure does not affect the control performance of the controller.

Interestingly, from the verification time formula (5), we can estimate the range of the number of agents that the decentralized real-time verification procedure can deal with. The idea is that, in each control period T_c, after computing the control signal, the remaining time bandwidth $T_c - \tau_{control}$ can be used for verification. Let $\bar{\tau}_e(\underline{\tau}_e)$, $\bar{\tau}_{tf}(\underline{\tau}_{tf})$, $\bar{\tau}_d(\underline{\tau}_d)$, $\bar{\tau}_c(\underline{\tau}_c)$ be the maximum (minimum) encoding, transferring, decoding and collision checking times on a quadcopter, $\bar{\tau}_{control}(\underline{\tau}_{control})$ be the maximum (minimum) control signal computation time for each control period T_c, then the number of agents that the decentralized real-time safety verification procedure can deal with (with assumption that the communication network works well) satisfies the following constraint:

$$\frac{T_c - \bar{\tau}_{control} - T_{runtime} - \bar{\tau}_e}{\bar{\tau}_{tf} + \bar{\tau}_d + \bar{\tau}_c} + 1 \leq N \leq \frac{T_c - \underline{\tau}_{control} - T_{runtime} - \underline{\tau}_e}{\underline{\tau}_{tf} + \underline{\tau}_d + \underline{\tau}_c} + 1. \tag{6}$$

Let consider our case study, from the Table, we assume that $\bar{\tau}_e = 0.0597$, $\underline{\tau}_e = 0.0525$, $\bar{\tau}_{tf} = 2.64$, $\underline{\tau}_{tf} = 1.05$, $\bar{\tau}_d = 0.022$, $\underline{\tau}_d = 0.0169$, $\bar{\tau}_c = 0.14$, $\underline{\tau}_c = 0.03$ milliseconds. Also, we assume that $\bar{\tau}_{control} = 10$ and $\underline{\tau}_{control} = 5$ milliseconds. We can theoretically estimate the number of quadcopters that our verification approach can deal with is $64 \leq N \leq 168$.

6.3 Verifying the geospatial free property

To illustrate how our approach verifies the global behavior of a distributed CPS, we consider the geospatial free property which requires that the some (or all) quadcopters never go into a specific region at the same time. For simplification, we reconsider the distributed search application with two quadcopters (quad 1 and quad 2) whose forbidden region is defined by $900 < x_0 < 1200 \wedge 900 < x_1 < 1200$. Figure 6 describes a sample of events describing that the quadcopter 2 can verify (based on its local clock) that it will not collide with the quadcopter 1 and the global geospatial free property is guaranteed in the next 1.838 seconds.

7 Discussion

Software architecture. The current implementation of our approach deploys the safety verifier of each agent inside the controller, and a single thread is used to execute the control and verification tasks. The main drawback of this implementation is that it may decrease the overall performance of the controller and even cause the controller to crash. To prevent that happens, in practice,

```
quadcopter0 computes it reach set from 2019-09-16 17:26:20.957 to 2019-09-16 17:26:22.956
quadcopter0 encodes its reach set to send out in 0.017749 milliseconds
quadcopter0 broadcasts its reach set to others
quadcopter0 may violates its local safety specification at time 2019-09-16 17:26:22.956
quadcopter1 receives reach set (hull) from quadcopter0
Time for transferring this reach set over network is around (not considering clock mismatch) 66
milliseconds
Decoding message from quadcopter0 takes 0.029057 milliseconds
Reach set (hull) of quadcopter0 that is valid from 2019-09-16 17:26:20.957 to 2019-09-16
17:26:22.956 of its local time is:
dim = 0 -> [89.18, 280.21]
dim = 1 -> [39.31, 58.43]
dim = 2 -> [804.58, 1240.06]
dim = 3 -> [91.37, 126.26]
Current reach set (hull) of quadcopter1 that is valid from 2019-09-16 17:26:20.119 to 2019-09-16
17:26:22.118 of its local time is:
dim = 0 -> [1946.28, 2060.66]
dim = 1 -> [25.17, 33.29]
dim = 2 -> [787.92, 1566.04]
dim = 3 -> [184.97, 203.86]
Current local time of quadcopter1 is 2019-09-16 17:26:21.112
Useful time for checking collision and global safety property is 1838 milliseconds
The received reachable set from quadcopter0 is useful
quadcopter1 will not collide with quadcopter0 in the next 1.838 seconds
The geospatial free property is guarantee in the next 1.838 seconds
```

