

Sebastian Biewer

Software Doping – Theory and Detection

Software Doping

Theory and Detection

*Dissertation submitted towards the degree Doctor of Engineering (Dr.-Ing.) of the
Faculty of Mathematics and Computer Science of Saarland University*

Sebastian Biewer
Saarbrücken, 2023

Date of the colloquium: 25 May 2023
Dean of the faculty: Prof. Dr. Jürgen Steimle
Chair of the committee: Prof. Dr. Verena Wolf
Examination Board: Prof. Dr. Holger Hermanns
Prof. Dr. Pedro D'Argenio
Prof. Dr. Gerardo Schneider
Academic assistant: Dr. Andreas Schmidt

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <https://dnb.dnb.de> abrufbar.

© 2023 Sebastian Biewer

Dedicated to the memory of

Felix

2018 – 2022

Auch wenn du nicht mehr bei uns bist, lebst du durch all die Spuren, die du hinterlassen hast weiter. In den knapp fünf Jahren auf dieser Welt hast du bei sehr vielen Menschen Spuren hinterlassen. Wir werden dich immer in Erinnerung behalten und bei jedem Familientreffen an dich denken.

Meine Gedanken sind auch bei deinen Eltern, Carina und Simon, die so unfassbar stark sind und sich um deine drei Geschwister Carlotta, Leo und Paul genauso liebevoll kümmern, wie sie sich um dich gekümmert haben.

Sie haben das Symbol der Kerze gewählt, um sich von dir zu verabschieden und an dich zu denken. Jede Kopie dieser Dissertation enthält eine Kerze, die für dich brennt. In jeder Kopie dieser Dissertation hinterlässt du eine weitere Spur.



Acknowledgements

This endeavour would not have been possible without my supervisor Holger Hermanns. It is a top priority for you to care for everyone at the chair, to listen to our wishes, and to find solutions for our problems. You were supporting me in many ways: you shared your scientific advice in many situations, you bought a (private) car for emissions tests (that turned out to be a real polluter), and much more. It was a great joy to have so many social events with you and the other Dependables, which also made the work at our chair so enjoyable. Thank you very much! I am also extremely grateful to Pedro D'Argenio for agreeing to review this thesis and for becoming an important advisor at the beginning of my PhD time. You taught me many things that became important during my PhD time! Many thanks also go to Gerardo Schneider, who kindly agreed to review this thesis.

I want to thank Sarah Sterz, who provided ethically sound justifications for many stupid things we did (at least that is how I see it). Most important, I am very grateful for the close friendship that we developed during the pandemic! Thank you very much, Sabine Nermerich and Christa Schäfer. You are/were the defence line against the central administration of the university. I am also grateful that we successfully stole Florian Schießl from a bicycle fair to become the master of our technical infrastructure. During the pandemic, I very much appreciated that you regularly came to university when nobody else did, which made my working days significantly more social. I still try to understand how Felix Freiburger can know so many things – be it nasty details of the examination regulations, the safety architecture of roller coasters, or the rights of owners of bee swarms that decided to relocate. Thank you very much for sharing all of this knowledge! I am also thankful to Gregory Stock, who was always ready to help, in particular regarding technical problems. Michaela Klauck became particularly important whenever social events or trips had to be organised. While everyone else desperately suffered from acute procrastination, you did the organisation with ease. Thank you very much! Many thanks also to Maximilian Köhl, with whom I had many important conversations about coffee and the perfect way to brew it. Unfortunately, our master plan to buy a very expensive coffee machine and to hire a student who becomes our personal barista was never put into action. Special thanks also go to Lena Becker, in particular for her great services as a supervision tutor for Programming 1 before she became a Dependable. Thanks

must also go to Kevin Baum for a lot of very valuable insights into topics beyond the computer science horizon.

I also want to thank all the other Dependables with whom I had a very good time at the chair: Alexander Graf-Brill, Andreas Schmidt, Daniel Stan, Gereon Fox, Gilles Nies, Hassan Hatefi, Juan Fraire, Vahid Hashemi, and Yuliya Butkova. I had many great projects and papers with a lot of talented people. Many thanks to Rayna Dimitrova, Maciej Gazda, Mohammad Mousavi, Yannik Schnitzer, Maximilian Schwenger, Bernd Finkbeiner, Sebastian Holler, Michael Fries, Thomas Heinze, and Gilles Barthe. Thanks also go to Evelyn Kraska and Sabine Lessel for their administrative support.

I also want to thank all my friends that made my time in Saarbrücken so great. Thank you Noemi, for letting me move into your WG when I needed a room and thanks for all the great pastries! I also want to thank Jana; you helped me to survive the lockdowns during the pandemic. Thanks also must go to Nikolai for being a great office mate. Thanks for so many things to Norine, Chris, Yannick, Kathrin, Caro, Fabian, Nicolas, Jan, Paul, and many more! Special thanks go to Clara. You gave me insights into a world I did not know and you were always there for me when I needed help!

Finally, I also want to thank my family; I will do so in German. Zuerst möchte ich meinen Eltern, Gaby und Thomas danken, die nie Zweifel daran hatten, dass ich da hinkomme, wo ich jetzt bin. Ihr habt mir die Unterstützung gegeben, die ihr euch für euch selbst gewünscht hättet, vielen Dank dafür! Danke auch an meine Schwestern, Anna und Franziska, natürlich auch an Jakob und Jonah, an meine Omas Renate und Ursula und an Isi und Onki. Besonderer Dank geht auch an Armin und Jürgen. Einen großen Einfluss hatten meine Opas Roman und Matthias, die mich oft in ihre Werkstatt mitgenommen haben, mir grundlegende praktische Dinge beigebracht haben und mein Interesse für Naturwissenschaften geweckt haben. Ich bin euch sehr dankbar, auch wenn ich euch das nicht mehr persönlich sagen kann.

Abstract

Software is *doped* if it contains a hidden functionality that is intentionally included by the manufacturer and is not in the interest of the user or society. This thesis complements this informal definition by a set of formal *cleanness* definitions that characterise the absence of software doping. These definitions reflect common expectations on clean software behaviour and are applicable to many types of software, from printers to cars to discriminatory AI systems. We use these definitions to propose white-box and black-box analysis techniques to detect software doping. In particular, we present a provably correct, model-based testing algorithm that is intertwined with a probabilistic-falsification-based test input selection technique. We identify and explain how to overcome the challenges that are specific to real-world software doping tests and analyses.

The most prominent example of software doping in recent years is the Diesel Emissions Scandal. We demonstrate the strength of our cleanness definitions and analysis techniques by applying them to emission cleaning systems of diesel cars. All our car related research is unified in a *Car Data Platform*. The mobile app *LolaDrives* is one building block of this platform; it supports conducting real-driving emissions tests and provides feedback to the user in how far a trip satisfies driving conditions that are defined by official regulations.

