Deductive Verification of Stochastic Hybrid Systems with Isabelle/HOL

Christian Pardillo Laursen

PhD

University of York

Computer Science

September 2025

Abstract

Stochastic hybrid systems are increasingly used to model safety-critical applications in robotics and autonomous systems, where uncertainty arises from sensor noise, actuator errors, and unpredictable environments. Ensuring the safety of these systems requires rigorous formal guarantees, yet existing verification tools often struggle with the combined complexity of continuous, discrete, and stochastic behavior.

In this thesis, we introduce a framework for the deductive verification of stochastic hybrid systems (SHS). This is centered around the Isabelle/HOL implementation of stochastic differential dynamic logic (SdL), a logic which allows for the verification of SHS by modelling them as stochastic hybrid programs (SHP). To complement our SdL implementation, we develop the foundational mathematical theories for defining and reasoning about continuous-time stochastic processes. We also present a simulation tool for SHPs, which aids the verification effort by enabling the validation of models before verification is attempted.

Contents

List of Accompanying Material

All the Isabelle theories and Haskell source files that have been developed for, and discussed within, this thesis have been supplied in a zip file, and are available on GitHub at `https://github.com/cplaursen/PhD_meta_repository`, together with instructions for their execution and clearly labelled dependencies.

Acknowledgements

No man is an island. Without the support of the wonderful people I have somehow managed to surround myself with, this thesis would not have been written. I'd like to thank the following people:

Simon, thank you for your bottomless patience and ever-so-excellent feedback. I couldn't have asked for a better supervisor.

Mark, thank you for your enthusiasm and for bringing a perspective that helped me put my work into context.

Arjun, Kit, Charles, thanks for accepting that I am much better at squash than all of you.

Fernando, thank you for putting me in your thesis acknowledgements.

Most importantly, Esher, you have supported me through the good and the bad, and you have given me the strength to continue during my most stressful moments. Out of everyone, I could never have done it without you. I was lucky to meet you at the start of this journey, and I am the luckiest man alive to be finishing it with our wedding on the horizon.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Collaboration Sections 3.1 and 3.2 were originally contributed to a journal paper titled "Scalable Automated Verification for Cyber-Physical Systems in Isabelle/HOL" [88]. They are entirely my own work.

Chapter 4 was developed in collaboration with Simon Foster. In particular, Simon developed the theory of scenes and implemented the `alphabet_scene_space` command, while I developed the theory of scene spaces and frames.

Chapter 1

Introduction

1.1 Motivation

Cyber-physical systems (CPS) combine continuous behaviour with discrete control. They are ubiquitous and their operation is often safety-critical. Common examples include robots, autonomously operated vehicles, and control systems for power plants. Ensuring that the software controlling these systems behaves in a safe manner is an important and difficult problem. In particular, robotic systems are a domain where uncertainty and safety are paramount concerns.

To address this, we can apply formal verification, which consists of applying mathematical methods to prove properties about software. This allows us to produce strong guarantees that programs satisfy some conditions by using mathematical models. This is in contrast to validation and testing approaches, which examine the behaviour of the program directly, by executing it. One of the most popular formal methods is model checking, which exhaustively explores the state space of a model to ensure that some safety condition is never broken. Another option is deductive verification, which applies formal logic to reason symbolically about a system without needing to explore the state space.

Verification tools for robotics need to account for both the control software, which has a discrete state space, and the physical robot model, which is continuous. In order to model both, we model robots as hybrid systems, which can perform discrete jumps or evolve their state according to a system of ordinary differential equations (ODEs) [54]. Hybrid systems provide a powerful abstraction by combining a discrete state machine with continuous dynamics, allowing us to represent both the logical decisions of the controller and the physical movement of the robot. When attempting to formally verify such systems, with their continuous state spaces, simply applying state exploration is not tractable — we need to either apply approximations or reason symbolically about the program behaviour. Applying deductive verification lets us use symbolic reasoning in order to avoid approximations, and it has been successfully applied to hybrid systems [39, 88]. In particular, KeYmaera X is a proof assistant developed for the deductive verification of hybrid systems, and it has been applied to numerous case studies [67, 85]. Hybrid systems are useful models for robots, but they can only express uncertainty using nondeterminism, which cannot be used to make statements about the probabilistic likelihood of different outcomes. Uncertainty arises in many forms in robotics - for example, roboticists must consider sensor noise and actuator error, as well as the complexities that arise from navigating unknown environments and using approximate algorithms [110].

A natural way of dealing with uncertainty is modelling it using probability. We can characterise

sensing errors as probability distributions, and uncertain motion using stochastic differential equations (SDEs). By extending hybrid systems to incorporate these stochastic elements, we get stochastic hybrid systems (SHS) [97]. Stochastic hybrid systems (SHS) are hybrid systems with discrete and stochastic elements. These can be introduced by making the discrete transitions probabilistic, and/or adding stochastic noise to the continuous evolution.

SHS are very difficult to reason about using state exploration-based tools. The state space for these is both continuous and stochastic, which makes it difficult to provide any reasonable guarantees of safety. Instead, in this thesis we apply deductive verification to reason symbolically about the structure of the program and the behaviour of the probabilistic system, to produce the same approximation-free guarantees that we can provide for hybrid systems, without a state explosion issue. This involves adapting Stochastic Differential Dynamic Logic (SdL) a logical framework for SHS originally introduced by Platzer [96], to be used with a proof assistant. The objects of this logic are models of SHS as programs, called stochastic hybrid programs (SHPs), which can evolve along SDEs and execute discrete assignments.

We use the Isabelle/HOL proof assistant [65] for deductive verification. Isabelle/HOL is an interactive theorem prover for higher-order logic (HOL), which has been widely applied to the formalisation of mathematical theories [59, 30] and the verification of software [88, 70].

1.2 Research questions

We frame the research goals of this project by posing the following research questions:

RQ 1: How can we model uncertainty in the context of cyber-physical systems in a way that enables formal verification?

RQ 2: To what extent can we mechanise the mathematical foundations of stochastic process theory in Isabelle/HOL to support the verification of stochastic hybrid systems?

RQ 3: How can we develop a tool that will enable practical, compositional verification for these stochastic models?

1.3 Contributions

We identify several contributions within this thesis.

- **Modelling and verification of uncertainty**: We demonstrate how uncertainty can be modelled using both nondeterminism and probability. We implemented a computer algebra system (CAS) integration for Isabelle/HOL [88] to automate the verification of hybrid systems, and ported existing proofs from KeYmaera X to Isabelle/HOL.
- **Theory of scene spaces**: We developed a formal theory of scenes and scene spaces in Isabelle/HOL. This provides a rigorous foundation for reasoning about sets of variables in formal logics, which is a necessary component for compositional reasoning in logics such as SdL.
- **Stochastic hybrid system simulation**: We designed and implemented a modelling language and simulation tool for SHS. This tool supports a model-simulate-verify workflow by allowing practitioners to validate their models before proceeding to formal verification.

- **Formalisation of stochastic processes**: We developed a foundational theory of stochastic processes in Isabelle/HOL, based on a standard probability theory text [71]. Our mechanisation includes the Kolmogorov-Chentsov theorem and the construction of finite product kernels.
- **Implementation of SdL**: We implemented an axiomatic version of SdL in Isabelle/HOL, based on the logic proposed by Platzer [96]. This includes the mechanisation of SdL proof rules and a case study demonstrating the verification of a stochastic flight controller.

Together, our contributions enable a methodology for the formal verification of cyber-physical systems, where the physical system is modelled as an SHP, validated via simulation, and then formally verified using our SdL implementation, which is backed by an underlying mathematical theory.

1.4 Thesis structure

The remainder of this thesis is structured as follows:

In **Chapter 2**, we provide a literature survey of formal verification for robotics, identifying the current state of the art and the gaps that this thesis aims to fill. We also provide the necessary theoretical background on hybrid and stochastic hybrid systems.

In **Chapter 3**, we explore the modelling of uncertainty in cyber-physical systems. We present our integration of computer algebra systems into Isabelle/HOL and demonstrate its utility through a case study on planar flight. We also compare the use of nondeterminism and probability for modelling uncertainty.

In **Chapter 4**, we present our theory of scenes and scene spaces. We motivate the need for these structures in the context of formal verification and develop their algebraic properties in Isabelle/HOL.

In **Chapter 5**, we describe the design and implementation of our simulation language for SHPs. We provide a formal semantics for the language and illustrate its usage through examples of a water tank and a flight controller.

In **Chapter 6**, we detail our mechanisation of stochastic process theory in Isabelle/HOL. We recount the formalisation of measure and probability theory, and present our results on the Kolmogorov-Chentsov theorem and transition kernels.

In **Chapter 7**, we present our implementation of stochastic differential dynamic logic (SdL). We describe the mechanisation of the logic and its proof rules, and provide a case study verifying a stochastic version of the planar flight controller.

Finally, in **Chapter 8**, we conclude the thesis, reflecting on our research questions and discussing potential avenues for future work.

Chapter 2

Literature Survey

In this chapter we will be surveying the literature on formal verification for robotics and providing relevant definitions and theoretical background. Throughout, we will be addressing the three research questions presented in the introduction, in order to better demonstrate the gap in the literature that this thesis fills.

2.1 Formal Verification for Robotics

Luckcuck et al. have already published a survey of formal methods for robotics [80]. We follow a similar approach to the one taken by them: we selected keywords for a literature search on Google Scholar, filtering to papers from only the past 10 years and considering the first 100 papers. The results were ordered by relevance as determined by Google Scholar's search algorithm. We chose Google Scholar for our search as it provides a comprehensive index of scholarly literature across both computer science and robotics, covering most major conferences and journals. The query employed is the following: "hybrid systems" OR robot OR robotic OR robotics "formal verification". This aims to cover as much ground as possible, to ensure that we get a thorough sample of the work in this area. We discard a few which did not consist of novel contributions, and were instead surveys or book chapters. We supplement the works returned by this search with relevant background literature and seminal works that did not surface in the results — such as Platzer's introduction of hybrid programs — which we identify as such where they appear. Throughout, we address the three research questions set out in the introduction.

In order to formally verify robot algorithms we need to model the robots mathematically, since formal verification works with models. There are a number of ways of modelling robots, each with their own trade-offs, but models can never be perfectly faithful - it is always necessary to make approximations and assumptions. The accuracy of these approximations will determine how applicable the formal verification is to the real robot - the difference between a model and the real world is known as the reality gap, and it is an ever-present problem.

The main formal verification approaches we consider are the following:

- Model checking, where the state space of the model is exhaustively explored in order to verify some property.
- Reachability analysis, where the goal is to verify whether the system enters a given set of states. This is checked in a similar manner to model checking, by first over-approximating

the state space of the program with sets that are easy to compute, and then mechanically checking whether the system reaches the states.

- Deductive verification, where a formal proof of correctness is constructed using a proof assistant.

2.1.1 Discrete State Machines

Discrete state machines can be used to model robot behaviour, where the tasks of the robot are abstracted into a finite set of states and discrete transitions between them. This can allow us to reason about the robot's high-level goals. Many tools employ this modelling paradigm; an example is is RoboChart [1], a graphical modelling language based on UML which has an associated suite of tools for analysis and verification, based on model checking.

Model checking

In the surveyed literature we find a number of articles which use many flavours of discrete state machines. Since they are discrete by nature they are well-suited for model checking, so they are often formally verified using this method. Model checking is indeed the most common approach, with 45 of the papers surveyed applying it.

We find two main approaches to encoding problems in robotics as state machines: we can either consider high-level problems that abstract away the physical presence of the robot, or we can discretise the real world. We see this dichotomy in the approach to verifying mobile robot navigation: Germanos and Secco verify the BUG algorithms by modelling them as Petri nets, abstracting away the location of the robots and the obstacles [42], while Bérard et al. discretise the state space and model navigation protocols as finite state automata [9].

There are many methods for specifying high-level robot behaviour as discrete state machines. A popular one is Behaviour Trees, which are tree-based task switching structures. Biggar et al. propose a method for formally verifying these by converting them to linear temporal logic, which is a popular language for model checkers that is internally converted into a finite state automaton [13]. Another such work is by Kiekbusch et al., who use model checking to verify behaviour trees in the presence of sensor failures [69].

Other contributions propose new methodologies for modelling aspects of robots using finite state machines so that they can be model checked. Examples include security and survivability properties [12], ethical choices [24], communication between ROS nodes [50], scheduling [36, 37], IEC61499 block diagrams [28, 105], time-aware computation [29], and human-robot interaction [107, 103, 7].

Another approach is to translate existing modelling methodologies into a model-checkable format. This is an accessible way to introduce formal verification to practitioners who might already be using model-based methods. In the literature survey, we find that Lu et al. translate the Ptolemy Discrete-Event Model into UPPAAL timed automata [79]. Model transformation can also be applied to transform between different formal notations - Rathmair et al. give a method to combine a number of model checkers using model transformation in order to verify a system [102].

So far all of the methods presented here do not deal with uncertainty, which we will address as per research question 1. The main way of doing this is using probability, and probabilistic model checking is automated by established tools such as PRISM [72] and Storm [52].

Probability is used as an abstraction in the work of Porfirio et al., who encode social norms into

robots probabilistically [99], and Mikael, who uses probability as an abstraction for swarm robotics, similar to the work by Konur et al. discussed in the motivation [83]. By contrast Lahijanian et al. develop an abstraction so they can model check discrete-time stochastic systems using PRISM [73], which would enable the modelling of aleatoric uncertainty but only for a discrete state space.

It is also possible to model aleatoric uncertainty using nondeterminism. For example, Kiekbusch et al. use it to model sensor failures in behaviour trees [69]. However, this is limited because we cannot express the relative likelihood of different outcomes - failures will always have some chance of occurring but we can establish acceptable bounds on their likelihood.

Finally, we find several case studies that apply model checking to real-world robotic applications: Dixon et al. verify the safety of the Care-O-Bot robotic assistant [25], Farrell et al. show security properties for cooperative awareness messages between vehicles for preventing collisions [32], and Tahir et al. develop a model for a firefighting robot using various sensors [108]. However, because these case studies model the robots as discrete state machines, they capture only the high-level, digital behaviour of the system and not its low-level physical behaviour — such as the robot's continuous motion and the uncertainty that accompanies it. They are therefore not sufficient as case studies for the full development of a mobile robot.

2.1.2 Hybrid Systems

Hybrid systems combine continuous and discrete dynamics - the continuous dynamics are given by a system of ODEs, while the discrete dynamics can be modelled as a state machine. This allows us to model how the controller interacts with the real world. An instructive formalism here is that of hybrid automata [54], which make apparent the fact that hybrid systems are extensions of state machines.

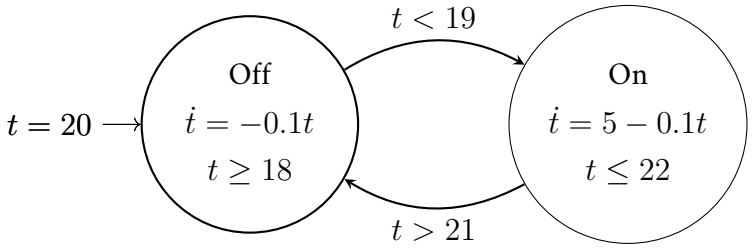
Definition 2.1 (Hybrid Automaton)**.** A hybrid automaton is formally defined as a tuple

$$\mathcal{H} = (\mathbb{R}^n, V, E, \mathit{init}, \mathit{inv}, \mathit{flow}, \mathit{jump})$$

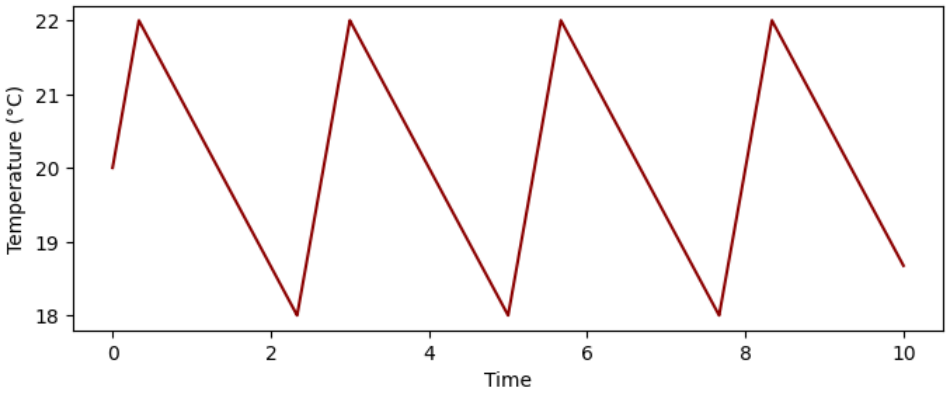
consisting of:

- A continuous state space in $\mathbb{R}^n$.
- A discrete graph (V, E) representing the modes and transitions.
- An initial condition $\mathit{init} : (V \times \mathbb{R}^n) \to \mathbb{B}$.
- An invariant $\mathit{inv} : (V \times \mathbb{R}^n) \to \mathbb{B}$.
- A flow condition $\mathit{flow} : (V \times \mathbb{R}^n) \to \mathbb{R}^n$.
- A jump condition $\mathit{jump} : (E \times \mathbb{R}^n \times \mathbb{R}^n) \to \mathbb{B}$.

A hybrid automaton provides a rigorous mathematical framework for modelling systems that exhibit interacting continuous and discrete dynamics. The discrete operational modes and allowable transitions are captured by the graph (V, E), while the continuous state variables reside in $\mathbb{R}^n$. Within any given discrete mode $v \in V$, the continuous state evolves over time according to the flow condition, provided that the system strictly satisfies the inv (invariant) condition. Discrete state transitions are governed by the jump condition along an edge $e \in E$, which acts as a boolean guard determining when a mode change is permissible and how the continuous state variables are mapped or reset to new values upon transitioning.



(a) Thermostat diagram



(b) Thermostat trace

Figure 2.1: Thermostat hybrid automaton, as seen in [54]

As an example of a hybrid automaton, consider the simple hybrid automaton presented in Figure 2.1. This has two states: off and on. It begins execution in the off state, with a temperature of 20. The system then evolves according to its flow condition $\dot{t} = -0.1t$ until temperature reaches 19, at which point the system may take the transition to the on state until the invariant condition $t \geq 18$ is no longer met, at which point the transition is forced.

Model checking

Model checking is not limited to purely discrete models — it can also be used to verify the continuous, physical behaviour of hybrid systems, but only by first applying some abstraction. For example, it is possible to directly model check linear hybrid automata using the HyTech model checker, and [21] transform DEV&DESS (Discrete Event and Differential Equation System Specification) models into linear hybrid automata to formally verify with HyTech.

Hybrid automata are useful for model checking, being adopted by tools such as HyTech [53]. This tool targets *linear* hybrid automata, where the dynamics of the continuous variables are expressions of the form $A\dot{x} \geq b$, as these can be model checked exactly.

Reachability

Reachability analysis is a popular approach for hybrid systems, as it has many of the same advantages as model checking. The reachability problem consists of deciding whether a system, starting from its initial states, can ever reach a given target state or set of states. For systems with continuous variables we can use approximations, as Huang et al. propose for weakly-hard systems – real-time systems that can occasionally meet deadlines [61]. The tool Ariadne computes reachability for hybrid automata through over-approximation of the reachable set using sets that are

more easily computable. Ariadne is used by Bresolin et al. to formally verify a robotic puncturing task [17], and Geretti et al. extend the tool to make it parametric [41]. Kekatos et al. take a different approach, formally verifying hybrid system models using the SpaceEx model checker [38] by first over-approximating the hybrid systems as piecewise affine models.

Deductive Verification

Hybrid automata align well with model checking and automated approaches, being a form of automata themselves, but they are less well-suited for deductive verification as they are difficult to algebraically decompose. Instead, hybrid systems are often characterised by deductive verification tools using hybrid programs, which were introduced by Platzer for his KeYmaera X proof assistant [94]. These have the following syntax:

$$\alpha, \beta ::= \mathsf{a} \mid x := \theta \mid x := * \mid ?\psi \mid x' = \theta \& \psi \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^*$$

Hybrid programs consist of constants, standard imperative programming operators such as assignment ($:=$), sequential composition ($;$), and iteration ($*$). The hybrid part comes from the continuous evolution operator ($x' = \theta \& \psi$), where the state evolves following some differential equation θ for a non-deterministic amount of time, until it reaches the boundary ψ. Finally, there are three nondeterministic operators: nondeterministic assignment ($x := *$), which assigns an arbitrary value to x; test ($?\psi$), which aborts the program if the condition ψ is not met; and nondeterministic choice ($\alpha \cup \beta$), which executes either program α or program β. Hybrid programs are useful because they are compositional, which allows us to decompose a proof of some property of a hybrid program into sub-proofs of smaller programs.

All of the papers from our literature survey that applied deductive verification modelled the robots as hybrid systems. There were a number of tools used for deductive verification, but the most popular is KeYmaera X [39], a proof assistant for differential dynamic logic which stands out due to its maturity and ease of use. A noteworthy feature of its proof system is that it has a number of ways of showing properties of continuous evolution. The most interesting one is differential induction, which allows us to specify invariants of the differential equation without solving it, although it is sometimes possible to outright solve the equations by using an integration with Mathematica.

Of the papers reviewed, KeYmaera X was used for verifying collision avoidance for a variety of systems [67, 84, 2], a speed controller that respects speed limits [85], and it was integrated with Simulink [55, 3]. KeYmaeraD, a variant of KeYmaera X used for formally verifying distributed hybrid systems, was applied to the problem of collision avoidance for an arbitrary number of aircraft [78].

The other proof assistants that appear in this survey – HOL4 [106], HOL Light [51], PVS [92], and Coq [63] – all had one thing in common that is different from KeYmaera X - they are all foundational. The mathematics in these tools are developed from fundamental axioms, in contrast to KeYmaera X which is constructed on top of the axioms of dL. This makes these proof assistants more flexible, but they also tend to be more difficult to use, and generally require more familiarity with the mathematical background.

One of the papers is dedicated exclusively to developing background theories and formalising the solutions of a class of differential equations in HOL4 [104]. Of the others, three [76, 101, 89] use the solutions of the ODEs to verify control algorithms. Chan et al. [19] use Rocq with differential

induction to prove stability properties, and the other papers [11, 10] develop methodologies for modelling and analysis.

Finally, Nakamura et al. use the CafeOBJ/OTS method to model the robots as observational transition systems and formally verify them using equational reasoning [89].

A notable mention that did not appear in the survey is IsaVODEs, an Isabelle/HOL approach developed by Munive et al.[88]. This tool implements a proof calculus inspired by KeYmaera X, but rather than stating its proof rules as axioms, its proof rules are based on the mathematical theories developed in Isabelle/HOL. This allows for greater extensibility and flexibility, as we can make use of any Isabelle results within the tool. IsaVODEs will be covered further in Chapter 3.

A recurring distinction among these tools is whether they use a *deep* or a *shallow* embedding of the object logic. In a deep embedding, the syntax of the logic is represented explicitly as a datatype within the prover and its meaning is given by interpretation functions; this exposes the logic itself to formal reasoning, but verifying a concrete system then requires manipulating syntax trees and repeatedly applying the semantic functions. In a shallow embedding, the constructs of the object logic are instead identified directly with native constructs of the prover, so that its existing automation and libraries can be reused when verifying systems, at the cost of being unable to reason about the syntax as an object. This choice has a direct bearing on how the tools above are used: KeYmaera X axiomatises dL directly, whereas Bohrer et al. [15] give a deep embedding of dL in both Rocq and Isabelle/HOL and use it to establish the soundness of the KeYmaera X proof calculus — a verification of the logic's metatheory rather than of any particular hybrid system. IsaVODEs, by contrast, uses a shallow embedding, reusing Isabelle's ODE libraries and proof automation to verify systems directly. Isabelle/HOL provides a substantially higher degree of proof automation than Rocq; the Rocq development of Chan et al. [19], for instance, applies differential induction to prove specific stability properties but says little about the automation of such proofs, while approaches that instead rely on explicit closed-form solutions of the ODEs are inherently restricted to equations that can be solved symbolically. We are not aware of any other implementation of differential induction or of differential dynamic logic in Rocq.

2.1.3 Other verification approaches

In this section we briefly survey approaches that do not simply verify properties of models.

A few papers apply statistical model checking (SMC), an approach where the system is simulated a number of times rather than exhaustively checked, and a probability of correctness is produced instead of a boolean answer. This is appealing for very large or stochastic systems, for which model checking can be intractable, and it can be seen as a middle ground between formal verification and simulation. SMC is for example applied by Lestingi et al.[75] for the verification of a human-following robot, where the robot is modelled using a set of hybrid automata and the humans are modelled stochastically. Gjondrekaj et al. use SMC for distributed, stochastic robot models [46], and Foughali uses it in his verification approach when regular model checking does not scale [35]. An advantage of SMC is that it can be very easily parallelised. This is used in the SyLVaaS tool developed by Mancini et al., which enables distributed model checking of hybrid systems with disturbances [82].

Machine learning is popular in robotics, but notoriously difficult to verify. Despite this, there are some attempts to verifying machine learning models. There are a wide range of approaches to this. One is to integrate a formal verification tool into the machine learning controller: Guidotti et al. integrate the dReal SMT solver into an ML-based controller [47], and Pore et al. apply deep

reinforcement learning with model checking in the loop in order to increase the safety of the system [98]. On the other hand, Huang et al. transform the ML models to labelled transition systems, which can then be model checked directly [62].

Runtime verification is applied by Wang et al.[113]. Here they present a tool for runtime verification where you describe the safety properties in a domain-specific language (DSL) which is automatically translated into a real-time verifier. They extend this work to target ROS programs [112].

Choi et al. verify robot simulations by specifying properties in a temporal logic and checking that the simulation conforms to the property [20]. Fang et al. integrate the PHAVer reachability tool with Matlab for simulation [31]. Savicks integrates simulation and formal verification in a different way, by developing a tool extension for the Rodin platform to work with Event-B. Mancini et al. combine model checking with simulation in order to obtain better scalability [81].

It is also possible to integrate model checking with simulation. It can be used for model checking, where all the possible scenarios are simulated [81]. It can also be used for visualisation and validation of the formal models [100, 22, 40].

A number of the papers present modelling approaches for robotics which are suitable for formal verification. Askarpour et al. present a methodology for modelling human-robot interactions using a logical language, which can then be checked using the Zot SAT checker [8, 6]. They later extended this work to perform formal risk analysis [111, 7].

2.1.4 Limitations of current approaches

We have seen that discrete state machines are a very versatile paradigm, and they have been applied in a wide variety of scenarios. Despite this, they are not suitable for modelling and verifying more low-level goals, such as motion planning, since they have to be expressed in terms of the real world. Indeed, the works surveyed that tackle such issues with discrete state machines can only do so by discretising the real world, which is not accurate enough for the purpose of motion planning. Hence, this modelling approach is not sufficient for addressing research question 1. For this, we need a modelling approach that can allow us to reason about robot interaction with the real world, which is continuous in nature.

Hybrid systems are more suitable for this aim, but they can only model uncertainty through nondeterminism, which does not allow us to reason about some events being more likely than others. In the literature we do find a few contributions that do use probability with hybrid systems, but they all use approximate techniques such as statistical model checking which can only provide verification up to some probability.

Hence, developing a deductive verification tool for stochastic hybrid systems, which could provide proven guarantees of the correctness of these systems, would be a valuable and novel contribution to the field.

2.2 Stochastic Hybrid Systems in Isabelle/HOL

In order to address the gap in the literature identified in the previous section, we propose developing a logic for stochastic hybrid systems in Isabelle/HOL, which will enable formal verification for continuous, stochastic robot models.

2.2.1 Stochastic Hybrid Systems

Stochastic hybrid systems (SHS) are hybrid systems with stochastic elements. These can be introduced by making the discrete transitions probabilistic, or adding stochastic noise to the continuous evolution, as detailed by Pola et al. [97]. The simplest class of SHS they present are partially-deterministic Markov processes (PDMP) [18, 23]. These have probabilistic transitions, and deterministic continuous dynamics. They extend hybrid systems by replacing the transition function with a transition rate function and a transition probability measure.

There are two approaches to stochastic dynamics - switching diffusion processes [43] and stochastic hybrid systems [60]. Both extend hybrid systems with stochastic differential equations (SDEs), which add random noise to ODEs:

$$dX_t = \mu(X_t, t)dt + \sigma(X_t, t)dW_t$$

Here, σ is a function from state and time to a covariance matrix, and dW stands for the differential of the Wiener process, i.e. Brownian motion. The semantics of SDEs can be given using the Itô integral [90]:

$$X_t = X_0 + \int_0^t \mu(X_s, s)ds + \int_0^t \sigma(X_s, s)dW_s$$

The first integral is a standard Riemann integral, accumulating the deterministic drift μ over time. The second integral accounts for the random fluctuations, integrating the diffusion σ against the Wiener process. It cannot be treated as an ordinary integral, since the sample paths of Brownian motion are too irregular – they are nowhere differentiable, so there is no well-defined rate of change to integrate against. Instead, the Itô integral is defined as a limit of sums over ever finer partitions of the time interval, where in each subinterval the diffusion is evaluated at the start of the subinterval, so that it does not depend on the future behaviour of the process.

2.2.2 Proposed Logics

Systems with stochastic dynamics can model imperfect actuators, but they are much more complex than their deterministic counterpart.

A few logics have been proposed for the formal verification of SHS, but none of them have yet been implemented:

- Platzer [96] proposes an extension of differential dynamic logic which adds a stochastic element to the continuous evolution and probabilistic composition.
- Wang et al.[114] extend hybrid CSP [77] with stochastic evolution.
- Hahn et al. [49] extend Modest, a modelling language for stochastic timed systems, with differential equations.

All of them extend some language in order to tackle SHS, by introducing either stochastic or continuous behaviour. The approach we propose is similar, extending the work of Foster et al. [34] to work with stochastic hybrid systems. The most appealing work to base the tool off is Platzer's stochastic dL (SdL), since the hybrid verification tool in Isabelle/HOL is already similar to dL. The modelling approach for SHS is called stochastic hybrid programs (SHP), which have the following syntax:

$$\alpha, \beta ::= x := \theta \mid x := * \mid ?H \mid dx = bdt + \sigma dW \& H \mid \lambda\alpha \oplus \nu\beta \mid \alpha; \beta \mid \alpha^*$$

SHPs consist of the following operators:

- Assignment ($x := \theta$): updates the value of variable x to the value of expression θ.
- Nondeterministic assignment ($x := *$): updates x to an arbitrary value.
- Test ($?H$): aborts the execution if condition H is not satisfied.
- Stochastic evolution ($dx = bdt + \sigma dW \& H$): evolves the state according to an SDE with drift b and noise σ, as long as condition H is satisfied.
- Probabilistic choice ($\lambda\alpha \oplus \nu\beta$): executes program α with probability λ and β with probability ν.
- Sequential composition ($\alpha; \beta$): executes α followed by β.
- Nondeterministic iteration (α^*): executes α zero or more times.

2.2.3 Isabelle/HOL probability

Isabelle is a generic proof assistant based on higher order logic [65]. It follows the LCF architecture: theorems are a datatype, and new theorems can only be constructed from existing ones or axioms. The most popular logic is Isabelle/HOL, which has been used to formalise a large amount of mathematics, from pure mathematical results such as the Jordan curve theorem to computer science structures, such as the formalisation of van Emde Boas trees [5]. Much of this is available either as part of the standard distribution, or in the Archive of Formal Proofs[1], a large collection of user-supplied Isabelle theories.

The probability theory available in Isabelle/HOL is based on measure theory, developed by Hölzl and Heller [59]. This is the study of measure functions - real-valued functions on sets that satisfy the properties of positivity and countable additivity. A probability measure is then simply a measure function that assigns value 1 to its universe. Many of the concepts in probability theory have generalised versions in measure theory, such as random variables and null sets.

This was developed further in Hölzl's PhD [57], where he develops a number of probability theory concepts, including Markov chains. Finally, he formalised continuous time Markov chains [58]. In order to reason about stochastic differential equations we need to extend the state space of these theories from discrete to continuous and develop a theory of martingales: continuous sequences of random variables. The Itô integral is a special case of these, and its formalisation would allow us to give the appropriate semantics to our stochastic hybrid systems. Measure-theoretic probability is not unique to Isabelle/HOL: comparable libraries exist in Lean's mathlib [109], which has a substantial theory of martingales and conditional expectation, and in Coq through Coquelicot [16] and the MathComp-Analysis development of Affeldt et al. [4]. We compare our own contributions against these developments in detail in Chapters 6 and 7.

[1] `https://www.isa-afp.org`

2.3 Summary

We have found that the majority of formal verification approaches in robotics rely on model checking of discrete state machines, which is effective for high-level tasks but insufficient for verifying low-level, continuous, or real-world interactions. Hybrid systems provide a more faithful representation of robotic systems by combining discrete and continuous dynamics, and various verification approaches have been applied to them, including deductive verification.

However, despite their strengths, hybrid systems cannot be used to model uncertainty probabilistically. This a critical limitation in verifying real-world robotic systems, as noise and probabilistic effects are inherent in these. Stochastic hybrid systems extend hybrid systems by incorporating stochastic behaviour, which lets us model these effects. However, they are more difficult to verify, and current tools can only provide approximate guarantees.

A few theoretical logics have been proposed for stochastic hybrid systems (SHS), such as Platzer's stochastic differential dynamic logic (SdL), but none have been implemented in widely-used or foundational proof assistants. This is a notable gap in the literature: there is no implementation of a deductive verification framework for SHS.

It is possible, however, to model uncertainty using hybrid systems by using nondeterminism. In the following chapter, we shall give an overview of how deductive verification can be applied to hybrid systems, how we can use them to model uncertainty, and finally of the drawbacks of this approach.