Figure 6 A sample of events for verifying the geospatial free property.

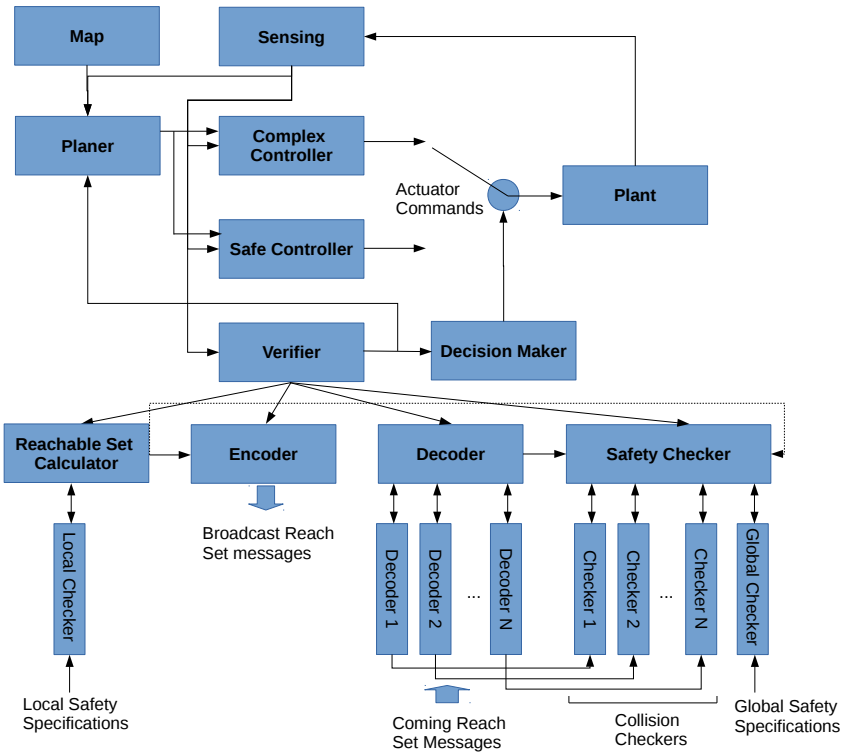


Figure 7 Software architecture for deploying decentralized real-time safety verification approach on a real platform.

the controller and verifier should be implemented in two separate software components. In this case, the computation burden for safety checks in the verifier does not affect the performance of the controller. The control task and the verification task can be executed efficiently in parallel as depicted in Figure 7. More importantly, this software architecture adopts the architecture of a fault-tolerant system [15] to prevent the propagation of failure from one component to others. It also benefits the use of a simplex-architecture for safety control in the case of dangerous circumstances.

As shown in Figure 7, the verifier component consists of four sub-components including a reachable set calculator, a encoder, a decoder, and a safety checker. These sub-components should also be implemented conveniently for parallel execution. The local safety property is verified inside the reachable set calculator at runtime. As the number of reachable set messages that need to be decoded increases with the number of participating agents, it is necessary to have multiple decoders working in parallel. These decoders listen to upcoming reachable set messages on different ports assigned to them by the verifier and immediately decode any arrived message. This parallel decoding helps to reduce the decoding time significantly. The decoded reachable sets are then sent to the safety checker containing multiple checkers run in parallel in which each checker is responsible for checking collision between the agent with another. The i^{th} checker and the i^{th} decoder is a pair worker, i.e., the checker only waits for the decoded reachable set of its corresponding co-worker. Therefore, the pair to pair collision detection task can be done very quickly. The safety checker also has a global checker which is responsible for checking global properties. The global checker is only triggered when the decoder component finishes decoding all arrived reachable set messages. For this reason, having parallel working decoders is essential to speed up the overall verification time which is required to be very small to work in the real-time setting.