Zusammenfassung

Software ist *gedopt* wenn sie eine versteckte Funktionalität enthält, die vom Hersteller beabsichtigt ist und deren Existenz nicht im Interesse des Benutzers oder der Gesellschaft ist. Die vorliegende Arbeit ergänzt diese nicht formale Definition um eine Menge von *Cleanness*-Definitionen, die die Abwesenheit von Software Doping charakterisieren. Diese Definitionen spiegeln allgemeine Erwartungen an "sauberes" Softwareverhalten wider und sie sind auf viele Arten von Software anwendbar, vom Drucker über Autos bis hin zu diskriminierenden KI-Systemen. Wir verwenden diese Definitionen um sowohl white-box, als auch black-box Analyseverfahren zur Verfügung zu stellen, die in der Lage sind Software Doping zu erkennen. Insbesondere stellen wir einen korrekt bewiesenen Algorithmus für modellbasierte Tests vor, der eng verflochten ist mit einer Test-Input-Generierung basierend auf einer Probabilistic-Falsification-Technik. Wir identifizieren Hürden hinsichtlich Software-Doping-Tests in der echten Welt und erklären, wie diese bewältigt werden können.

Das bekannteste Beispiel für Software Doping in den letzten Jahren ist der Diesel-Abgasskandal. Wir demonstrieren die Fähigkeiten unserer *Cleanness*-Definitionen und Analyseverfahren, indem wir diese auf Abgasreinigungssystem von Dieselfahrzeugen anwenden. Unsere gesamte auto-basierte Forschung kommt in der *Car Data Platform* zusammen. Die mobile App *LolaDrives* ist eine Kernkomponente dieser Plattform; sie unterstützt bei der Durchführung von Abgasmessungen auf der Straße und gibt dem Fahrer Feedback inwiefern eine Fahrt den offiziellen Anforderungen der EU-Norm der Real-Driving Emissions entspricht.

Contents

1	Introduction	1
1.1	Contributions of this Thesis	2
1.2	Organisation of the Thesis	6
2	Preliminaries	9
2.1	Sets, Functions, and Distances	9
2.2	Traces	10
2.3	Labelled Transition Systems & Model-Based Conformance Tests	12
2.4	Conformance Relations	15
2.5	Hyperproperties & Self-composition	16
2.6	Temporal Logics	18
2.6.1	HyperLTL	18
2.6.2	STL	20
2.6.3	Probabilistic Falsification	20
2.6.4	HyperSTL*	22
2.7	Diesel Emissions	23
2.8	Runtime Monitoring for Real Driving Emissions	26
2.8.1	RTLOLA	27
2.8.2	From Regulation to Specification	29
3	Notions of Software Doping	35
3.1	Sequential Programs	36
3.1.1	Strict cleanness	37
3.1.2	Robust cleanness	42
3.1.3	Func-cleanness	54
3.2	Reactive Systems	60
3.2.1	Strict cleanness	61
3.2.2	Robust cleanness	63
3.2.3	Func-cleanness	73
3.2.4	Past-Forgetful Distance Functions & Trace Integrity . . .	79
3.3	Mixed Input-Output Systems	82
3.3.1	Robust cleanness	85
3.3.2	Func-cleanness	89
3.3.3	Trace Integrity	89

3.4	Hybrid Systems	90
3.4.1	Conformance-Based Cleanness	94
3.4.2	Synchronised Retiming	100
3.5	Summary	104
3.6	Related Work & Contributions	104
4	Model-Aware Software Doping Analysis	109
4.1	Analysis through self-composition	109
4.2	HyperLTL	114
4.2.1	Experimental Results	123
4.3	Related Work & Contributions	125
5	Model-Agnostic Software Doping Analysis	127
5.1	Cleanness of Labelled Transition Systems	127
5.2	Reference Implementation for Robust Cleanness	131
5.3	Model-Based Doping Tests	139
5.4	HyperSTL	146
5.5	An Integrated Testing Approach	160
5.6	Related Work & Contributions	161
6	Hands-On: Diesel Doping Tests	163
6.1	Model-Based Testing in Practice	163
6.1.1	The Volkswagen Case	164
6.1.2	The Nissan Case	167
6.2	Conformance-Based Testing in Practice	169
6.3	Car Data Platform and LolaDrives	180
6.3.1	LolaDrives	181
6.3.2	Technical Setup	184
6.3.3	Demonstration	186
6.4	CDP-Based Test Input Selection	189
6.5	Related Work & Contributions	193
7	Conclusion & Future Work	195
7.1	Summary	195
7.2	Future Work	197
7.3	Effective Human Oversight with Func-Cleanness	198
7.3.1	Individual Fairness	200
7.3.2	Fairness Monitoring	203
	Bibliography	209

1 Introduction

Should we trust software manufacturers? Sociology defines trust as “the willingness of a party to be vulnerable to the actions of another party based on the expectation that the other will perform a particular action important to the trustor, irrespective of the ability to monitor or control that other party.” [89] Thus, having trust into a software manufacturer translates to the willingness of the software licensee or society to be vulnerable to the actions of the software manufacturer.

Whether or not to trust another party depends largely on the extent of the vulnerability one is willing to accept. We, as individuals and as society, become increasingly vulnerable, because software becomes ubiquitous in ever more domains of our daily lives. People increasingly use digital personal assistants like smart phones [112], smart watches [59], and smart speakers [111]. The Internet of Things [135] puts software in devices like refrigerators, stoves, and other kitchen devices. Heating systems, smart lighting systems, home video surveillance systems, smart door bells are only a few examples for a trend that adds computing power and software in ever more things of daily life.

Software also becomes increasingly important in sensitive areas like health care, banking, and critical infrastructure like power plant control, power grid management or automated agriculture [73]. In the automotive sector, cars are nowadays equipped with a variety of advanced driver-assistance systems up to fully autonomous driving; electric vehicles need a comprehensive software to coordinate the electrical engine, the battery and the cooling system of the car [28] and cars with combustion engines need specialised software for the cleaning of emissions to meet the strict requirements that are enforced by law [118]. Lastly, the avionic sector is increasingly relying on software; aircraft control software and flight control software [114] are only two examples. These are several examples where a lot is at stake; we are more vulnerable in these domains, because our health, life or wealth is at risk.

Examples of breaches of trust are numerous. The most severe in recent history is a fault in the design of the Maneuvering Characteristics Augmentation System (MCAS) of Boeing 737 MAX aircrafts, which caused two plane crashes and, altogether, 346 casualties [134]. Later, it turned out that not only the MCAS system was flawed, but also other parts of the aircraft control system [79]. This was a massive breach of trust. In this thesis I will particularly focus on two

elements of trust [97, Table 2.1]: *benevolence trust* involves trust in the benevolence of the manufacturer, and in particular trust in the lack of opportunism and egoism. *Competence trust* involves trust in skills of the manufacturer and the ability to use the technology necessary to build a software. Clearly, Boeing breached competence trust. In general, every unintended software fault – regardless of the severity of the consequences – can be considered as a breach of competence trust.