Chapter 3

Modelling uncertainty

Uncertainty comes in many forms; aleatoric uncertainty is inherent in the world, e.g. sensor noise, whereas epistemic uncertainty is related to our knowledge, and can be resolved as we acquire more — an example is learning about the layout of an unfamiliar room. The objective of this thesis is to develop a framework to model and formally verify systems which incorporate uncertainty. Probability is a way to quantify this, but not the only one: we can use nondeterminism too.

Nondeterminism has already been applied to modelling uncertainty in the context of cyber-physical systems. Mistch and Platzer [87] use KeYmaera X to mechanise hybrid system models that exhibit uncertainty by letting the system take values in some interval centered at the true value.

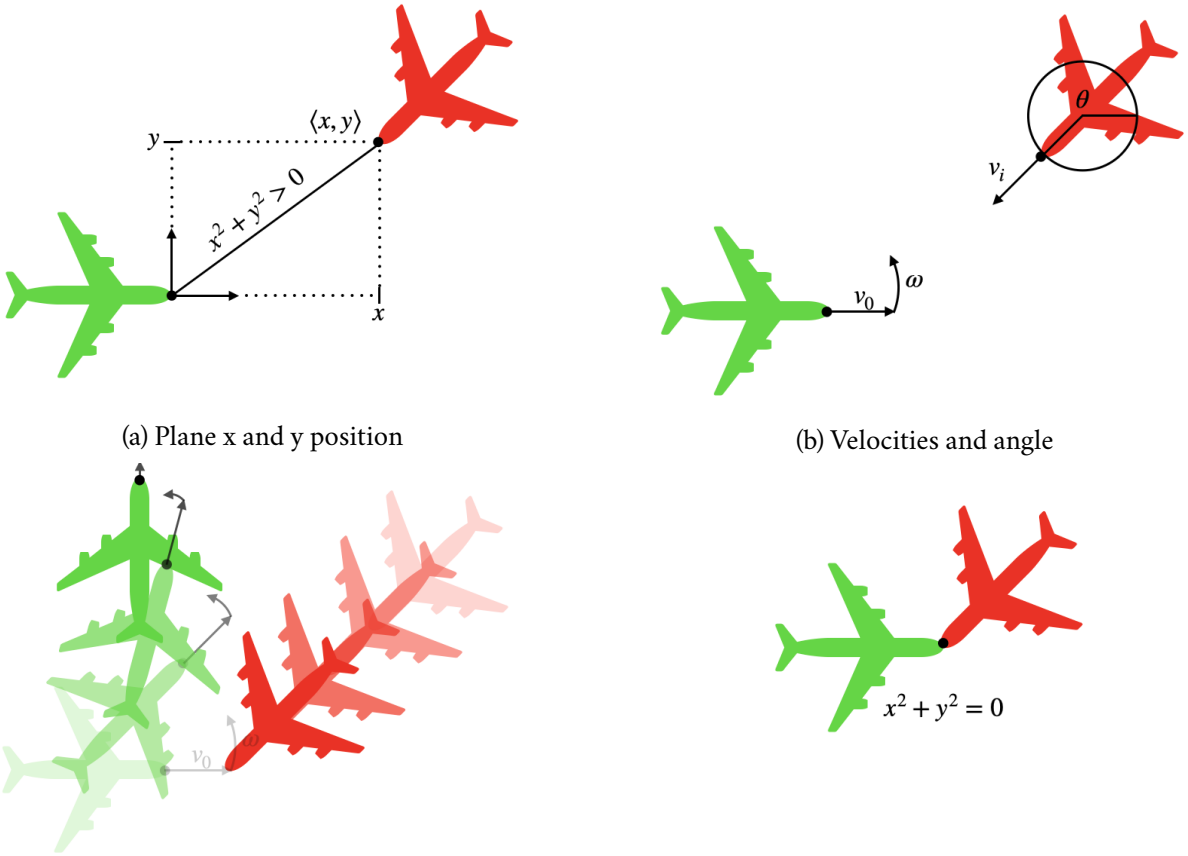
This approach is attractive, as it leverages existing tools and is easy to comprehend. In this chapter, we will explore how these two different models of uncertainty can be applied to the same systems, and the trade-offs that occur. We begin by introducing a case study on planar flight in IsaVODEs in Section 3.1. In Section 3.2, we describe the implementation of a computer algebra system integration for solving differential equations in Isabelle/HOL. Finally, in Section 3.3, we discuss the trade-offs of modelling uncertainty using nondeterminism.

In particular, we will explore how Isabelle/HOL, our proof assistant of choice, can be applied to this problem. For this, we can use IsaVODEs [88] to formally verify hybrid systems with a logic that mirrors that of KeYmaera X.

Research questions These questions are related to the broader research questions from the introduction, specifically focusing on the trade-offs between nondeterminism and probability for modelling uncertainty (RQ 1) and the suitability of Isabelle/HOL for mechanising these systems (RQ 2).

1. What is the best way to model uncertainty in the context of cyber-physical systems?
2. To what extent is Isabelle a good choice for modelling cyber-physical systems which exhibit uncertainty?

3.1 IsaVODEs: Planar flight



(a) Plane x and y position

(b) Velocities and angle

(c) Evasive maneuvers

(d) Collision

Figure 3.1: Diagrams illustrating the flight dynamics example

In this section, we motivate and demonstrate the usage of IsaVODEs with a worked example: an aircraft collision avoidance scenario that was first presented by Mitsch et al. [87]. It describes an aircraft trying to avoid a collision with a nearby intruding aircraft travelling at the same altitude. We can model this intruding aircraft by considering its coordinates x and y, and angle ϑ, in the reference frame of our own ship. The intruding plane has fixed velocity $v_i > 0$, and our plane has fixed velocity $v_o > 0$ and variable angular velocity ω. This is illustrated in Figure 3.1a and 3.1b.

Using IsaVODEs, we can model this using a **dataspace** command, as shown below:

```
dataspace planar_flight =
  constants
    v_o :: real (* own_velocity *)
    v_i :: real (* intruder velocity *)
  assumes
    v_o_pos: "v_o > 0" and
    v_i_pos: "v_i > 0"
  variables (* Coordinates in reference frame of own ship *)
    x :: real (* Intruder x *)
    y :: real (* Intruder y *)
    θ :: real (* Intruder angle *)
    ω :: real (* Angular velocity *)
```

This command allows us to define our constants, any assumptions about the verification problem, and the system's variables. In this case, we postulate two constants v_o and v_i, both of type `real`, for the velocity of the aircraft and intruder, respectively. Here, `real` is the type of precise mathematical real numbers, as opposed to floating points or rationals.

As per the problem statement, we assume that both of these constants are strictly positive. This is specified by the assumptions called `v_o_pos` and `v_i_pos`, respectively, which can be used as hypotheses in proofs. We supply the variables of this system, which give the relative position of the intruder (`x` and `y`), its orientation θ, and its angular velocity ω.

We next specify the dynamics that model the physical system by defining a constant called `plant`. The system of ODEs is non-linear and uses trigonometric functions:

```
definition "plant ≡ {x` = v_i * cos θ - v_o + ω * y,
                     y` = v_i * sin θ - ω * x,
                     θ` = -ω}"
```

The ODEs can be specified in a user-friendly manner, as physicists and engineers would informally state them, in terms of the constants and variables of the system `x`, `y`, and φ.

Next, we define a simple controller to avoid collisions, as explained below:

```
abbreviation "I ≡ (v_i * sin θ * x - (v_i * cos θ - v_o) * y
                  > v_o + v_i)^e"

abbreviation "J ≡ (v_i * ω * sin θ * x - v_i * ω * cos θ * y
                 + v_o * v_i * cos θ
                 > v_o * v_i + v_i * ω)^e"
```

```
definition "ctrl ≡ (ω ::= 0; ¿I?) ⊓ (ω ::= 1; ¿J?)"
```

This is based on two invariants, I and J, for two different scenarios. These are properties that remain true after the unfolding of each scenario. The invariant I holds when going straight is safe, while J holds when an evasion manoeuvre is allowed. Our controller selects whether to set our aircraft's angular velocity to 0 or 1 depending on which invariant holds (I and J respectively). The evasive manoeuvres that occur when the angular velocity is set to 1 are illustrated in Figure 3.1c.

Finally, our model follows the usual structure of an iteration (*) of control choices (`ctrl`) followed by a nondeterministic evolution of the system dynamics (`plant`):

```
definition "flight ≡ (ctrl; plant)*"
```

The behaviour of the system is characterised by all states reachable by executing the controller followed by the plant a finite number of times.

Next, we show how we can formally verify the collision avoidance of this system. We do this by specifying a Hoare triple within an Isabelle lemma:

```
lemma flight_safe: "{x² + y² > 0} flight {x² + y² > 0}"
proof -
  have ctrl_post: "{x² + y² > 0} ctrl {(ω = 0 ∧ @I) ∨ (ω = 1 ∧ @J)}"
    unfolding ctrl_def by wlp_full

  have plant_safe_I: "{ω = 0 ∧ @I} plant {x² + y² > 0}"
    unfolding plant_def apply (dInv "($ω = 0 ∧ @I)ᵉ", dWeaken)
    using v_o_pos v_i_pos sum_squares_gt_zero_iff by fastforce

  have plant_safe_J: "{ω = 1 ∧ @J} plant {x² + y² > 0}"
    unfolding plant_def apply (dInv "(ω=1 ∧ @J)ᵉ", dWeaken)
    by (smt (z3) cos_le_one mult_if_delta mult_le_cancel_iff2
          mult_left_le sum_squares_gt_zero_iff v_i_pos v_o_pos)

  show ?thesis
    unfolding flight_def
    apply (intro hoare_kstar_inv hoare_kcomp[OF ctrl_post])
    by (rule hoare_disj_split[OF plant_safe_I plant_safe_J])
qed
```

We formulate collision avoidance using $x^2 + y^2 > 0$ as the invariant in the Hoare triple. A collision occurs when $x^2 + y^2 = 0$, as illustrated in Figure 3.1d. We use Isabelle's Isar scripting language to break down the verification into several intermediate properties via the Isar command "**have**", which takes a label followed by a property specification.

We first calculate the postconditions that arise from running the controller (`ctrl`) by using our tactic `wlp_full`. This gives us two possible execution branches – one where $I \wedge \omega = 0$ holds, and one where $J \wedge \omega = 1$ holds. We give this first property the name `ctrl_post`.

The postconditions provide two possible initial states for the plant, which we consider using the properties `plant_safe_I` and `plant_safe_J`. We show that both preconditions guarantee the problem's postcondition $x^2 + y^2 > 0$ by applying differential induction, a technique that proves

an invariant of a system of ODEs without computing a solution. In each case, we use our Eisbach-designed method `dInv`, which takes as a parameter the invariant we wish to prove. The invariant is simply the precondition of each Hoare triple. However, we then need to prove that this invariant implies the postcondition, which is done using a further method called `dWeaken`. The remaining proof obligations can be solved using the **`Sledgehammer`** tool, which calls external automated theorem provers to find a solution and reconstructs it using the names of previously proven theorems in Isabelle's libraries. In particular, the property `plant_safe_J` is discharged with a call to Sledgehammer, which uses trigonometric identities formalised in HOL-Analysis. We display the proof state resulting from applying differential induction below:

```
`$ω = 1 ∧
 v₀ * vᵢ + vᵢ * $ω
 < vᵢ * $ω * sin ($ϑ) * $x - vᵢ * $ω * cos ($ϑ) * $y +
   v₀ * vᵢ * cos ($ϑ) ⟶
 0 < ($x)² + ($y)²`
```

In the case that **`sledgehammer`** is unable to find a proof, this proof obligation is quite readable, and so a manual proof or refutation could be given.

Finally, we put everything together to show that the whole system is safe, using the Isar command **`show`**, which is used to conclude proofs and restate the overall goal of the lemma (`?thesis`). The proof uses a couple of high-level Hoare logic laws, and the properties that were proven to complete the proof. This final step can be completely automated using Isabelle's classical reasoner, but we leave the details for the purpose of demonstration.

It is also possible to provide a proof of this verification problem using the aforementioned first workflow. Namely, users can provide the solutions to the system of ODEs `plant`. This completes our overview of our tool and its capabilities. In the remainder of this chapter, we expound the technical foundations of our CAS integration.

3.2 Solutions from Computer Algebra Systems

We have integrated the Wolfram Engine, a Computer Algebra System (CAS), into Isabelle/HOL to supply symbolic solutions to ODEs. The user can make use of the integration via the **`find_local_flow`** command, which supplies a solution to the first ODE it finds within the current subgoal.

The workflow for our CAS integration is illustrated in Figure 3.2. It consists of a multi-stage process that extracts the problem from the proof state, solves it using the external CAS, and reconstructs the solution in Isabelle/HOL.

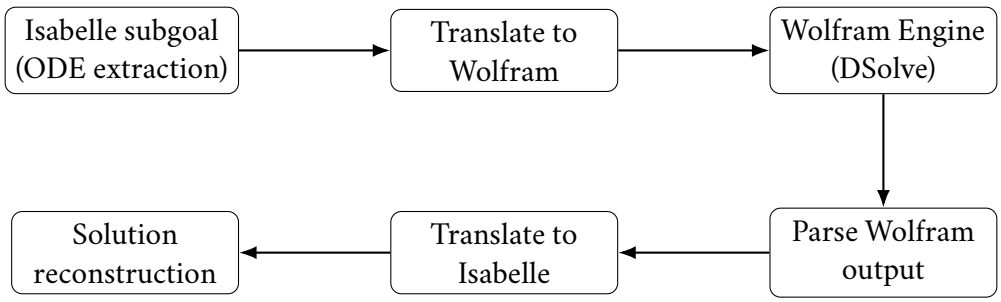


Figure 3.2: CAS integration workflow

Below, we show an application of our integration and its corresponding output in Isabelle. Users can click the greyed-out area to automatically insert the suggestion.

```
lemma local_flow_example:
  "{x > 0} x ::= 1; {x´ = 1} {x > 1}"
  find_local_flow
Output:
  Calling Wolfram...
  λt. [x ⇝ t + $x]
  try this: apply (wlp_solve "λt. [x ⇝ t + $x]")
```

Here, our plugin requests a solution to the ODE $x' = 1$ from the Wolfram Engine. A λ-abstraction denotes the solution, which inputs the current time and has in its body the constructed substitution. It provides the value for each continuous variable at time t. In this case, the substitution $[x \rightsquigarrow t+x]$ represents the solution $x\,t = t + x\,0$.

Our integration performs the following steps in order to produce a solution:

1. Retrieve an Isabelle term describing the system of ODEs.
2. Convert the term into an intermediate representation.
3. Use one of the CAS backends to solve the ODE.
4. Convert the solution to an Isabelle term.
5. Certify the solution using the `wlp_solve` Isabelle tactic.

We consider each of these stages below.

Intermediate representation To have an extensible and modular interface, we implement (1) an intermediate data structure for representing ODEs and their solutions, and (2) procedures for translating back and forth between Isabelle and our intermediate representation. This allows us to capture the structure of the ODEs without the added overhead of Isabelle syntax. It also gives us a unified interface between Isabelle and any CAS.

Our plugin assumes that a system of ODEs is expressed as a substitution of the form $[x \rightsquigarrow e, y \rightsquigarrow f, \cdots]$. Each expression $(e, f, \cdots)$ gives the derivative expression for each corresponding variable, potentially in terms of other variables. Naturally, for an ODE, these expressions are formed using only arithmetic operators and mathematical functions. We therefore derive the following constrained grammar for arithmetic expressions:

```
datatype AExp = NConst of string | UOp of string * AExp |
           BOp of string * AExp * AExp | NNat of int | NInt of int |
           NReal of real | CVar of string | SVar of string | IVar
```

`NConst` represents numeric constants, such as e or π. `UOp` is for unary operations and `BOp` is for binary operations. Both of these take a name, and the number of required parameter expressions. The name corresponds to the internal name for the operator in Isabelle/HOL. The operator "+", for example, has the name `Groups.plus_class.plus`. We then have three constructors for numeric constants: `NNat` for naturals, `NInt` for integers and `NReal` for reals. We have three characterisations of variables: `CVar` are arbitrary but fixed variables, `SVar` are mutable (state) variables, and

`IVar` represents the independent variable of our system, usually time.

A system of ODEs is then modelled simply as a finite map from variable names to derivative expressions. Converting back and forth between Isabelle terms (i.e. elements of the `term` datatype in Isabelle/ML) and the `AExp` datatype consists of recursing through the structure of the term, using a lookup table to translate each named arithmetic function.

Integrating a CAS into IsaVODEs consists of three separate components: a translation from the intermediate representation to the CAS input format, an interface with the CAS to obtain a solution, and a translation from the solution back into the required format.

Wolfram Engine The Wolfram Engine is the CAS behind WolframAlpha and Mathematica [116]. It is one of the leading CASs on the market, with powerful features for solving differential equations, amongst many other applications. The Wolfram language provides the DSolve function which produces solutions to various kinds of differential equations.

The basic building block for this integration is a representation of Wolfram expressions as an ML datatype. In the Wolfram language, everything is an expression [115], so this is sufficient for a complete interface. We represent these expressions as follows:

```
datatype expr = Int of int
              | Real of real
              | Id  string
              | Fun string * expr list
              | CurryFun string * expr list list
```

`Real` and `Int` represent real and integer numbers respectively. `Id` represented an identifiers, and `Fun` represents functions with only one list of arguments, which are distinguished from curried functions `CurryFun` with multiple sets of arguments.

Our approach for translation from `AExp` to `expr` is the following:

1. Generate an alphabetically ordered variable mapping to avoid name clashes and ease solution reconstruction.
2. Traverse the expression tree and translate each term to Wolfram.
3. Wrap in the Wolfram `DSolve` function, supplying the state and independent variables along with our ODE.

Once we have a well-formed Wolfram expression, we use the `wolframscript` command-line interface to obtain a solution in the form of a list of Wolfram `Rule` expressions, which are isomorphic to our substitutions. We can parse the result back into the `expr` type, and translate this into the `AExp` type using a translation table for function names and the variable name mapping we constructed.

Capabilities Aside from the CAS itself failing to find a solution, there are two cases where the integration is unable to produce an Isabelle certification despite the CAS returning a solution.

The first case happens when the provided solution contains a function not implemented in Isabelle. Instances of this case include functions defined in terms of integrals such as the error function or the Bessel function.

The second case concerns solutions which are undefined somewhere within their domain. Examples of this case include divisions by 0, which users can address by manually specifying the domain of the solution in Isabelle.

Despite these shortcomings, the CAS integration proves very effective at leveraging the solutions provided by CASs, as a large class of ODEs can be automatically solved and certified with minimal input from the user.

3.3 Uncertainty as nondeterminism

We now illustrate how uncertainty can be modelled using nondeterministic hybrid systems, by considering the example of a water tank that has to be filled to a certain level. We adapt this from a KeYmaera X example given by Mitsch and Platzer [86].

We begin by considering a water tank which exhibits no uncertainty, which we can model by using a hybrid program. First, we define the state space that our program will be operating in:

```
dataspace water_tank =
  constants
    m :: real
    ε :: real
  assumes
  ε_pos: "ε > 0"
  variables
    f :: real
    l :: real
    c :: real
```

Here, we fix two arbitrary constants m and ε, and assume that $\epsilon > 0$. These will stand for the maximum volume of our water tank and the frequency at which our controller operates respectively. Then, we define three state variables: f will be the flow of the water tank, l the water tank level, and c the elapsed time since the last controller operation.

We now proceed to define the elements of our system:

```
definition "plant ≡  { l‘ = f, c‘ = 1 | (0 ≤ l ∧ c ≤ ε)}"
```

The "plant", or physical system, is given by an ODE. $l' = f$ lets the flow determine the change in water level, and $c' = 1$ makes c track time. The ODE executes until the time exceeds ε.

```
definition "ctrl ≡ f ::= ? ; ¿-1 ≤ f ∧ f ≤ ((m - l) / ε)? ; c ::= 0"
```

The controller can only set the flow. It chooses a valid flow value by first setting the flow nondeterministically to any value, and then using the test operator to ensure that it falls within a safe limit, which is given by expression $f \leq \frac{m-l}{\varepsilon}$. If f is chosen to be equal to $\frac{m-l}{\varepsilon}$ then the water tank level l will be filled up to the maximum level m after an execution of length ε.

Now, we can show that the water tank level never exceeds the maximum:

```
lemma water_tank_safe:
  "H{0 ≤ $l ∧ $l ≤ m} (ctrl ; plant)* {0 ≤ $l ∧ $l ≤ m}"
  apply (rule hoare_kstar_inv)
  unfolding ctrl_def plant_def
  apply (wlp_solve "λt. [l ⇝ $f * t + l, c ⇝ t + c]")
```

```
  apply expr_simp
  by (smt (verit) mult_left_mono pos_le_divide_eq zero_less_mult_iff)
```

We prove this by using the solution to the system that the **find_local_flow** tool provides. Since the system is linear, the solution is given by $l' = ft + l, c' = t + c$. Then, a call to the simplifier returns a straightforward arithmetic goal that can be automatically discharged.

Water tank with uncertainty How can we model the uncertainty in this system? The actuation of the valve may have some error, as will the reading of the water level. One approach using nondeterminism is to keep track of the true, physical values of the water level and flow rate, and add some error quantity to the actuation and sensor values:

```
dataspace uncertain_water_tank =
  constants
    m :: real
    ε ::real
    U :: real
  assumes
    epsilon_nonneg: "0 ≤ ε" and
    U_nonneg: "0 ≤ U"
  variables
    f :: real
    l :: real
    lu :: real
    c :: real
```

The plant remains the same, but we add a measurement stage.

```
definition "tank_measure ≡ lu ::= ?;
¿l - U ≤ lu ∧ lu ≤ l + U?"
```

Now, we can still construct a hybrid program which never overfills the water tank by accounting for the error in the control inputs.

```
definition "ctrl ≡ f ::= ?;
¿-1 ≤ f ∧ f ≤ (m - (lu + U)) / ε?; c ::= 0"
```

We can now prove that this water tank operating with uncertainty is still safe, using the same solution approach as before.

```
lemma uncertain_water_tank_safe:
 "H{0 ≤ l ∧ l ≤ m ∧ l - U ≤ lu ∧ lu ≤ l + U}
     (ctrl; plant; tank_measure)*
   {0 ≤ l ∧ l ≤ m}"
```

Uncertain water tank is safe

This model is quite restrictive, and arguably imposes unnecessary restrictions on the controller. With probability, we have an additional knob to turn by being able to give the safety bound that we are expecting to satisfy. We can also provide more accurate models of the behaviour of the valve - rather than being restricted to simply giving all possible values, we can give the relative likelihood of each one, which lets us give less restrictive models of behaviour. For the water tank example, we can model both sensor and flow value using a Gaussian distribution centered at the actual value, and adjust the variance to match the real system.

3.4 Summary

We have successfully adapted KeYmaera X proofs to IsaVODEs, and evidenced that uncertainty can be modelled using nondeterminism while still being able to provide verification guarantees. Porting proofs from KeYmaera X to IsaVODEs is straightforward — the tactics and proof rules in KeYmaera X are generally available in IsaVODEs, with the notable exception of quantifier elimination, which makes the translation task mostly mechanical. Despite this, we did have to implement the `dInv` tactic for inserting an invariant into an ODE execution in order to more closely mirror the KeYmaera proof.

As we have argued, nondeterminism does have its downsides. By modelling using nondeterminism, we are forced to consider the worst-case scenario, as we must prove that the system is safe in any reachable state for a safety proof to be successful. By contrast, probabilistic modelling lets us give different weights to states that are more or less likely, which lets us prove that safety properties hold within some probabilistic threshold. This approach mirrors how cyber-physical systems are tested and certified, and gives us the flexibility required to accurately model real systems.

Chapter 4

Modelling Variable Sets: Scenes and Scene Spaces

In this chapter, we develop the theory of scenes and scene spaces, which provide a foundation for reasoning about sets of variables in Isabelle/HOL. We begin by introducing lenses, which are used to model variables in IsaVODEs, and discuss their limitations for representing sets of variables. We then introduce scenes, which sacrifice some type information for a more versatile algebraic structure, and scene spaces, which add the necessary structure for set-like reasoning.

4.1 Optics primer: Lenses

Definition 4.1. A *lens* is a quadruple $X \triangleq (\mathcal{V}, \mathcal{S}, get, put)$[33]. Here $\mathcal{V}$ and $\mathcal{S}$ are non-empty sets called the *view type* and *state space*, respectively, $get : \mathcal{S} \to \mathcal{V}$ and $put : \mathcal{S} \times \mathcal{V} \to \mathcal{S}$ are total.

In the standard implementation of IsaVODEs, lenses are used to model variables. As part of the experiments we did with IsaVODEs, we discovered that the tool lacks some compositional reasoning that requires that we characterise the sets of variables a program uses. For example, compositional reasoning in differential dynamics logic requires formalisation of the sets of bound and free variables, which are called BV and FV [93]. Unfortunately, lenses cannot be collected in sets due to issues with their types, nor can they be combined in an arbitrary way to characterise a set, so we can currently not construct these sets in IsaVODEs.

Intuitively, a lens $X : \mathcal{V} \to \mathcal{S}$ describes a $\mathcal{V}$-shaped region of the state space $\mathcal{S}$. Lenses can capture either individual variables of type $\mathcal{V}$, or several variables arranged in a hierarchy.

They usually obey a number of algebraic laws. To support the practical use of these theories, we provide an implementation in Isabelle/HOL, including an `alphabet_scene_space` command that automatically generates scene spaces for a given state space. In the following sections, we will distinguish between the mathematical theory and its Isabelle implementation where relevant.

Definition 4.2. A lens is said to be *total* if it obeys the following laws:

$$get(put(s, v)) = v, \qquad put(put(s, v'), v) = put(s, v), \qquad put(s, get(s)) = s.$$

These laws characterise the expected behaviours of the put and get functions. First, if we put some value to our lens, then we should retrieve the same value with get. Next, the latest put should

override any previous operations. Finally, putting the current value of the lens back into the state should result in an unchanged state. If only the first two laws hold, then the lens is called partial — partial lenses are useful for modelling variables that may not exist in certain states, such as pointer variables. Here, we focus only on total lenses.

A basic model is the n-ary Cartesian product space $\mathcal{S} \triangleq \mathcal{V}_1 \times \mathcal{V}_2 \times \cdots \times \mathcal{V}_n$, for which we can form a lens for each projection $\pi_i : \mathcal{S} \to \mathcal{V}_i$. However, in practice state spaces are more complex than this, often consisting of hierarchy, and lenses support such a generalisation.

Definition 4.3. Two lenses X and Y are *independent* if their put functions commute, and the getter from one lens is unaffected by putting to the other:

$$put_X\big(put_Y(\sigma, v), u\big) = put_Y\big(put_X(\sigma, u), v\big) \tag{4.1}$$

$$get_X\big(put_Y(\sigma, v)\big) = get_X(\sigma) \tag{4.2}$$

$$get_Y\big(put_X(\sigma, u)\big) = get_Y(\sigma) \tag{4.3}$$

Independence is an important property of lenses, which tells us that altering one lens will have no effect on the other. We can commute assignments of independent lenses, and generally independence allows us to combine properties of programs if they have been proven about independent sections.

Finally, lenses can be combined using the $(+)_L$ (lens addition) operator. Given two lenses X and Y with view types $\mathcal{V}_X$ and $\mathcal{V}_Y$ which share a state space $\mathcal{S}$, $X +_L Y$ will be a lens with state space $\mathcal{S}$ and view type $\mathcal{V}_X \times \mathcal{V}_Y$.

At first glance, this solves the lenses-as-sets problem - if we have some finite collection of scenes that we want to treat as a single object, we can simply take the iterated lens addition of them. Unfortunately, it is not possible to give such an object an adequate type in Isabelle if we do not know the number of lenses in this collection ahead of time, as the type of n-tuples for some arbitrary n cannot be given in Isabelle. Instead, we find a solution in scenes.

4.2 Scenes

We cannot collect lenses in sets, because they may have different types, and sets in Isabelle are homogeneous. Being able to characterise a region of the state is important, as it lets us answer questions such as “what variables are modified by this program?”.

Therefore, we need a set-like object which lets us combine different lenses on the same state space. To address this issue, we introduce scenes, which are isomorphic to lenses but do not have an explicit view type. To define them, we first define overriding functions:

Definition 4.4 (Overrider)**.** An associative function $\triangleright : S \to S \to S$ is an overrider if it satisfies the overriding law:

$$x \triangleright y \triangleright z = x \triangleright z \tag{4.4}$$

(4.5)

```
locale overrider =
  fixes F  :: "'s ⇒ 's ⇒ 's" (infixl "▷" 65)
  assumes
    ovr_overshadow_left: "x ▷ y ▷ z = x ▷ z" and
    ovr_overshadow_right: "x ▷ (y ▷ z) = x ▷ z"
```

For example, we can construct an overrider on the product type `'a × 'b` which overrides the value of the first element in the pair:

Definition 4.5 (Overrider on product type)**.** Define

$$(a, b) \triangleright_{\text{fst}} (c, d) \triangleq (a, d)$$

Then $\triangleright_{\text{fst}}$ is an overrider.

```
abbreviation fst_overrider :: "('a × 'b) ⇒ ('a × 'b) ⇒ ('a × 'b)"
  (infixl "▷fst" 65) where
"x ▷fst y ≡ (fst x, snd y)"

interpretation overrider "(▷fst)"
  by (unfold_locales; force)
```

We further distinguish idempotent overriders, which will characterise total lenses.

Definition 4.6 (Idempotent overrider)**.** An overrider $\triangleright$ is idempotent if it satisfies $x \triangleright x = x$

```
locale idem_overrider = overrider +
  assumes ovr_idem: "x ▷ x = x"
```

The overrider we define above is an idempotent overrider.

Definition 4.7 ($\triangleright_{\text{fst}}$ is idempotent.)**.**

```
interpretation idem_overrider "(▷fst)"
  by (unfold_locales; force)
```

Now, a scene is simply an overriding function. We define a new type for this, as we will be making use of the Isabelle type class system.

```
typedef 's scene = "{F :: 's ⇒ 's ⇒ 's. overrider F}"
  by (rule_tac x="λ x y. x" in exI, simp, unfold_locales, simp_all)
```

Scenes are closely linked to lenses. We can embed a lens as a scene by simply making use of the lens override, defined in the Optics library:

Definition 4.8 (Lens override)**.**

```
definition lens_override :: "('b ⟹ 'a) ⇒ 'a ⇒ 'a ⇒ 'a" (infixl "◁ı"
95) where
[lens_defs]: "S₁ ◁ₓ S₂ = putₓ S₁ (getₓ S₂)"
```

Definition 4.9 (Lens scenes)**.** A lens l can be recast as a scene through the lens override operation:

$$x \triangleright y = x \oplus_L y \text{ on } l$$

```
lift_definition lens_scene :: "('v ⟹ 's) ⇒ 's scene" ("⟦_⟧∼") is
"λ X s₁ s₂. if (mwb_lens X) then s₁ ⊕L s₂ on X else s₁"
```

Scenes can now be understood as equivalence classes of lenses. If two lenses map to a scene, then they have the same effect on the state space, and their types will be equal up to isomorphism. For example, we can show that $+_L$ is commutative up to isomorphism.

Lemma 4.1 (Lens addition commutes up to scene equality)**.** *For independent, total lenses X and Y, the following holds: $[\![X +_L Y]\!]_\sim = [\![Y +_L X]\!]_\sim$*

```
lemma lens_plus_commute_scene:
  assumes "mwb_lens X" "mwb_lens Y" "X ⋈ Y"
  shows "⟦X +L Y⟧∼ = ⟦Y +L X⟧∼"
```

Idempotent scenes correspond to total lenses:

Lemma 4.2 (Idempotent scenes are equivalent to total lenses)**.**

```
lemma idem_scene_eq_vwb:
  assumes "mwb_lens X"
  shows "vwb_lens X = idem_scene ⟦X⟧∼"
```

However, we cannot recover a lens from a scene unless we know the type of the original lens. By embedding the lens as a scene, we forget the type information, which gives rise to better compositionality — we can now collect these objects in sets.

By defining lens scenes, we establish a homomorphism. Every relevant lens operation lifts to scenes in a natural way, e.g. lens product, independence, etc.

Definition 4.10 (Independent scenes)**.** Two scenes are independent if their overriders commute.

```
lift_definition scene_indep :: "'a scene ⇒ 'a scene ⇒ bool" (infix "⋈S"
50)
is "λ F G. (∀ s1 s2 s3. G (F s1 s2) s3 = F (G s1 s3) s2)" .
```

We are also now in a position to construct set-like operators: union, intersection, and complement. These allow us to construct sets, which is great, but they do not have properties we like, because of this odd relation called "compatibility" which appears as a side-condition of some laws, and is quite nebulous. Therefore, we resolve this problem in the following section with scene spaces.