Let $\bar{\tau}_{rs}, \bar{\tau}_e, \bar{\tau}_{tf}$ and $\bar{\tau}_d$ respectively be the worst case times of reachable set computation, encoding, transferring and decoding, $\bar{\tau}_{cc}$ and $\bar{\tau}_{gc}$ be the worst case times of peer-to-peer collision detection and global safety verification. For a system with N agents, the total worst-case verification time is $\bar{\tau}_{total} = \bar{\tau}_{rs} + \bar{\tau}_e + \bar{\tau}_{tf} + \bar{\tau}_d + \bar{\tau}_{cc} + \bar{\tau}_{gc}$. If we do the verification in *sequential way*, i.e., using only one port for reachable set communication and one checker for all peer-to-peer collision detection and global safety verification, the total worst-case verification is: $\bar{\tau}^*_{total} = \bar{\tau}_{rs} + \bar{\tau}_e + \bar{\tau}_{tf} + N\bar{\tau}_d + N\bar{\tau}_{cc} + \bar{\tau}_{gc} >> \bar{\tau}_{total}$.

Scalability. From the above discussion, one can see that the software architecture plays an important role when we implement our approach in a real platform. In practice, if each participating agent has the powerful hardware for communication and computation, and the software for our approach is implemented in a parallel manner as proposed above, then the worst-case verification time does not depend on the number of agents in the system. Therefore, our decentralized real-time safety verification approach is scalable for systems with a large number of agents. Also, the proposed software architecture is especially useful in the case that there are losses of reachable set messages. In this hazardous situation, the agent still has some partial information to check if a collision occurs based on the available, reachable set messages. Therefore, the planner still can re-perform path planning algorithm based on the current information and past information it has to find the safest path for the agent for this incomplete information situation.

Effect of time synchronization error. The time synchronization error directly affects the ability to receive useful reachable sets and globally useful reachable sets for verification. If the time synchronization error is too large, all exchanged reachable sets may not be useful for verification. In this case, we cannot verify the collision avoidance and global safety properties. For example, in

the definition of the globally useful reachable set, it is required that after all exchanged reachable sets have been received and decoded at an agent A_i at time local current time t_c^i, each received reachable set is only useful if and only if $T + min\{t_{rs}^i - \delta_i - \delta_j\} > t_c^i$. One can see that if the time synchronization error is too large, then the related requirement cannot be satisfied. Therefore, the maximum tolerance for the time synchronization error can be estimated roughly as $T + min\{t_{rs}^i - 2\delta_{max}\} >> t_c^i \implies \delta_{max} << (T + t_{rs}^i - t_c^i)/2$.

8 Related Work

Our work is inspired by the static and dynamic analysis of timed distributed traces [11] and the real-time reachability analysis for verified simplex design [6]. The former one proposes a sound method of constructing a global reachable set for a distributed CPS based on the recorded traces and time synchronization errors of participating agents. Then the global reachable set is used to verify a global property using Z3 [10]. This method can be considered to be a *centralized analysis* where the reachable set of the whole system is constructed and verified by one analyzer. Such a verification approach is offline which is fundamentally different from our approach as we deal with online verification in a decentralized manner. Our real-time verification method borrows the face-lifting technique developed in [6] and applies it to a distributed CPS.

Another interesting aspect of real-time monitoring for linear systems was recently published in [8]. In this work, the authors proposed an approach that combines offline and online computation to decide if a given plant model has entered an uncontrollable state which is a state that no control strategy can be applied to prevent the plant go to the unsafe region. This method is useful for a single real-time CPS, but not a distributed CPS with multiple agents.