A breach of trust that is fundamentally different from the Boeing case is the Diesel Emissions Scandal. In 2014 and 2015, the US environmental protection agency discovered that several cars manufactured by Volkswagen were shipped with tampered emission cleaning systems [131]. These systems contained defeat devices that were able to tell emissions tests and real driving apart [40, 47]. During the emissions tests the cars were set to comply to the regulations, while on the road the behaviour was optimised for parameters other than clean emissions. Later, it turned out that not only Volkswagen cars showed such fraudulent behaviour, but also cars by other manufacturers [51]. In the end, millions of cars were affected and had to be recalled [93]. Notably, the problem was the *software* of the car, rather than the hardware.

The Diesel Scandal was a massive breach of benevolence trust. The actions that the car owner and society reasonably (and well-known to the manufacturer) expect from the car manufacturer and its actual actions were contradictory. Car owners and society were badly vulnerable – the damage caused by this behaviour was immense. Nitric oxides are toxic gases that cause human diseases [122]. Owners of diesel cars suffered from driving bans [5] and a loss in value of their cars [126]. From a computer science view, the emission cleaning malfunction is not a *bug*, because it is intended by the software manufacturer. In recent work by Barthe et al. [11], such behaviour has been called *software doping*. A bug is a breach of competence trust, while software doping is a breach of benevolence trust. Technically, the difference between software doping and ordinary bugs is threefold: (1) Only for the former there is a basic mismatch in intentions about what the software should do. (2) While a bug is most often rooted in a small coding error, software doping can occupy a considerable portion of the implementation. (3) Bugs can potentially be detected during production by the manufacturer, whereas software doping by its nature can – if at all – only be discovered after production, by the other party facing the final product.

1.1 Contributions of this Thesis

At the core of this thesis is the concept of software *cleanness*. If a software violates a cleanness definition, then we will say that it is *doped* (w.r.t. this

particular cleanness notion). We first provide multiple cleanness definitions, each fitting a multitude of types of software it can be applied to. Based on these, we develop analysis techniques to detect violations of cleanness – and, therefore, the presence software doping. We propose techniques that assume knowledge about the internals of the system (i.e., white-box analyses) and techniques that do not need such knowledge (i.e., black-box analyses). Every technique is backed by formal foundations and is demonstrated using a concrete example.

For black-box techniques, it is important that our cleanness definitions reason about the observations of the system. That is, they do not make any assumptions regarding how the system works internally to produce the observed behaviour. This is an important characteristic of these definitions, because they are supposed to provide software licensees, users or society – irrespective of whether they trust the manufacturer – a way to check for undesired behaviour. Notably, software is almost always *licensed*. While the difference between licensing and buying a software is subtle for the user of the software, it makes a huge difference regarding the analysis of it. License agreements typically hinder the licensee from an inspection of the internals of the software. For example, the de-compilation of the executable binary file is typically forbidden in such agreements.

An important aspect of trust are the expectations of the software licensees (or society) on the manufacturer. Many expectations that are ethically or morally justifiable may not be justifiable juridically. Hence, a breach of trust does not always imply a breach of law. For cars there is relatively strict law defining what a car is allowed to do and what is forbidden (and this law has been tightened after the Diesel Scandal). There are other types of software to which specialised law applies, e.g., fully automatic decision making based on personalised data [123, 39]. However, the pervasive nature of software makes it impossible to enforce precise regulations for every possible software. Even for software that is regulated, not every possible behaviour is covered by the regulation. In case of the Diesel Emissions Scandal, the emissions were measured only for a single test cycle; the speed trajectory of this test cycle was specified in the regulation. The cleanness definitions in this thesis are designed in such a way that they seamlessly define requirements on system behaviours that have been left out by the regulation (but taking the regulated behaviour into account).

Our running and most motivating example is the Diesel Emissions Scandal; most demonstration use cases are strongly related to it. Nevertheless, we emphasise that the cleanness definitions and analysis techniques presented in this thesis can be applied to many types of systems.

Example 1.1. To get permission to sell a new car model in the European Union (and in many other countries) the manufacturer must prove that this car model complies to certain effective regulations. For example, car manufacturers have to do tests on a chassis dynamometer under precisely defined conditions (in-

cluding details like ambient air temperature, etc.). The car has to follow a (single) predefined speed trajectory, which is called test cycle. While the car is driven according to this test cycle, several characteristics of the car are measured; for example, the fuel consumption and the amount of nitric oxides and other emissions that leave the exhaust pipe of the car. In 2014, before the Diesel Emissions Scandal surfaced, the car had to follow the *New European Driving Cycle* (NEDC) [124], which is depicted in Figure 2.2 on page 24.

One of the cleanness notions I will present is *robust cleanness*. Intuitively, in the context of car admissions, the emission cleaning system of a car is robustly clean if test cycles that are similar to the NEDC lead to amounts of nitric oxide emissions that are not vastly different from the amount emitted during the NEDC. For example, if a car emits an average of 80 mg/km of nitric oxides while driving the NEDC, but it emits on average 240 mg/km when driving the NEDC constantly 5 km/h slower, then this would be considered as a violation of robust cleanness. While the input deviation is small (at most 5 km/h), the output deviation is extreme (three times the NEDC emissions).

Obviously, robust cleanness adds constraints on the outputs of a car (e.g., the amount of emitted nitric oxides) for inputs that are not covered by the official regulation (i.e., test cycles other than the NEDC). Still, the outputs measured for the NEDC, which is part of the official regulation, have influence on the additional constraints imposed by robust cleanness. Thus, robust cleanness defines restrictions that seamlessly extend the requirements imposed by the regulation.

In this thesis I will navigate along two dimensions: a *stakeholder dimension* and a *level-of-abstraction dimension*. The level-of-abstraction dimension is bounded by a foundational (or abstract) perspective on software doping on the one end, and a practical (or applied) perspective on the other end. The thesis begins with the foundational perspective; it provides the cleanness definitions and compares the different variants of cleanness. It presents definitions for different computational models; in particular sequential programs, reactive systems and hybrid systems. Some of the cleanness notions can express the same class of cleanness in different ways; we prove their equivalences in these cases. We then move closer towards the practical perspective by proposing a formal foundation for doping tests. We use a model-based testing theory to construct an algorithm that is able to propose test cases. We continue by discussing the practical challenges regarding this theoretical test generation algorithm and demonstrate how the main ideas can be used to arrive at an executable implementation. Finally, we present actual doping tests; most interestingly emissions tests that we executed with a real car on a chassis dynamometer.

The stakeholder dimension along which this thesis navigates ranges from a manufacturer's perspective on software doping towards a research perspective

further to a layperson’s perspective. A manufacturer has full access to the internals and externals of a system. Thus, they are able to verify for the whole input space the absence of software doping w.r.t. a suitable notion of cleanness.