Definition 4.11 (Scene operations)**.** We can define scene union, intersection, complement. They can be ordered and form a lattice.

```
instantiation scene :: (type) "{bot, top, uminus, sup, inf}"
begin
  lift_definition bot_scene    :: "'a scene" is "λ x y. x" by (unfold_locales,
simp_all)
  lift_definition top_scene    :: "'a scene" is "λ x y. y" by (unfold_locales,
simp_all)
  lift_definition uminus_scene :: "'a scene ⇒ 'a scene" is "λ F x y.
F y x"
    by (unfold_locales, simp_all)
```

Scene union requires that the two scenes are at least compatible. If they are not, the result is the bottom scene.

```
  lift_definition sup_scene :: "'a scene ⇒ 'a scene ⇒ 'a scene"
```

```
    is "λ F G. if (∀ s₁ s₂. G (F s₁ s₂) s₂ = F (G s₁ s₂) s₂) then (λ s₁
s₂. G (F s₁ s₂) s₂) else (λ s₁ s₂. s₁)"
    by (unfold_locales, auto, metis overrider.ovr_overshadow_right)
  definition inf_scene :: "'a scene ⇒ 'a scene ⇒ 'a scene" where
    [lens_defs]: "inf_scene X Y = - (sup (- X) (- Y))"
  instance ..
end
```

Definition 4.12 (Scene compatibility)**.** This is a prerequisite for well-behaved union, intersection, etc.

```
lift_definition scene_compat :: "'a scene ⇒ 'a scene ⇒ bool" (infix
"##ₛ" 50)
is "λ F G. (∀ s₁ s₂. G (F s₁ s₂) s₂ = F (G s₁ s₂) s₂)" .
```

4.3 Scene spaces

Working in the space of all scenes means that we cannot guarantee that any two scenes will be compatible, and hence do not know if their union is defined. A first step towards fixing this is to work in a space where all scenes are compatible and which is closed under union.

To construct such a space, we take clues from linear algebra and define a scene space to be the set generated by taking unions of some collection of *basis scenes*. These basis scenes should be mutually independent and in some way "atomic". Then, we will be able to deconstruct any scene in the space into its constituent basis scenes.

The scene space is unique to every program. We can extract a scene from some section of the program, and use it to answer questions like "what are the free/bound variables in this program?" by obtaining its basis scene decomposition.

To define scene spaces, we first define some notions for collections of scenes:

Definition 4.13 (Scene fold)**.** Given some list of scenes, its distributed scene union can be taken by folding with scene union and using the bottom scene as the initial element.

```
abbreviation foldr_scene :: "'a scene list ⇒ 'a scene" ("⨆ₛ") where
"foldr_scene as ≡ foldr (⊔ₛ) as ⊥ₛ"
```

Definition 4.14 (Independent scene collection)**.** Set S forms an independent family if its elements are pairwise independent, i.e. $\forall x, y \in S.\ x \bowtie_S y$

```
definition scene_indeps :: "'s scene set ⇒ bool" where
"scene_indeps = pairwise (⋈ₛ)"
```

Definition 4.15 (Scene span)**.** A list of scenes S form a span if they combine to form the top scene $\top_S$: $\bigsqcup_S S = \top_S$.

```
definition scene_span :: "'s scene list ⇒ bool" where
"scene_span S = (foldr (⊔ₛ) S ⊥ₛ = ⊤ₛ)"
```

Intuitively, the whole space is covered by the scenes in a scene span.

We use lists rather than sets because of a focus on computability and finiteness: when characterising the variables of a given program, we are only ever going to consider a finite collection of scenes.

With the above definitions, we can give the conditions for a scene space to be defined:

Definition 4.16 (Scene space class)**.** A type forms a scene space if we can provide a list of mutually independent basis scenes which form a span.

```
class scene_space =
  fixes Vars :: "'a scene list"
  assumes idem_scene_Vars [simp]: "⋀ x. x ∈ set Vars ⟹ idem_scene x"
  and indep_Vars: "scene_indeps (set Vars)"
  and span_Vars: "scene_span Vars"
```

We name the basis scenes `Vars`, as conceptually they will denote individual variables in our program.

Definition 4.17 (Scene space)**.** The scene space of some suitable collection of basis scenes is the closure under scene union of the basis scenes and the bottom scene $\bot_S$.

```
inductive_set scene_space :: "'a scene set" where
bot_scene_space [intro]: "⊥S ∈ scene_space" |
Vars_scene_space [intro]: "x ∈ set Vars ⟹ x ∈ scene_space" |
union_scene_space [intro]: "⟦ x ∈ scene_space; y ∈ scene_space ⟧
    ⟹ x ⊔S y ∈ scene_space"
```

Here, command `inductive_set` constructs the least set of scenes that is closed under the listed properties.

We show that any scene in a scene space is idempotent, and any two scenes in a scene space are compatible. Further, scene spaces are finite, and they are closed under intersection and complement.

Any scene in a scene space can be decomposed into its basis scenes.

Definition 4.18 (Scene space basis decomposition)**.** Given some scene space $\mathcal{S}$ wit basis scene set $\mathcal{B}$, any scene $s \in \mathcal{S}$ can be decomposed into some set d such that $d \subseteq \mathcal{B}$ and $\bigsqcup_S d = s$.

```
lemma scene_space_vars_decomp:
  assumes "a ∈ scene_space"
  shows "∃xs. set xs ⊆ set Vars ∧ a = ⨆S xs"
```

These decompositions are not unique, as we allow scene spaces to contain the bottom scene as part of its basis, and we give the decomposition in terms of lists. However, they will be unique up to these factors. This construction lets us recover the set of basis scenes that make up some set, which will be useful in characterising free variables via set reasoning. This information can also be recovered via independence reasoning - a basis scene is independent of some other scene in the scene space if and only if it is not a part of its decomposition.

4.4 Frames

To make the usage of scenes practical for program verification, we define the type of scenes belonging to the scene space. We refer to elements of this type as frames.

Definition 4.19 (Frame type)**.**

```
typedef (overloaded) 'a::scene_space frame = "scene_space :: 'a scene
set"
  morphisms of_frame mk_frame
```

This is very advantageous for proof automation, as we can now remove the scene space membership or compatibility assumptions which were previously required whenever performing any operation on two scenes. At this point, the abstraction of scenes as set-like objects is complete. The implementation details of compatibility and overriders fade into the background, and we can construct frames using a set-like operator which is a list under the hood:

Definition 4.20 (Insert lens into frame)**.** Analogously to inserting elements into a set, we define an operator for inserting a lens into a frame. Given a frame F and a lens X, define lens insertion by

$$\text{insert}(X, F) = \begin{cases} [\![X]\!]_S \sqcup_S F & \text{if X corresponds to a basis scene} \\ F & \text{otherwise} \end{cases}$$

.

```
definition lens_insert :: "('a ⟹ 's::scene_space) ⇒ 's frame ⇒ 's
frame"
  where "lens_insert x a = sup (lens_frame x) a"

syntax
  "_frame_set" :: "args ⇒ 'a::scene_space frame"    ("{|(_)|}_F")
translations
  "{|x, xs|}_F" ⇌ "CONST lens_insert x {|xs|}_F"
  "{|x|}_F" ⇌ "CONST lens_insert x {||}_F"
```

With this syntax, we can treat frames as sets, which makes the development much more ergonomic for characterising the various sets of variables that a user might care about, such as free variables. We support this set-like characterisation by showing that frames form a complete lattice and a boolean algebra, which give us many of the properties of interest for sets. We do this by showing that the frame type is an instance of the relevant Isabelle typeclasses:

Lemma 4.3 (Frames form a boolean algebra)**.**

```
instantiation frame :: (scene_space) boolean_algebra
begin

lift_definition minus_frame :: "'a frame ⇒ 'a frame ⇒ 'a frame" is "λ
a b. a ⊓_S - b"
  by blast

lift_definition uminus_frame :: "'a frame ⇒ 'a frame" is "uminus"
  by blast

instance
  by (intro_classes; transfer)
     (simp_all add: scene_inter_indep)
end
```

Lemma 4.4 (Frames form a complete lattice)**.**

```
instance frame :: (scene_space) complete_lattice
```

4.5 Summary

We have introduced scenes, a set-like object for characterising the state space of programs. These objects circumvent the type issues that prevented lenses to be collected in, and treated like, sets, but their inherent lack of structure led the development of scene spaces in order to make them useful. Finally, for some given scene space, we can give the type of all its members, which we refer to as frames.

With frames, we now arrived at a way of characterising sets of variables in a shallow embedding which is applicable to practical verification. This will allow us to characterise, amongst other things, the free and bound variables in an arbitrary program, something which is essential for applying compositional reasoning.

Chapter 5

A modelling language for stochastic hybrid systems

In this chapter, we describe the design of a language for modelling reactive stochastic hybrid systems, and the implementation of an interpreter for this language. The language is based on Platzer's SdL [96].

A simulation language complements the formal logic by allowing practitioners to explore the behaviour of their models. By mirroring the language that will be used for formal modelling, the simulator can be seamlessly integrated into the verification workflow, which can enable prototyping of both model and verification conditions.

We have developed the language in the Haskell programming language. At the core of our implementation is a type-indexed abstract syntax tree, where each language construct is associated with a Haskell type. This lets us leverage the Haskell type-checker to ensure that a program can only be represented if it is well-typed.

To execute our programs, we give a semantic function based on the reader-writer-state (RWS) monad. This exposes the semantics of the language in a clear way, which makes it suitable for formalisation.

5.1 Language Design

The language we develop is based on the stochastic hybrid programs (SHPs) introduced by Platzer [96]. SHPs include a test operator; this discards an execution path if the test is not satisfied. This is not useful for simulation, so we replace these with control flow statements.

We give the syntax of our programs as a BNF grammar. We write $\mathtt{x}^{term}_{;}$ for a list of x, each terminated by ";", and similarly $\mathtt{x}^{sep}_{,}$ for a ","-separated list of x.

⟨*expr*⟩ ::= real
| ⟨*ident*⟩
| bool
| ⟨*expr*⟩ bop ⟨*expr*⟩
| uop ⟨*expr*⟩
| (⟨*expr*⟩)

We begin with our expressions. The primitives are identifiers, which can either be constants or variables, booleans, and real numbers. These can be composed with unary (uop) and binary (bop) operators, and they can be parenthesised.

⟨*Stmt*⟩ ::= ⟨*ident*⟩ ':=' ⟨*expr*⟩;
| ⟨*ident*⟩ ':=' { ⟨*expr*⟩ ", ⟨*expr*⟩ };
| `input` ⟨*ident*⟩;
| `if` ⟨*expr*⟩ { ⟨*SHP*⟩ } (`else` { ⟨*SHP*⟩ })
| `while` ⟨*expr*⟩ { ⟨*SHP*⟩ }
| `loop` { ⟨*SHP*⟩ }
| ⟨*SDE*⟩;

⟨*SHP*⟩ ::= [⟨*Stmt*⟩]

A program consists of a sequence of statements, to be executed in order. Since we terminate atomic statements with ";", we can interpret concatenation as sequential composition without risk of ambiguity. In order, the statements in our language are: assignment, random assignment, external input, conditional, while loop, infinite loop, and SDE execution.

Assignment, if statements, and while loops function as expected. Random assignment assigns a value drawn from a uniform distribution over the range $\{l, h\}$. The infinite loop operator executes the contained SHP until all the time allocated for simulation has passed. The input operator reads a value from an external stream, which can either be user input or programmatically provided. This gives our language a reactive element.

Finally, SDEs are expressions of the form:

⟨*SDE*⟩ ::= { ⟨*diffs*⟩ } dt + { ⟨*diffs*⟩ } dW & ⟨*expr*⟩
| {⟨*diffs*⟩} dt & ⟨*expr*⟩
| {⟨*diffs*⟩} dW & ⟨*expr*⟩

⟨*diff*⟩ ::= ⟨*ident*⟩ ' = ⟨*expr*⟩

⟨*diffs*⟩ ::= ⟨*diff*⟩
| ⟨*diffs*⟩ , ⟨*diff*⟩

An SDE is given as two sets of "differential terms", which indicate the derivative of variables in a context, one for the drift **`dt`** and another for the noise **`dW`**. We allow users to omit the **`dt`** or **`dW`** terms for completeness - if they do, they are set to 0.

Each SHP file is structured into a number of blocks. We implement four kinds of blocks:

- `Variables`: defines the program variables for the stochastic hybrid program, optionally initialised to a value.
- `Enums`: declares the strings that will be interpreted as members of an enumeration type, rather than as variable names.
- `Constants`: a list of constant definitions.
- `SHP`: the stochastic hybrid program, which is executed in the scope of other blocks.

The syntax for them is the following.

⟨*block*⟩ ::= `Variables` { ⟨*var*⟩$;^{term}$ }
| `Definitions` { ⟨*def*⟩$;^{term}$ }

| Enums { $\langle ident \rangle^{sep}_{,}$ }
| SHP { $\langle SHP \rangle$ }

$\langle var \rangle$ ::= $\langle type \rangle$ $\langle ident \rangle$ = $\langle expr \rangle$;
| $\langle type \rangle$ $\langle ident \rangle$;

$\langle def \rangle$ ::= $\langle ident \rangle$ = $\langle expr \rangle$;

5.1.1 Thermostat example

To illustrate our language, consider the following example, where we simulate the operation of a thermostat by monitoring a temperature variable, and switching between two operation modes depending on the temperature.

```
Constants {
    Heating = 0.3;
    Ambient = -0.1;
}
```

First, we declare our constants. We set fixed values for the heating factor when the thermostat is on, and the ambient temperature decay when it is off.

```
Enums { On, Off }
```

We define two Enum objects, which we will use to keep track of whether the thermostat is on or off.

```
Variables {
    enum state;
    real temp;
}
```

Finally, the variables for our thermostat are the state, which is enum-valued, and the temperature, which is real-valued.

```
SHP {
    temp := {19, 21};
    state := On;
    loop {
        if temp < 19 {
            state := On;
        }
        if temp > 21 {
            state := Off;
        }
        if state = On {
            {temp'=Heating} dt + {temp' = 0.1} dW & temp <= 22;
        } else {
            {temp'=Ambient} dt + {temp' = 0.1} dW & temp >= 18;
        }
    }
}
```

We begin by setting the temperature to a random value between 19 and 21, and beginning in the On state. Then, while the time has not reached 10, we continuously check whether the temperature has fallen outside the acceptable range and set the state accordingly, and depending on the state execute the SDE associated with a thermostat that is either on or off, making use of the constants we defined earlier.

We trace the execution of this thermostat in Figure 5.1. Here, we plot 6 runs of the simulation to demonstrate the variability that can arise within these systems. While the runs slowly diverge over time, we can observe that they always remain within the prescribed temperature bounds, as expected.

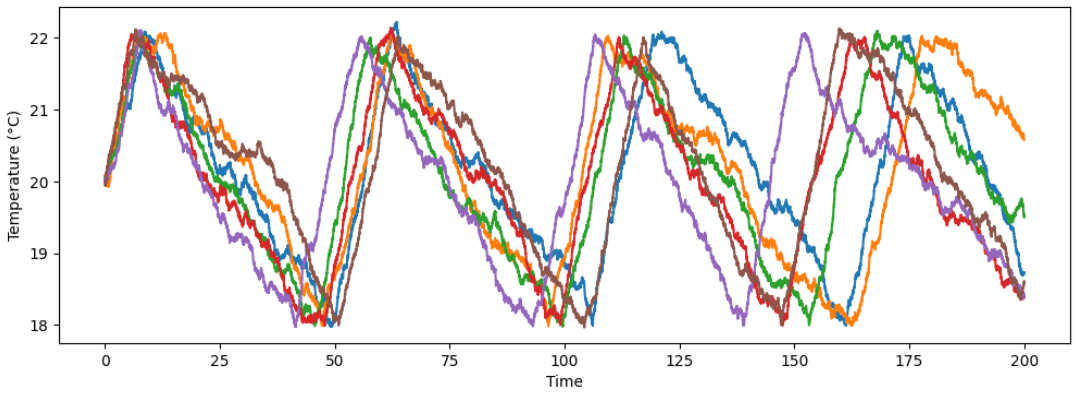


Figure 5.1: Stochastic thermostat traces

5.2 Semantics

We can define the context of our program as a quintuple (Γ, t, I, R, tr), where Γ is a string-value map, t is the current execution time, I is an input stream, R is a stream of random numbers, and tr is a trace. Given some context S, we define the projections to each of their components with $S_\Gamma, S_t, \ldots$.

We give the semantics of our programming language in terms of its traces in Figure 5.2.

The semantics of the language are standard because we conservatively extend an imperative language with probabilistic operators. Assignment `x:=e` updates the value of `x` in state Γ to the evaluation of `e` in context. Random assignment `x := {m,n}` uses a random number from stream R to calculate a uniformly distributed random number between m and n to assign to x. The **`input x`** instruction updates x directly with a value from the input stream. If, while, and sequential composition behave as expected from an imperative language, and loop behaves as **`while true`**, though it will differ in practice as we will halt the loop once the execution time has exceeded the maximum simulation time.

Finally, for our SDE evolution operator `b` **`dt`** $+$ σ **`dW`** `&` `H`, we depend on obtaining a process X that solves the SDE, starting at 0. Then, we execute X until it fails to satisfy H, and update our context to reflect the time it took, the state it reached, and its trace.

$$\frac{}{\langle(\Gamma, t, I, R, tr), \texttt{x := e}\rangle \Downarrow (\Gamma\{x \mapsto [\![e]\!]^{\Gamma}\}, t, I, R, tr)}\ \text{assn}$$

$$\frac{}{\langle(\Gamma, t, I, r\#R, tr), \texttt{x := \{n,m\}}\rangle \Downarrow (\Gamma\{x \mapsto r * (m - n) + n\}, t, I, R, tr)}\ \text{rassn}$$

$$\frac{}{\langle(\Gamma, t, i\#I, R, tr), \texttt{input x}\rangle \Downarrow (\Gamma\{x \mapsto i\}, t, I, R, tr)}\ \text{input}$$

$$\frac{[\![b]\!]^{S_\Gamma} \qquad \langle S, \alpha\rangle \Downarrow S'}{\langle S, \texttt{if b \{}\alpha\texttt{\} else \{}\beta\texttt{\}}\rangle \Downarrow S'}\ \text{cond\_t}$$

$$\frac{\neg[\![b]\!]^{S_\Gamma} \qquad \langle S, \beta\rangle \Downarrow S'}{\langle S, \texttt{if b \{}\alpha\texttt{\} else \{}\beta\texttt{\}}\rangle \Downarrow S'}\ \text{cond\_f}$$

$$\frac{\langle S, \alpha\rangle \Downarrow R \qquad \langle R, \beta\rangle \Downarrow T}{\langle S, (\alpha; \beta)\rangle \Downarrow T}\ \text{seq}$$

$$\frac{[\![b]\!]^{S_\Gamma} \qquad \langle S, \alpha\texttt{; while b \{}\alpha\texttt{\}}\rangle \Downarrow S'}{\langle S, \texttt{while b \{}\alpha\texttt{\}}\rangle \Downarrow S'}\ \text{while\_f}$$

$$\frac{\neg[\![b]\!]^{S_\Gamma}}{\langle S, \texttt{while b \{}\alpha\texttt{\}}\rangle \Downarrow S}\ \text{while\_t}$$

$$\frac{\langle S, \alpha\texttt{; loop \{}\alpha\texttt{\}}\rangle \Downarrow S'}{\langle S, \texttt{loop \{}\alpha\texttt{\}}\rangle \Downarrow S'}\ \text{loop}$$

$$\frac{X \text{ solves } \texttt{b dt + } \sigma \texttt{ dW \& H} \qquad X^R(S_\Gamma) \text{ hits H at time } t'}{\langle(\Gamma, t, I, R, tr), \texttt{b dt + } \sigma \texttt{ dW \& H}\rangle \Downarrow (X^R(S_\Gamma)'_t, t + t', I, R, tr \bullet X^R(S_\Gamma)_{\{0..t'\}})}\ \text{sde}$$

Figure 5.2: Semantics for our SHP simulation language

5.3 Parsing

5.3.1 Haskell for DSLs

Gibbons identifies two approaches to developing domain-specific languages (DSLs), which he calls embedded and standalone [44]. The embedded approach consists of developing a self-contained library within a host language which acts as a language of its own. By contrast, a standalone language involves developing a full language with custom syntax.

Embedding a language in a host language gives us access to all the tooling developed for the host, but the DSL developer becomes bound to its syntax. We choose to develop a standalone language for the implementation flexibility it provides, together with a shallower learning curve and better syntax. This involves writing a parser from scratch, together with implementing the AST manipulation and type-checking.

5.3.2 Parser generator

When parsing a string of text into an abstract syntax tree, there are two main options: parser generators and parser combinators. A parser generator will take a representation of the language, written as a context free grammar, and produce a parser that adheres to these rules. By contrast, parser combinator libraries provide basic parsing functions and a variety of ways of combining them. For this application we choose to use Alex[27] and Happy[45] – respectively the lexer and parser generators for Haskell. We chose this over parser combinators because it leads to more efficient code which can serve as documentation for the structure of the language, and we don't need the flexibility afforded by parser combinators.

We adopt a multi-stage approach for the processing of our language. We have an untyped representation of programs, which is designed to be compatible with the happy parser. Then, typechecking amounts to transforming from the untyped to the typed program representation, and type-correctness becomes encoded in the data structure. This gives us more flexibility in the parsing stage, and allows the use of generic traversals prior to typechecking.

First, we introduce variables and expressions:

```
type Variable = String

data PExpr = PReal Double
           | PBool Bool
           | PEnum String
           | PVar Variable
           | PBop String PExpr PExpr
           | PUop String PExpr
    deriving (Show, Eq, Data)
```

Expressions are at the core of the parser. We define primitive expressions for real numbers, booleans, variables, and enumeration constants. Unary and binary operators are encoded as strings; we opt to not encode them in their own type for easier extensibility.

We derive three typeclasses from our type definition: Show lets us convert instances of the class to a string, Eq provides an equality function, and finally Data allows us to define generic traversals by making use of the SYB library [74]. These generic traversals let us, for example, obtain all variables in a program without having to write the traversal functions for the various data structures contained in a program by hand.

Now, we define the type of programs:

```
data PSHP
  = PAssn Variable PExpr
  | PRandAssn Variable PExpr PExpr
  | PInput Variable
  | PComp PSHP PSHP
  | PWhile PExpr PSHP
  | PLoop PSHP
  | PCond PExpr PSHP PSHP
  | PSkip
  | PSDE [PDiff] [PDiff] PExpr
  deriving (Show, Eq, Data)
```

```
data PDiff = PDiff Variable PExpr
    deriving (Show, Eq, Data)
```

These do not contain any type information yet, and simply encode the structure. Of note here is the `PDiff` constructs, which we use to represent the derivatives of the variables in an SDE evolution. Finally, we give the possible types of our variables:

```
data PSHPType = HPReal | HPBool | HPEnum
    deriving (Show, Eq, Data)
```

We only allow reals, booleans, and enumeration objects. To declare variables, constants, and enumeration members, we introduce a number of blocks:

5.4 Type Checking

We use generalized algebraic data types (GADTs) to represent our typechecked expressions. They are extensions of ADTs which carry type variables which are not necessarily associated with a value of that type.

5.4.1 GADT primer

To understand GADTs, consider the following ADT for linked lists:

```
data List a = Nil | Cons a (List a)
```

Here, we are stating that the type of lists has two constructors: one which takes no parameters, and another which takes two: a value of type `a` and another of type `List a`. Using GADT notation, we can define the above type by giving these types explicitly:

```
data List a where
    Nil :: List a
    Cons :: a -> List a -> List a
```

GADTs extend regular ADTs by letting us specify the return type of the constructor via additional type variables, known as phantom types. This lets us encode additional information about our objects which can be verified by the typechecker. For example, we can define vectors which expose their length to the type system:

```
data Nat = Zero | Succ Nat

data Vector a n where
    VNil :: Vector a Zero
    VCons :: a -> Vector a n -> Vector a (Succ n)
```

Note how this definition only differs from the above list definition in the second type parameter. Now, we can define a safe head function for this type which will cause the compiler to throw a type error if it is ever applied to an empty list:

```
vhd :: Vector a (Succ n) -> a
vhd (VCons x _) = x
```

5.4.2 Representing SHPs with GADTs

Because we can embed type information in our datatypes, the process of typechecking consists simply of converting from the untyped representation we used for parsing to the typed representation, which can only encode type-correct programs. This ensures that all type errors will be restricted to the typechecking phase.

For this purpose, we define the type universe of SHPs as a GADT:

```
data SHPType a where
    SHPReal :: SHPType Double
    SHPBool :: SHPType Bool
    SHPEnum :: SHPType String
    (:->)  :: forall arg res. Typeable res =>
        SHPType arg -> SHPType res -> SHPType (arg -> res)

data Var a where
    Var :: Typeable a => String -> SHPType a -> Var a
```

Here `SHPType a` describes the types that SHP expressions. Each constructor acts as a witness to the type it represents, which enables type-safe manipulation of variables without resorting to ad hoc casting or unchecked operations. The main benefit of a closed universe is the ability to enumerate types fully, which is very useful in the typechecking phase.

The `Var a` type represents a named variable with an associated type witness. Each variable stores a string identifier along with its corresponding `SHPType a` tag. This tagging ensures that the type of the variable is always known at the type level and preserved at runtime.

The use of the `Typeable` constraint is essential for later stages where we need to store heterogeneous variables in a key-value store, for which we will be using the `Dynamic` type. `Typeable` allows us to embed the type in the `Dynamic` value, which can be later recovered through the associated `SHPType` tag.

```
data UOperator a b where
    Neg :: forall a. Num a => UOperator a a
    Not :: UOperator Bool Bool
    Sin :: UOperator Double Double
    Cos :: UOperator Double Double
```

This type encodes unary operators - we also define an interpretation function which maps each constructor to a function of the correct type.

```
evalUOperator :: UOperator a b -> (a -> b)
evalUOperator Neg = negate
evalUOperator Not = not
evalUOperator Sin = sin
evalUOperator Cos = cos
```

Note that the type system enforces that the function we produce match the type specified by the operator. In the same way, we define binary operators and their evaluation function.

```
data BOperator a b c where
    Plus :: forall a. Num a => BOperator a a a
```

```
    Minus :: forall a. Num a => BOperator a a a
    Times :: forall a. Num a => BOperator a a a
    Divide :: forall a. Fractional a => BOperator a a a
    Power :: forall a. Integral a => BOperator a a a

    Eq :: forall a. Eq a => BOperator a a Bool
    Leq :: forall a. Ord a => BOperator a a Bool
    Le :: forall a. Ord a => BOperator a a Bool
    Geq :: forall a. Ord a => BOperator a a Bool
    Ge :: forall a. Ord a => BOperator a a Bool
```

Having defined the building blocks for our typed representation, the extension from the untyped parsing representation to our GADTs amounts to replicating the constructors with extended type information.

```
data Expr a where
    EReal :: Double -> Expr Double
    EBool :: Bool -> Expr Bool
    EEnum :: String -> Expr String
    EVar :: Var a -> Expr a
    EApp :: forall a b. (Typeable a, Typeable b) =>
        Expr (a -> b) -> Expr a -> Expr b
    EBop :: forall a b c. (Typeable a, Typeable b, Typeable c) =>
        BOperator a b c -> Expr (a -> b -> c)
    EUop :: forall a b. (Typeable a, Typeable b) =>
        UOperator a b -> Expr (a -> b)
```

We introduce an application constructor `EApp` to reduce code duplication, and enabling the future addition of user-defined functions. Outside of this, everything mirrors the untyped representation.

We define a type `AExpr` which hides the type information in order to pass it around without the need for polymorphism - objects of this type carry around an `SHPType a` which lets us reconstruct the type when needed.

```
data AExpr = forall a. Typeable a => Expr a ::: SHPType a
```

Typechecking the expressions amounts to constructing values of type `AExpr`. The final result has type `Either String AExpr`, which lets us account for type mismatch errors. We use the `Refl` constructor to prove to the compiler that the types of two expressions are equal - if this construction fails, we capture it by returning a `Left` containing an error message.

```
typecheckPExpr :: Env -> PExpr -> Either String AExpr
typecheckPExpr env e =
  case e of
    PReal r -> Right $ EReal r ::: SHPReal
    PBool b -> Right $ EBool b ::: SHPBool
    PEnum e -> Right $ EEnum e ::: SHPEnum
```

Converting primitive values is simple: we can just return the corresponding typed expression constructor with the appropriate `SHPType` witness. No type errors can occur at this stage, so we wrap our values in `Right`.

```
PVar v -> do
  ASHPType typ <- maybeToEither
    (env !? v)
    [i|Error - undefined variable in expression: #{v}|]
  return $ EVar (Var v typ) ::: typ
```

To typecheck variables, we look up their assigned type from the environment, and use it to construct the `EVar` expression. If the variable is undefined in the environment we return a `Left` value with an error message instead.

```
PUop op f -> do
  f_typed ::: f_t <- typecheckPExpr env f
  op_typed ::: (t1 :-> ret_typ) <- typeUOp op f_t
  Refl <- testEqualityEither "" f_t t1
  return $ EApp op_typed f_typed ::: ret_typ
```

To typecheck operator expressions, we have to compare the type the operator expects with the type it is supplied. To do this, we construct the following value by pattern-matching both types: `Refl :: a :~: a`. This proves to the compiler that the two types are equal, which lets us satisfy the conditions necessary to construct an `EApp` value.

```
PBOp op f g -> do
   f_typed ::: f_t <- typecheckPExpr env f
   g_typed ::: g_t <- typecheckPExpr env g
   op_typed ::: (t1 :-> t2 :-> ret_typ) <- typeBOp op f_t g_t
   Refl <- testEqualityEither "" f_t t1
   Refl <- testEqualityEither "" g_t t2
   return $ EApp (EApp op_typed f_typed) g_typed ::: ret_typ
```

Similarly for two-argument expressions, we have to do two tests, but the result is the same.

Finally, we can use these expressions to parse our SHP statements. The type information that our expressions carry is used to enforce that variables are only assigned to expressions whose types match, and to ensure that control flow conditions return a boolean.

```
data Diff = Diff (Var Double) (Expr Double)
 deriving Show

data SHP where
    Assn :: forall a. Var a -> Expr a -> SHP
    RandAssn :: Var Double -> Expr Double -> Expr Double -> SHP
    Input :: forall a. Var a -> SHP
    Composition :: SHP -> SHP -> SHP
    While :: Expr Bool -> SHP -> SHP
    Skip :: SHP
    Cond :: Expr Bool -> SHP -> SHP -> SHP
    SDE :: [Diff] -> [Diff] -> Expr Bool -> SHP
```

Values of SHP type are straightforward to construct now that we have expression typechecking. We have to ensure that the expressions contained within have the appropriate types, and that the referenced variables are defined and correctly typed.

```
typecheckPSHP env shp = case shp of
  PAssn var exp -> do
    ASHPType var_typ <- maybeToEither (env !? var) "Type error"
    exp_typed ::: typ <- typecheckPExpr env exp
    Refl <- testEqualityEither "Type error" var_typ typ
    return $ Assn (Var var typ) exp_typed

  PCond exp left right -> do
    exp_typed ::: SHPBool <- typecheckPExpr env exp
    left_shp <- typecheckPSHP env left
    right_shp <- typecheckPSHP env right
    return $ Cond exp_typed left_shp right_shp
```

Above, we show a sample of the typechecking for SHPs. The process is very similar to how we typecheck expressions — for assignment, we have to prove that the variable has the same type as the expression we are assigning via the `Refl` constructor, and for conditionals we have to ensure that the condition has a boolean type.

Now that we have type-correct SHP programs, we can implement execution without needing to worry about type errors.

5.5 Simulation

For execution, we need to keep track of the state. We store values in a key-value map, where the `Dynamic` type lets us store values with different types in the same map.

```
type Store = Map String Dynamic
```

Expressions are evaluated in the context of the store. Variables with type `typ` extract their value from the state, and if the type of the stored dynamic value does not match we throw an error. However, we perform static checks of the program code to ensure this never happens with a well-formed store.

```
evalExpr :: (String -> Dynamic) -> Expr a -> a
evalExpr ctx expr =
    case expr of
      EReal r -> r
      EVar (Var x typ) -> case fromDynamic (ctx x) of
                          Just v -> v
                          Nothing -> error "Type error"
      EBool b -> b
      EBop op -> evalBOperator op
      EUop op -> evalUOperator op
      EApp f e -> evalExpr ctx f (evalExpr ctx e)
      EEnum x -> x
```

With the ability to evaluate expressions, we proceed to define SHP executions using the RWS (reader, writer, state) monad. This monad encapsulates common patterns that arise in defining imperative programs by combining three different monads. The reader monad threads immutable parameters through program execution, the writer monad collates an append-only list of logs that the program can emit, and the state monad contains mutable state that can be read and written by

the program. In this case, the reader carries simulation parameters and variable types, which are known statically; the writer collects the points visited during differential equation evolution and the state contains our mutable store and time.

Our execution type is

```
type Execution m a = PrimMonad m =>
    RWST (Config m) (Endo [Vector Double]) State m a
```

Here, RWST is the RWS *transformer*, which adds RWS functionality to any monad. In our case, we require that the monad at the bottom of the stack be a `PrimMonad` - a class of monad that can perform unsafe operations, which we need for generating random values.

We can understand an object with type `Execution m a` as a program which runs in monad `m` which supports random value generation (generally `IO` or `ST`), takes as input a `Config m` object, writes logs to `Endo [Vector Double]`, has access to a state of type `State`, and returns a value of type `a`.