Additionally, there has been other significant works for verifying distributed CPS. Authors of [12, 29, 33] presented a real-time software for distributed CPS but did not perform a safety verification of individual components and a whole system. The works presented in [3,19,21] can be used to verify distributed CPS, but they do not consider a real-time aspect. An interesting work proposed in [26] can formally model and verify a distributed car control system against several safety objectives such as collision avoidance for an arbitrary number of cars. However, it does not address the verification problem of distributed CPS in a real-time manner. The novelty of our approach is that it can over-approximate of the reachable set of each agent whose dynamics are non-linear with a high precision degree in real-time.

The most related work to our scheme was recently introduced in [25]. The authors proposed an online verification using reachability analysis that can guarantee safe motion of mobile robots with respective to walking pedestrians modeled as hybrid systems. This work utilizes CORA toolbox [1] to perform reachability analysis while our work uses a face-lifting technique. However, this work does not consider the time-elapse for encoding, transferring and decoding the reachable set messages between each agent, which play an important role in distributed systems.

9 Conclusion and Future Work

We have proposed a decentralized real-time safety verification method for distributed cyber-physical systems. By utilizing the timing information and the reachable set information from exchanged reachable set messages, a sound guarantee about the safety of the whole system is obtained for each participant based on its local time. Our method has been successfully applied for a distributed search application using quadcopters built upon the StarL framework. The main benefit of our approach is that it allows participants to take advantages of formal guarantees available locally in real-time to perform intelligent actions in dangerous situations. This work is a

fundamental step in dealing with real-time safe motion/path planing for distributed robots. For future work, we seek to deploy this method on a distributed autonomous driving testbed using the F1Tenth racing platform [28] and extend it to distributed CPS with heterogeneous agents where the agents can have different motion dynamics and thus they have different control periods. In addition, the scalability of the proposed method can be improved by exploiting the benefit of parallel processing, i.e., each agent handles multiple reachable set messages and checks for collisions in parallel.

References

1 Matthias Althoff. An introduction to cora 2015. In *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.

2 Karl Johan Aström, Tore Hägglund, and Karl J Astrom. *Advanced PID control*, volume 461. ISA-The Instrumentation, Systems, and Automation Society, 2006.

3 Kyungmin Bae, Joshua Krisiloff, José Meseguer, and Peter Csaba Ölveczky. Designing and verifying distributed cyber-physical systems using multirate pals: An airplane turning control system case study. *Science of Computer Programming*, 103:13–50, 2015.

4 Stanley Bak and Parasara Sridhar Duggirala. Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, pages 173–178. ACM, 2017.

5 Stanley Bak and Parasara Sridhar Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In *International Conference on Computer Aided Verification*, pages 401–420. Springer, 2017.

6 Stanley Bak, Taylor T Johnson, Marco Caccamo, and Lui Sha. Real-time reachability for verified simplex design. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 138–148. IEEE, 2014.

7 Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for nonlinear hybrid systems. In *International Conference on Computer Aided Verification*, pages 258–263. Springer, 2013.

8 Xin Chen and Sriram Sankaranarayanan. Model predictive real-time monitoring of linear systems. In *Real-Time Systems Symposium (RTSS), 2017 IEEE*, pages 297–306. IEEE, 2017.

9 Thao Dang and Oded Maler. Reachability analysis via face lifting. In *Hybrid Systems: Computation and Control (HSCC '98)*, pages 96–109. Springer, 1998. LNCS 1386.

10 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

11 Parasara Sridhar Duggirala, Taylor T Johnson, Adam Zimmerman, and Sayan Mitra. Static and dynamic analysis of timed distributed traces. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 173–182. IEEE, 2012.

12 John C Eidson, Edward A Lee, Slobodan Matic, Sanjit A Seshia, and Jia Zou. Distributed real-time software for cyber–physical systems. *Proceedings of the IEEE*, 100(1):45–59, 2012.

13 Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification*, pages 379–395. Springer, 2011.