If there is reason to mistrust the manufacturer, software-driven systems can be analysed by researchers, NGOs and alike, which constitute a completely different user group along the stakeholder dimension. Such third parties typically do not have (enough) knowledge about the internals of a system to use exhaustive verification techniques. The analyses can solely base on observations of the system, i.e., the available analyses are model-agnostic. This restriction has certain drawbacks. Most of the cleanness definitions in this thesis are based on the comparison of two execution traces of a system. For example, for robust cleanness (from the above example) a standard execution of a system (e.g., the NEDC test) must be compared to non-standard executions of the system. In the presence of nondeterminism it might even become necessary to consider infinitely many trajectories at the same time. Properties over multiple traces are called hyperproperties [35]. In this respect, expressing robust cleanness as a hyperproperty needs both universal and existential quantifications of system trajectories. Formulas containing only one type of quantifiers can be analysed efficiently, e.g., using model-checking techniques, but checking properties with alternating quantifiers is known to be computationally hard [34, 57]. Moreover, testing of such problems is in general not possible. Assume, for example, a property requiring for a (nondeterministic) system that for every input i , there exists the output $o = i$, i.e., one of the system’s possible behaviours computes the identity function. For black-box systems with infinitely large input and output domains the property can neither be verified nor falsified through testing. In order to verify the property, it is necessary to iterate over the infinite input set. For falsification one must show that for some i the system cannot produce i as output. However, due to nondeterminism, not observing an output in finitely many trials does not rule out that this output can indeed be generated. A large fraction of the contents of this thesis focusses on model-agnostic software doping analyses.

Regarding the stakeholder dimension along which this thesis is aligned, the testing-based approaches above are located between the manufacturer and the layperson ends. Using the testing techniques still requires a solid understanding of the theoretical concepts and proper adjustments to apply it to concrete systems. Continuing along the stakeholder dimension, the thesis presents a tool for laypersons to detect software doping without detailed knowledge about the verification techniques inside this tool. To provide such a tool and to hide the formal verification interface from the layperson, we had to instantiate the analysis framework for a concrete use case; we chose the cleanness of emission cleaning systems in diesel cars. I will demonstrate a mobile phone application

called LolaDrives that allows laypersons to conduct diesel emissions tests without expensive equipment.

If the reader decides to mistrust a certain software manufacturer, then this thesis provides a multitude of techniques to initiate an investigation on undesired software functionality, i.e., investigations on software doping. The largest part of this thesis is applicable to any kind of software. Still, motivated by the impudent past actions of various car manufacturers, a significant part covers the doping of emission cleaning systems in diesel cars.

1.2 Organisation of the Thesis

The contents in this thesis are based on the following publications.

- [42] Pedro R. D’Argenio, Gilles Barthe, Sebastian Biewer, Bernd Finkbeiner, and Holger Hermanns. Is your software on dope? - Formal analysis of surreptitiously “enhanced” programs. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 83–110. Springer, 2017. doi: 10.1007/978-3-662-54434-1.4.
- [17] Sebastian Biewer, Pedro R. D’Argenio, and Holger Hermanns. Doping tests for cyber-physical systems. In David Parker and Verena Wolf, editors, *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings*, volume 11785 of *Lecture Notes in Computer Science*, pages 313–331. Springer, 2019. doi: 10.1007/978-3-030-30281-8.18.
- [18] Sebastian Biewer, Pedro R. D’Argenio, and Holger Hermanns. Doping tests for cyber-physical systems. *ACM Trans. Model. Comput. Simul.*, 31(3):16:1–16:27, 2021. doi: 10.1145/3449354.
- [46] Rayna Dimitrova, Maciej Gazda, Mohammad Reza Mousavi, Sebastian Biewer, and Holger Hermanns. Conformance-based doping detection for cyber-physical systems. In Alexey Gotsman and Ana Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing*

- Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12136 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 2020. doi: 10.1007/978-3-030-50086-3_4.
- [20] Sebastian Biewer, Rayna Dimitrova, Michael Fries, Maciej Gazda, Thomas Heinze, Holger Hermanns, and Mohammad Reza Mousavi. Conformance relations and hyperproperties for doping detection in time and space. *Log. Methods Comput. Sci.*, 18(1), 2022. doi: 10.46298/lmcs-18(1:14)2022.
- [21] Sebastian Biewer, Bernd Finkbeiner, Holger Hermanns, Maximilian A. Köhl, Yannik Schnitzer, and Maximilian Schwenger. RTLOLA on board: Testing real driving emissions on your phone. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 365–372. Springer, 2021. doi: 10.1007/978-3-030-72013-1_20.
- [22] Sebastian Biewer, Bernd Finkbeiner, Holger Hermanns, Maximilian A. Köhl, Yannik Schnitzer, and Maximilian Schwenger. On the road with RTLOLA: Testing real driving emissions on your phone. *Int. J. Softw. Tools Technol. Transf.*, 2023. doi:10.1007/s10009-022-00689-5.
- [25] Sebastian Biewer and Holger Hermanns. On the detection of doped software by falsification. In Einar Broch Johnsen and Manuel Wimmer, editors, *Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13241 of *Lecture Notes in Computer Science*, pages 71–91. Springer, 2022. doi: 10.1007/978-3-030-99429-7_4.

The concrete structure of the thesis is summarised below.

Chapter 2 This chapter contains the background information on which the remaining chapters build on.

Chapter 3 This chapter presents a collection of cleanness definitions targeted for different execution models (e.g., sequential programs, reactive systems, hybrid systems, etc.) and contexts with different structural requirements

concerning system cleanness. We discuss the differences between the different definitions and, where appropriate, identify the circumstances under which cleanness definitions are equivalent.

Chapter 4 In this chapter we demonstrate how existing verification techniques can be adapted for software doping analysis. The internals of the analysed systems must be known. For sequential programs we use a verification technique based on self-composition that allows to (manually) prove cleanness of programs. For reactive systems, we propose HyperLTL formulas that allow model-checkers to automatically perform cleanness analyses.

Chapter 5 This chapter develops analysis techniques for cases in which knowledge about the internals of a system is unavailable. We model a largest implementation that satisfies robust cleanness and prove that this model can be instantiated into a model-based testing framework. To generate promising test inputs, the chapter demonstrates how to combine a probabilistic falsification approach with the model-based testing approach. Limitations (caused by the model agnosticism and by the fact that cleanness is a hyperproperty) are clearly highlighted and discussed.

Chapter 6 The theoretical results are applied to the Diesel Emissions Scandal. We use the Volkswagen defeat device to demonstrate how our cleanness definitions easily detect the Volkswagen doping. Without further knowledge about the internals of the software running inside a Nissan car, we successfully use the model-based testing technique to uncover software doping in this car. We use the same car to demonstrate why cleanness notions that explicitly include time reasoning are important. As a supplement to our formal cleanness-based software doping characterisation, we take up Real-Driving Emissions (RDE) tests as a regulation-defined software doping characterisation. We demonstrate the mobile application *LolaDrives* and explain how it is integrated into a *Car Data Platform* that serves as an interface between RDE-based emissions tests and cleanness-based emissions tests.

Chapter 7 This chapter provides a summary of the contents in this thesis, possible future work, and a preview on an interdisciplinary ongoing research project that is related to software doping analysis.