Now, we can interpret our typed SHP representation as an `Execution m a` object, which can finally be executed by supplying the appropriate state parameters.

```
runSHP ::
     TracingMode w
  -> (forall a. String -> SHPType a -> Config m -> m (Expr a))
  -> SHP
  -> Execution m w ()
```

TracingMode indicates the type of the trace - we define utilities for tracing the full path that the SHP takes, as well as keeping track of minimum value or whether a predicate is always satisfied. The next function in the type lets the execution obtain user input as a string, which is converted to an expression. As before, we define execution as a case match on all the SHP constructors.

```
runSHP traceMode inputFun shp =
  case shp of
    Assn (Var v typ) exp -> do
      disc <- use store
      store . at v ?= toDyn (evalExprStore disc exp)
```

To interact with the various parts of the state, we use lenses. `use` accesses the component identified by a lens within the state, and `at` lets us further index a map in order to update it with `(?=)`. Since our map type is heterogeneous, we use `toDyn` to map all expressions into the `Dynamic` type before storing them.

```
RandAssn (Var v typ) low high -> do
  g <- asks gen
  disc <- use store
  val <- MWC.uniformR (evalExprStore disc lo,
                       evalExprStore disc hi) g
  store . at v ?= toDyn val
```

Similarly for random assignment, we update the state using lenses. To obtain the random value, we first get the random number seed from the reader, and use it to produce a uniformly distributed value between the values that the supplied expressions evaluate to.

```
Input (Var v typ) -> do
  config <- ask
  enums <- asks enums
  disc <- use store
  expression <- lift $ inputFun v typ config
  store . at v ?= toDyn (evalExprStore disc expression)
```

For input, we resort to the supplied input function to obtain an expression from the user, which we can then write to the state.

```
Composition n m -> runSHP traceMode inputFun n
                >> runSHP traceMode inputFun m
Cond p n m -> do
  disc <- use store
  if evalExprStore disc p
    then runSHP traceMode inputFun n
    else runSHP traceMode inputFun m
While p m -> do
  disc <- use store
  when (evalExprStore disc p) $
    runSHP traceMode inputFun m >>
    runSHP traceMode inputFun shp
```

Finally, the combinators are defined recursively. SHS `Composition` becomes monadic composition (`>>`), `While` loops execute their body and recursively call themselves until the condition becomes false, and `Cond` expressions are lifted to Haskell's conditional.

5.5.1 SDE simulation

We use Euler-Maruyama integration for simulating SDEs. This is an extension of the Euler method from ODEs to the stochastic domain. To solve an SDE $\mathrm{d}X_t = b(X_t, t)\,\mathrm{d}t + \sigma(X_t, t)\,\mathrm{d}W$ with a chosen timestep size δ, let $\tau_t = t\delta$ be the time value at timestep t, and $W_n\,\mathcal{N}(0, \delta)$ be a sequence of i.i.d. vectors with dimensions matching our SDE. Then, given some initial conditions $X_0 = I$, we can calculate our solution iteratively:

$$X_{t+1} = X_t + b(X_t, \tau_t)\delta + \sigma(X_t, \tau_t)\Delta W_n$$

The types for our execution are the following:

```
type Flow = Vector Double -> Double -> Vector Double
type Noise = Vector Double -> Double -> Vector (Vector Double)
```

In our simulation, we define a helper function to obtain ΔW_t, with parameters `dt` (timestep), `k` (vector length) and gen (random generation seed).

```
dWiener ::
  (PrimMonad m) =>
  Double -> -- step size
  Int -> -- length of vector
  Gen (PrimState m) -> -- random seed
```

```
  m (Vector Double)
dWiener dt k gen = generateM k (\_ -> normal 0 (sqrt dt) gen)
```

We generate random numbers using `normal 0 (sqrt d) gen`, which is an action in monad `m` that draws a value from distribution $\mathcal{N}(0, dt)$. Then, `generateM k` runs our monadic action `k` times and collects the results into a vector.

We use this to implement the core Euler-Maruyama step:

```
eulerMaruyamaStep :: PrimMonad m
  => Flow -> Flow -> Vector Double -> Double -> Double
  -> Gen (PrimState m) -> m (Vector Double)
eulerMaruyamaStep flow noise state t dt gen = do
    w_n <- dWiener dt (length state) gen
    return $ zipWith (+) state
            (zipWith (+) (map (*dt) (flow state t))
            (zipWith (*) (noise state t) w_n))
```

Each step, we sample the random vector and store the result in variable `w_n`. Then, the $\sigma\,\mathrm{d}W$ term is `zipWith (*) (noise state t) w_n`, the $b\,\mathrm{d}t$ term is `map (*dt) (flow state t)`, and we can calculate the updated state by adding the three in a pointwise manner using `zipWith (+)`.

An important consideration for executing SDE simulation in the context of the program is how its execution should be traced. There are a few different traces we could care about, depending on the situation - the natural choice is to store the state at each timestep, but we can also trace the value of an expression, or simply keep track of the maximal value. In order to add this flexibility, we define a type of tracing modes, where each constructor encodes the type of its associated trace:

```
type Trace m w = Vector Double -> Execution m w ()

data TracingMode w where
    RawTrace :: TracingMode (Endo [Vector Double])
    FullTrace :: Expr Double -> TracingMode (Endo [Double])
    MinTrace :: Expr Double -> TracingMode (Maybe (Min Double))
    MaxTrace :: Expr Double -> TracingMode (Maybe (Max Double))
    AnyTrace :: Expr Bool -> TracingMode Any
    AllTrace :: Expr Bool -> TracingMode All
    NoTrace :: TracingMode ()

data SomeTracingMode = forall a. Monoid a =>
     SomeTracingMode (TracingMode a)
```

A tracing function is one that writes to the trace without modifying the state. To define the various tracing behaviours, we defer to the monoid instances of the carrier types to perform the appending for us. For instance, we use the `Endo` monoid to store full traces efficiently - appending in this monoid has time complexity O(1), as opposed to O(n) for the standard list monoid.

```
eulerMaruyamaTraceDiag ::
  (PrimMonad m, Monoid w) =>
  Tracing.Trace m w ->
  Flow ->
  Flow ->
```

```
  (Double -> Vector Double -> Bool) ->
  Vector Double ->
  Execution m w (Vector Double)
eulerMaruyamaTraceDiag traceF flow noise boundary state =
  do
    t <- use time
    Config {maxTime = maxTime, dt = dt, gen = gen} <- ask
    eulerMaruyamaLoop maxTime t dt gen state
 where
  eulerMaruyamaLoop maxTime t dt gen st =
    if maxTime <= t || not (boundary t st)
      then do
        time .= t -- we only update time in RWS at the end
        return state
      else do
        nextState <- eulerMaruyamaStepDiag flow noise st t dt gen
        traceF nextState
        eulerMaruyamaLoop maxTime (t + dt) dt gen nextState
```

Our parameters are the tracing mode, the flow and noise components of the SDE, the boundary, and finally initial state. We return a program execution which itself returns the final state of the SDE.

The function proceeds in three stages. First, we obtain the parameters we need from the environment: initial time, maximum simulation time, timestep, and random generator. Then, we execute the step function in a tight loop, tracing the state with every step. Finally, once the SDE hits the boundary or the maximum simulation time is reached, we write the new time back to the state monad and return the state of the SDE.

With this, we can implement the evaluation function for the **SDE** constructor:

```
SDE drift noise boundary -> do
      state <- use store
      let flowVars = map getDiffVar drift
      let flowVarMap = fromList $ zip flowVars [0 ..]
      let eval = evalExprVector state flowVarMap
                   :: Expr a -> Vector Double -> a
      let flowF v t = fromList $ map (flip eval v . getDiffExpr) drift
      let noiseF v t = V.fromList $ map (flip eval v . getDiffExpr) noise
      let evalBoundary t = evalExprVector state flowVarMap boundary
      new_state <- eulerMaruyamaTraceDiag (runTrace traceMode flowVarMap)
         flowF noiseF evalBoundary (storeToVec flowVars state)
      store .= vecToStore flowVars state new_state
```

5.6 Usage examples

In this section, we will simulate some example SHPs, mirroring those presented in Chapter 3. For this, we use the command-line interface which we developed for this simulation tool. The command-line interface for the tool provides several options for controlling the simulation, including the ability to specify the input file, output file, and the simulation mode. The available

options can be seen by running the tool with the `-help` flag:

```
$ stochastic-hybrid-sim --help
Usage: stochastic-hybrid-sim (FILENAME | --prog PROGRAM | --stdin)
                  [-o FILENAME]
                  [--parse | --typecheck | --simulate | --benchmark]
                  [--rawtrace | --notrace | --fulltrace ARG |
                   --mintrace ARG | --maxtrace ARG]
                  [-t FLOAT] [--timestep FLOAT] [-n INT]
```

5.6.1 Water tank

We begin by implementing the water tank example in our simulator, as displayed in Figure 5.3. This mirrors the IsaVODEs system defined in Chapter 3, though we have to provide concrete values for all constants.

In order to execute the program and produce a trace, we can run the following command:

```
$ stochastic-hybrid-sim water_tank.shp -o tank_trace.txt
```

This will execute the water tank SHP and save the trace to `tank_trace.txt`. We can use this data to produce a plot of the execution, as shown in Figure 5.4.

We can choose the format in which traces are collected, which is useful for producing statistics over multiple runs of an SHP. This is aided by option `-n`, which informs how many times a program is to be run. For example, we can gather statistics on the maximum water level observed across multiple simulation runs with the following command:

```
$ stochastic-hybrid-sim water_tank.shp --maxtrace l -n 100 \
    -o tank_trace.txt
```

The results of this are shown in Figure 5.4.

The water level generally stays within the prescribed range, though on most of the traces it does exceed the target maximum level m. This is due to the added variability in the SDE not being accounted for in the flow rate selection. Note, however, that despite this it rarely exceeds the maximum by much, and so a more conservative flow rate selection function would be effective in preventing overflows.

5.6.2 Planar flight

We can also reproduce the planar flight controller example, as shown in Figure 5.5. This SHP models the interaction between our own aircraft and an intruding aircraft, where our controller must decide on an angular velocity ω to avoid collision based on uncertain observations of the intruder's position.

The program consists of a control loop where the intruder's position (x, y) and angle θ are observed with some uncertainty, and the controller sets ω accordingly. The continuous dynamics are then simulated according to the SDEs that describe the relative motion of the two aircraft.

```
Constants {
    epsilon = 1;
    m = 5;
    U = 0.1;
}

Variables {
    real f, l, lu, c, c_old;
}

SHP {
    l := {0, m};
    c := 0;
    loop {
        lu := {l-U, l+U};
        f := {-1, m - (lu + U) / epsilon};
        c_old := c;

        { l' = f, c' = 1 } dt + {l' = 0.1, c'=0} dW
            & 0 <= l /\ c <= c_old + epsilon;
    }
}
```

Figure 5.3: Uncertain water tank SHS

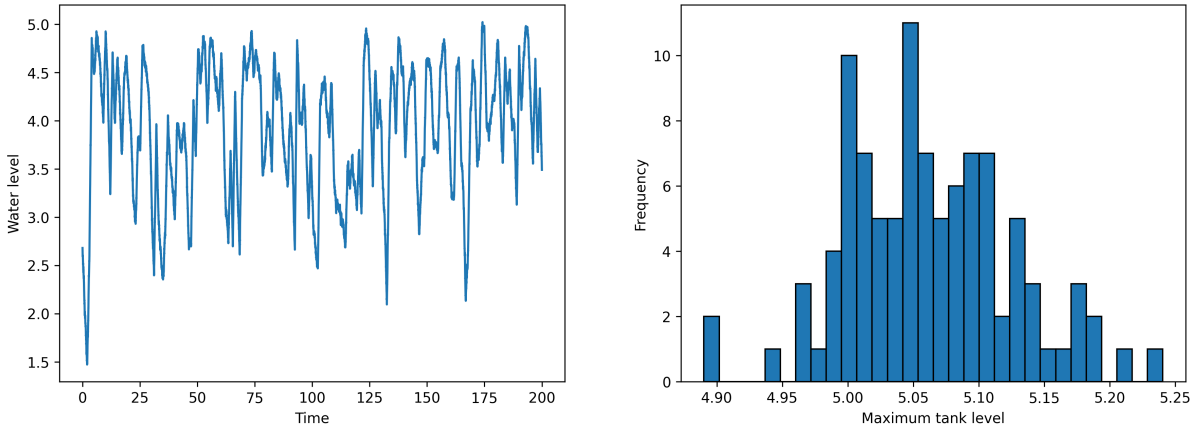
We plot a few runs of this SHP in Figure 5.6. We can observe that both the $\omega = 1$ and $\omega = 0$ paths are represented, and that indeed there are no collisions or close calls. We can also gather statistics on the minimum distance to the intruder observed across multiple runs using the `-mintrace` option, which can help in identifying potential safety violations.

5.7 Summary

In this chapter we have introduced a domain-specific language for reactive stochastic hybrid systems, along with a corresponding simulator and preliminary verification infrastructure. The language is inspired by Platzer's stochastic hybrid programs and supports a model–simulate–verify workflow, where simulation provides intuition and debugging support, and formal verification in Isabelle delivers rigorous guarantees.

In our implementation, we use Haskell to define a standalone language with a strongly typed abstract syntax. This choice allows clear separation between the surface syntax and execution semantics, and facilitates future translation to other back-ends, such as theorem provers, code generators, or compilation to machine code.

We were able to leverage a number of advanced features of Haskell in order to make our code simpler, more correct, and more extensible. We implemented the program analysis stages using one datatype for each stage, which let us apply generic program transformations before typechecking. We used GADTs to carry the type information for the programs, and make use of the typechecker to ensure correctness of the implementation. Using the reader-writer-state monad let us write the



(a) Water tank execution trace

(b) Histogram of maximum observed water level

Figure 5.4: Water tank simulation results

semantics of our program out very clearly while retaining extensibility, for instance in the ability to select the tracing monoid at runtime.

```
Constants {
  v_own = 1; // Own velocity
  v_int = 1; // Intruder velocity
  delta = 0.1; // Disturbance
}

Variables {
    real x;
    real y;
    real theta;
    real w;
}

SHP {
    // ctrl
    x := {0,5};
    y := {0,5};
    theta := {0,3};
    if (v_int * sin(theta) * x -
         (v_int * cos(theta) - v_own) * y
        > v_own + v_int) {
        w := 0;
    } else {
        w := 1;
    }
    {
        x' = v_int * cos(theta) - v_own + w*y,
        y' = v_int * sin(theta) - w*x,
        theta' = w,
        w' = 0
    } dt +
    { x'=delta, y'=delta, theta'=delta, w'=0 } dW
    & true;
}
```

Figure 5.5: Flight controller with uncertainty

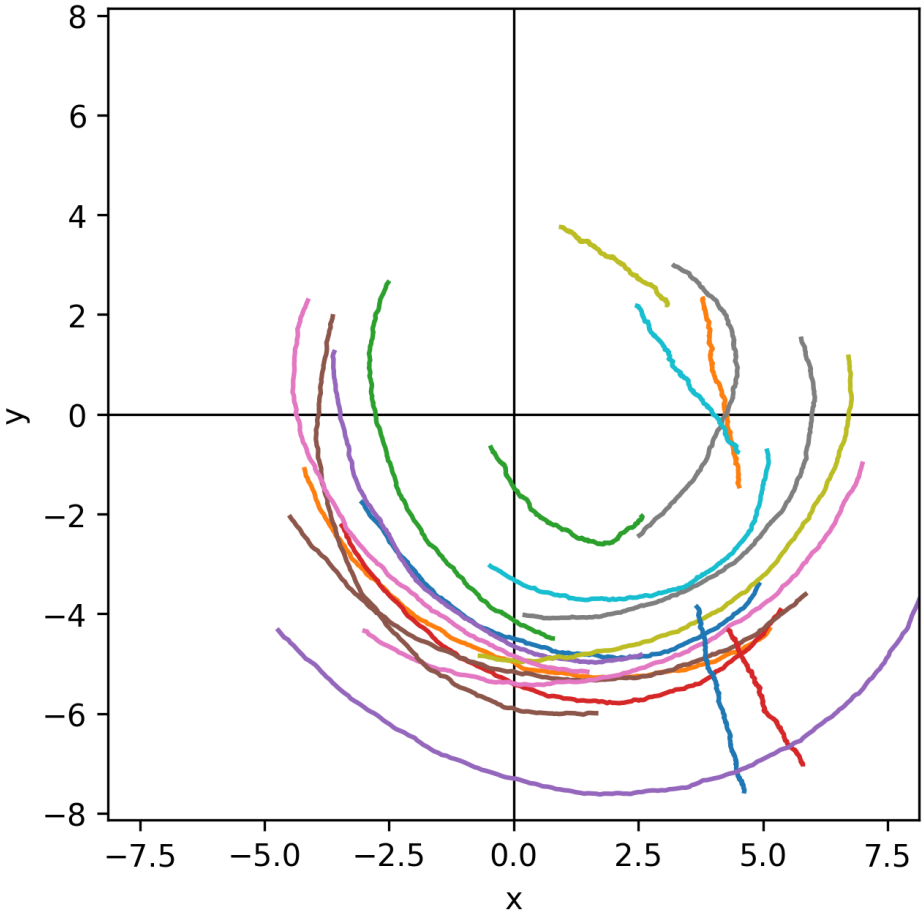


Figure 5.6: Planar flight simulation results

Chapter 6

Towards Brownian Motion in Isabelle/HOL

6.1 Introduction

We attempt to mechanise Brownian motion in Isabelle/HOL. This is an important continuous-time stochastic process which is used to give semantics to stochastic differential equations. This in turn describes the dynamics of SHS, forming the foundation of our proposed logic. Brownian motion is a central object in probability theory — it models the random motion of particles in a fluid, but it can be applied to model a wide range of phenomena, such as thermal properties [66] and stock prices [91].

In this chapter, we mechanise stochastic processes in Isabelle. This includes formalising a number of properties which allow us to define the properties that characterise Brownian motion. In addition, we prove the Kolmogorov-Chentsov theorem, which lets us construct processes with almost-surely continuous paths. Finally, we develop the theory of transition kernels, which let us characterise and construct stochastic processes with desirable properties. All of our results have been formalised in Isabelle/HOL[1], and mainly follow Klenke's "Probability theory: A Comprehensive Course" text; one of the key textbooks in probability theory [71]. Unfortunately, we were unable to complete the full construction of Brownian motion, due in part to an error in a textbook proof, but our development remains a significant contribution.

Our main contribution is not the theory of stochastic processes itself, but instead the engineering effort involved in formalising it. The mechanisation process involves much more work than simply reiterating the results: pen-and-paper proofs are often laden with omissions, and their mechanisation strengthens their argument by spelling out every detail of the proof. This often involves creativity, especially in the case where the textbook definitions or problem statements have to be altered to fit within the logic, or enable better automation.

We also present our insights into the mechanisation of non-trivial continuous mathematics, and how it can be structured to best leverage proof automation. In our development, we benefitted from Isabelle's proof automation — both automated theorem proving with Sledgehammer [14], but also more traditional and bespoke proof methods.

These results have not been mechanised before to the authors' knowledge, and they are comprised

[1]The results are available at `https://github.com/cplaursen/Brownian_Motion`

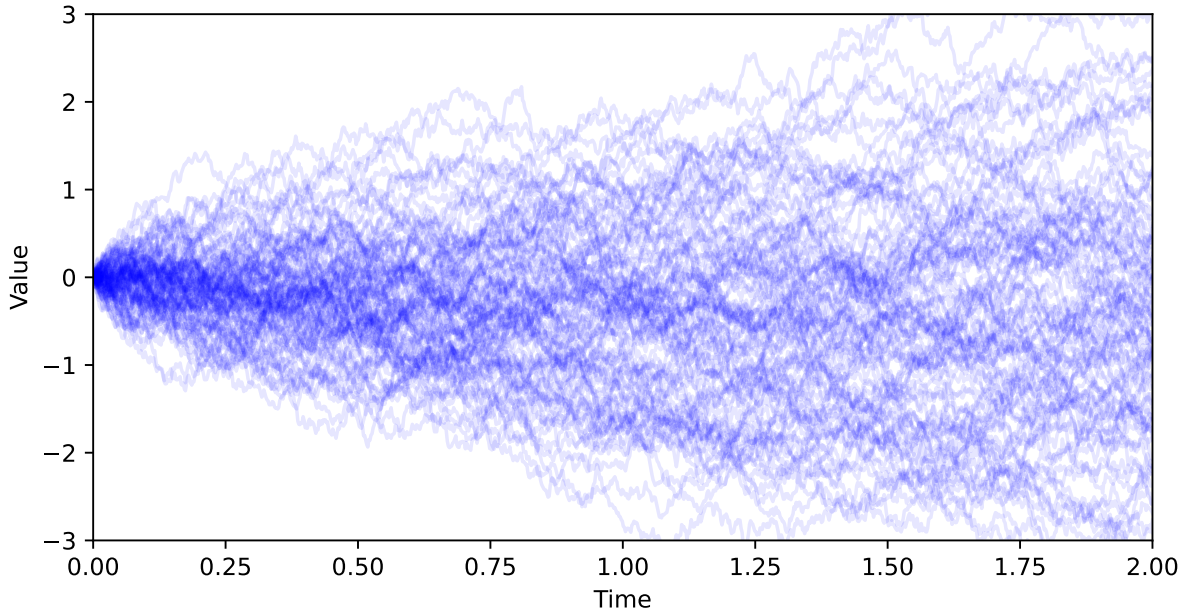


Figure 6.1: Paths of Brownian motion

of some deep, yet fundamental, results in stochastic process theory. Our mechanisation has required us to formalise a number of smaller novel technical results, particularly the iterated kernel product and a constructive definition for intervals of dyadic rationals. The results are also applicable outside of constructing Brownian motion; transition kernels can be used to give semantics to probabilistic programs [56], and the Kolmogorov-Chentsov theorem is applicable for constructing an important class of stochastic processes.

6.2 A probability theory primer

Our development is based on the Isabelle/HOL analysis and probability libraries, namely HOL-Analysis and HOL-Probability [59, 57]. To provide some context, we will recount some notions from probability and measure theory, together with how they are implemented in Isabelle.

Measure spaces The central objects of measure theory are measure spaces: triples $(\Omega, \mathcal{F}, \mu)$ consisting of the space Ω, the measurable sets $\mathcal{F}$ and a measure function $\mu : \mathcal{F} \to \mathbb{R}$. The measurable sets form a σ-algebra: a family of sets closed under complements w.r.t. Ω and countable unions, and which contains Ω. Together, $(\Omega, \mathcal{F})$ is called a measurable space. μ is only defined on the measurable sets, and must satisfy $\mu(A) \geq 0$ and countable additivity:

$$\sum^{i} \mu(A_i) = \mu\left(\bigcup^{i} A_i\right)$$

In Isabelle, measures are objects of type `'a measure`, which are internally composed of the above triple. To access the components of the measure objects, we use projection functions `sets`, `space` and `emeasure` respectively. The measure function is `ennreal` (extended non-negative real)–valued, as we need to allow infinity, but we can also use projection `measure` for the real-valued version.

Whenever we apply the measure to some set A, we need to show that A is measurable - if it is not, the measure defaults to 0. Therefore, goals of the form `A ∈ sets M` are frequent, and we can use tactic `measurable` to simplify or discharge them.

A measure is called σ-finite if there exists some countable set $\mathcal{A} \subseteq \mathcal{F}$ with $\bigcup \mathcal{A} = \Omega$, such that for any $A \in \mathcal{A}$, $\mu(A) < \infty$. A measure is finite if $\mu(\Omega) < \infty$, and finally it is a probability space if $\mu(\Omega) = 1$.

In Isabelle, these are encoded using locales — bundles of fixed variables and assumptions. Locale `sigma_finite_measure` fixes a measure M and assumes the existence of $\mathcal{A}$. `finite_measure` extends this with the finiteness assumption, and finally `prob_space` does the same with assumption `emeasure M (space M) = 1`.

Almost everywhere We state that a property P holds almost everywhere (a.e.), or equivalently almost surely (a.s.) in a probability space, if $\{x \mid \neg P(x)\}$ is a subset of a measurable set with measure 0 (a null set).

In Isabelle, we use a binder to define almost-everywhere properties: `AE x in M. P x` means that the set of x that do not satisfy P are a subset of a null set.

Measurable functions and random variables Given measurable spaces $(X, \mathcal{F})$ and $(Y, \mathcal{G})$, $f : X \to Y$ is $\mathcal{F} - \mathcal{G}$ measurable if for any $S \in \mathcal{G}$, $\{x \mid f(x) \in S\} \in \mathcal{F}$.

In Isabelle, we write $M \to_M N$ for the set of measurable functions between measures M and N. It is a very common prerequisite for any lemmas on probability spaces that any functions be measurable, and we can again use the `measurable` tactic to discharge or simplify these goals.

When the source is a probability space, measurable functions are known as random variables. When typesetting random variables, it is standard practice to omit the points of the source measure: we write $\{X \neq Y\}$ instead of $\{\omega \in \Omega. X(\omega) \neq Y(\omega)\}$, though we often have to defer to the latter form in the Isabelle text.

Given some $M \to_M N$ measurable function X, its distribution is the push-forward measure it induces on its target, defined by $\mathcal{D}_X(S) = \mu(\{\omega \in \Omega \mid X(\omega) \in S\})$. In Isabelle, this is given by `distr M N X`; this function treats N as simply a measurable space, returning a measure space with the same space and sets as N, but equipped with the distribution of X as its new measure.

Independent variables Exclusive to probability is the notion of independence: two families of events $\mathcal{A}$ and $\mathcal{B}$ are independent with respect to probability measure μ if $\forall A \in \mathcal{A}, B \in \mathcal{B}. \mu(A) \cdot \mu(B) = \mu(A \cap B)$. Then, random variables are independent if their preimages form independent families.

In Isabelle, we have predicates `indep_sets` and `indep_vars` which characterise the corresponding concepts.

Product measure The product measure $\prod_{i \in I}(\Omega_i, \mathcal{F}_i, \mu_i)$ is generated by sets of the form $\{A \mid \forall i \in I. A_i \in \mathcal{F}_i\}$. The variable X_J for $J \subseteq I$ is the projection from the product space with index I to index J - this allows us to characterise product spaces with infinite index sets by considering their finite-dimensional distributions.

In Isabelle, the product measure is given by function `PiM I M`. Elements of this space are domain-restricted functions — the Isabelle type system does not allow us to directly specify products with

a given index set, so instead we use functions which are set to `undefined` on elements outside the index set.

Measure integral Finally, we can take the integral with respect to a measure. Isabelle defines the Bochner integral, which lets us integrate over functions into any Banach space; this extends the more commonly seen Lebesgue integral.

In Isabelle we write $\int$`x. f x ∂M` for the integral of `f x` with respect to measure `M`. If f is `ennreal`-valued, we can use the nonnegative integral instead, which is better for proof automation when applicable: $\int^+$`x. f x ∂M`.

6.3 Formalising Stochastic Processes

This section presents our formalisation of stochastic processes in Isabelle/HOL, which builds on the background theory presented in Section 6.2. We introduce stochastic processes, modifications and indistinguishability, and convergence of processes in measure, all of which are necessary to prove the existence of Brownian motion.

A stochastic process is an indexed family of random variables from some probability space $\{X_t \mid t \in I\}$ for random variables X_t and index set I. In many instances I represents time; common index sets include the natural numbers or the real interval $[0, \infty)$.

An example of a stochastic process is the Bernoulli process, a discrete time stochastic process with $I = \mathbb{N}$, which has two outcomes, 0 and 1, and can be used to model a sequence of coin tosses. Another example is a one dimensional simple random walk, where in each time-step a position is incremented or decremented by 1, based on the outcome of the Bernoulli process. We define stochastic processes in Isabelle using a locale:

Definition 6.1 (Stochastic process locale)**.**

```
locale stochastic_process = prob_space +
  fixes M' :: "'b measure"
    and I :: "'t set"
    and X :: "'t ⇒ 'a ⇒ 'b"
  assumes random_process[measurable]: "⋀i. random_variable M' (X i)"
```

Locales allow us to implement algebraic structures, including their carrier sets, operations and axioms. A locale is a named set of fixed variables and assumptions [48]. One can prove theorems within a locale relative to the locale assumptions and fixed variables, whilst making use of other theorems in the locale.

Our stochastic process locale extends `prob_space`, another locale which fixes `M :: 'a measure`, i.e. a measure space taking values of type `'a`, and assumes it is a probability space. To these assumptions, we add three fixed variables: M' is the target measurable space of our process, I is the index set, and X is the process itself.

We then assume that X_i is a random variable into space M' for all i. Here, $\bigwedge$ is a *meta-quantifier*. It has the same semantics as $\forall$, but it is more convenient for proof automation as its operation is built into Isabelle, rather than requiring introduction and elimination laws.

We add the `[measurable]` theorem attribute to our assumption, which makes it available to proof method `measurable`. This proof method automates reasoning related to proving that functions

or sets are measurable; we improve its success rate by tagging relevant theorems so that they may be automatically used, rather than having to supply them every time.

Locale definitions give us a predicate that encodes the locale assumptions. We can use this predicate to define the type of stochastic processes:

Definition 6.2 (Stochastic process type)**.** A stochastic process is a quadruple (M, M', I, X) where X is a process with index set I, which is $M - M'$-measurable for any choice of index.

```
typedef ('t, 'a, 'b) stochastic_process =
 "{(M :: 'a measure, M' :: 'b measure, I :: 't set, X :: 't ⇒ 'a ⇒ 'b).
   stochastic_process M M' X}"
```

Here we introduce a type of objects that satisfy the `stochastic_process` predicate defined by the above locale. Types bring the advantage of proof automation. By defining a new type, the measurability assumption becomes built into the stochastic process objects, and no longer needs to be stated as an assumption in every theorem.

An example of a stochastic process is the simple random walk, which at each timestep will increase or decrease its value by 1 with uniform probability. To define this, we can use a stream space as our sample space, which contains all streams (infinite lists) of a given type. We define the action of our random walk R on the stream space for booleans:

$$R_n(\omega) \triangleq \sum_{j \in \{1..n\}} \text{if } \omega!!j \text{ then } 1 \text{ else } -1$$

Here, $\omega!!j$ is a function that takes the jth element of stream ω. Then, at each timestep, our stochastic process will increase by one if the corresponding stream element is `True`, and decrease otherwise.

Locales define a predicate that characterises their fixed variables and assumptions. We use the predicate defined by the stochastic process locale to define the type of stochastic processes as a quadruple of source measure, target measure, index set and family of random variables, which satisfy the locale predicate.

We define the distributions of the process: given some $J \subseteq I$, we define a product measure where the distribution at each index is given by the process:

Definition 6.3 (Stochastic process distributions)**.** For some stochastic process (X, μ, I), its finite-dimensional distribution for some finite $T \subseteq I$ are given by the joint distribution of all $X_t,\ t \in T$.

```
lift_definition distributions :: "('t, 'a, 'b) stochastic_process ⇒ 't
set ⇒ ('t ⇒ 'b) measure"
is "λ(M, M', _, X) T. (Π_M t∈T. distr M M' (X t))"
```

Here, `lift_definition` lets us define a function on some type in terms of its underlying structure.

Many properties of stochastic processes, including some necessary for defining Brownian motion, are given in terms of their increments — expressions of the form $X_t - X_{t-1}$. In textbooks, collections of increments are often defined by obtaining some $t_1, t_2, \ldots, t_n$ such that $t_1 \leq t_2 \leq \cdots \leq t_n$. We define one such construction below.

Definition 6.4 (Independent increments)**.** A stochastic process has independent increments if for all $t_0, \ldots, t_n$ with $t_0 < t_1 < \cdots < t_n$, all $(X_{t_i} - X_{t_{i-1}})_{i=1\ldots n}$ are mutually independent.

```
lift_definition indep_increments :: "('t :: linorder, 'a, 'b :: minus)
stochastic_process ⇒ bool" is
"λ(M, M', I, X).
  (∀l. set l ⊆ I ∧ sorted l ∧ length l ≥ 2 ⟶
     prob_space.indep_vars M (λ_. M') (λk v. X (l!k) v - X (l!(k-1)) v)
{1..<length l})" .
```

In order to define this in Isabelle, we use sorted lists of indices. This involves predicate `sorted`, which establishes that a list is in sorted order. We define our collection of independent increments in terms of a sorted index list, and establish that a process has independent increments if any choice of index list leads to an independent family of increments.

Brownian motion, or the Wiener process, is a process W with $W_0 = 0$, independent normally distributed increments, and almost surely continuous paths. It serves as a fundamental model for real-world systems with uncertainty, naturally arising in various types of noise.

Its independent increments ensure that past events do not influence future outcomes. The normal distribution of increments is supported by the central limit theorem, which states that the sum of many independent random variables tends toward a normal distribution as the number of variables grows large. This property justifies modelling cumulative small effects with normally distributed increments. Finally, the continuous paths make the process suitable for modelling physical phenomena that cannot exhibit sudden jumps or discontinuities.

In Isabelle, we define Brownian motion using a locale:

Definition 6.5 (Brownian motion)**.** A process W is a Brownian motion if it starts at 0, has independent, normally distributed increments, and almost-surely continuous paths.

```
locale brownian_motion =
fixes W :: "(real, 'a, real) stochastic_process"
assumes init_0[simp]: "𝒫(x in proc_source W. W 0 x = 0) = 1"
    and indep_increments: "indep_increments W"
    and normal_increments: "⋀s t. 0 ≤ s ∧ s < t ⟹
            distributed (proc_source W) borel (λv. W t v - W s v)
             (normal_density 0 (sqrt (t-s)))"
    and AE_continuous: "AE x in proc_source W. continuous_on {0..} (λt.
W t x)"
```

In order to construct this, one can make use of the Daniell-Kolmogorov theorem, which is available in Isabelle [64], in order to produce a process with an uncountable state space and specified distributions. However, this does not guarantee almost surely continuous paths; for this we will prove the Kolmogorov-Chentsov theorem, which will allow us to modify a process in order to make its paths almost surely continuous.

Often in probability theory we extend common arithmetic properties so that they hold outside of some set with probability 0. This is the case for equality between stochastic processes, and we present two different ways of extending it in this way.

In order to compare processes, we first introduce the concept of compatible processes, which share the same index set, target measure, and source σ-algebra.