14 Antoine Girard, Colas Le Guernic, and Oded Maler. Efficient computation of reachable sets of linear time-invariant systems with inputs. In *Hybrid Systems: Computation and Control*, pages 257–271. Springer, 2006.

15 Alwyn E Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions, 2010.

16 T. A. Henzinger. The theory of hybrid automata. In *IEEE Symposium on Logic in Computer Science (LICS)*, page 278, Washington, DC, USA, 1996. IEEE Computer Society.

17 Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In *Computer aided verification*, pages 460–463. Springer, 1997.

18 Taylor T. Johnson, Stanley Bak, Marco Caccamo, and Lui Sha. Real-time reachability for verified simplex design. *ACM Trans. Embed. Comput. Syst.*, 15(2), February 2016. doi:10.1145/2723871.

19 Taylor T Johnson and Sayan Mitra. Parametrized verification of distributed cyber-physical systems: An aircraft landing protocol case study. In *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pages 161–170. IEEE, 2012.

20 Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. *dReach: δ-Reachability Analysis for Hybrid Systems*, pages 200–205. Springer, 2015.

21 Pratyush Kumar, Dip Goswami, Samarjit Chakraborty, Anuradha Annaswamy, Kai Lampka, and Lothar Thiele. A hybrid approach to cyber-physical systems verification. In *Proceedings of the 49th Annual Design Automation Conference*, pages 688–696. ACM, 2012.

22 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

23 Colas Le Guernic and Antoine Girard. Reachability analysis of hybrid systems using support functions. In *Computer Aided Verification*, pages 540–554. Springer, 2009.

24 Yixiao Lin and Sayan Mitra. Starl: Towards a unified framework for programming, simulating and verifying distributed robotic systems. *CoRR*, abs/1502.06286, 2015. arXiv:1502.06286.

25 Stefan B Liu, Hendrik Roehm, Christian Heinzemann, Ingo Lütkebohle, Jens Oehlerking, and Matthias Althoff. Provably safe motion of mobile robots

in human environments. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 1351–1357. IEEE, 2017.

26 Sarah M Loos, André Platzer, and Ligia Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In *International Symposium on Formal Methods*, pages 42–56. Springer, 2011.

27 Nancy Lynch, Roberto Segala, Frits Vaandrager, and Henri B Weinberg. *Hybrid i/o automata*. Springer, 1996.

28 Matthew O'Kelly, Hongrui Zheng, Dhruv Karthik, and Rahul Mangharam. F1tenth: An open-source evaluation environment for continuous control and reinforcement learning. *Proceedings of Machine Learning Research*, 123, 2020.

29 Qinghui Tang, Sandeep KS Gupta, and Georgios Varsamopoulos. A unified methodology for scheduling in distributed cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(S2):57, 2012.

30 Hoang-Dung Tran, Luan Viet Nguyen, Nathaniel Hamilton, Weiming Xiang, and Taylor T. Johnson. Reachability analysis for high-index linear differential algebraic equations (daes). In *17th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'19)*. Springer International Publishing, August 2019.

31 Hoang-Dung Tran, Luan Viet Nguyen, Patrick Musau, Weiming Xiang, and Taylor T. Johnson. Decentralized real-time safety verification for distributed cyber-physical systems. In Jorge A. Pérez and Nobuko Yoshida, editors, *Formal Techniques for Distributed Objects, Components, and Systems (FORTE'19)*, pages 261–277, Cham, June 2019. Springer International Publishing.

32 Hoang-Dung Tran, Luan Viet Nguyen, Weiming Xiang, and Taylor T Johnson. Order-reduction abstractions for safety verification of high-dimensional linear systems. *Discrete Event Dynamic Systems*, 27(2):443–461, 2017.

33 Yuanfang Zhang, Christopher Gill, and Chenyang Lu. Reconfigurable real-time middleware for distributed cyber-physical systems with aperiodic events. In *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*, pages 581–588. IEEE, 2008.