2 Preliminaries

2.1 Sets, Functions, and Distances

In this thesis, we will regularly refer to the *natural numbers* \mathbb{N} , the *real numbers* \mathbb{R} and the set of *Boolean values* $\mathbb{B} = \{\text{true}, \text{false}\}$. In some contexts, we use the symbol \top for true and \perp for false. The set of *extended real numbers* $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, \infty\}$ contains the regular real numbers but also includes positive and negative infinity [102, Chapter 1]. Infinity is integrated into the order of real numbers (e.g., as defined by Rudin [104, Chapter 1]) such that for all $x \in \overline{\mathbb{R}}$ it holds that $x \geq -\infty$ and $x \leq \infty$. Further variations of \mathbb{R} are the set of non-negative real numbers $\mathbb{R}_{\geq 0} := \{x \in \mathbb{R} \mid x \geq 0\}$, and the set of non-negative extended real numbers $\overline{\mathbb{R}}_{\geq 0} := \{x \in \overline{\mathbb{R}} \mid x \geq 0\}$. We denote by \mathbb{N}_+ the set $\mathbb{N} \setminus \{0\}$, i.e., the natural numbers without zero.

For a finite set $S = \{s_1, \dots, s_n\}$ we denote by $|S|$ the number of elements in S , i.e., $|S| = n$. For a set S with infinitely many elements, we define $|S| = \infty$. For a positive natural number n and a set S , we denote by S^n the $(n-1)$ cross-product of S with itself, i.e., $S^n := \{(s_1, \dots, s_n) \mid \forall 1 \leq i \leq n. s_i \in S\}$. We denote by 2^S the *power set* of S , i.e., $2^S = \{T \mid T \subseteq S\}$.

A *function* $f : A \rightarrow B$ assigns to every value $a \in A$ exactly one value $b = f(a) \in B$. We call A the *domain* of f , denoted $\text{dom}(f)$, and B the *codomain* of f , denoted $\text{cod}(f)$. f is *injective* if and only if for all $a_1, a_2 \in A$, $f(a_1) = f(a_2)$ implies that $a_1 = a_2$. f is *surjective* if and only if for every $b \in B$, there is some $a \in A$, such that $b = f(a)$. f is *bijective* if and only if f is injective and surjective. The *restriction* $f|_{A'}$ of f to a smaller domain $A' \subseteq A$ is the function $f' : A' \rightarrow B$ with $f'(a) := f(a)$. If f is bijective, then the *inverse of f* is the function $f^{-1} : B \rightarrow A$ that satisfies $f^{-1}(b) = a$ if and only if $f(a) = b$. For some function $g : B \rightarrow C$, $h = g \circ f$ is the *function composition* of f and g resulting in a new function $h : A \rightarrow C$, where $h(x) = g(f(x))$.

Let S be a set, and $R = S \times S$ a binary relation on S . R is *reflexive* if and only if for every $x \in S$, $(x, x) \in R$. R is *symmetric* if and only if for every $x, y \in S$, it holds that $(x, y) \in R$ if and only if $(y, x) \in R$.

Let $R \subseteq \overline{\mathbb{R}}$ be a subset of the extended real numbers. Based on Rudin [104, Chapter 1], a value $b \in \overline{\mathbb{R}}$ is an *upper bound* of R if and only if $x \leq b$ for all $x \in R$. If and only if such a b exists, R is *bounded above*. Similarly, a value $b \in \overline{\mathbb{R}}$

is a *lower bound* of R if and only if $b \leq x$ for all $x \in R$. If and only if such a b exists, R is *bounded below*.

Let $b \in \overline{\mathbb{R}}$ be an upper bound for a set $R \subseteq \overline{\mathbb{R}}$. b is the *least upper bound* or *supremum* of R if and only if for every upper bound $b' \in \overline{\mathbb{R}}$ of R it holds that $b \leq b'$. If b is the supremum of R , we write $b = \sup R$. Analogously, let $b \in \overline{\mathbb{R}}$ be a lower bound for a set $R \subseteq \overline{\mathbb{R}}$. b is the *greatest lower bound* or *infimum* of R if and only if for every lower bound $b' \in \overline{\mathbb{R}}$ of R it holds that $b \geq b'$. If b is the infimum of R , we write $b = \inf R$. Notice that $\sup \emptyset = -\infty$ and $\inf \emptyset = \infty$ [102]. Let C be a set, P be a predicate over values in C and f a function from C to $\overline{\mathbb{R}}_{\geq 0}$. For the expression $\sup\{f(x) \mid x \in C \wedge P(x)\}$ we also use the abbreviated notation $\sup_{P(x)} f(x)$. Similarly, we write $\inf_{P(x)} f(x)$ instead of $\inf\{f(x) \mid x \in C \wedge P(x)\}$.

We say that a function $d : A \times A \rightarrow \overline{\mathbb{R}}$ is a *distance function*¹ if and only if for all $a \in A$, $d(a, a) = 0$ and for all $a, b \in A$, $d(a, b) \geq 0$ and $d(a, b) = d(b, a)$. The *Hausdorff distance*

$$\mathcal{H}(d)(A, B) := \max\{\sup_{a \in A} \inf_{b \in B} d(a, b), \sup_{b \in B} \inf_{a \in A} d(a, b)\}$$

defines a distance between two sets A and B based on a distance function d on the elements of the sets.

2.2 Traces

A trace describes an execution of a system as a sequence of observations during such an execution. Formally, a trace is defined over a set of observations or symbols – the trace is a sequence of symbols that must be in this set. Let X be a set of symbols. A *finite trace over X* is a finite sequence $t = x_1 x_2 \cdots x_n$, where $x_i \in X$ for all $1 \leq x \leq n$. The *length* of t is n , denoted $|t| = n$. The set X^* is the set of all finite traces over X . An *infinite trace over X* is an infinite sequence $t = x_1 x_2 x_3 \cdots$, where $x_i \in X$ for all $i \in \mathbb{N}_+$. We say that the length of t is infinite and write $|t| = \infty$. The set X^ω is the set of all infinite traces over X . We denote the *empty trace*, i.e., a trace that contains no symbol, by ϵ and say that its length is zero, denoted $|\epsilon| = 0$. A *trace over X* is the empty trace, a finite trace over X or an infinite trace over X . Let $t_1 = x_1 x_2 \cdots x_n$ be a finite trace and $t_2 = y_1 y_2 \cdots$ an infinite trace. Then the concatenation of t_1 and t_2 is defined by $t_1 \cdot t_2 := x_1 x_2 \cdots x_n y_1 y_2 \cdots$.