Definition 6.6 (Compatible processes)**.** Stochastic processes are compatible if they share the same index set, source measurable space, and target σ-algebra

```
lift_definition compatible :: "('t,'a,'b) stochastic_process ⇒ ('t,'a,'b)
stochastic_process ⇒ bool"
is "λ(Mx, M'x, Ix, _) (My, M'y, Iy, _). Mx = My ∧ sets M'x = sets M'y
∧ Ix = Iy" .
```

Definition 6.7 (Modifications)**.** Any two compatible processes X and Y are modifications of each other if, for any given $t \in I$, $P[X_t = Y_t] = 1$. Equivalently, X is a modification of Y if and only if there exists a family of null sets N_t such that $\{X_t \neq Y_t\} \subseteq N_t$ for all $t \in I$.

```
definition modification :: "('t,'a,'b) stochastic_process ⇒ ('t,'a,'b)
stochastic_process ⇒ bool" where
"modification X Y ⟷ compatible X Y ∧ (∀t ∈ proc_index X. AE x in proc_source
X. X t x = Y t x)"
```

Definition 6.8 (Indistinguishable processes)**.** Two processes X and Y are indistinguishable if the set $\{\exists t.\ X_t \neq Y_t\}$ is a null set.

```
definition indistinguishable :: "('t,'a,'b) stochastic_process ⇒ ('t,'a,'b)
stochastic_process ⇒ bool" where
"indistinguishable X Y ⟷ compatible X Y ∧
 (∃N ∈ null_sets (proc_source X). ∀t ∈ proc_index X. {x ∈ space (proc_source
X). X t x ≠ Y t x} ⊆ N)"
```

Indistinguishability clearly implies modification, and we prove that the other direction of implication holds for a countable index set, and almost surely continuous processes on an interval:

Lemma 6.1 (Modification implies indistinguishability when the processes are almost surely continuous)**.** *Let X and Y be stochastic processes into a metric space (S, d). Assume that X and Y are modifications of each other, that they have indices of the form $[0, T]$, and that they are almost surely continuous. Then X and Y are indistinguishable.*

```
lemma modification_continuous_indistinguishable:
  fixes X :: "(real, 'a, 'b :: metric_space) stochastic_process"
  assumes modification: "modification X Y"
    and interval: "∃T > 0. proc_index X = {0..T}"
    and continuous: "AE ω in proc_source X. continuous_on (proc_index
X) (λt. X t ω)"
            "AE ω in proc_source Y. continuous_on (proc_index Y) (λt.
Y t ω)"
  shows "indistinguishable X Y"
```

The proof of this theorem relies on obtaining a null set for a countable dense subset of the index space, and extending the null set to cover the whole space by continuity. We also rely on this technique later for the proof of the Kolmogorov-Chentsov theorem.

6.4 Convergence in measure

The usual notion of convergence for functions is pointwise convergence: a sequence of functions $f_i : X \to Y$ converges pointwise to f if $\forall x \in X.\ \lim_{n\to\infty} f_n(x) = f(x)$. There are two ways of generalising this within a measure space, where we want to consider convergence up to measure 0: convergence in measure and almost sure convergence.

The texts we consulted [71, 26], only define convergence in measure for sequences, and leave it to the reader to extrapolate to real-valued functions. We mechanise this by defining convergence in measure in terms of filters. These are commonly used in topology, and particularly Isabelle/HOL and Lean, to indicate the "mode of convergence" that we are interested in. Examples of filters are the convergence of a sequence as $n \to \infty$, or the limit of the function at a point $\lim_{x \to a} f(x)$.

In Isabelle, to specify that a function f converges pointwise to some value l along filter F, we write `(f ⟶ l) F`, which can be read as "f converges to l along filter F". A common filter is `sequentially`, which specifies convergence of a sequence. Another is `at t within S`, which specifies convergence of a function to a point within a domain. So we can write $(\frac{1}{n} \longrightarrow 0)$ `sequentially`, and $(\frac{1}{n} \longrightarrow \infty)$ `at 0 within` $[0, \infty)$.

We extend this notion to convergence in measure by providing an alternate "converges to" operator:

Definition 6.9 (Convergence in measure)**.** A family of measurable functions f converges in measure μ to measurable function l in filter F if for all measurable sets A of finite measure and $\varepsilon > 0$:

$$\mu(\{\omega . d(f_n(\omega), l(\omega)) > \varepsilon\} \cap A)) \longrightarrow_F 0)$$

```
definition tendsto_measure :: "'b measure
   ⇒ ('a ⇒ 'b ⇒ 'c :: {second_countable_topology,metric_space})
   ⇒ ('b ⇒ 'c) ⇒ 'a filter ⇒ bool"
  where "tendsto_measure M X l F ≡ (∀n. X n ∈ borel_measurable M)
   ∧ l ∈ borel_measurable M
   ∧ (∀ε > 0. ∀A ∈ fmeasurable M.
  ((λn. measure M ({ω ∈ space M. dist (X n ω) (l ω) > ε} ∩ A)) ⟶
0) F)"
```

Intersecting with a set of finite measure helps to "localise" the convergence - in spaces of infinite measure, we can define pathological functions which diverge but converge in measure, and this finite set prevents that.

In finite measure spaces, our definition of convergence in measure is equivalent to one without the intersection with the set of finite measure. This lets us provide a simpler introduction rule in the `finite_measure` locale:

```
lemma (in finite_measure) finite_tendsto_measureI:
  assumes [measurable]: "⋀n. f' n ∈ borel_measurable M" "f ∈ borel_measurable
M"
    and "⋀ε. ε > 0 ⟹
     ((λn. measure M {ω ∈ space M. dist (f' n ω) (f ω) > ε}) ⟶ 0)
F"
  shows "tendsto_measure M f' f F"
```

We also define convergence almost everywhere, which is yet another way to define convergence in a measure space:

Definition 6.10 (Convergence almost everywhere)**.** A family of measurable functions f converges almost everywhere to measurable function l in filter F if

$$f_n \longrightarrow_F l \qquad \text{almost everywhere}$$

```
definition tendsto_AE :: "'b measure ⇒ ('a ⇒ 'b ⇒ 'c :: topological_space)
⇒ ('b ⇒ 'c) ⇒ 'a filter ⇒ bool" where
"tendsto_AE M f' l F ⟷ (AE ω in M. ((λn. f' n ω) ⟶ l ω) F)"
```

We show that convergence almost everywhere implies convergence in measure, which are both implied by standard pointwise convergence.

```
lemma measure_conv_imp_AE_sequentially:
  assumes [measurable]: "⋀n. f' n ∈ borel_measurable M"
                        "f ∈ borel_measurable M"
    and "tendsto_AE M f' f sequentially"
  shows "tendsto_measure M f' f sequentially"
```

We show that the limit in measure — and by consequence the limit almost everywhere — is unique almost everywhere.

```
lemma (in sigma_finite_measure) LIMSEQ_measure_unique_AE:
  fixes f :: "nat ⇒ 'a ⇒ 'b :: {second_countable_topology,metric_space}"
  assumes [measurable]: "⋀n. f n ∈ borel_measurable M"
      "l ∈ borel_measurable M" "l' ∈ borel_measurable M"
    and tendsto: "tendsto_measure M f l sequentially"
        "tendsto_measure M f l' sequentially"
  shows "AE x in M. l x = l' x"
```

6.5 Dyadic intervals

To prove the Kolmogorov-Chentsov theorem, we rely on the existence of the dyadic rationals to provide a countable, dense subset of the reals with evenly-sized intervals that can be made arbitrarily small. This will allow us to establish continuity of our process on this countable set, which we can then extend to the whole space by leveraging continuity.

The dyadic rationals are rationals whose denominator is a power of 2 — $\{k/2^n\}$ for integer k and natural n. In this section, we formalise dyadic *intervals*, which contain all dyadic rationals which are a subset of some real-valued interval.

We define dyadic intervals in terms of steps:

Definition 6.11 (Dyadic interval step)**.** For real numbers S, T and natural number n, dyadic interval step $D_n(S,T)$ contains all dyadic rationals in interval $[S,T]$ with denominator 2^n:

$$D_n(S,T) = \left\{ \frac{k}{2^n} \;\middle|\; k \in [\lceil 2^n S \rceil, \lfloor 2^n T \rfloor] \right\}$$

```
definition dyadic_interval_step :: "nat ⇒ real ⇒ real ⇒ real set"
  where "dyadic_interval_step n S T ≡ (λk. k / (2 ^ n)) ` {⌈2^n * S⌉..⌊2^n
* T⌋}"
```

Then, we can define the dyadic interval as the union of all dyadic steps.

Definition 6.12 (Dyadic interval)**.** The set of dyadic rationals contained in real interval $[S,T]$ is given by $D(S,T) = \bigcup_n D_n(S,T)$

```
definition dyadic_interval :: "real ⇒ real ⇒ real set"
  where "dyadic_interval S T ≡ (⋃n. dyadic_interval_step n S T)"
```

Conceptually, this coincides with understanding the dyadic steps as a sequence, and taking their limit as $n \to \infty$. The dyadic steps provide a constructive way of defining properties of the dyadic

interval by defining a property which holds on step n and taking a limit to extend it to the entire set.

We prove some expected properties of the dyadic interval, beginning with finiteness and countability:

Lemma 6.2 (Any dyadic interval step $D_n(S,T)$ is finite)**.**

```
lemma dyadic_interval_step_finite[simp]: "finite (dyadic_interval_step
n S T)"
```

Lemma 6.3 (Dyadic interval $D(S,T)$ is countable)**.**

```
lemma dyadic_interval_countable[simp]: "countable (dyadic_interval S T)"
```

Our concern with this theory is to aid the development of the Kolmogorov-Chentsov theorem, where we will only be considering intervals of the form $[0,T]$. For convenience, we write $D(T)$ for $D(0,T)$, and likewise for steps $D_n(T)$ for $D_n(0,T)$.

```
lemma dyadic_interval_dense: "closure (dyadic_interval 0 T) = {0..T}"
```

Dyadic intervals are dense in the realsDyadic interval $D(T)$ is dense in real interval $[0,T]$ — any point in $[0,T]$ is the limit of a sequence of points in $D(T)$

To prove this lemma, we define sequence $\xi_n \triangleq \frac{\lfloor 2^n x \rfloor}{2^n}$, and show that $\xi_n \in D_n(T)$, and $\lim_{n\to\infty} \xi_n = x$.

Dyadic expansions We define the binary expansion of a dyadic rational, as we will use it in the Kolmogorov-Chentsov proof. Any dyadic admits a binary expansion, consisting of some integer k for whole part, and a bit list b for the fractional part:

Definition 6.13 (Binary expansion of dyadic rational)**.** (b,k) is the n-bit expansion of dyadic rational $x \in D_n(T)$ if b is a list of integers drawn from $\{0,1\}$ with length n and

$$x = k + \sum_{m=1}^{n} b_{m-1}$$

```
definition "dyadic_expansion x n b k ≡ set b ⊆ {0,1}
  ∧ length b = n ∧ x = real_of_int k + (∑m∈{1..n}. real (b ! (m-1))
/ 2 ^ m)"
```

The dyadic expansion is defined for dyadic rationals in a given step.

Definition 6.14 (Dyadic expansion for dyadic interval step)**.** Any dyadic rational $x \in D_n(T)$ has an n-bit expansion

```
lemma dyadic_expansion_ex:
  assumes "x ∈ dyadic_interval_step n 0 T"
  shows "∃b k. dyadic_expansion x n b k"
```

It then follows that any dyadic rational has a dyadic expansion:

Definition 6.15 (Dyadic expansion for dyadic intervals)**.** Any dyadic rational $x \in D(T)$ has some dyadic expansion k, b

```
lemma dyadic_expansion_ex':
  assumes "x ∈ dyadic_interval 0 T"
  shows "∃n b k. dyadic_expansion x n b k"
```

Finally, the dyadic expansion is unique for a given step.

```
lemma dyadic_expansion_unique:
  assumes "dyadic_expansion x n b k"
    and "dyadic_expansion x n c j"
  shows "b = c ∧ j = k"
```

6.6 Hölder continuity

Hölder continuity lies between uniform continuity and Lipschitz continuity. We will need it for the Kolmogorov-Chentsov theorem, as we will be producing Hölder continuous functions, but it also finds applications in functional analysis.

Definition 6.16 (Hölder continuity)**.** Let (E, d) and (E', d') be two metric spaces. A function $\varphi : E \to E'$ is Hölder continuous of order γ (Hölder-γ-continuous) for $0 < \gamma \leq 1$ on set D if for all $r, s \in D$ there is a $C \geq 0$ such that

$$\forall r, s \in D.\, d(\varphi(r), \varphi(s)) \leq C d'(r, s)^\gamma$$

```
definition holder_on :: "real
              ⇒ 'a :: metric_space set
              ⇒ ('a ⇒ 'b :: metric_space)
              ⇒ bool" ("_-holder'_on" 1000) where
  "γ-holder_on D φ ⟷
   (∃C ≥ 0. (∀r∈D. ∀s∈D. dist (φ r) (φ s) ≤ C * dist r s powr γ))"
```

Here, `powr` is the power function for $\mathbb{R}$-valued exponents, as opposed to `^` for exponents in $\mathbb{N}$.

We can also define a local version of this notion, where a function needs to satisfy the continuity condition only within some $\varepsilon > 0$, rather than the entire space.

Definition 6.17 (Local Hölder continuity)**.**

```
definition local_holder_on :: "real ⇒ 'a :: metric_space set ⇒ ('a ⇒
'b :: metric_space) ⇒ bool" where
  "local_holder_on γ D φ ≡
  (∀t∈D. ∃ε > 0. ∃C ≥ 0. (∀r∈D. ∀s∈D. dist s t < ε ∧ dist r t < ε ⟶
dist (φ r) (φ s) ≤ C * dist r s powr γ))"
```

An example of a Hölder-continuous function is the power function for fractional powers:

Definition 6.18 (Fractional powers are Hölder)**.** For some real γ where $0 < \gamma \leq 1$, function x^γ is Hölder-γ-continuous on domain $[0, \infty]$.

```
lemma holder_powr:
  assumes "α ∈ {0<..1}"
  shows "α-holder_on {0..} (λx. x powr α)"
```

Further, x^γ is *exactly* Hölder-γ-continuous - it is not Hölder for any value larger than γ.

If φ is Hölder-γ-continuous on D, it is also locally Hölder-γ-continuous on D — we can simply choose the entire space as the "local" neighbourhood of continuity. The other direction holds if D is compact — we prove this by constructing the constant C required by global continuity, using the constants given by local continuity on the finite open cover of D.

Lemma 6.4 (Local Hölder continuity implies Hölder continuity in a compact domain.)**.** *If f is locally Hölder-γ-continuous on domain D and D is compact, then f is Hölder-γ-continuous.*

```
lemma local_holder_compact_imp_holder:
  assumes "γ > 0" "compact I" "local_holder_on γ I φ"
  shows "γ-holder_on I φ"
```

Finally, we prove that Hölder continuity is related in the expected ways to other forms of continuity which are already available in the Isabelle distribution. We show that global 1-Hölder continuity is Lipschitz continuity and local 1-Hölder continuity is local Lipschitz; global γ-Hölder implies uniform continuity and both the global and local versions imply regular continuity.

6.7 The Kolmogorov-Chentsov Theorem

The Kolmogorov-Chentsov theorem allows us to construct Hölder-continuous modifications of processes which satisfy an upper bound on their expectation.

Before we begin, our first step will be to prove a generalised version of Markov's inequality, which extends theorem `nn_integral_Markov_inequality` from the HOL-Analysis library.

Theorem 6.1 (Markov's inequality)**.** *Let $(\Omega, \mathcal{A}, \mu)$ be a measure space, Let X be a real-valued measurable function on $\mathcal{A}$. Let $f : \mathbb{R} \to [0, \infty)$ be monotonically increasing on the set $(0, \infty) \cup ran(X)$. Then, for any $\varepsilon > 0$ where $f(\varepsilon) > 0$:*

$$\mu[X(p) \geq \varepsilon] \leq \frac{\int_\Omega f(X(x))\mu(dx))}{f(\varepsilon)}$$

```
theorem nn_integral_Markov_inequality_extended:
  fixes f :: "real ⇒ ennreal" and ε :: real and X :: "'a ⇒ real"
  assumes mono: "mono_on (range X ∪ {0<..}) f"
    and finite: "⋀x. f x < ∞"
    and e: "ε > 0" "f ε > 0"
    and [measurable]: "X ∈ borel_measurable M"
  shows "emeasure M {p ∈ space M. (X p) ≥ ε}
                ≤ (∫⁺ x. f (X x) ∂M) / f ε"
```

This theorem lets us give an upper bound to the measure of set $\{X > \epsilon\}$ given the expected value of $f(X)$.

6.7.1 Theorem premises and general properties

There are several steps to the proof of the Kolmogorov-Chentsov theorem, so we define a locale that encodes the necessary premises and give each step as a separate lemma.

Theorem 6.2 (Assumptions for the Kolmogorov-Chentsov theorem)**.** *Let X be a stochastic process defined in a probability space with measure μ, which takes values in a Polish space equipped with metric d,*

and is indexed by $[0, \infty)$*. Assume we have some* $a, b, C < 0$*, and some* $\gamma \in (0, b/a)$*, such that for any* $s, t \geq 0$*, the following inequality holds:*

$$\int d(X_t(x), X_s(x))^a \mathrm{d}\mu(x) \leq C d(t, s)^{1+b}$$

Then, we will construct a modification X' *of* X *such that* X' *is locally* γ*-Hölder-continuous.*

```
locale Kolmogorov_Chentsov =
  fixes X :: "(real, 'a, 'b :: polish_space) stochastic_process"
    and a b C γ :: real
  assumes index[simp]: "proc_index X = {0..}"
    and target_borel[simp]: "proc_target X = borel"
    and gt_0: "a > 0" "b > 0" "C > 0"
    and b_leq_a: "b ≤ a"
    and gamma: "γ ∈ {0<..<b/a}"
    and expectation: "⋀s t. ⟦s ≥ 0; t ≥ 0⟧ ⟹
          (∫⁺ x. dist (X t x) (X s x) powr a ∂proc_source X)
        ≤ C * dist t s powr (1+b)"
```

Here, our process takes values in a Polish space, which are defined in Isabelle as complete, separable metric spaces. Common examples of Polish spaces include rationals, reals, and real normed vectors.

As part of our assumptions, we have a bound on the expectation of X. We use can use the above theorem to convert this into a bound on the measure of increments of X by transitivity:

Lemma 6.5 (Upper bound on increments of X)**.** *For* $s, t \geq 0$*,* $\varepsilon > 0$*, the following bound holds:*

$$P[d(X_s, X_t) \leq \varepsilon] \leq \frac{E[d(X_t, X_s)]^\alpha}{\varepsilon^\alpha}$$

```
lemma (in Kolmogorov_Chentsov) markov_X:
  assumes "s ≥ 0" "t ≥ 0" "ε > 0"
  shows "𝒫(x in source. ε ≤ dist (X t x) (X s x))
              ≤ (C * dist t s powr (1 + b)) / ε powr a"
```

This lets us show that at any $t \geq 0$, X converges in probability to X_t — it can be understood as "continuous in measure".

Lemma 6.6 (X converges to X_t in measure)**.**

```
lemma (in Kolmogorov_Chentsov) conv_in_prob:
  assumes "t ≥ 0"
  shows "tendsto_measure (proc_source X) X (X t)
         (at t within {0..})"
```

We will construct our modification as a limit of the process as $s \to t$, and because X_s converges in probability this will allow us to show that the limit is equal to X_t with probability 1, and hence a modification.

6.7.2 Constructing the modification for index set $[0, T]$

We will now construct a Hölder-continuous modification of X restricted to $[0, T]$ for any positive T. Later, we will use countable additivity properties to extend this to an index set of $[0, \infty)$.

```
locale Kolmogorov_Chentsov_finite = Kolmogorov_Chentsov +
  fixes T :: real
  assumes zero_le_T: "0 < T"
```

The idea is to construct our modification process on the dyadic rationals by discarding a null set of dyadic increments that do not satisfy a Hölder condition, and using uniform continuity to cover the whole space. The null set in question is set N, defined as follows:

Definition 6.19 (Null set of non-continuous paths)**.** Define

$$A_n = \{\text{Max}\,\{d(X_{2^{-n}(k-1)}, X_{2^{-n}k}) \mid k \in 1..\lfloor 2^n T\rfloor\} \geq 2^{-\gamma n}\}$$

This set characterises the increments that do not satisfy a Hölder condition. We now construct the limit set as $n \to \infty$ by letting $B_n = \bigcup_m A_{n+m}$, and finally $N = \bigcap_n B_n$

```
definition "A ≡ λn. if 2 ^ n * T < 1 then space source else
  {x ∈ space source.
    Max {dist (X (real_of_int (k - 1) * 2 powr - real n) x)
              (X (real_of_int k * 2 powr - real n) x)
       | k. k ∈ {1..⌊2^n * T⌋}} ≥ 2 powr (-γ * real n)}"

abbreviation "B ≡ λn. (⋃m. A (m + n))"

abbreviation "N ≡ ⋂(range B)"
```

We can obtain the measure of set A from its construction by using the Markov bound from lemma `markov_X`:

Lemma 6.7 (Measure of A_n)**.**

```
lemma emeasure_A_leq:
  fixes n :: nat
  assumes [simp]: "2^n * T ≥ 1"
  shows "emeasure source (A n) ≤ C * T * 2 powr (- n * (b - a * γ))"
```

We can then show that $P(B_n)$ tends to 0 as $n \to \infty$, which allows us to finally show that N is a null set.

Lemma 6.8 (Limit of B_n is 0)**.**

```
lemma lim_B: "(λn. measure source (B n)) ⟶ 0"
```

Having identified this null set, we will construct our modification by ignoring the null set, and showing that every path that does not belong to it is Hölder continuous. For this, we fix some $\omega \in \Omega - N$, for which we will show that $X(\omega)$ is Hölder continuous on the dyadic rationals, and then use this to construct the desired modification.

Definition 6.20 (Fixed ω outside of N)**.** We define a context (an unnamed locale) for some arbitrary but fixed ω, and obtain a positive value n_0 for which $\omega \notin \bigcup_n A_{n+n_0}$

```
context
  fixes ω
  assumes ω: "ω ∈ space source - N"
begin

definition "n₀ ≡ SOME m. ω ∉ (⋃n. A (n + m)) ∧ m > 0"
```

Here, we use Isabelle's `SOME` operator to define n_0. This gives us an object which satisfies a given predicate, invoking the axiom of choice, and we need to prove the existence of such an object in order to use it in proofs.

Dyadic increments Here, we will show that $X(\omega)$ is Hölder continuous on the dyadic rationals. To recap, lemma `markov_X` uses the Markov inequality to give an upper bound on the probability that the increments of X change too quickly for Hölder continuity, and defined family of sets A_i to characterise the paths of X which exhibit Hölder-discontinuous increments of size 2^{-n}. Then, at the limit, set N characterises paths with discontinuous sections of any size. Finally, by matching the definition of A with the Markov inequality, we are able to prove the following theorem:

Theorem 6.3 (Dyadic increments of $X(\omega)$)**.** *For any $\omega \in \Omega - N$, and for rationals of the form $s = \frac{k}{2^n}, t = \frac{k-1}{2^n} \in 0..T$, the s, t increment of X satisfies a Hölder condition:*

$$d(X_s(\omega), X_t(\omega)) < 2^{-\gamma n}$$

```
lemma X_dyadic_incr:
  assumes "k ∈ {1..⌊2^n * T⌋}" "n ≥ n₀"
  shows "dist (X ((real_of_int k-1)/2^n) ω)
              (X (real_of_int k/2^n) ω) < 2 powr (- γ * n)"
```

Our task now is to prove that the above property holds for any two dyadic rationals. We will do this in stages, by proving the property for increasingly more permissive choices of index, and leveraging the construction of the dyadic rationals as a limit of steps. For convenience, we write D^n to mean $D^n_{0,T}$ as defined previously.

Lemma 6.9 (Dyadic increments 1)**.** *For some $n_0 \leq n \leq m$, let t, u be dyadic rationals such that $u \leq t$, $t \in D^m$, $u \in D^n$ and $t - u \leq \frac{1}{2^{n-1}}$. Then,*

$$d(X_u(\omega), X_t(\omega)) \leq \frac{2^{-\gamma n}}{1 - 2^{-\gamma}}$$

```
lemma dist_dyadic_mn:
  assumes mn: "n₀ ≤ n" "n ≤ m"
    and t_dyadic: "t ∈ dyadic_interval_step m 0 T"
    and u_dyadic_n: "u ∈ dyadic_interval_step n 0 T"
    and ut: "u ≤ t" "t - u < 2/2^n"
  shows "dist (X u ω) (X t ω) ≤ 2 powr (- γ * n) / (1 - 2 powr - γ)"
```

To prove this, we rely on the dyadic expansion of $t - u$, i.e. a list $b \subseteq 0, 1$ such that $\sum_{i=1}^{m} 2^{-i} b_i$. Note that, for simplicity, we present this theorem with a 1-indexed b, but in the Isabelle text b is 0-indexed. We use this to construct a sequence ξ which interpolates between t and u in 2^{-k}-sized steps:

$$\xi_l = u + \sum_{i=n}^{l} \frac{b_i}{2^i}$$

Then, $\xi_n = u$, $\xi_m = t$ and $\xi_{k+1} - \xi_k = \begin{cases} 2^{-k}, & \text{if } b_k = 1, \\ 0, & \text{if } b_k = 0. \end{cases}$ for $k \in [n, m)$

Now, we can apply theorem `X_dyadic_incr` to each of these increments: $d(X_{xi_l}, X_{xi_l - 1}) \leq 2^{-\gamma l}$

And finally, by applying the triangle inequality for sums:

$$d(X_t, X_u) \leq \sum_{i=n}^{m} d(X_{\xi_{i+1}}, X_{\xi_i}) \leq \sum_{i=n}^{m} 2^{-\gamma i} \leq \frac{2^{-\gamma n}}{1 - 2^{-\gamma}}$$

Lemma 6.10 (Dyadic increments 2). *For some $n_0 \leq n \leq m$, let t and s be dyadic rationals such that $s \leq t, t, s \in D^m$, and $t - s \leq \frac{1}{2^n}$. Then,*

$$d(X_t(\omega), X_s(\omega)) \leq 2\frac{2^{-\gamma n}}{1 - 2^{-\gamma}}$$

```
lemma dist_dyadic_fixed:
  assumes mn: "n₀ ≤ n" "n ≤ m"
    and s_dyadic: "s ∈ dyadic_interval_step m 0 T"
    and t_dyadic: "t ∈ dyadic_interval_step m 0 T"
    and st: "s ≤ t" "t - s ≤ 1/2^n"
  shows "dist (X t ω) (X s ω) ≤ 2 * 2 powr (- γ * n) / (1 - 2 powr -
γ)"
```

To show this, we obtain an intermediate value $u \triangleq \frac{\lfloor 2^n s \rfloor}{2^n}$. We can apply the above theorem to intervals t, u and u, s, and with the triangle inequality obtain the required bound.

Definition 6.21 (Hölder constant C_0). Define $C_0 = 2\frac{2^\gamma}{1-2^{-\gamma}}$

```
definition "C₀ ≡ 2 * 2 powr γ / (1 - 2 powr - γ)"
```

This is the limit of the sum of 2^n, which will serve as a Hölder constant for the following lemma:

Lemma 6.11 (Dyadic increments 3). *For any two dyadic rationals where $|s-t| \leq \frac{1}{2_0^n}$, $d(X_t(\omega), X_s(\omega) \leq C_0 * |s-t|^\gamma$*

```
lemma dist_dyadic:
  assumes t: "t ∈ dyadic_interval 0 T"
    and s: "s ∈ dyadic_interval 0 T"
    and st_dist: "dist t s ≤ 1 / 2 ^ n₀"
  shows  "dist (X t ω) (X s ω) ≤ C₀ * (dist t s) powr γ"
```

We no longer restrict the step that our increment resides in, but from the construction of dyadic intervals we know that any dyadic rational belongs to some interval construction step. In order to apply the previous theorem, we obtain an m such that both s and t belong to D^m. Now, we can apply our previous bound directly.

In order to give a bound for dyadic increments of any size, we define another constant:

Definition 6.22 (Hölder constant K). Define $K = C_0(2^{\lceil 2^{n_0} T \rceil})^{1-\gamma}$"

```
definition "K ≡ C₀ * (2^nat ⌈2^n₀ * T⌉) powr (1 - γ)"
```

Now, we can finally extend the bound for all increments.

Lemma 6.12 (Dyadic increments 4). *For any two dyadic rationals $s, t \in [0, T]$*

```
lemma X_dyadic_le_K:
  assumes dyadic: "s ∈ dyadic_interval 0 T" "t ∈ dyadic_interval 0 T"
  shows "dist (X s ω) (X t ω) ≤ K * dist s t powr γ"
```

The reasoning for this mirrors the argument for step 1: we already have a bound on increments of size 2^{-n_0}, and we can extend the bound using the triangle inequality to cover the entire interval. To this end, we define $f_i = s + (t-s)\frac{i}{2^n}$, where $n = \lceil 2^{n_0} T \rceil$ is defined so that $\frac{|s-t|}{n} \leq \frac{1}{2^{n_0}}$ and hence satisfy `dist_dyadic`. Now, we can take the sum $\sum_i = 1^{2^n} d(X_{f_i}, X_{f_i - 1})$ and apply the previous step, and obtain our bound by the triangle inequality.

Hence, we have arrived at the following theorem:

Theorem 6.4 (X is Hölder-continuous on the dyadic rationals)**.**

```
corollary holder_dyadic: "γ-holder_on (dyadic_interval 0 T) (λt. X t ω)"
  apply (intro holder_onI exI[where x=K])
  using K_pos X_dyadic_le_K by force
```

Hölder continuity on the real interval In order to extend our result from the dyadics to the reals, we define the following limit:

Definition 6.23 (Limit of $X(\omega)$)**.**

$$L_t = \lim_{s \to t \in D_{0,T}} X_s(\omega)$$

```
definition "L ≡ (λt. (Lim (at t within dyadic_interval 0 T) (λs. X s
ω)))"
```

The existence of this limit follows from the uniform continuity of $X(\omega)$ (which is implied by Hölder continuity), and the fact that the dyadics form a dense cover of the reals. Here is where the polish space assumption comes into play — if the space weren't polish, we would not be able to guarantee the existence of this limit.

Constructing the modification Having defined a suitable Hölder continuous function for a fixed ω, we now stop fixing ω and define our modification:

Definition 6.24 (Definition of $\tilde{X}$)**.** Given some $\omega \in \Omega - N$, we define

$$\tilde{X}_t(\omega) = \lim_{s \to t \in D_{0,T}} X_s(\omega)$$

```
definition X_tilde :: "real ⇒ 'a ⇒ 'b" where
  "X_tilde ≡ (λt ω. if ω ∈ N then undefined
     else (Lim (at t within dyadic_interval 0 T) (λs. X s ω)))"
```

Showing that $\tilde{X}(\omega)$ is Hölder continuous for any $\omega \in \Omega$ is straightforward: we have already shown it for $\omega \in \Omega - N$, and since it is constant for $\omega \in N$, it is trivially continuous for this, too.

We now need to show that $\tilde{X}$ is a modification. This follows from the almost-sure uniqueness of limits in probability — we proved earlier that X_s converges to X_t in probability as $s \to t$, and it follows from the convergence of L that $\tilde{X}_s$ will converge almost-surely to X_t as $s \to t \in D_{0,t}$. Together, these facts allow us to prove the following:

Lemma 6.13 (For $0 \leq t \leq T$, $X_t = \tilde{X}_t$ almost surely)**.**

```
lemma X_eq_X_tilde_AE:
  assumes "t ∈ {0..T}"
  shows "AE ω in source. X t ω = X_tilde t ω"
```

6.7.3 Extending the modification from $[0, T]$ to $[0, \infty)$

We have constructed our required modification on index set $[0, T]$, and it now remains to show that this can be extended to a modification covering index set $[0, \infty)$. The idea is to take the following limit:

$$\lim_{n\to\infty} \tilde{X}_{[0,n]}$$

for $\mathbb{N}$-valued n.

To do this, we first discard the details of our construction of $\tilde{X}$ — all that matters is that it is a Hölder continuous modification of X:

Definition 6.25 (Arbitrary modification of X on index set [0,T).)]

```
definition "Mod ≡ λT. SOME Y. modification (restrict_index X {0..T})
Y ∧
    (∀x∈space source. local_holder_on γ {0..T} (λt. Y t x))"
```

Lemma 6.14 (Indistinguishable on restricted set). *For any two $S, T > 0$, $Mod(S)$ is indistinguishable from $Mod(T)$ on index set $min(S, T)$.*

```
lemma indistinguishable_mod:
  assumes "S > 0" "T > 0"
  shows "indistinguishable
     (restrict_index (Mod S) {0..min S T})
     (restrict_index (Mod T) {0..min S T})"
```

We get this by deriving modification first, and then obtaining indistinguishability from the path continuity of Mod. This lets us obtain a null set on which $Mod(S)$ and $Mod(T)$ are different, for any S, T:

Definition 6.26 (Null set for Mod S and Mod T). For $S, T > 0$ we have a null set $N_{S,T}$ such that $X^T \neq X^S \subseteq N_{S,T}$.

```
definition "N S T ≡ SOME N. N ∈ null_sets source ∧
   {ω ∈ space source. ∃t ∈ {0..min S T}. (Mod S) t ω ≠ (Mod T) t ω}
⊆ N"
```

Then, $N_{\text{inf}} = \bigcup_{S,T\in\mathbb{N}} N_{S,T}$ is a null set by countable subadditivity, which characterises

Lemma 6.15 (Countable union of $N_{S,T}$ is a null set).

```
definition N_inf where "N_inf ≡ (⋃S ∈ ℕ - {0}. (⋃ T ∈ ℕ - {0}. N S
T))"

lemma N_inf_null: "N_inf ∈ null_sets source"
```

With this, we can finally construct a modification of X on index set $[0, \infty)$ by ignoring paths in N_{inf}.

Definition 6.27 (Construction of a modification of X on $[0, \infty)$)**.**

```
definition "X_mod ≡ λt ω. if ω ∈ space source - N_inf
                          then Mod ⌊t+1⌋ t ω else undefined"

definition "X_mod_process ≡ prob_space.process_of source
                    (proc_target X) {0..} X_mod undefined"
```

We show that this process is a modification of X, and that it is locally Hölder-γ-continuous, completing the proof of the Kolmogorov-Chentsov theorem.

```
lemma modification_X_mod_process: "modification X X_mod_process"

lemma local_holder_X_mod: "local_holder_on γ {0..} (λt. X_mod t ω)"
```

Discussion In order to prove this theorem, we were able to closely follow the structure of the textbook proof, but there were many mechanisation gaps to fill in, and some departures:

- The textbook proof assumes $T = 1$ without loss of generality (WLOG) in the construction of the modification on a bounded index set. We believe that this WLOG claim stems from a similarity in reasoning between the $T = 1$ case and the arbitrary T case. However, without a formal proof in Isabelle of this fact, we had to resort to proving the theorem for an arbitrary T directly. This motivated the development of arbitrary dyadic intervals, and complicated the proofs somewhat, as it forced us to adjust for cases $T < 1$ and $T > 1$.
- Proving that X is continuous on the dyadic intervals was the most arduous part of the proof. It involved over 400 lines over 5 lemmas, as well as the formalisation of dyadic rationals and their binary expansion.
- There was no mention in the textbook of any measurability or summability requirement for the constructed objects, though none of them were too complex to prove.

We have submitted this proof to the AFP, together with the necessary supplementary material. The submission consists of 3830 lines, of which 1368 are the actual proof of the Kolmogorov-Chentsov theorem.

6.8 Transition kernels

Transition kernels can be understood as generalisations of Markov chain transition functions. They can be used to characterise and construct Markov processes in a natural way. In this section, we develop the machinery required for defining stochastic processes, by using well-behaved families of kernels. This involves defining the infinite product space generated by a family of kernels, building on the Daniell-Kolmogorov theorem, whose coordinate process will have the desired properties. In particular, we are concerned with Markov semigroups: consistent families of kernels where composing kernels is equivalent to summing their indices.

Unfortunately, we were not able to complete this construction, owing in part to an error in the textbook proof we were following. Transition kernels have been constructed before in Isabelle, but the product kernel construction is novel, and will still prove valuable as a means to construct and characterise stochastic processes.

Definition 6.28 (Transition kernel)**.** We define a transition kernel from measurable space $(\Omega, \mathcal{A})$ to $(\Omega', \mathcal{A}')$ as a function $\kappa : \Omega \times \mathcal{A}' \to \mathbb{R}^+$ such that:

$$\forall A' \in \mathcal{A}'. \quad \kappa(-, A') \text{ is } \mathcal{A} - \mathcal{B} \text{ measurable}$$
$$\forall \omega \in \Omega. \quad \kappa(\omega, -) \text{ is a measure on } (\Omega', \mathcal{A}')$$

```
locale transition_kernel =
  fixes M :: "'a measure"
    and M' :: "'b measure"
    and κ :: "'a ⇒ 'b set ⇒ ennreal"
  assumes
    sets_target_measurable [measurable]:
      "⋀A'. A' ∈ sets M' ⟹ (λω. κ ω A') ∈ M →M borel" and
    space_source_measure: "⋀ω. ω ∈ space M ⟹
      measure_space (space M') (sets M') (λA'. κ ω A')"
```

Here, $\mathcal{B}$ denotes the Borel σ-algebra. This is the sigma-algebra generated by the open sets of a topological space, and is very commonly used when dealing with topological spaces.

Intuitively, given some current state ω, $\kappa(\omega, A')$ computes the probability of transitioning to some state in A'. Transition kernels provide a natural way of defining processes by characterising their increments. For example, we can characterise a random walk on the integers which increases with probability p and decreases with probability $(1 - p)$ by the following kernel on the integers:

Definition 6.29.

$$\kappa(i, \{m\}) = \begin{cases} p & \text{if } i = m - 1 \\ 1 - p & \text{if } i = m + 1 \\ 0 & \text{otherwise} \end{cases}$$

Because we are working with a discrete sigma algebra, i.e. the power set of the integers, the kernel is fully defined by its operation on singleton sets. We make use of countable additivity to compute the probability of some event, by simply taking the sum over its elements.

A transition kernel $\kappa : \Omega \to A'$ is stochastic iff $\forall \omega \in \Omega. \kappa(\omega, -)$ is a probability measure. In order to compose kernels together, we need to integrate with respect to the kernel measure. We show that this integral is measurable, which is essential if we are composing several integrals, or defining a kernel in terms of the integral of another kernel.

Lemma 6.16 (Kernel integral is measurable)**.** *Let f be an $(\Omega_1 \times \Omega_2 - \mathcal{B})$ measurable function. Let κ be a finite kernel on $\Omega_1 - \Omega_2$. Define*

$$I_f(\omega_1) = \int f(\omega_1, \omega_2) \kappa(\omega_1, d\omega_2)$$

Then I_f is a Borel measurable function.

```
lemma kernel_measure_pair_integral_measurable [measurable]:
  fixes f :: "'a × 'b ⇒ ennreal"
    and K :: "('a, 'b) kernel"
  assumes f_measurable[measurable]: "f ∈ borel_measurable (kernel_source
K ⨂M kernel_target K)"
      and finite_K: "finite_kernel K"
  defines "I ≡ λ f. (λω1. ∫+ω2. f(ω1, ω2) ∂kernel_measure K ω1)"
  shows "I f ∈ borel_measurable (kernel_source K)"
```

The proof for this lemma follows an approximation approach for Borel measurable functions, where we first prove that the theorem holds for indicator functions, then simple functions, and finally for Borel measurable functions as they can be represented as countable sums of simple functions, which are measurable because of the monotone convergence of the Lebesgue integral.

6.8.1 Kernel Product

The binary kernel product combines two kernels to operate on a product space.

Definition 6.30 (Binary kernel product)**.** Let κ_1 be an $\Omega_1 - \Omega_2$ kernel, and κ_2 be a $\Omega_1 \times \Omega_2 - \Omega_3$ kernel. Then the kernel product $\otimes_K$ is a kernel $\Omega_1 - \Omega_2 \times \Omega_3$ defined by

$$(\kappa_1 \otimes_K \kappa_2)(\omega_0, A) \triangleq \int \left(\int \mathbb{1}_A(\omega_1, \omega_2) \kappa_2((\omega_0, \omega_1) d\omega_2) \right) \kappa_1(\omega_0, d\omega_1)$$

```
definition pair_kernel :: "('a, 'b) kernel ⇒ ('a × 'b, 'c) kernel ⇒
('a, 'b × 'c) kernel" (infixr "(⨂K)" 90) where
"pair_kernel K_1 K_2 ≡ kernel_of (kernel_source K_1) (kernel_target K_1
⨂M kernel_target K_2)
  (λω0 A. ∫+ω1. (∫+ω2. indicator A (ω1, ω2)∂kernel_measure K_2 (ω0,
ω1))∂kernel_measure K_1 ω0)"
```

We also define $\otimes_P$ as the kernel product of $\kappa_1 : \Omega_1 - \Omega_2$ and $\kappa_2 : \Omega_2 - \Omega_3$ by understanding κ_2 as a kernel that ignores the elements of Ω_1. This operation has more reasonable types which makes it more compositional, and therefore more useful in defining stochastic processes.

Definition 6.31 (Partial kernel product)**.**

```
definition pair_kernel_partial :: "('a, 'b) kernel ⇒ ('b, 'c) kernel
⇒ ('a, 'b × 'c) kernel" (infixr "(⨂P)" 90) where
"pair_kernel_partial K_1 K_2 ≡ K_1 ⨂K (kernel_of (kernel_source K_1
⨂M kernel_source K_2) (kernel_target K_2)
 (λω A2. kernel K_2 (snd ω) A2))"
```

We show that our definitions form a transition kernel if the kernels are finite. For this, we need to prove that the inside of the integral is measurable, which requires an additional proof. The arguments for these are quite standard - approximation using Dynkin systems. If both kernels are finite then the product is σ-finite, and if they are stochastic or substochastic then the product is too.

Our objective is to define the finite product kernel analogously to the following recursion:

$$\bigotimes_{i=1}^{k+1} \kappa_i = (\kappa_1 \otimes \kappa_2 \otimes \cdots \otimes \kappa_k) \otimes \kappa_{k+1}$$

This is not possible within the type system of Isabelle, as it is not possible to construct a type of n-fold Cartesian products where n is allowed to vary. Instead, we represent these n-fold products as functions from a finite index set, and define the recursion using the product measure `PiM` as our target.

Definition 6.32 (Finite product kernel)**.** Let n be some natural number and K a family of kernels

with index set $\{0..n\}$. Then define $\bigotimes_{k=0}^{n} K_k$ is a kernel $\Omega - \Omega^{\{0..n\}}$, defined recursively as follows:

$$\left(\bigotimes_{i=0}^{0} K_i\right)(\omega, A') = K_0(\omega, \pi_0^{-1}(A')),$$

$$\left(\bigotimes_{i=0}^{n+1} K_i\right)(\omega, A') = \int_{\omega_f} \left[\int_{\omega'} \mathbf{1}_{A'}(\omega_f, \omega')\, dK_{n+1}(\omega_f(n), \omega')\right] d\left(\bigotimes_{i=0}^{n} K_i\right)(\omega, \omega_f).$$

```
primrec PiK :: "nat ⇒ (nat ⇒ ('a, 'a) kernel) ⇒ ('a, (nat ⇒ 'a)) kernel"
where
"PiK 0 K = kernel_of (kernel_source (K 0)) (PiM {0} (λi. kernel_target
(K i)))
   (λ ω A'. (K 0) ω ((λx.(λn∈{0}. x)) -‘ A' ∩ space (kernel_target (K
0))))" |
"PiK (Suc n) K = kernel_of (kernel_source (K 0)) (PiM {0..Suc n} (λi.
kernel_target (K i)))
  (λω A'. ∫⁺ωf. (∫⁺ω. indicator A' (ωf (Suc n:=ω)) ∂kernel_measure
(K (Suc n)) (ωf n))
                ∂kernel_measure (PiK n K) ω)"
```

In our definition, the base case lifts kernel K_0 to operate on the product space of functions, and the recursive case defines the product kernel as an integral with respect to $K(n+1)$, analogously to our binary product construction. We show that this kernel is well-formed by induction on the dimension n of the product, and we are able to reuse proofs from our binary product construction.

We now define a way of obtaining a measure from a kernel. Let $(\Omega, \mathcal{F}, \mu)$ be a measure space, and let $\kappa : \mathcal{F} - \mathcal{G}$ be a finite kernel. Then

$$\mu \otimes_S \kappa \triangleq \kappa_\mu \otimes_P \kappa$$

defines the semidirect product. We prove that integrals over the semidirect product can be split into two integrals:

$$\int f(\omega)(\mu \otimes_S \kappa)(d\omega) = \int \left(\int f(\omega_1, \omega_2)\kappa(\omega_1, d\omega_2)\right) \mu(d\omega_1)$$

Finally, we define kernel composition. Let κ_1 be a finite $(\Omega_0, \mathcal{A}_0) - (\Omega_1, \mathcal{A}_1)$ transition kernel, and let κ_2 be a $(\Omega_1, \mathcal{A}_1) - (\Omega_2, \mathcal{A}_2)$ transition kernel. We can now define kernel composition by

$$\kappa_1 \circ \kappa_2(\omega_0, A_2) = \int_{\Omega_1} \kappa_2(\omega_1, A_2)\kappa_1(\omega_0, d\omega_1)$$

Kernels and convolution In Klenke's text, theorem 14.31 makes the following claim [71, p.314]:

Given a family of independent variables X_i and index I, define the kernels

$$\kappa_i(\omega, -) \triangleq \delta_\omega \star \mathcal{D}_{X_i}$$

Then, the product kernel of κ_i at state 0 is equal to the distribution of X_i:

$$\left(\bigotimes_{k=0}^{n} \kappa_{I_k}\right) 0 = \prod_{t \in I([0,n])} \mathcal{D}(X_t)$$

However, we have been unable to reproduce the associated proof in Isabelle. The chain of reasoning relies on the following statement:

$$\mathbb{P}_{(S_1,\ldots,S_n)}(\varphi_n(A_1 \times \cdots \times A_n)) = \prod_{k=1}^{n} \mu_k(A_k)$$

Where $S_k = X_1 + \cdots + X_k$ and $\varphi_k(x_1, \ldots, x_k) = (x_1, x_1 + x_2, \ldots, x_1 + \cdots + x_k)$. We believe that the above does not hold in general — the argument seems to be that the image of φ is the inverse of taking sums of each coordinate via the S random variable, but we were unable to prove this. As this is a required lemma for this approach to constructing Brownian motion, we were unable to continue.

6.9 Discussion

We present the conclusions that we have drawn about mechanisation of textbook mathematics in the form of answers to the following questions.

Question 1: How well do textbook mathematics lend themselves to mechanisation? We found that on a number of occasions, the textbook proofs omitted a significant amount of detail, at times skipping proof obligations entirely. This was most common with measurability proofs; these can be quite tedious, but were often dismissed by the author. Most of the work in formalising pen-and-paper mathematics lies in making the implicit – explicit – and filling in the gaps between the broad strokes of the proofs.

In particular, one of the major deviations from Klenke's text was the development of the theory of dyadic intervals. In the proof of Kolmogorov-Chentsov, the author assumes "without loss of generality" that $T = 1$, implicitly arguing that the reasoning for any other value of T would be identical. This kind of reasoning cannot be carried out ad hoc in a proof assistant, and therefore we had to develop a theory that allowed us to deal with arbitrary values of T in the proof, leading us to dyadic intervals.

Importantly, we claim to have found a proof error. This highlights the importance of mechanisation in enforcing mathematical rigour, as it is impossible for such errors to get past mechanisation. Unfortunately, this did also prevent us from completing the construction of Brownian motion, though other possible constructions are possible and left to future work.

Our development consists of ∼7000 lines of proofs, comprising 391 lemmas and 33 definitions. The formalisation covers about 20 pages of the textbook, including some preliminary material from the early chapters that was not previously available in Isabelle/HOL. We consider this to be a substantial piece of work; the pen-and-paper proofs are terse, and significant effort was expended in adapting textbook definitions to work with the Isabelle/HOL libraries and type system.

Question 2: How do the proof facilities of Isabelle/HOL support mechanisation of textbook mathematics? Isabelle/HOL was expressive enough for all of the definitions we wanted to write, but at times we had to make concessions for the type system. The finite kernel product was defined in the textbook as a recursion using the binary product. This would need dependent types to work in Isabelle, so we had to make this construction explicit. Other such occasions included having to restrict the types of measures to be homogeneous, when they could in theory each have a different

type, although we doubt this will have any impact in practice. The advantage of using simple types is the wealth of automation facilities available; much of the low-level reasoning can be deferred to automated theorem provers via Sledgehammer [14], making the development simpler. Throughout the development, we count almost 700 subgoals proven via Sledgehammer.

Despite this, we often found ourselves decomposing large proof goals by hand. Sledgehammer is not a panacea, and it goes hand-in-hand with Isabelle's other theorem provers, such as the simplifier and the measurability prover. We made use of two main approaches for improving automation within our theories: firstly, we made use of type definitions when practical, as these leverage the type system to discharge assumptions automatically. Secondly, theorem annotations give domain-specific knowledge to the automated provers by indicating which theorems can be applied to a specific situation, e.g. simplification rules can be unconditionally applied to simplify a proof goal. Despite this, we still had to write routine proofs by hand. In particular, proofs for measurability goals involving the product measure `PiM` were very difficult to automate; this re-interprets functions as dependent products, which are cumbersome to reason about in Isabelle's type system.

6.10 Related work

The probability theory of Isabelle/HOL was developed during Hölzl's PhD thesis [57], which introduces basic concepts such as the construction of probability spaces, as well as product spaces and Markov Chains. Immler formalised the Kolmogorov extension theorem [64], which proves the existence of stochastic processes with arbitrary indices. We make use of these results, which are all available in the HOL-Analysis and HOL-Probability libraries of Isabelle/HOL.

There are overlaps between our work and other Isabelle/HOL developments. In their formalisation of quasi-Borel spaces, Hirata et al. developed a theory of measure kernels in order to describe probabilistic programs [56]. Their definition is compatible with ours, but they do not define any form of the product kernel and thus it is not enough to describe stochastic processes. However, we do make use of their formalisation of measurable isomorphisms. Stochastic processes have been covered in Isabelle/HOL too — Keskin et al. have developed a theory of martingales in Isabelle/HOL [68]. The theorems they prove in their development are largely disjoint from ours, but their definition of stochastic processes is compatible with ours.

Beyond Isabelle/HOL, comparable probability-theory formalisations exist in other interactive theorem provers, though to our knowledge none reach the construction of Brownian motion. Lean's mathlib [109] provides an extensive measure- and probability-theory library, including conditional expectation and a substantial theory of martingales, with work towards stochastic calculus and the Itô integral ongoing; its foundational, measure-theoretic design is close to ours, but the Kolmogorov–Chentsov theorem and the product-kernel construction we develop here are absent. In the Coq/Rocq ecosystem, the Coquelicot library [16] supplies real analysis, while the MathComp-Analysis and Infotheo developments of Affeldt et al. [4] formalise measure-theoretic probability on top of the Coq mathematical components library. These likewise share a foundational basis with our work, but they target general analysis and discrete information theory rather than continuous-time stochastic processes, and so do not provide the modifications and continuity results that the construction of Brownian motion requires.

Our development is thus distinguished less by the underlying measure theory — which is broadly comparable across these systems — than by the machinery specific to continuous-time processes: convergence in measure, almost-surely continuous modifications via Kolmogorov–Chentsov, and the iterated kernel product.

6.11 Summary

We have developed a theory of stochastic processes in Isabelle/HOL, based on the existing probability theory library. We formalised stochastic processes and stochastic kernels, and proved the Kolmogorov-Chentsov theorem for producing continuous modifications of well-behaved processes. The groundwork has been laid for the construction of Brownian motion, and with it the mechanisation of stochastic calculus. Even if the attempted approach failed, there are many different constructions of Brownian motion that can be developed with the use of our theory.

Chapter 7

Stochastic Differential Dynamic Logic for Stochastic Hybrid Systems

So far, we have developed a simulator for SHS and mechanised the underlying theory of stochastic processes. We will now bring these two together by mechanising an axiomatic logic for SHS in Isabelle/HOL.

In this chapter, we detail our implementation of stochastic differential dynamic logic (SdL) in Isabelle/HOL. While the logic itself was proposed by Platzer [95], its mechanisation and the supporting proof infrastructure are original contributions of this thesis.

7.1 Semantics for SdL

Let us first recount some notions from Platzer's paper. He defines SHPs by the following grammar:

$$\alpha, \beta ::= x_i := \theta \mid x_i := * \mid ?H \mid dx = bdt + \sigma dW \& H \mid \lambda\alpha \oplus \nu\beta \mid \alpha; \beta \mid \alpha^*$$

Here θ is a real-valued expression, H is a predicate on the state, b denotes the flow and σ denotes the noise component of an SDE, and ν and λ are real numbers.

These are, in order: assignment, random assignment, test, SDE evolution, probabilistic choice, sequential composition and iterated composition. They behave as expected, analogously to the SHPs we defined for simulation; we will later see how they can be implemented in Isabelle.

As an example, we can rewrite the thermostat SHP presented in Chapter 5 to fit in this language:

$$\begin{aligned}
&\text{Thermostat} \triangleq x := *; \\
&?x \geq 19 \wedge x \leq 24; \, (\\
&0.5(\mathrm{d}x = 5dt + 0.1x \, \mathrm{d}W \& \, x \leq 24) \\
\oplus\, &0.5(\mathrm{d}x = -0.1 \, \mathrm{d}t + 0.1x \, \mathrm{d}W \& \, x \geq 19) \\
&)^*
\end{aligned}$$

This language operates more non-deterministically than ours - the test operation discards any executions that fail its predicate. The differences are the following:

- Random assignment in this language assumes a distribution which is not specified. In our simulation language, we use a uniform distribution over a range.
- Tests perform the same function as our if statement. Tests are impractical for simulation, since they require that we throw away an execution trace, but for many practical purposes they are equivalent.
- We choose not to implement random choice in our simulation language, as we can instead use random assignment followed by conditional.
- There are no while loops in this language - it is geared towards system modelling rather than allowing some amount of traditional programming.

SHPs are given semantics by the following pair of functions.

$$[\![\alpha]\!] : (\Omega \to \mathbb{R}^d) \to ([0,\infty) \times \Omega \to \mathbb{R}^d); Z \mapsto [\![\alpha]\!]^Z = ([\![\alpha]\!]_t^Z)_{t \geq 0}$$

and

$$(\!|\alpha|\!) : (\Omega \to \mathbb{R}^d) \to (\Omega \to \mathbb{R}); Z \mapsto (\!|\alpha|\!)^Z$$

Here, our state space is $\mathbb{R}^d$ and our probability space is Ω. The function $[\![\alpha]\!]_t^Z$ takes as input a random variable Z defining its initial state and returns an $\mathbb{R}^d$-valued stochastic process, indexed by a positive time t. This notion of stochastic process matches what we developed in order to mechanise Brownian motion, with the added complication of an initial state parameter.

$(\!|\alpha|\!)^Z$ is a stopping time, after which α is no longer allowed to evolve and instead becomes $\perp$. Together, the two functions define a class of objects which evolve until they hit some boundary condition.

In order to reason about SHPs, Platzer introduces stochastic differential dynamic logic (SdL) formulas. An SdL formula f maps a random variable describing the current state to a real-valued random variable. They are given semantics by function $[\![f]\!] : (\Omega \to \mathbb{R}^d) \to (\Omega \to \mathbb{R})$.

Terms of SdL are given by

$$f, g ::= F \mid Bf \mid \lambda f + \nu g \mid \langle \alpha \rangle f$$

Where F is a measurable function, B is a boolean combination of terms, $+$ is linear combination and $\langle \alpha \rangle f$ (diamond) represents the supremal value of f along the evolution of SHP α. An SdL formula is a boolean expression regarding two SdL terms:

$$\phi ::= f \leq g \mid f = g$$

The central operator of the logic is the diamond, which has the following semantics

$$[\![\langle \alpha \rangle f]\!]^Z = \sup\{[\![f]\!]^{[\![\alpha]\!]_t^Z} : 0 \leq t \leq (\!|\alpha|\!)^Z\}$$

Term $\langle \alpha \rangle f$ denotes the maximum value attained by SdL term f during the execution of program α, between times 0 and $(\!|\alpha|\!)$. Intuitively, f would denote some safety condition which is continuously tracked, where higher values are more unsafe. This value is continuously tracked and the highest value is returned - if it was greater than some threshold then the system entered an unsafe state.

For example, the SdL term $\langle \text{Thermostat} \rangle x$ denotes the highest temperature reached during the execution of the thermostat program. This is a random variable - we obtain a distribution over possible maximum values.

7.2 A Semantic Domain for SdL

In order to mechanise this logic, we will be taking a different approach to Platzer in defining the logic. Rather than define a syntax for the logic and give each syntax element semantics, we are fixing a semantic domain for the logic and then using this to define the syntax. The main benefit here is that we escape the limitations of a fixed syntax tree, while allowing us to directly make use of Isabelle's proof automation.

This is a shallow embedding, where we aim to re-use the facilities of the implementation language as much as possible, rather than working with an abstract syntax tree and implementing the target language from scratch. We applied this approach to the type system of the simulation language in Chapter 5 by making use of Haskell types via GADTs rather than defining our type universe from scratch.

For this, we work in the type of functions, rather than with an abstract datatype which defines the constructors for SHPs and SdL. This will require first identifying the relevant type of functions, and then producing a set of conditions that characterise functions that correspond to SHP and SdL terms.

Our contributions here are the following:

- We augment the space and the operations to make the absorbing state precise.
- We identify a set of healthiness conditions for functions to represent SHPs and SdL terms.

Absorbing state

In order to characterise the behaviour of processes that result from a failed test, or those which have exceeded their stopping time, Platzer relies on a notion of 'absorbing state' which these processes enter [95, p.4], but this is never precisely defined. Our first contribution is to make this notion precise by extending the target space of SHPs with an unreachable element that will characterise an absorbing state.

Labelling the disjoint point characterising absorbing state as $\perp$, we will be writing $S_\perp$ as shorthand for $S \cup \{\perp\}$. Then, the type of the random variable denoting SHP state becomes:

$$Z : \Omega \to \mathbb{R}^d_\perp$$

Z is a random variable which, at any given point, is either $\mathbb{R}^d$-valued or $\perp$. We can use this to define SHPs as functions directly:

Definition 7.1. An SHP is a function α of type:

$$\alpha : (\Omega \to \mathbb{R}^d_\perp) \to ((\mathbb{R} \to \Omega \to \mathbb{R}^d_\perp) \times (\Omega \to \mathbb{R}))$$

Instead of defining semantic functions to give the stopping time and stochastic process associated with a given SHP, the SHP now *is* the function, which takes as input initial state and . We can recover the two components by

For consistency, we write $[\![\alpha]\!]$ and $(\!|\alpha|\!)$ to identify the first and second elements of this tuple, respectively. In addition, we write $[\![\alpha]\!]^Z$ to mean the process associated with α started in state Z, and $[\![\alpha]\!]_t$ to indicate the value taken at time t by α.

SHP healthiness

The type we constructed in definition 7.1 allows behaviours that we want to disallow for SHPs. Functions with this type can recover from an absorbing state, they can negative stopping times, and there are no constraints on how the process evolves. In order to restrict the behaviours of our SHPs, we define the following healthiness conditions, which characterise "well-behaved" programs:

Definition 7.2 (SHP healthiness)**.**

$$(\!|\alpha|\!) \geq 0 \tag{7.1}$$

$$\text{If } Z(\omega) = \bot \text{ then } [\![\alpha]\!]^Z(\omega) = \bot \tag{7.2}$$

$$\text{Program } [\![\alpha]\!]^Z(\omega) \text{ may only apply } Z \text{ to } \omega \tag{7.3}$$

$$[\![\alpha]\!]^Z_t \text{ is a random variable} \tag{7.4}$$

$$[\![\alpha]\!]^Z \text{ is a.s. piecewise continuous.} \tag{7.5}$$

Condition 7.1 mandates that processes have a non-negative stopping time. This will be necessary when we define the diamond operator within SdL. Condition 7.2 requires that the process be unable to recover from a failed intial state. This is necessary for the behaviour of the operators which use the absorbing state to be as expected: we need the process that arises from a failed test to be discounted from any future calculations, so it cannot suddenly recover from failure.

Condition 7.3 is more unusual, but without it we cannot ensure that the process is using the initial state, rather than conjuring up some ω value which it returns. This is effectively a constraint on the syntax of the language.

We require condition 7.4 in order to reason about the probabilistic behaviour of the SHP. Finally, condition 7.5 ensures that the stochastic process has piecewise-continuous sample paths, allowing for discrete jumps. In practice, we will axiomatise the behaviour of the continuous processes, so we will not be including this condition in our mechanisation.

SdL term healthiness

Similarly to above, we define SdL terms to be denoted as functions of the type

$$(\Omega \to \mathbb{R}^d_\bot) \to (\Omega \to \mathbb{R})$$

They are maps from state-valued random variables to real-valued random variables. We also define healthiness conditions to SdL terms, analogously to the above definition for SHPs.

Definition 7.3 (SdL term healthiness)**.** An SdL term is healthy if it satisfies the following conditions:

$$f^Z(\omega) = 0 \text{ if } Z(\omega) = \bot \tag{7.6}$$

$$f^Z(\omega) \text{ may only apply } Z \text{ to } \omega \tag{7.7}$$

$$f^Z \text{ is a random variable} \tag{7.8}$$

7.3 Simplifying the Semantic Domain

While the healthiness conditions let us characterise the set of functions which correspond to valid terms, they are cumbersome to work with and lead to clunky definitions. We need to take care to handle $\bot$ terms explicitly, and condition 7.3 in particular is difficult to work with. Thankfully, we can refactor the type of these functions in order to simplify definitions and encode some healthiness conditions within the type. For SHPs, we argue that the following simplified type is equivalent:

$$\mathbb{R}^d \to \Omega \to ((\mathbb{R} \to \mathbb{R}^d_\bot) \times \mathbb{R})$$

There are a few changes here: instead of having a random variable representing additional state, we simply have an $\mathbb{R}^d$ value. Conceptually, condition 7.3 enforces that the only value that a process α can use with path ω is $Z(\omega)$, so we can simply pre-apply Z to ω and supply that value, rather than requiring that the process do it.

Next, we extract the path parameter Ω from the tuple in order to simplify definitions. This results in SHPs being two-argument functions which return a (path, time) tuple.

Finally, we exclude $\bot$ from the input. Condition 7.2 forces SHPs to take value $\bot$ when the input state is $\bot$ - we can enforce this directly without need for a healthiness condition by removing the option from the argument and instead, when attempting to run an SHP on the absorbing state, simply returning an $\bot$ and a stopping time of 0 directly.

The changes made to the type of SdL terms are analogous – the resulting type is the following:

$$\mathbb{R}^d \to (\Omega \to \mathbb{R})$$

SdL terms are now functions from state to real-valued random variables. Similarly to above, applying an SdL term to an absorbing state results in a random variable which always returns 0; the only healthiness condition that remains is measurability.

Formal proof of equivalence

We can show that the two definitions for SHP semantics are equivalent. For this, we construct a bijection φ between the two forms of SHP, which we will later prove correct:

$$[\![\varphi(\alpha)]\!]^Z_t(\omega) = \begin{cases} \bot & \text{if } Z(\omega) = \bot \\ [\![\alpha]\!]^{Z(\omega)}_t(\omega) & \text{otherwise} \end{cases}$$

$$(\!|\varphi(\alpha)|\!)^Z(\omega) = \begin{cases} 0 & \text{if } Z(\omega) = \bot \\ (\!|\alpha|\!)^{Z(\omega)}(\omega) & \text{otherwise} \end{cases}$$

Similarly, we give a bijection ξ between our definitions of SdL term:

$$[\![\xi(f)]\!]^Z(\omega) = \begin{cases} 0 & \text{if } Z(\omega) = \bot \\ [\![f]\!]^{Z(\omega)}(\omega) & \text{otherwise} \end{cases}$$

7.3.1 Definitions for the Semantic Domain

In addition to a simpler characterisation of the semantic domain, we provide new definitions which we argue simplify the logic and make it more extensible. In particular, we present the following 3:

- SHPs with stopping time 0 are generalised as discrete SHPs.
- The Iverson bracket lets us embed predicates in SdL.
- We define a number of mathematical operators for SdL terms, e.g. addition.

Discrete SHPs

There are a number of primitive constructors in SdL that have a stopping time of 0. These simply perform a discrete action on the state, and thus can be specified as this action on the state and lifted to act on the correct space.

Definition 7.4 (Discrete SHP)**.** Given some function $\sigma : (\mathbb{R}^d \times \Omega) \to \mathbb{R}^d$, the discrete SHP associated with it is

$$\begin{aligned} [\![\sigma]\!]^k_t(\omega) &= \sigma(k, \omega) \\ (\!|\sigma|\!)^k &= 0 \end{aligned}$$

These functions are probabilistic programming operators. For example, we can define deterministic assignment as a function which operates on the state and ignores the random variable. Given some substitution $s : \mathbb{R}^d \to \mathbb{R}^d$, we can define as $s'(k, \omega) = s(k)$.

Iverson bracket

Platzer's approach to defining SdL terms does not fit with our semantic embedding approach. As long as reasoning with the resulting terms is tractable, we can define a much richer set of constructors than Platzer provides, and since we will be implementing this in Isabelle there is no implementation overhead resulting from this, as we'll be able to reuse the existing theories.

In order to embed arbitrary predicates in SdL formulae, we define the Iverson bracket:

Definition 7.5 (Iverson bracket)**.** Given predicate $\mathcal{H} : \mathbb{R}^d \to \mathbb{B}$, the Iverson bracket $[H]$ is defined by

$$[\mathcal{H}]^Z(\omega) = \begin{cases} 0 & \text{if } Z(\omega) = \bot \text{ or } \mathcal{H}(Z(\omega)) \\ 1 & \text{otherwise} \end{cases}$$

Operations on SdL terms

In order to allow the combining of SdL terms, we can show that they form a vector space, a multiplicative ring, a lattice, and a partial order. Defining operations on SdL terms is generally straightforward, as it is usually sufficient to lift the relevant operation on real numbers to work on the SdL terms in a pointwise fashion. For example, addition (and identically multiplication, division, etc.) can be defined as follows:

$$[\![f + g]\!]^Z(\omega) = [\![f]\!]^Z(\omega) + [\![g]\!]^Z(\omega) \tag{7.9}$$

An exception to this is the ordering of SdL terms.

Definition 7.6 (SdL Order)**.** Let f and g be SdL terms. We define the (non-strict) order as

$$f \leq g \triangleq \forall Z, \omega \,.\, [\![f]\!]^Z(\omega) \leq [\![g]\!]^Z(\omega).$$

The strict order is then given by

$$f < g \triangleq f \leq g \wedge \exists Z, \omega \,.\, [\![f]\!]^Z(\omega) < [\![g]\!]^Z(\omega).$$

7.4 Mechanising Stochastic Hybrid Programs

In Isabelle, we define these objects by employing a shallow embedding, rather than describing the syntax and semantics separately. This allows us harness Isabelle's proof automation and existing theories without having to constantly apply the semantic functions. As covered in section 7.3, we use a simpler type than Platzer uses in his text, and define healthiness conditions to characterise well-formed programs.

```
definition wb_SHP :: "('s ⇒ 'a ⇒ (real ⇀ 's) × real) ⇒ bool" where
"wb_SHP α ≡ ∀k ω. snd (α k ω) ≥ 0"
```

Despite having identified a number of conditions, the only one that is practically necessary for this development is non-negative time. This is because conditions 7.2 and 7.3 are enforced by the type of SHPs, and we axiomatise the probability space and the stochastic evolution operations so conditions 7.5 and 7.4 are not necessary for verification.