Let $k \in \mathbb{N}_+$ be an index and let t be an infinite trace over X with $t = x_1 x_2 x_3 \cdots$, or a finite trace over X with $t = x_1 x_2 \cdots x_n$ and $n \geq k$. The *k th*

¹This definition of a distance function is a combination of pseudometrics and semimetrics: d is allowed to assign a zero distance to non-identical values in A and it does not require the triangle inequality.

projection of t is the operation $t[k] := x_k$, which returns the k th symbol in the sequence. The k -*prefix* of t is defined as $t[..k] := x_1 x_2 \cdots x_k$ and returns all symbols in the sequence up to and including the k th symbol. The prefix $t[..k]$ is a finite trace of length k . The zero-prefix $t[..0]$ is the empty trace ϵ . The k -*suffix* of t is defined as $t[k..] := x_k x_{k+1} \cdots$ and returns all symbols in the sequence from and including the k th symbol. The suffix of a trace t is infinite if and only if t is an infinite trace. For a set T of traces over X , each with length of at least k , the set $T[..k]$ contains the k -prefixes of all traces in T , i.e., $T[..k] := \{t[..k] \mid t \in T\}$. Similarly, $T[k..]$ contains the k -suffixes of all traces in T , i.e., $T[k..] := \{t[k..] \mid t \in T\}$.

Let $x \in X$. We denote by x^ω the infinite trace $x x x \cdots$ that consists of only x symbols. Let Y be a second set of symbols and let $T_X \subseteq X^\omega$, $T_Y \subseteq Y^\omega$ be sets of infinite traces over X , respectively Y . We denote by $T_X \times^\omega T_Y$ the product construction for infinite traces on the level of the individual symbols and call the result *traces of pairs over T_X and T_Y* . That is,

$$T_X \times^\omega T_Y := \{t \in (X \times Y)^\omega \mid \exists t_X \in T_X, t_Y \in T_Y. \forall k \in \mathbb{N}_+. t[k] = (t_X[k], t_Y[k])\}.$$

Moreover, for reasons of convenience, we introduce *trace-pair-equality*, which equates traces of pairs with pairs of traces. Concretely, let $t \in (X \times Y)^\omega$ be a trace of pairs and $(t_X, t_Y) \in X^\omega \times Y^\omega$ be a pair of traces. Then we say that $t = (t_X, t_Y)$ if and only if for every $k \in \mathbb{N}_+$, $t[k] = (t_X[k], t_Y[k])$.

To represent traces over continuous time domains, we use *generalised timed traces (GTTs)* [61]. Such a trace is a function with a discrete or continuous domain – which is called time domain – that maps time points to values. In contrast, the discrete traces presented above are sequences of values that have as an implicit time domain the positions k in the trace; the set of positions are represented by the positive natural numbers (for infinite traces) or a subset thereof (for finite traces). Formally, a generalised timed trace is a function $\mu : \mathcal{T} \rightarrow X$ such that $\mathcal{T} \subseteq \mathbb{R}_{\geq 0}$. We call $\text{dom}(\mu) = \mathcal{T}$ the *time domain* of μ and $\text{cod}(\mu) = X$ the *value domain* of μ . X^θ is the set of all generalised timed traces with value domain X .

Let $\mu : \mathcal{T} \rightarrow X$ be a GTT and $t \in \mathcal{T}$ a point in time. The t -*prefix* of μ is defined as μ but with the time domain restricted to $\mathcal{T} \cap [0, t]$, i.e., $\mu[..t] := \mu|_{\mathcal{T} \cap [0, t]}$. Likewise, for $s, t \in \mathcal{T}$ with $s \leq t$, the s - t -*infix* of μ is defined as μ but with the time domain restricted to $\mathcal{T} \cap [s, t]$, i.e., $\mu[s..t] := \mu|_{\mathcal{T} \cap [s, t]}$.

2.3 Labelled Transition Systems & Model-Based Conformance Tests

In Chapter 5 we will instantiate a model-based conformance testing framework. There, specifications are modelled as labelled transition systems (LTS) that explicitly distinguish between input and output transitions. A special instance of these LTS are input-output transition systems, which are LTS that are input enabled [121]. For ease of presentation, we do not consider internal transitions.

Definition 2.1. A *labelled transition system (LTS)* with inputs and outputs $L = \langle Q, \text{In}, \text{Out}, \rightarrow, q_{\text{init}} \rangle$ is a five-tuple where (i) Q is a (possibly uncountable) non-empty set of states; (ii) In and Out are disjoint (possibly uncountable) sets of input labels, respectively output labels, i.e., $\text{In} \cap \text{Out} = \emptyset$; (iii) with $L = \text{In} \cup \text{Out}$, $\rightarrow \subseteq Q \times L \times Q$ is the transition relation; (iv) $q_{\text{init}} \in Q$ is the initial state. Instead of $(q, a, q') \in \rightarrow$ we write $q \xrightarrow{a} q'$. An LTS is an *input-output transition system (IOTS)* if it is input-enabled in any state, i.e., for all $q \in Q$ and $a \in \text{In}$ there is some $q' \in Q$ such that $q \xrightarrow{a} q'$. An LTS (or IOTS) is *finite* if Q and \rightarrow are finite. It is *deterministic* if for every $q, q_1, q_2 \in Q$ and $a \in L$, $q \xrightarrow{a} q_1$ and $q \xrightarrow{a} q_2$ imply that $q_1 = q_2$.

A *finite path* p in a labelled transition system $L = \langle Q, \text{In}, \text{Out}, \rightarrow, q_{\text{init}} \rangle$ is a sequence $q_1 a_1 q_2 a_2 \cdots a_{n-1} q_n$ with $q_i \xrightarrow{a_i} q_{i+1}$ for all $1 \leq i < n$. We denote as $\text{last}(p)$ the last state occurring in p , i.e., $\text{last}(p) = q_n$. An *infinite path* p in L is a sequence $q_1 a_1 q_2 a_2 \dots$ with $q_i \xrightarrow{a_i} q_{i+1}$ for all $i \in \mathbb{N}_+$. Let $\text{paths}_*(q)$ and $\text{paths}_\omega(q)$ be the sets of all finite and infinite paths of L beginning in state q , respectively. If $q = q_{\text{init}}$, we also write $\text{paths}_*(L)$ ($\text{paths}_\omega(L)$) instead of $\text{paths}_*(q_{\text{init}})$ ($\text{paths}_\omega(q_{\text{init}})$). A finite trace $\sigma = a_1 a_2 \cdots a_n$ over $(\text{In} \cup \text{Out})$ is a finite trace of L beginning in $q_1 \in Q$, if there is a finite path $q_1 a_1 q_2 a_2 \dots a_n q_{n+1} \in \text{paths}_*(q_1)$. We denote as $\text{last}(\sigma)$ the last action label occurring in σ , i.e., $\text{last}(\sigma) = a_n$. An infinite trace of L beginning in $q_1 \in Q$ is an infinite trace $a_1 a_2 \cdots$ over $(\text{In} \cup \text{Out})$ such that there is an infinite path $q_1 a_1 q_2 a_2 \dots \in \text{paths}_\omega(q_1)$. If p is a path, we let $\text{trace}(p)$ denote the trace induced by p . For states $q \in Q$, let $\text{traces}_*(q)$ and $\text{traces}_\omega(q)$ be the set of all finite and, respectively, infinite traces beginning in q , and let $\text{traces}_*(L) = \text{traces}_*(q_{\text{init}})$ and $\text{traces}_\omega(L) = \text{traces}_\omega(q_{\text{init}})$. We will use $L_1 \subseteq L_2$ to denote that $\text{traces}_\omega(L_1) \subseteq \text{traces}_\omega(L_2)$.

A state $q \in Q$ is *reachable* from state $q_1 \in Q$ if there exists a path $q_1 a_1 \dots a_n q \in \text{paths}_*(q_1)$. A state $q \in Q$ is *reachable* in L if q is reachable from state q_{init} . A non-empty set $Q' \subseteq Q$ of states constitutes a *cycle* if every state in Q' is reachable from every other state in Q' via a path that induces a non-empty trace. A cycle Q' is *reachable* from a state $q \in Q$ if any of the states in Q' is reachable from q . Q' is *reachable* in L if it is reachable from state q_{init} .

We will rely on the model-based testing framework by Jan Tretmans [119, 120, 121]; in particular, we will use the *input-output conformance* (**io**co) notion. In this setting, it is assumed that the implemented system under test (IUT) \mathcal{I} can be modelled as an IOTS while the specification of the required behaviour is given in terms of an LTS *Spec*.

For the conformance check, it is necessary to explicitly indicate when the system does not produce an output, i.e., when the system is *quiescent*. In practice, an implementation can observe quiescence by means of a timeout mechanism. In LTS a state is quiescent whenever it cannot proceed autonomously, i.e., it cannot produce an output; we use a distinct label δ to indicate quiescence as part of the model. This label is typically included in the set of outputs. Given a set of outputs Out , the set $\text{Out}_\delta := \text{Out} \cup \{\delta\}$ is this output set with quiescence included.

In specifications, δ -transitions are often modelled as self-loops back to the quiescent state. These self-loops are added to all quiescent states of an LTS when applying the quiescence closure to it.

Definition 2.2. Let $L = \langle Q, \text{In}, \text{Out}, \rightarrow, q_{\text{init}} \rangle$ be an LTS. The *quiescence closure* (or *δ -closure*) of L is the LTS $L_\delta := \langle Q, \text{In}, \text{Out}_\delta, \rightarrow_\delta, q_{\text{init}} \rangle$ with $\rightarrow_\delta := \rightarrow \cup \{s \xrightarrow{\delta} s \mid \forall o \in \text{Out}, t \in Q: s \not\rightarrow t\}$. The *suspension traces* of L is the set $\text{traces}_*(L_\delta)$ of finite traces of the quiescence closure of L .

To define **io**co, we need the following LTS specific definitions. Let $L = \langle Q, \text{In}, \text{Out}, \rightarrow, q_{\text{init}} \rangle$ be an LTS, $\sigma = a_1 a_2 \dots a_n \in \text{traces}_*(L)$ a trace in L , and $Q' \subseteq Q$ a subset of the state space of L . The set L after σ is defined as $\{q_{n+1} \mid q_1 a_1 q_2 a_2 \dots a_n q_{n+1} \in \text{paths}_*(L)\}$ and Q' after a as $\{q' \mid \exists q \in Q': q \xrightarrow{a} q'\}$. For a state q , let $\text{out}(q) = \{o \in \text{Out} \mid \exists q': q \xrightarrow{o} q'\}$ and for a set of states Q' , let $\text{out}(Q') = \bigcup_{q \in Q'} \text{out}(q)$.

The idea behind **io**co is that any output produced by the IUT \mathcal{I} must have been foreseen by its specification *Spec*, and moreover, any input of \mathcal{I} not foreseen by *Spec* may introduce new functionality. This is captured by the following definition.

Definition 2.3. For every IUT \mathcal{I} and specification *Spec*, \mathcal{I} **io**co *Spec* holds if and only if for all $\sigma \in \text{traces}_*(\text{Spec}_\delta)$ it holds that $\text{out}(\mathcal{I}_\delta \text{ after } \sigma) \subseteq \text{out}(\text{Spec}_\delta \text{ after } \sigma)$.

The base principle of *conformance testing* now is to assess by means of testing whether the IUT conforms to its specification w.r.t. **io**co. Tretmans defines test cases as LTS. These LTS are described by means of a basic process algebra [121]. A *process* is a term defined in the language \mathcal{P} given by

$$p ::= \sum_{z \in Z} a_z; p_z \mid A$$

where Z is an index set, each a_z is a label, each p_z is a process, and A belongs to a set of constants called *process names*, which in turn can be defined by equations of the form $A := p$. (Following [121], we use the semicolon as action prefix operator.) We write $\sum_{z \in Z_1} a_z; p_z + \sum_{z \in Z_2} a_z; p_z$ for $\sum_{z \in Z_1 \cup Z_2} a_z; p_z$. A process has semantics in terms of LTS in the usual way: the set of states is the set of all possible processes and the transitions are defined according to the following rules.

$$\sum_{z \in Z} a_z; p_z \xrightarrow{a_z} p_z \qquad \frac{p \xrightarrow{a} p'}{A \xrightarrow{a} p'} \quad A := p$$

A *test case* t for an implementation with inputs in In and outputs in Out is defined as a deterministic LTS. Let t_0 be the initial state of t . t has the following restrictions: (i) from t_0 , any of the special processes **pass** and **fail** can be reached, where **pass** \neq **fail**, and they are defined by **pass** := $\sum\{a; \mathbf{pass} \mid a \in \text{Out}_\delta\}$ and **fail** := $\sum\{a; \mathbf{fail} \mid a \in \text{Out}_\delta\}$, (ii) t has no reachable cycles except those of **pass** and **fail**, and (iii) for any state q reachable from t_0 , the set $\{a \mid q \xrightarrow{a} q'\}$ contains the whole set Out of outputs, and also contains either exactly one input or δ (but not both). A *test suite* is a set of test cases, a *test run* of a test case t with an IUT \mathcal{I} is an experiment where the test case supplies inputs to the IUT while observing the outputs of the IUT or the absence of them [121]. This can be captured by parallel composition according to the following transition rule:

$$\frac{q \xrightarrow{a} q' \quad p \xrightarrow{a} p' \quad a \in \text{In} \cup \text{Out}_\delta}{q \parallel p \xrightarrow{a} q' \parallel p'}$$

Let p_0 be the initial state of \mathcal{I} . The IUT \mathcal{I} passes the test case t , notation \mathcal{I} **passes** t , if and only if there is no state p such that a state **fail** $\parallel p$ is reachable from $t_0 \parallel p_0$. Given a test suite T , we write \mathcal{I} **passes** T whenever \mathcal{I} **passes** t for all $t \in T$.

A test case can be generated by the algorithm TG shown below. Argument S is a subset of the state space of the specification LTS *Spec*. The algorithm nondeterministically returns a process, which induces a deterministic LTS. We write $t \in \text{TG}(S)$ to denote that t is one of the processes that can be generated by an execution of $\text{TG}(S)$.