In the type of the above definition, `'a ⇀ 'b` is synonymous with type `'a ⇒ 'b option`.

A quirk of working with Isabelle is that we cannot use the standard probability theory approach of working with random variables in a point-free way; instead we explicitly reference the member of the probability space $\omega \in \Omega$ that we are applying our random variable to. We call this ω the path, as it will characterise the (random) path that our stochastic process will take.

Using the above, we can now give an Isabelle proof of the equivalence between Platzer's definitions for SHPs and our own

```
lemma wb_SHP_bij: "bij_betw φ (Collect wb_SHP_alt) (Collect wb_SHP)"
```

Here, Collect is the set of all objects that satisfy a given predicate – in this case healthiness conditions for SHPs.

With this, we can define a type of SHPs, similarly to how we defined the type of stochastic processes and kernels in our Brownian motion development.

```
typedef ('a, 's) shp = "{α :: 's ⇒ 'a ⇒ ((real ⇀ 's) × real). wb_SHP
α}"
  morphisms dest_SHP mk_SHP
```

As mentioned above, we can define a generic constructor for discrete SHPs, which takes as argument a function on state which can potentially be absorbing.

```
lift_definition discrete_SHP :: "('s ⇒ 'a ⇀ 's) ⇒ ('a, 's) shp" is
"λf k ω. (λ_. f k ω, 0)"
```

Now, we can define assignment by lifting a substitution using this operation.

```
definition SHP_assign :: "'s subst ⇒ ('a, 's) shp" where
"SHP_assign σ ≡ discrete_SHP (λk ω. Some (σ k))"
```

We also define random assignment, which takes a lens and assigns it the value given by a random variable applied to the current path. This is a departure from Platzer's logic, which does not allow us to specify the distribution.

```
definition SHP_rand_assign :: "('b ⟿ 's) ⇒ ('a ⇒ 'b) ⇒ ('a, 's) shp"
where
"SHP_rand_assign l X ≡ discrete_SHP (λk ω. Some (put_l k (X ω)))"
```

With the `discrete_SHP` operator we can also define tests:

```
definition SHP_test :: "('s ⇒ bool) ⇒ ('a, 's) shp" where
"SHP_test H ≡ discrete_SHP (λx ω. if H x then Some x else None)"
```

Tests behave as an identity under composition unless the supplied predicate does not hold on the state, in which case they enter the absorbing state.

Our last primitive SHP is the SDE

```
consts SDE :: "(real ⇒ 's subst) — Drift
           ⇒ (real ⇒ 'a ⇒ 's ⇒ 's ⇒ 's) — Noise
           ⇒ ('s ⇒ bool) — Boundary
           ⇒ ('a, 's) shp"
```

We do not have the background theories required to properly define this operator in Isabelle, so instead we introduce it as a named constant for which we will define axioms. An SDE takes three parameters: a substitution defining the flow, a matrix defining the noise, and a boundary condition indicating when evolution should halt.

Semantically, the process associated with this SHP is the stochastic process X that solves the differential equation specified by $dx = bdt + \sigma dW$, with stopping time $\inf\{t \geq 0. X_t \notin H\}$. This expression returns the earliest time where the process fails to satisfy the boundary condition H, and informs when SDE evolution should halt and the rest of the program should continue executing.

We define two combinators for constructing SHP programs.

```
lift_definition SHP_oplus :: "real ⇒ ('a, 's) shp ⇒ ('a, 's) shp ⇒
('a, 's) shp"
  (infixl "(⊕_S)" 55) is
"λx α β. (λk ω. if U ω ≤ x then α k ω else β k ω)"
```

Given processes α and β, and real numbers x and y which are expected to add to 1, this operator makes a weighted random choice between them with the help of random variable U, which we axiomatise to have a uniform distribution in $[0, 1]$.

The definition for sequential composition is similar, but we must deal with the possibility of either SHP becoming absorbing:

```
lift_definition SHP_seq :: "('a, 's) shp ⇒ ('a, 's) shp ⇒ ('a, 's) shp"
(infixl ";" 65) is
"λα β. (λk ω. let (proc_α, stop_α) = α k ω
              in (case proc_α stop_α of
                None ⇒ (λt. None, stop_α)
              | Some k' ⇒ ((λt. if t < stop_α
                   then proc_α t
                   else fst (β k' ω) (t - stop_α)),
                stop_α + snd (β k' ω))))"
```

Here, we first execute α until it reaches its stopping time, at which point we execute β starting at the state at which α terminated. If either of the processes become absorbing, so does the composition.

Natural number powers are iterated sequential composition

```
primrec SHP_pow :: "nat ⇒ ('a, 's) shp ⇒ ('a, 's) shp" where
"SHP_pow 0 _ = SHP_test (λ_. True)" |
"SHP_pow (Suc n) α = α ; (SHP_pow n α)"
```

And finally, Kleene star executes the iterated sequential composition until the process becomes absorbing or reaches its stopping time.

```
definition SHP_star :: "('a, 's) shp ⇒ ('a, 's) shp" where
"SHP_star α ≡ mk_SHP (λk ω. (λt.
              SHP_process (SHP_pow (LEAST x. snd (SHP_pow x α k ω) ≥
t) α) k t ω,
              lim (λn. SHP_time (SHP_pow n α) k ω)))"
```

7.5 Stochastic differential dynamic logic

We define SdL formulae in Isabelle as functions with type $S \Rightarrow \Omega \Rightarrow real$. Here, S is the state space, and Ω is the sample space for our probability measure. Then, an SdL formula can be seen as a function from state to a real-valued random variable.

```
typedef ('a, 's) sdl = "UNIV :: ('s ⇒ 'a ⇒ real) set"
  morphisms app_sdl mk_sdl
  by simp
```

Similarly to SHPs, we prove that the two definitions for SdL terms are isomorphic, by constructing the following bijection;

```
definition wb_SdL :: "(('a ⇀ 's) ⇒ 'a ⇒ real) ⇒ bool" where
"wb_SdL f = (∀Z ω. (Z ω = None ⟶ f Z ω = 0) ∧
            f Z ω = f (λx∈{ω}. Z x) ω)"

definition ξ :: "('s ⇒ 'a ⇒ real) ⇒ ('a ⇀ 's) ⇒ 'a ⇒ real" where
"ξ f Z ω = (case Z ω of None ⇒ 0 | Some x ⇒ f x ω)"
```

```
lemma wb_SdL_bij: "bij_betw ξ UNIV (Collect wb_SdL)"
```

SdL formulae are effectively real-valued functions, which have a natural interpretation for arithmetic operators in terms of their pointwise application. We define a number of operators in this fashion using Isabelle's class mechanism. For example, to instantiate our SdL type as a real vector space we supply the following definitions:

```
instantiation sdl :: (type, type) real_vector begin

lift_definition uminus_sdl :: "('a, 'b) sdl ⇒ ('a, 'b) sdl" is "λf Z
ω. - f Z ω" .

lift_definition zero_sdl :: "('a, 'b) sdl" is "λZ ω. 0" .

lift_definition minus_sdl :: "('a, 'b) sdl ⇒ ('a, 'b) sdl ⇒ ('a, 'b)
sdl" is
  "λf g Z ω. f Z ω - g Z ω" .

lift_definition plus_sdl :: "('a, 'b) sdl ⇒ ('a, 'b) sdl ⇒ ('a, 'b)
sdl" is
  "λf g Z ω. f Z ω + g Z ω" .

lift_definition scaleR_sdl :: "real ⇒ ('a, 'b) sdl ⇒ ('a, 'b) sdl" is
  "λr f Z ω. r * f Z ω" .
```

Here, we use the lifting package to define terms on the underlying structure of SdL terms. For each of these operators, we apply the functions to their arguments and apply the relevant natural number operator to the result. We use the same approach to instantiate SdL terms as a lattice:

```
instantiation sdl :: (type, type) lattice begin

lift_definition inf_sdl :: "('a, 'b) sdl ⇒ ('a, 'b) sdl ⇒ ('a, 'b) sdl"
is
  "λf g Z ω. min (f Z ω) (g Z ω)" .

lift_definition sup_sdl :: "('a, 'b) sdl ⇒ ('a, 'b) sdl ⇒ ('a, 'b) sdl"
is
  "λf g Z ω. max (f Z ω) (g Z ω)" .
```

Defining an order proves more complicated - the pointwise approach does not yield the antisymmetry required. Instead, for SdL terms f and g, $f \leq g$ if for all initial states Z, and for all path parameters ω, $fZ\omega \leq gZ\omega$. To define $f < g$, we require $f \leq g$, and in addition that there be at least one state Z and path ω for which $f(Z,\omega) < g(Z,\omega)$.

```
instantiation sdl :: (type, type) order begin

lift_definition less_eq_sdl ::
  "('a, 'b) sdl ⇒ ('a, 'b) sdl ⇒ bool" is
  "λf g. ∀Z. f Z ≤ g Z" .

lift_definition less_sdl ::
  "('a, 'b) sdl ⇒ ('a, 'b) sdl ⇒ bool" is
```

```
  "λf g. f ≤ g ∧ (∃Z ω. f Z ω < g Z ω)" .
```

In order to lift boolean predicates to SdL, we define the Iverson bracket:

```
lift_definition iverson :: "('s ⇒ bool) ⇒ ('a, 's) sdl" is
  "λH k ω. if H k = True then 1 else 0" .
```

This lets us define implication as multiplication.

```
definition SdL_imp :: "('s ⇒ bool) ⇒ ('a, 's) sdl ⇒ ('a, 's) sdl" (infix
"(⟶$_S$)" 55) where
  "H ⟶$_S$ f = iverson H * f"
```

Finally, the core operator of SdL is the diamond, which provides the ability to reason about SHPs.

```
lift_definition SdL_fdia :: "('a, 's) shp ⇒ ('a,'s) sdl ⇒ ('a, 's) sdl"
is
  "λ α f k ω. (⨆ {(case fst (α k ω) t of Some v ⇒ f v ω | None ⇒ 0)
                | t. 0 ≤ t ∧ t ≤ snd (α k ω)})" .
```

It records the supremal value of f reached by SHP α during its execution. For discrete programs, this amounts to evaluating f on the final state, but for continuous programs this evaluates f along the path of the stochastic process until it reaches the stopping time, and returns the maximum reached.

We take the maximum rather than the expected value as this allows for worst-case verification – a system that reaches an unsafe state briefly is still unsafe, despite being safe on average.

We axiomatise a number of properties about the SdL diamond.

```
axiomatization where
  seq: "⟨α ; β⟩ f ≤ ⟨α⟩ (f ⊔ ⟨β⟩ f)" and
  seq': "f ≤ ⟨β⟩ f ⟹ ⟨α ; β⟩ f ≤ ⟨α⟩ (⟨β⟩ f)" and
  seq_cont: "⟨α ; β⟩ f ≤ ⟨α⟩ (⟨β⟩ f)" and
  scale: "⟨α⟩ (s *$_R$ f) = s *$_R$ ⟨α⟩ f" and
  pos: "0 ≤ f ⟹ 0 ≤ ⟨α⟩ f" and
  mon: "f ≤ g ⟹ ⟨α⟩ f ≤ ⟨α⟩ g" and
  mon': "(⋀k. (H ⟶$_S$ f) k ω ≤ l)
       ⟹ (⟨(SDE b σ H) :: ('a, 'b) shp⟩ f) k ω ≤ l" and
  ind: "(⟨α⟩ g) ≤ g ⟹ (⟨SHP_star α⟩ g) ≤ g"
```

All of these axioms, apart from `scale`, are geared towards reasoning with inequalities. Proofs in SdL are generally concerned with proving that some target unsafe property remains below a threshold. Examples of such properties include the temperature of a component, execution time, and, when negated, distance to obstacles.

The first three axioms allow compositional reasoning involving sequential composition. Axiom `seq`

To enable probabilistic reasoning we have two axioms regarding our probabilistic operators – random assignment and SDE evolution.

```
axiomatization where
  distr: "𝒫(ω in M. SdL_fdia (SHP_rand_assign l X) f Z ω ∈ S ω) =
      ∫ω. indicator (S ω) (SdL_fdia (SHP_assign (λx. put$_1$ x (X ω))) f
Z ω) ∂M" and
  diff: "⟦∀ω. SdL_fdia α (SdL_imp H f) Z ω ≤ k*p; SdL_imp H f ≥ 0; SdL_imp
H (L b σ f) ≤ 0⟧ ⟹
```

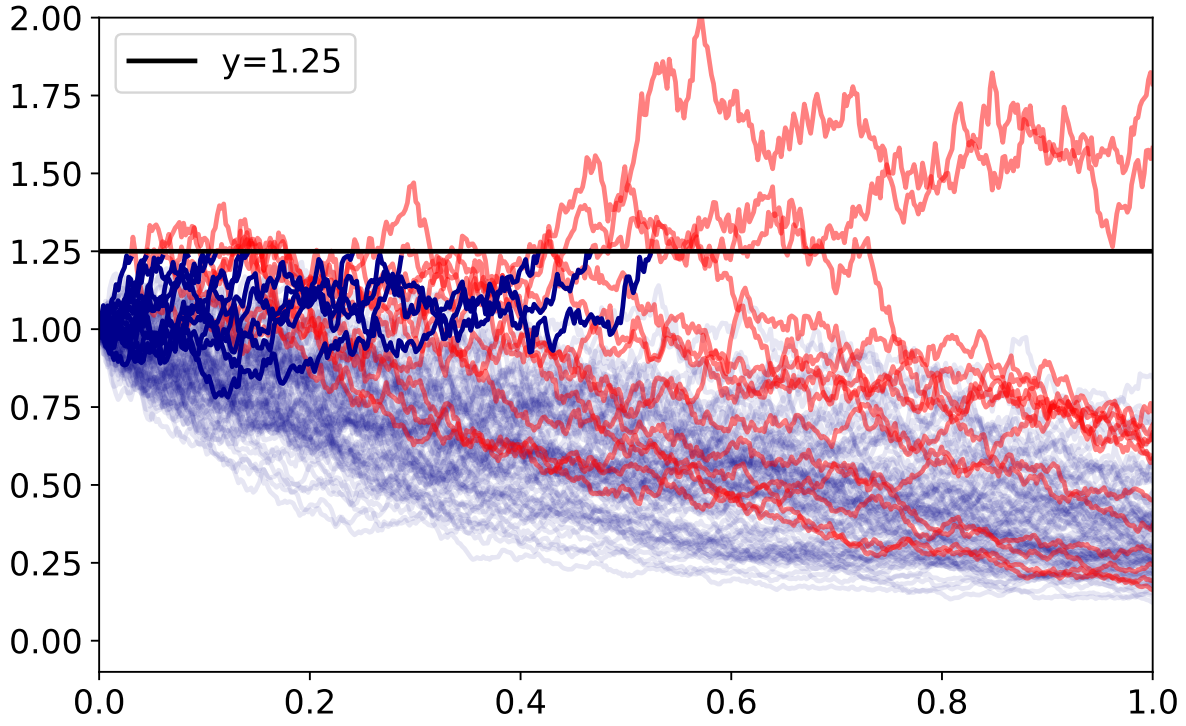


Figure 7.1: Illustration of diff rule for Geometric Brownian Motion with $k = 1.25$ and $p \leq 0.8$.

```
𝒫(ω in M. (SdL_fdia α (SdL_fdia (SDE b σ H) f) Z ω) ≥ k) ≤ p"
```

Here, we've axiomatised a measure space for which these axioms hold. The central axiom is diff, which lets us reason about the operation of an SDE without solving it. Instead, we need to prove properties about the differential generator L of the SDE, which is analogous to the derivative of an SDE. If we can show that the f-value of the SDE path has a negative derivative and the starting value is not too large, we can assert probability bounds on the maximum f-value. We also give an alternative form of the diff rule which uses sequential composition, rather than two diamonds.

```
lemma (in prob_space) diff':
  assumes "⋀ω. SdL_fdia α (SdL_imp H f) Z ω ≤ k*p" "SdL_imp H f ≥ 0"
"SdL_imp H (L b σ f) ≤ 0"
    shows "𝒫(ω in M. (SdL_fdia (α ; SDE b σ H) f) Z ω ≥ k) ≤ p"
```

In Figure 7.1 we give an example of an application of the diff rule to Geometric Brownian Motion (GBM), given by SDE $\mathrm{d}X_t = \mu X_t \,\mathrm{d}t + \sigma X_t \,\mathrm{d}W_t$.

When GBM has a negative drift, we can show that L is negative. We can then see that when the process is started at $kp = 1.25 * 0.8 = 1$, most of the paths in the figure stay below the threshold of k. The paths highlighted in red are those that have exceeded k at some point, and we can empirically verify that this is less than 1/5th of all paths - hence the diff rule holds.

7.6 Example

In this section, we use our tool to mechanise an example SdL proof first presented by Platzer [95, p.12]

Let $H = x^2 + y^2 < 10$:

$$P(\langle ?x^2 + y^2 \leq \frac{1}{3}; x := \frac{x}{2}; dx = \frac{-x}{2}dt - ydW, dy = \frac{-y}{2}dt + xdW \& H \rangle x^2 + y^2 \geq 1) \leq \frac{1}{3}$$

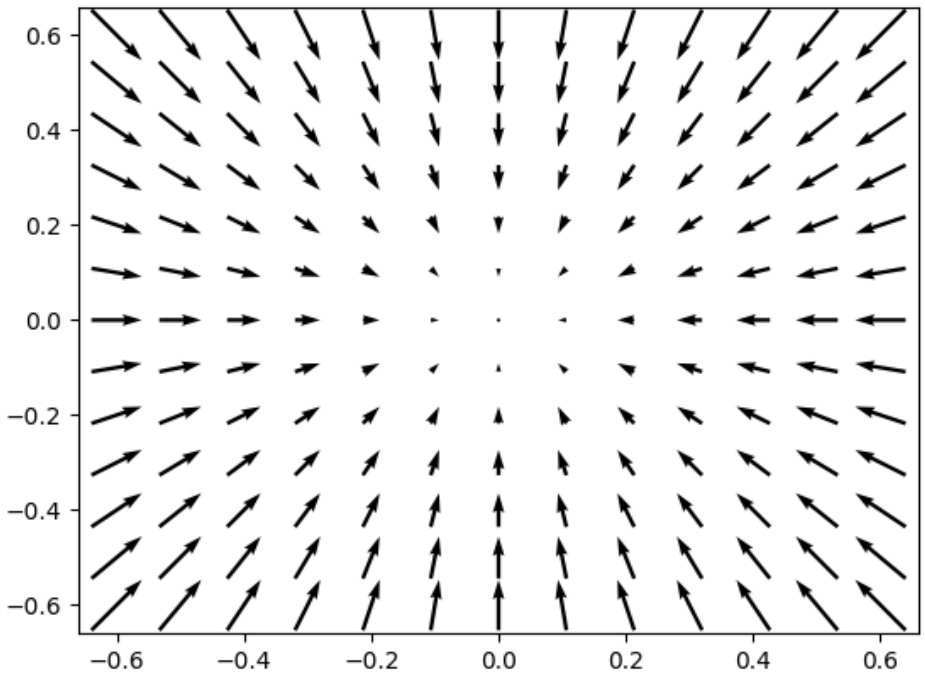


Figure 7.2: Drift of example SDE

In order to see this, we plot the drift of the SDE in Figure 7.2. This converges quickly to 0, and so it is only necessary to show that in most cases it will converge quickly enough to not exceed $x^2 + y^2 = 1$ despite the noise term.

We start by defining our variables using a dataspace — by using shallow expressions, we can reuse much of the tooling that this provides.

```
dataspace SdL_example =
  variables
    x :: real
    y :: real
```

In the context of this dataspace, we define the various parts of our system.

```
abbreviation "H ≡ (x² + y² < 10)ₑ"

abbreviation "example_SDE ≡ SDE
    (λ_. [x ⇝ -x/2, y ⇝ -y/2])
    (λ_ _ _. [x ⇝ -y, y ⇝ x])
    H"
```

H is the evolution domain for our SDE, which we define by giving the substitutions representing the differential equations directly.

```
definition "SHP_example ≡ ((SHP_test (x² + y² ≤ 1/3)ₑ
  ; SHP_assign [x ⇝ x/2])
  ; example_SDE)"
```

The SHP is defined as a series of compositions, where we give the expressions for test and assignment using the shallow expression package.

```
definition "SdL_system ≡ ⟨SHP_example⟩ (SdL_fun (x² + y²)ₑ)"
```

Finally, the SdL term requires us to lift an expression to the SdL type by using the `SdL_fun` constructor.

Now, we can restate the lemma in Isabelle:

```
lemma example:
  assumes "prob_space M"
  shows "𝒫(x in M. SdL_system Z x ≥ 1) ≤ 1/3"
  unfolding SdL_system_def SHP_example_def
proof (rule prob_space.diff'[OF assms])
```

We begin by applying rule `diff`. This gives us three subgoals:

$$H \longrightarrow f \geq 0 \tag{7.10}$$

$$\langle\alpha\rangle(H \longrightarrow f) \leq \lambda p \tag{7.11}$$

$$H \longrightarrow Lf \leq 0 \tag{7.12}$$

We can show the first subgoal by noting that we only need to show that f is positive, which can be handled by the auto prover. Proof method `transfer` unfolds definitions stated using `lift_definition` — in this case we undo the lifting from functions to SdL terms, and unfold the definition of $\leq$.

```
  let ?f = "SdL_fun (x² + y²)ₑ"
  show "0 ≤ H ⟶ₛ ?f"
```

For the second subgoal, we break down the term using the assignment rule and a chain of inequalities.

We begin by applying rule seq to turn the sequential composition diamond into two diamonds:

```
  have "⟨SHP_test (x² + y² ≤ 1/3)ₑ ; SHP_assign [x ⇝ x/2]⟩(H ⟶ₛ ?f)
≤
  ⟨SHP_test (x² + y² ≤ 1/3)ₑ⟩ (SdL_imp H ?f ⊔ ⟨SHP_assign [x ⇝ x/2]⟩ SdL_imp
H ?f)"
    by (rule seq)
```

We now have diamonds containing primitive operations, which we can simplify out.

```
  also have "... ≤ ⟨SHP_test (x² + y² ≤ 1/3)ₑ⟩ (SdL_imp H ?f)"
    apply (simp add: assign test)
    apply (insert indeps vwbs)
    unfolding SdL_imp_def apply (subst assign')
    apply transfer
    apply (expr_auto add: le_fun_def field_simps)
    done
```

Then, by transitivity, we get the following inequality:

```
  finally have *: "⟨SHP_test (($x)² + ($y)² ≤ 1 / 3)ₑ ; SHP_assign [x
↝ $x / 2]⟩ H ⟶S SdL_fun (($x)² + ($y)²)ₑ
  ≤ ⟨SHP_test (($x)² + ($y)² ≤ 1 / 3)ₑ⟩ H ⟶S SdL_fun (($x)² + ($y)²)ₑ"
.
```

We can show that the new term that we obtained satisfies our desired bound for any initial state and path parameter. We show this by simplifying the test diamond down and unfolding definitions.

```
  moreover have "⟨SHP_test (x² + y² ≤ 1/3)ₑ⟩ (H ⟶S ?f) Z ω ≤ 1 * (1/3)"
    apply (simp add: test SdL_imp_def)
    apply transfer
    apply auto
    done
```

Using these facts, we arrive at the required inequality by transitivity:

```
  ultimately show "(⟨SHP_test (($x)² + ($y)² ≤ 1 / 3)ₑ ; SHP_assign [x
↝ $x / 2]⟩
        (SdL_imp H (SdL_fun (($x)² + ($y)²)ₑ))) Z ω ≤ 1 * (1 / 3)"
    unfolding less_eq_sdl.rep_eq le_fun_def apply auto[1]
    apply (erule allE[where x=Z])
    apply (erule allE[where x=ω])
    by linarith
```

For our final subgoal, we need to supply the solution for the infinitesimal generator and verify that $Lf \leq 0$ holds.

```
  show "(H ⟶S L (λ_. [x ↝ - $x / 2, y ↝ - $y / 2])
        (λ_ _ _. [x ↝ - $y, y ↝ $x])
        (SdL_fun (($x)² + ($y)²)ₑ))
      ≤ 0"
```

As we have not mechanised L — the differential generator term — we rely on a pen-and-paper justification here:

$$\begin{aligned} Lf &= \frac{1}{2}\left(-x\frac{\partial f}{\partial x} - y\frac{\partial f}{\partial y} + y^2\frac{\partial^2 f}{\partial x^2} - 2xy\frac{\partial^2 f}{\partial x \partial y} + x^2\frac{\partial^2 f}{\partial y^2}\right) \\ &= \frac{1}{2}\left(-2x^2 - 2y^2 + 2y^2 + 2x^2\right) \\ &= 0 \end{aligned}$$

7.7 Related work

The mechanisation of dynamic logics for hybrid systems has received considerable attention, but, to our knowledge, always in the deterministic setting. The closest work is the formally verified differential dynamic logic of Bohrer et al. [15], who give a complete formalisation of dL — the deterministic ancestor of SdL — in both Coq and Isabelle/HOL, and use it to establish the soundness of the proof calculus underlying KeYmaera X [39]. Their work and ours share the goal of placing a Platzer-style hybrid-program logic on a machine-checked foundation, and both must contend with the semantics of differential equations within a prover. They differ in two key respects. First,

Bohrer et al. treat dL, whose continuous evolution is governed by ODEs and whose nondeterminism is set-valued; SdL instead evolves along SDEs and replaces nondeterministic choice with probabilistic choice, so its semantics are measure-theoretic, with formulae interpreted as real-valued random variables and the diamond denoting a supremum over stochastic evolutions. Second, they pursue a deep embedding with a concrete syntax in order to reason about the proof calculus itself, whereas we use a shallow embedding over a fixed semantic domain so as to reuse Isabelle's automation and probability libraries directly.

Within Isabelle/HOL, the IsaVODEs framework of Foster, Munive et al. [88, 34] mechanises a dL-style verification calculus for deterministic hybrid systems, derived foundationally from Isabelle's ODE libraries rather than postulated axiomatically. Our SdL implementation adopts the same shallow-expression infrastructure (Chapter 4), but extends it from the deterministic to the stochastic setting; conversely, IsaVODEs is foundational where our SDE constant and diamond axioms are, for now, axiomatic, pending the stochastic-analysis groundwork of Chapter 6.

The other logics proposed for stochastic hybrid systems — the stochastic Hybrid CSP of Wang et al. [114] and the extension of Modest by Hahn et al. [49] discussed in Chapter 2 — remain, to our knowledge, unmechanised. We are therefore not aware of any other theorem-prover formalisation of a *stochastic* dynamic logic: the deterministic case has been treated in both Coq and Isabelle/HOL, but the stochastic case, which this chapter addresses, has not.

7.8 Summary

In this chapter, we developed an axiomatic prover for stochastic hybrid systems. This is the first work of its kind, and provides a proof of concept that deductive verification can be applied to reasoning about systems with continuous state spaces and probabilistic behaviour.

By employing the shallow expressions library, we were able to define our constants in plain Isabelle instead of needing to construct a deep embedding. This makes the development small yet powerful — we were able to replicate the example from Platzer's theoretical paper in only 100 lines, excluding reasoning about the differential generator term.

By design, this logic can be given a fully formal background once the theory of SDEs has been formalised, the groundwork of which was laid in the previous chapter. With a mechanised background, we will be able to replace the SDE constant with a proper Isabelle definition, and the axioms can be replaced with Isabelle lemmas without affecting any results derived from them.

Chapter 8

Conclusion

In this thesis, we have laid the groundwork for the development of a toolset that enables the deductive verification of stochastic hybrid systems.

At the core of this goal is our SdL mechanisation, which serves as evidence that it is possible to prove meaningful properties about stochastic systems deductively. This development is backed up by our stochastic process mechanisation, which could be extended to provide a foundational basis for our proof system, in the same way that IsaVODEs has been.

To aid the mechanisation, and visualise our systems, we developed a language for modelling and simulating SHS. This tool will allow users to test their hypotheses before having to dedicate significant effort to formal verification. Further, by implementing the language in Haskell, using a monad for execution, and providing a semantics for the language, we enable a direct comparison between this language and the SHPs of our Isabelle SdL implementation, which makes the simulation results more trustworthy.

We posed three research questions in the introduction, and we are now in a position to address them:

RQ 1: How can we model uncertainty in the context of cyber-physical systems in a way that enables formal verification?

We have given evidence that while using nondeterminism to model uncertainty is possible and supported by existing tools, it comes with expressivity drawbacks that may lead to over-conservative models. By modelling uncertainty using probability instead, we allow for more accurate models which can provide probabilistic safety guarantees, which are often required by regulatory frameworks.

RQ 2: To what extent can we mechanise the mathematical foundations of stochastic process theory in Isabelle/HOL to support the verification of stochastic hybrid systems?

We have found that while mechanising the foundational theories is a significant undertaking, it is possible to implement advanced results in stochastic process theory such as the Kolmogorov-Chentsov theorem. While our formalisation does not yet provide a complete basis for verifying all SHS, it provides a rigorous framework that can be extended to include more advanced results.

The mechanisation process for this kind of mathematics is arduous but tractable, and the approach we have taken in this work can be applied to developing further results, including the potential construction of Brownian motion and eventually SDEs.

RQ 3: How can we develop a tool that will enable practical, compositional verification for these stochastic models?

We have addressed this by developing a framework centered around an implementation of SdL in Isabelle/HOL. Applying deductive verification to the domain of stochastic models lets us provide concrete safety guarantees while allowing us to model uncertainty accurately with SDEs. By giving compositional rules for the probabilistic diamond operator, our logic facilitates the verification of larger systems, and with the differential generator we can prove properties about SDE evolution without needing to solve or simulate them. The practicality of this tool is supplemented by our simulation language, which lets users model and visualise the behaviour of their systems ahead of verification, while using a language which mirrors the constructs of SdL's SHPs.

8.1 Future work

The material presented in this thesis can be extended in a number of ways, with the end goal of providing a powerful CPS verification tool for real-world systems.

We identify three main avenues for future work: extensions to the mathematical mechanisation work, improvements to the SdL logic and the simulation tool, and applications of the tools to the formal verification of CPS.

8.1.1 Mathematical mechanisation

The mechanisation we have presented in this thesis does not yet suffice to provide a complete foundation for SdL. A central article of future work is to complete it, so that SdL can be built entirely on the axioms of higher-order logic, with no further unfounded assumptions.

The immediate obstacle is the construction of Brownian motion. As discussed in Chapter 6, the route we attempted via iterated kernel products was blocked by an error in Klenke's Theorem 14.31, so the full construction remains open. However, the Kolmogorov-Chentsov theorem we have mechanised now opens a more direct path: one can use the Daniell-Kolmogorov extension theorem [64] to build a process with the correct Gaussian finite-dimensional distributions, and then apply Kolmogorov-Chentsov to obtain an almost-surely continuous modification, which is precisely Brownian motion. The required moment bound is immediate for Gaussian increments.

Building on Brownian motion, the next step is a theory of stochastic integration. This means mechanising the Itô integral, which we have discussed throughout this thesis as the construct that gives meaning to SDEs. The integral is defined as a limit of sums in which the integrand is evaluated at the left endpoint of each subinterval, and so its formalisation rests directly on the theory of martingales developed by Keskin et al. [68], whose definitions are compatible with ours, together with the Itô isometry that establishes the integral as a limit in L^2. Once the integral is in place, one can define the solution of an SDE as the process satisfying the corresponding integral equation, prove existence and uniqueness under Lipschitz and growth conditions on the drift and noise coefficients, and establish Itô's formula, which is the chain rule for stochastic calculus and the result from which the differential generator is derived. This would let us replace the axiomatic basis of SdL with a foundational one, and would also benefit from completing the product-kernel construction, which remains valuable in its own right for characterising and constructing Markov processes.

Finally, a full mechanisation of the differential generator L would remove the last pen-and-paper step in our SdL case study (Chapter 7), where $Lf \leq 0$ is currently justified by hand. With Brow-

nian motion, stochastic integration, and the generator all mechanised, the SDE constant and the diamond axioms of SdL can be replaced by Isabelle definitions and derived lemmas, without invalidating any results already proved on top of them.

8.1.2 Improvements to the SdL tool

Despite being applicable to the mechanisation of a toy example, there is a long way to go before the SdL implementation can be used for industrial verification.

A first task is to complete the mechanisation of the language constructs that we have only partially treated. In particular, the Kleene star is currently defined but its reasoning principles are underdeveloped, and the diamond operator's treatment of unbounded executions deserves attention — a discounted infinite-horizon semantics may offer a more tractable account of non-terminating systems than the present formulation.

A second task is proof automation. Our SdL case study is short, but each step is currently driven by hand. We expect that the bespoke Eisbach methods we developed for IsaVODEs, such as `dInv` and `dWeaken` (Chapter 3), can be adapted to discharge the recurring patterns in SdL proofs, in particular the application of the `diff` rule and the simplification of diamonds over primitive operations. Extending our computer algebra system integration (Section 3.2) to supply and certify solutions to SDEs, mirroring the `find_local_flow` workflow for ODEs, would further reduce manual effort.

A third task is to enable genuinely compositional reasoning over larger programs. The theory of scenes and scene spaces developed in Chapter 4 was motivated precisely by this need: it lets us characterise the sets of bound and free variables of a program within the logic, which is a prerequisite for the substitution-based compositional rules that make logics such as dL scale. Integrating frames into SdL would let us state and prove such rules, allowing the verification of larger systems by decomposition.

8.1.3 Improvements to the simulation tool

Regarding the simulation tool, future work includes adding support for statistical model checking, which would allow for producing certainty bounds for the results obtained from simulations. To improve performance for large-scale simulations, the modelling language could be compiled rather than interpreted, for example by using Haskell's LLVM integration. There are also more advanced simulation techniques that could be investigated, such as zero-crossing detection for more accurate simulation of hybrid dynamics.

The simulator could also be related more directly to the logic. Our Haskell semantics mirror the SHPs of SdL, so one option is to generate the interpreter from the Isabelle definitions via code generation; another is to prove the interpreter sound with respect to the SdL semantics. Either would let simulation results be used as evidence alongside the formal development, rather than as a separate artefact.

8.1.4 Applications of the work

Finally, we aim to apply our framework to more complex and diverse case studies in the domain of autonomous systems, in order to further validate its effectiveness and identify areas for improvement in both the logic and the supporting tools. The case studies in this thesis are deliberately small, and larger examples — multi-agent collision avoidance, navigation under sensor noise, or

systems with several interacting controllers — would test whether the compositional rules and proof automation discussed above scale as intended.

In particular, we are interested in exploring how our approach can be integrated into existing development workflows for safety-critical systems. The model-simulate-verify workflow is intended to fit alongside the tools that practitioners already use, and connecting it to robotics middleware such as ROS would let the models be derived from, or checked against, deployed controllers, rather than being maintained separately.

Bibliography

[1] A. Miyazawa, P. Ribeiro, L. Wei, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock. "RoboChart: modelling and verification of the functional behaviour of robotic applications". In: *Software & Systems Modeling* 18 (Jan. 23, 2019), pp. 3097–3149. issn: 1619-1374. doi: 10.1007/s10270-018-00710-z. url: https://doi.org/10.1007/s10270-018-00710-z.

[2] Aakash Abhishek, Harry Sood, and Jean-Baptiste Jeannin. "Formal verification of braking while swerving in automobiles". In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*. 27. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–11. isbn: 9781450370189. url: https://doi.org/10.1145/3365365.3382217 (visited on 05/25/2022).

[3] Julius Adelt, Timm Liebrenz, and Paula Herber. "Formal Verification of Intelligent Hybrid Systems that are Modeled with Simulink and the Reinforcement Learning Toolbox". en. In: *Formal Methods*. Ed. by Marieke Huisman, Corina Păsăreanu, and Naijun Zhan. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 349–366. isbn: 9783030908706. doi: 10.1007/978-3-030-90870-6_19.

[4] Reynald Affeldt. "An Introduction to MathComp-Analysis". In: (2025).

[5] Thomas Ammer and Peter Lammich. "van Emde Boas Trees". In: *Archive of Formal Proofs* (Nov. 2021). https://isa-afp.org/entries/Van_Emde_Boas_Trees.html, Formal proof development. issn: 2150-914x.

[6] Mehrnoosh Askarpour. "Safer-HRC: a methodology for safety assessment through formal verification in human-robot collaboration". In: (2018).

[7] Mehrnoosh Askarpour, Dino Mandrioli, Matteo Rossi, and Federico Vicentini. "Formal model of human erroneous behavior for safety analysis in collaborative robotics". en. In: *Robotics and Computer-Integrated Manufacturing* 57 (June 2019), pp. 465–476. issn: 0736-5845. doi: 10.1016/j.rcim.2019.01.001. url: https://www.sciencedirect.com/science/article/pii/S0736584518303247 (visited on 05/25/2022).

[8] Mehrnoosh Askarpour, Dino Mandrioli, Matteo Rossi, and Federico Vicentini. "SAFER-HRC: Safety Analysis Through Formal vERification in Human-Robot Collaboration". en. In: *Computer Safety, Reliability, and Security*. Ed. by Amund Skavhaug, Jérémie Guiochet, and Friedemann Bitsch. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 283–295. isbn: 9783319454771. doi: 10.1007/978-3-319-45477-1_22.

[9] Béatrice Bérard, Pascal Lafourcade, Laure Millet, Maria Potop-Butucaru, Yann Thierry-Mieg, and Sébastien Tixeuil. "Formal verification of mobile robot protocols". en. In: *Distributed Computing* 29.6 (Nov. 2016), pp. 459–487. issn: 1432-0452. doi: 10.1007/s00446-016-0271-1. url: https://doi.org/10.1007/s00446-016-0271-1 (visited on 05/25/2022).

[10] Cinzia Bernardeschi, Pierpaolo Dini, Andrea Domenici, Maurizio Palmieri, and Sergio Saponara. "Formal Verification and Co-Simulation in the Design of a Synchronous Motor Control Algorithm". en. In: *Energies* 13.16 (Jan. 2020), p. 4057. issn: 1996-1073. doi: 10.3390/en13164057. url: https://www.mdpi.com/1996-1073/13/16/4057 (visited on 05/25/2022).

[11] Cinzia Bernardeschi, Andrea Domenici, and Paolo Masci. "A PVS-Simulink Integrated Environment for Model-Based Analysis of Cyber-Physical Systems". In: *IEEE Transactions on Software Engineering* 44.6 (June 2018), pp. 512–533. issn: 1939-3520. doi: 10.1109/TSE.2017.2694423.

[12] S. Bernardi, U. Gentile, S. Marrone, J. Merseguer, and R. Nardone. "Security modelling and formal verification of survivability properties: Application to cyber–physical systems". en. In: *Journal of Systems and Software* 171 (Jan. 2021), p. 110746. issn: 0164-1212. doi: 10.1016/j.jss.2020.110746. url: https://www.sciencedirect.com/science/article/pii/S0164121220301710 (visited on 05/25/2022).

[13] Oliver Biggar and Mohammad Zamani. "A Framework for Formal Verification of Behavior Trees With Linear Temporal Logic". In: *IEEE Robotics and Automation Letters* 5.2 (Apr. 2020), pp. 2341–2348. issn: 2377-3766. doi: 10.1109/LRA.2020.2970634.

[14] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. "Hammering towards QED". In: *Journal of Formalized Reasoning* 9.1 (2016).

[15] Rose Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp, and André Platzer. "Formally verified differential dynamic logic". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. 2017, pp. 208–221.

[16] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. "Coquelicot: A user-friendly library of real analysis for Coq". In: *Mathematics in Computer Science* 9.1 (2015), pp. 41–62.

[17] Davide Bresolin, Luca Geretti, Riccardo Muradore, Paolo Fiorini, and Tiziano Villa. "Formal verification of robotic surgery tasks by reachability analysis". en. In: *Microprocessors and Microsystems* 39.8 (Nov. 2015), pp. 836–842. issn: 0141-9331. doi: 10.1016/j.micpro.2015.10.006. url: https://www.sciencedirect.com/science/article/pii/S014193311500160X (visited on 05/25/2022).

[18] Manuela L Bujorianu and John Lygeros. "Reachability questions in piecewise deterministic Markov processes". In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2003, pp. 126–140.

[19] Matthew Chan, Daniel Ricketts, Sorin Lerner, and Gregory Malecha. "Formal verification of stability properties of cyber-physical systems". In: *Proc. CoqPL* (2016).

[20] Benjamin J. Choi, Juyoun Park, and Chung Hyuk Park. "Formal Verification for Human-Robot Interaction in Medical Environments". In: *Companion of the 2021 ACM/IEEE International Conference on Human-Robot Interaction*. HRI '21 Companion. New York, NY, USA: Association for Computing Machinery, Mar. 2021, pp. 181–185. isbn: 9781450382908. doi: 10.1145/3434074.3447155. url: https://doi.org/10.1145/3434074.3447155 (visited on 05/25/2022).

[21] Han Choi, Sungdeok Cha, Jae Yeon Jo, Junbeom Yoo, Hae Young Lee, and Won Tae Kim. "Formal verification of basic DEV&DESS formalism using hytech". In: *Information (Japan)* 16.1 B (Jan. 2013), pp. 821–826. issn: 1343-4500. url: http://www.scopus.com/inward/record.url?scp=84874051403&partnerID=8YFLogxK (visited on 05/25/2022).

[22] Luís Diogo Couto, Stylianos Basagiannis, El Hassan Ridouane, Alie El-Din Mady, Miran Hasanagic, and Peter Gorm Larsen. "Injecting Formal Verification in FMI-Based Co-simulations of Cyber-Physical Systems". en. In: *Software Engineering and Formal Methods*. Ed. by Antonio Cerone and Marco Roveri. Lecture Notes in Computer Science. Cham: Springer Interna-

tional Publishing, 2018, pp. 284–299. isbn: 9783319747811. doi: 10.1007/978-3-319-74781-1_20.

[23] Mark HA Davis. "Piecewise-deterministic Markov processes: A general class of non-diffusion stochastic models". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 46.3 (1984), pp. 353–376.

[24] Louise Dennis, Michael Fisher, Marija Slavkovik, and Matt Webster. "Formal verification of ethical choices in autonomous systems". en. In: *Robotics and Autonomous Systems* 77 (Mar. 2016), pp. 1–14. issn: 0921-8890. doi: 10.1016/j.robot.2015.11.012. url: https://www.sciencedirect.com/science/article/pii/S0921889015003000 (visited on 05/25/2022).

[25] Clare Dixon, Matt Webster, Joe Saunders, Michael Fisher, and Kerstin Dautenhahn. ""The Fridge Door is Open"–Temporal Verification of a Robotic Assistant's Behaviours". en. In: *Advances in Autonomous Robotics Systems*. Ed. by Michael Mistry, Aleš Leonardis, Mark Witkowski, and Chris Melhuish. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 97–108. isbn: 9783319104010. doi: 10.1007/978-3-319-10401-0_9.

[26] Szymon Dolecki and Frédéric Mynard. *Convergence foundations of topology*. World Scientific Publishing Company, 2016.

[27] Chris Dornan and Simon Marlow. *Alex: A Lexical Analyser Generator*. url: https://hackage.haskell.org/package/alex (visited on 01/11/2024).

[28] Dmitrii Drozdov, Sandeep Patil, Victor Dubinin, and Valeriy Vyatkin. "Formal verification of cyber-physical automation systems modelled with timed block diagrams". In: *2016 IEEE 25th International Symposium on Industrial Electronics (ISIE)*. ISSN: 2163-5145. June 2016, pp. 316–321. doi: 10.1109/ISIE.2016.7744910.

[29] Dmitrii Drozdov, Sandeep Patil, Victor Dubinin, and Valeriy Vyatkin. "Towards formal verification for cyber-physically agnostic software: A case study". In: *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*. Oct. 2017, pp. 5509–5514. doi: 10.1109/IECON.2017.8216953.

[30] Mnacho Echenim, Hervé Guiol, and Nicolas Peltier. "Formalizing the Cox–Ross–Rubinstein Pricing of European Derivatives in Isabelle/HOL". en. In: *Journal of Automated Reasoning* 64.4 (Apr. 2020), pp. 737–765. issn: 1573-0670. doi: 10.1007/s10817-019-09528-w. url: https://doi.org/10.1007/s10817-019-09528-w (visited on 08/31/2022).

[31] Huixing Fang, Jian Guo, Huibiao Zhu, and Jianqi Shi. "Formal Verification and Simulation: Co-verification for Subway Control Systems". In: *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*. July 2012, pp. 145–152. doi: 10.1109/TASE.2012.11.

[32] Marie Farrell, Matthew Bradbury, Michael Fisher, Louise A. Dennis, Clare Dixon, Hu Yuan, and Carsten Maple. "Using Threat Analysis Techniques to Guide Formal Verification: A Case Study of Cooperative Awareness Messages". en. In: *Software Engineering and Formal Methods*. Ed. by Peter Csaba Ölveczky and Gwen Salaün. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 471–490. isbn: 9783030304461. doi: 10.1007/978-3-030-30446-1_25.

[33] Simon Foster, James Baxter, Ana Cavalcanti, Jim Woodcock, and Frank Zeyda. "Unifying semantic foundations for automated verification tools in Isabelle/UTP". In: *Science of Computer Programming* 197 (2020), p. 102510. issn: 0167-6423. doi: https://doi.org/10.1016/j.scico.2020.102510. url: https://www.sciencedirect.com/science/article/pii/S0167642320301192.

[34] Simon Foster, Jonathan Julián Huerta y Munive, Mario Gleirscher, and Georg Struth. "Hybrid Systems Verification with Isabelle/HOL: Simpler Syntax, Better Models, Faster Proofs". In: *FM2021* (June 2021). arXiv: 2106.05987 [cs.LO].

[35] Mohammed Foughali. "Formal verification of the functionnal layer of robotic and autonomous systems". en. PhD thesis. INSA de Toulouse, Dec. 2018. url: https://hal.laas.fr/tel-02080063 (visited on 05/25/2022).

[36] Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Pierre-Emmanuel Hladik, Félix Ingrand, and Anthony Mallet. "Formal Verification of Complex Robotic Systems on Resource-Constrained Platforms". In: *2018 IEEE/ACM 6th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. ISSN: 2575-5099. May 2018, pp. 2–9.

[37] Mohammed Foughali and Pierre-Emmanuel Hladik. "Bridging the gap between formal verification and schedulability analysis: The case of robotics". en. In: *Journal of Systems Architecture* 111 (Dec. 2020), p. 101817. issn: 1383-7621. doi: 10.1016/j.sysarc.2020.101817. url: https://www.sciencedirect.com/science/article/pii/S1383762120301090 (visited on 05/25/2022).

[38] 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: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 379–395.

[39] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. "KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems". In: *CADE*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. LNCS. Springer, 2015, pp. 527–538. doi: 10.1007/978-3-319-21401-6_36.

[40] Amjad Gawanmeh, Ali Alwadi, and Sazia Parvin. "Formal Verification of Control Strategies for a Cyber Physical System". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. ISSN: 2332-5666. June 2017, pp. 91–96. doi: 10.1109/ICDCSW.2017.59.

[41] Luca Geretti, Riccardo Muradore, Davide Bresolin, Paolo Fiorini, and Tiziano Villa. "Parametric formal verification: the robotic paint spraying case study". en. In: *IFAC-PapersOnLine*. 20th IFAC World Congress 50.1 (July 2017), pp. 9248–9253. issn: 2405-8963. doi: 10.1016/j.ifacol.2017.08.1287. url: https://www.sciencedirect.com/science/article/pii/S2405896317318050 (visited on 05/25/2022).

[42] Vasileios Germanos and Emanuele Lindo Secco. "Formal Verification of Robotics Navigation Algorithms". In: *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. Aug. 2016, pp. 177–180. doi: 10.1109/CSE-EUC-DCABES.2016.181.

[43] Mrinal K Ghosh, Aristotle Arapostathis, and Steven I Marcus. "Ergodic control of switching diffusions". In: *SIAM Journal on Control and Optimization* 35.6 (1997), pp. 1952–1988.

[44] Jeremy Gibbons. "Functional programming for domain-specific languages". In: *Central European Functional Programming School*. Springer, 2013, pp. 1–28.

[45] Andy Gill and Simon Marlow. *happy: Happy is a parser generator for Haskell*. url: https://hackage.haskell.org/package/happy (visited on 01/11/2024).

[46] Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese, Francesco Tiezzi, Carlo Pinciroli, Manuele Brambilla, Mauro Birattari, and Marco Dorigo. "Towards a Formal Verification Methodology for Collective Robotic Systems". en. In: *Formal Methods and Software Engineering*. Ed. by Toshiaki Aoki and Kenji Taguchi. Lecture Notes in Computer Science. Berlin,

Heidelberg: Springer, 2012, pp. 54–70. isbn: 9783642342813. doi: 10.1007/978-3-642-34281-3_7.

[47] Dario Guidotti, Francesco Leofante, Armando Tacchella, and Claudio Castellini. "Improving Reliability of Myocontrol Using Formal Verification". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 27.4 (Apr. 2019), pp. 564–571. issn: 1558-0210. doi: 10.1109/TNSRE.2019.2893152.

[48] Florian Haftmann and Makarius Wenzel. "Local theory specifications in Isabelle/Isar". In: *Types for Proofs and Programs: International Conference, TYPES 2008 Torino, Italy, March 26-29, 2008 Revised Selected Papers*. Springer. 2009, pp. 153–168.

[49] Ernst Moritz Hahn, Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen. "A compositional modelling and analysis framework for stochastic hybrid systems". In: *Formal Methods in System Design* 43.2 (Aug. 2012), pp. 191–232. doi: 10.1007/s10703-012-0167-z.

[50] Raju Halder, José Proença, Nuno Macedo, and André Santos. "Formal Verification of ROS-Based Robotic Applications Using Timed-Automata". In: *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. May 2017, pp. 44–50. doi: 10.1109/FormaliSE.2017.9.

[51] John Harrison. "HOL light: An overview". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 60–66.

[52] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. "The probabilistic model checker Storm". In: *International Journal on Software Tools for Technology Transfer* 24.4 (2022), pp. 589–610.

[53] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. "HyTech: A model checker for hybrid systems". In: *International Journal on Software Tools for Technology Transfer* 1.1-2 (1997), pp. 110–122.

[54] Thomas A. Henzinger. "The Theory of Hybrid Automata". In: *Verification of Digital and Hybrid Systems*. Ed. by M. Kemal Inan and Robert P. Kurshan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 265–292. isbn: 978-3-642-59615-5. doi: 10.1007/978-3-642-59615-5_13. url: https://doi.org/10.1007/978-3-642-59615-5_13.

[55] Paula Herber, Julius Adelt, and Timm Liebrenz. "Formal Verification of Intelligent Cyber-Physical Systems with the Interactive Theorem Prover KeYmaera X." In: *Software Engineering (Satellite Events)*. 2021.

[56] Michikazu Hirata, Yasuhiko Minamide, and Tetsuya Sato. "Semantic Foundations of Higher-Order Probabilistic Programs in Isabelle/HOL". In: *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Ed. by Adam Naumowicz and René Thiemann. Vol. 268. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 18:1–18:18. isbn: 978-3-95977-284-6. doi: 10.4230/LIPIcs.ITP.2023.18. url: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.18.

[57] Johannes Hölzl. "Construction and Stochastic Applications of Measure Spaces in Higher-Order Logic". PhD thesis. Universität München, 2013.

[58] Johannes Hölzl. "Markov processes in Isabelle/HOL". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. 2017, pp. 100–111.

[59] Johannes Hölzl and Armin Heller. "Three chapters of measure theory in Isabelle/HOL". In: *International Conference on Interactive Theorem Proving*. Springer. 2011, pp. 135–151.

[60] Jianghai Hu, John Lygeros, and Shankar Sastry. "Towards a theory of stochastic hybrid systems". In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2000, pp. 160–173.

[61] Chao Huang, Wenchao Li, and Qi Zhu. "Formal verification of weakly-hard systems". In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. HSCC '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 197–207. isbn: 9781450362825. doi: 10.1145/3302504.3311811. url: https://doi.org/10.1145/3302504.3311811 (visited on 05/25/2022).

[62] Wei Huang, Yifan Zhou, Youcheng Sun, Alec Banks, Jie Meng, James Sharp, Simon Maskell, and Xiaowei Huang. "Formal Verification of Robustness and Resilience of Learning-Enabled State Estimation Systems for Robotics". In: *arXiv preprint arXiv:2010.08311* (2020).

[63] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. "The coq proof assistant a tutorial". In: *Rapport Technique* 178 (1997), p. 113.

[64] Fabian Immler. "Generic construction of probability spaces for paths of stochastic processes in Isabelle/HOL". In: *Master's thesis, TU München (October 2012)* (2012).

[65] Isabelle Development Team. *Isabelle*. 2023. url: https://isabelle.in.tum.de/index.html.

[66] Seok Pil Jang and Stephen US Choi. "Role of Brownian motion in the enhanced thermal conductivity of nanofluids". In: *Applied physics letters* 84.21 (2004), pp. 4316–4318.

[67] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Ryan Gardner, Aurora Schmidt, Erik Zawadzki, and Andre Platzer. "Formal verification of ACAS X, an industrial airborne collision avoidance system". In: *2015 International Conference on Embedded Software (EMSOFT)*. Oct. 2015, pp. 127–136. doi: 10.1109/EMSOFT.2015.7318268.

[68] Ata Keskin, Tobias Nipkow, and Katharina Kreuzer. "On the Formalization of Martingales". BA thesis. Technische Universität München, Sept. 15, 2023.

[69] Lisa Kiekbusch, Christopher Armbrust, and Karsten Berns. "Formal Verification of Behaviour Networks Including Hardware Failures". en. In: *Intelligent Autonomous Systems 13*. Ed. by Emanuele Menegatti, Nathan Michael, Karsten Berns, and Hiroaki Yamaguchi. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2016, pp. 1571–1582. isbn: 9783319083384. doi: 10.1007/978-3-319-08338-4_113.

[70] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[71] Achim Klenke. *Probability theory: a comprehensive course*. 3rd. Springer Science & Business Media, 2020.

[72] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[73] Morteza Lahijanian, Sean B. Andersson, and Calin Belta. "Formal Verification and Synthesis for Discrete-Time Stochastic Systems". In: *IEEE Transactions on Automatic Control* 60.8 (Aug. 2015), pp. 2031–2045. issn: 1558-2523. doi: 10.1109/TAC.2015.2398883.

[74] Ralf Lämmel and Simon Peyton Jones. "Scrap your boilerplate: a practical design pattern for generic programming". In: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. TLDI '03. New Orleans, Louisiana, USA: Association for Computing Machinery, 2003, pp. 26–37. isbn: 1581136498. doi: 10.1145/604174.604179. url: https://doi.org/10.1145/604174.604179.

[75] Livia Lestingi, Mehrnoosh Askarpour, Marcello M. Bersani, and Matteo Rossi. "Formal Verification of Human-Robot Interaction in Healthcare Scenarios". en. In: *Software Engineering and Formal Methods*. Ed. by Frank de Boer and Antonio Cerone. Lecture Notes

in Computer Science. Cham: Springer International Publishing, 2020, pp. 303–324. isbn: 9783030587680. doi: 10.1007/978-3-030-58768-0_17.

[76] Liming Li, Zhiping Shi, Yong Guan, Chunna Zhao, Jie Zhang, and Hongxing Wei. "Formal verification of a collision-free algorithm of dual-arm robot in HOL4". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. ISSN: 1050-4729. May 2014, pp. 1380–1385. doi: 10.1109/ICRA.2014.6907032.

[77] Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. "A calculus for hybrid CSP". In: *Asian Symposium on Programming Languages and Systems*. Springer. 2010, pp. 1–15.

[78] Sarah M. Loos, David Renshaw, and André Platzer. "Formal verification of distributed aircraft controllers". In: *Proceedings of the 16th international conference on Hybrid systems: computation and control*. HSCC '13. New York, NY, USA: Association for Computing Machinery, Apr. 2013, pp. 125–130. isbn: 9781450315678. doi: 10.1145/2461328.2461350. url: https://doi.org/10.1145/2461328.2461350 (visited on 05/25/2022).

[79] Zhihao Lu, Rui Wang, and Yong Guan. "Formal verification of discrete event model". In: *Proceedings of the 4th ACM SIGSOFT International Workshop on Testing, Analysis, and Verification of Cyber-Physical Systems and Internet of Things*. TAV-CPS/IoT 2020. New York, NY, USA: Association for Computing Machinery, July 2020, pp. 3–4. isbn: 9781450380324. doi: 10.1145/3402842.3407159. url: https://doi.org/10.1145/3402842.3407159 (visited on 05/25/2022).

[80] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. "Formal Specification and Verification of Autonomous Robotic Systems: A Survey". In: *ACM Comput. Surv.* 52.5 (Sept. 2019). doi: 10.1145/3342355. url: https://doi.org/10.1145/3342355.

[81] Toni Mancini, Federico Mari, Annalisa Massini, Igor Melatti, Fabio Merli, and Enrico Tronci. "System Level Formal Verification via Model Checking Driven Simulation". en. In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 296–312. isbn: 9783642397998. doi: 10.1007/978-3-642-39799-8_21.

[82] Toni Mancini, Federico Mari, Annalisa Massini, Igor Melatti, and Enrico Tronci. "SyLVaaS: System Level Formal Verification as a Service *". en. In: *Fundamenta Informaticae* 149.1-2 (Jan. 2016), pp. 101–132. issn: 0169-2968. doi: 10.3233/FI-2016-1444. url: https://ieeexplore.ieee.org/document/7092763 (visited on 05/25/2022).

[83] Lenaertz Mikaël. "Formal verification of flexibility in swarm robotics". PhD thesis. Citeseer, 2012.

[84] Stefan Mitsch, Khalil Ghorbal, David Vogelbacher, and André Platzer. "Formal verification of obstacle avoidance and navigation of ground robots". en. In: *The International Journal of Robotics Research* 36.12 (Oct. 2017), pp. 1312–1340. issn: 0278-3649. doi: 10.1177/0278364917733549. url: https://doi.org/10.1177/0278364917733549 (visited on 05/25/2022).

[85] Stefan Mitsch, Sarah M. Loos, and Andre Platzer. "Towards Formal Verification of Freeway Traffic Control". In: *2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*. Apr. 2012, pp. 171–180. doi: 10.1109/ICCPS.2012.25.

[86] Stefan Mitsch and André Platzer. "ModelPlex: verified runtime validation of verified cyber-physical system models". In: *Formal Methods in System Design* 49.1 (2016), pp. 33–74. doi: 10.1007/s10703-016-0241-z. url: https://doi.org/10.1007/s10703-016-0241-z.

[87] Stefan Mitsch and André Platzer. "Verified Runtime Model Validation for Partially Observable Hybrid Systems". In: 2018.

[88] Jonathan Julián Huerta y Munive, Simon Foster, Mario Gleirscher, Georg Struth, Christian Pardillo Laursen, and Thomas Hickman. *Scalable Automated Verification for Cyber-Physical Systems in Isabelle/HOL*. 2024. arXiv: 2401.12061 [cs.LO].

[89] Masaki Nakamura, Kazutoshi Sakakibara, Yuki Okura, and Kazuhiro Ogata. "Formal Verification of Multitask Hybrid Systems by the OTS/CafeOBJ Method". In: *International Journal of Software Engineering and Knowledge Engineering* 31.11n12 (Dec. 2021), pp. 1541–1559. issn: 0218-1940. doi: 10.1142/S0218194021400118. url: https://www.worldscientific.com/doi/abs/10.1142/S0218194021400118 (visited on 05/25/2022).

[90] Bernt Oksendal. *Stochastic differential equations: an introduction with applications*. Springer Science & Business Media, 2013.

[91] Maury FM Osborne. "Brownian motion in the stock market". In: *Operations research* 7.2 (1959), pp. 145–173.

[92] Sam Owre, John M Rushby, and Natarajan Shankar. "PVS: A prototype verification system". In: *International Conference on Automated Deduction*. Springer. 1992, pp. 748–752.

[93] André Platzer. "A Uniform Substitution Calculus for Differential Dynamic Logic". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 467–481. isbn: 978-3-319-21401-6.

[94] André Platzer. "Differential Dynamic Logic for Hybrid Systems." In: *J. Autom. Reas.* 41.2 (2008), pp. 143–189. issn: 0168-7433. doi: 10.1007/s10817-008-9103-8.

[95] André Platzer. *Stochastic Differential Dynamic Logic for Stochastic Hybrid Programs*. Tech. rep. CMU-CS-11-111. School of Computer Science, Carnegie Mellon University, 2011.

[96] André Platzer. "Stochastic Differential Dynamic Logic for Stochastic Hybrid Programs". In: *CADE*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Vol. 6803. LNCS. Springer, 2011, pp. 446–460. doi: 10.1007/978-3-642-22438-6_34.

[97] G. Pola, M.L. Bujorianu, J. Lygeros, and M.D. Di Benedetto. "Stochastic Hybrid Models: An Overview". In: *IFAC Proceedings Volumes* 36.6 (June 2003), pp. 45–50. doi: 10.1016/s1474-6670(17)36405-4.

[98] Ameya Pore, Davide Corsi, Enrico Marchesini, Diego Dall'Alba, Alicia Casals, Alessandro Farinelli, and Paolo Fiorini. "Safe Reinforcement Learning using Formal Verification for Tissue Retraction in Autonomous Robotic-Assisted Surgery". In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. ISSN: 2153-0866. Sept. 2021, pp. 4025–4031. doi: 10.1109/IROS51168.2021.9636175.

[99] David Porfirio, Allison Sauppé, Aws Albarghouthi, and Bilge Mutlu. "Authoring and Verifying Human-Robot Interactions". In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. UIST '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 75–86. isbn: 9781450359481. doi: 10.1145/3242587.3242634. url: https://doi.org/10.1145/3242587.3242634 (visited on 05/25/2022).

[100] Adithya T. Praveen, Anubhav Gupta, Siddhartha Bhattacharyya, and Raja Muthalagu. "Assuring Behavior of Multirobot Autonomous Systems With Translation From Formal Verification to ROS Simulation". In: *IEEE Systems Journal* (2022), pp. 1–9. issn: 1937-9234. doi: 10.1109/JSYST.2022.3149677.

[101] Adnan Rashid and Osman Hasan. "Formal Verification of Robotic Cell Injection systems up to 4-DOF using HOL Light". en. In: *Formal Aspects of Computing* 32.2 (July 2020), pp. 229–250. issn: 1433-299X. doi: 10.1007/s00165-020-00514-3. url: https://doi.org/10.1007/s00165-020-00514-3 (visited on 05/25/2022).

[102] Michael Rathmair, Thomas Haspl, Titanilla Komenda, Bernhard Reiterer, and Michael Hofbaur. "A Formal Verification Approach for Robotic Workflows". In: *2021 20th International Conference on Advanced Robotics (ICAR)*. Dec. 2021, pp. 670–675. doi: 10.1109/ICAR53236.2021.9659366.

[103] Michael Rathmair, Christoph Luckeneder, Thomas Haspl, Berhnard Reiterer, Ralph Hoch, Michael Hofbaur, and Hermann Kaindl. "Formal Verification of Safety Properties of Collaborative Robotic Applications including Variability". In: *2021 30th IEEE International Conference on Robot & Human Interactive Communication (RO-MAN)*. IEEE. 2021, pp. 1283–1288.

[104] Muhammad Usman Sanwal and Osman Hasan. "Formal Verification of Cyber-Physical Systems: Coping with Continuous Elements". en. In: *Computational Science and Its Applications – ICCSA 2013*. Ed. by Beniamino Murgante, Sanjay Misra, Maurizio Carlini, Carmelo M. Torre, Hong-Quang Nguyen, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 358–371. isbn: 9783642396373. doi: 10.1007/978-3-642-39637-3_29.

[105] Viktor Shatrov and Valeriy Vyatkin. "Formal Verification of IEC 61499 Enhanced with Timed Events". en. In: *Technological Innovation for Life Improvement*. Ed. by Luis M. Camarinha-Matos, Nastaran Farhadi, Fábio Lopes, and Helena Pereira. IFIP Advances in Information and Communication Technology. Cham: Springer International Publishing, 2020, pp. 168–178. isbn: 9783030451240. doi: 10.1007/978-3-030-45124-0_16.

[106] Konrad Slind and Michael Norrish. "A brief overview of HOL4". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2008, pp. 28–32.

[107] Richard Stocker. "Towards the formal verification of human-agent-robot teamwork". PhD thesis. Citeseer, 2013.

[108] Anum Tahir, Kashif Saghar, Harris Bin Khalid, Umar Shadab Butt, Umar Shahbaz Khan, and Usman Asad. "Formal Verification and Development of an Autonomous Firefighting Robotic Model". In: *2019 International Conference on Robotics and Automation in Industry (ICRAI)*. Oct. 2019, pp. 1–6. doi: 10.1109/ICRAI47710.2019.8967388.

[109] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: ACM, Jan. 2020. doi: 10.1145/3372885.3373824. url: https://doi.org/10.1145/3372885.3373824.

[110] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probalistic robotics*. MIT Press, 2006.

[111] Federico Vicentini, Mehrnoosh Askarpour, Matteo G. Rossi, and Dino Mandrioli. "Safety Assessment of Collaborative Robotics Through Automated Formal Verification". In: *IEEE Transactions on Robotics* 36.1 (Feb. 2020), pp. 42–61. issn: 1941-0468. doi: 10.1109/TRO.2019.2937471.

[112] Rui Wang, Yong Guan, Houbing Song, Xinxin Li, Xiaojuan Li, Zhiping Shi, and Xiaoyu Song. "A Formal Model-Based Design Method for Robotic Systems". In: *IEEE Systems Journal* 13.1 (Mar. 2019), pp. 1096–1107. issn: 1937-9234. doi: 10.1109/JSYST.2018.2867285.

[113] Rui Wang, Yingxia Wei, Houbing Song, Yu Jiang, Yong Guan, Xiaoyu Song, and Xiaojuan Li. "From Offline Towards Real-Time Verification for Robot Systems". In: *IEEE Transactions on Industrial Informatics* 14.4 (Apr. 2018), pp. 1712–1721. issn: 1941-0050. doi: 10.1109/TII.2017.2788901.

[114] Shuling Wang, Naijun Zhan, and Lijun Zhang. "A Compositional Modelling and Verification Framework for Stochastic Hybrid Systems." In: *Formal Aspects Comput.* 29.4 (2017), pp. 751–775.

[115] Wolfram Research. *Expressions - Wolfram Language*. url: https://reference.wolfram.com/language/guide/Expressions.html.
[116] Wolfram Research, Inc. *Wolfram Engine*. https://www.wolfram.com/engine/. 2025.