$\text{TG}(S) :=$ choose nondeterministically one of the following processes:

1. **pass**
2. $i; t_i$ where $i \in \text{In}$, S after $i \neq \emptyset$ and $t_i \in \text{TG}(S$ after $i)$
 - + $\sum\{o; \mathbf{fail} \mid o \in \text{Out} \wedge o \notin \text{out}(S)\}$
 - + $\sum\{o_j; t_{o_j} \mid o_j \in \text{Out} \wedge o_j \in \text{out}(S)\}$,

where for each $o_j, t_{o_j} \in \text{TG}(S$ after $o_j)$

3. $\sum \{o; \mathbf{fail} \mid o \in \mathbf{Out} \cup \{\delta\} \wedge o \notin \mathbf{out}(S)\}$
 $+ \sum \{o_j; t_{o_j} \mid o_j \in \mathbf{Out} \cup \{\delta\} \wedge o_j \in \mathbf{out}(S)\},$
 where for each $o_j, t_{o_j} \in \mathbf{TG}(S \text{ after } o_j)$

Given a specification $Spec$ with initial state s_0 , $\mathbf{TG}(\{s_0\})$ generates a *test suite* for $Spec$. The first possible option in the algorithm states that at any moment the test process can stop indicating that the execution up to this point has been satisfactory. The second option may exercise input i and continue with test t_i . Alternatively it can accept any possible output. If the output is not included in the specification, the test fails. If instead the output is considered, it is accepted and the testing process continues. The third option is similar to the previous one except that it considers the possibility of quiescence instead of inputs. When the absence of an output (i.e., label δ) is observed, the test fails if quiescence is not accepted and otherwise continues with the selected execution. \mathbf{TG} can produce a (possibly infinitely large) test suite T , for which a system \mathcal{I} **passes** T if \mathcal{I} **io**co $Spec$ and, conversely, \mathcal{I} **io**co $Spec$ if \mathcal{I} **passes** T . The former property is called *soundness* and the latter is called *exhaustiveness*. A test suite is *complete*, if and only if it is both sound and exhaustive. We refer to the original work of Tretmans [120, 121] for more details and intuitions about **io**co, \mathcal{P} and \mathbf{TG} .

2.4 Conformance Relations

Input-output conformance is a convenient vehicle to check whether interacting discrete state systems adhere to some specification. For continuous systems (such as, for example, cyber-physical systems), input-output conformance is not sufficient, because it is not suitable to check conformance regarding the timing behaviour of a system. To explicitly consider time, we here use generalised timed traces (cf. Section 2.2) to define conformance relations. A straightforward conformance notion considers two GTTs as conformant if at every time instant the distance between the values of the individual traces is below some fixed threshold ϵ .

Definition 2.4. Let $\mu_1 : \mathcal{T}_1 \rightarrow X$ and $\mu_2 : \mathcal{T}_2 \rightarrow X$ be two GTTs with time domains \mathcal{T}_1 and \mathcal{T}_2 . Furthermore, let $d : X \times X \rightarrow \overline{\mathbb{R}}_{\geq 0}$ be a distance function on X and $\epsilon \in \overline{\mathbb{R}}_{\geq 0}$ be a threshold for tolerated deviations of values. Then, μ_1 and μ_2 are *trace conformant* with d and ϵ , denoted $\mathbf{TraceConf}_{d,\epsilon}(\mu_1, \mu_2)$, if and only if $\mathcal{T}_1 = \mathcal{T}_2$ and for all $t \in \mathcal{T}_1$ it holds that $d(\mu_1(t), \mu_2(t)) \leq \epsilon$.

More advanced conformance notions (based on [4, 81]) use the explicit occurrence of time in GTTs to consider two traces with certain time shifts as conformant. We present two conformance notions that will be relevant in this thesis.

Both are parametrised by a value threshold $\epsilon \in \overline{\mathbb{R}}$ and a time threshold $\tau \in \overline{\mathbb{R}}$. *Hybrid conformance* relates two GTTs as conformant, whenever the two traces can be shifted by up to τ on the time axis, such that, after such a retiming, at every time instant the distance between the values of the individual traces is below ϵ . Notably, hybrid conformance allows at every time t to move along the time axis independent of the retiming before t . *Skorokhod conformance* is stricter in that sense. It considers a retiming function r that describes for the full time domain how to move along the time axis. To prevent the aforementioned characteristic of hybrid conformance, Skorokhod conformance requires r to be a strictly increasing continuous bijection.

Definition 2.5. Let $\mu_1 : \mathcal{T}_1 \rightarrow X$ and $\mu_2 : \mathcal{T}_2 \rightarrow X$ be two GTTs with time domains \mathcal{T}_1 and \mathcal{T}_2 . Furthermore, let $d : X \times X \rightarrow \overline{\mathbb{R}}_{\geq 0}$ be a distance function on X , $\epsilon \in \overline{\mathbb{R}}_{\geq 0}$ be a threshold for tolerated deviations of values, and $\tau \in \overline{\mathbb{R}}_{\geq 0}$ a threshold for tolerated time deviations. Then, μ_1 and μ_2 are

- a) *hybrid conformant* with d , τ and ϵ , denoted $\text{HybridConf}_{d,\tau,\epsilon}(\mu_1, \mu_2)$, if and only if
- $\forall t_1 \in \mathcal{T}_1. \exists t_2 \in \mathcal{T}_2. |t_2 - t_1| \leq \tau \wedge d(\mu_2(t_2), \mu_1(t_1)) \leq \epsilon$ and
 - $\forall t_2 \in \mathcal{T}_2. \exists t_1 \in \mathcal{T}_1. |t_1 - t_2| \leq \tau \wedge d(\mu_1(t_1), \mu_2(t_2)) \leq \epsilon$
- b) *Skorokhod conformant* with d , τ and ϵ , denoted $\text{SkorConf}_{d,\tau,\epsilon}(\mu_1, \mu_2)$, if and only if \mathcal{T}_1 and \mathcal{T}_2 are intervals and there is a strictly increasing continuous bijection $r : \mathcal{T}_1 \rightarrow \mathcal{T}_2$, called *retiming*, such that
- for all $t \in \mathcal{T}_1$, $|r(t) - t| \leq \tau$, and
 - for all $t \in \mathcal{T}_1$, $d(\mu_1(t), \mu_2(r(t))) \leq \epsilon$.

We write $\text{Conf}_1 \sqsubseteq \text{Conf}_2$ whenever for all $\mu_1 : \mathcal{T}_1 \rightarrow X$ and $\mu_2 : \mathcal{T}_2 \rightarrow X$, we have that $\text{Conf}_1(\mu_1, \mu_2) \Rightarrow \text{Conf}_2(\mu_1, \mu_2)$. We write $\text{Conf}_1 \sqsubset \text{Conf}_2$ whenever $\text{Conf}_1 \sqsubseteq \text{Conf}_2$ and $\neg \text{Conf}_2 \sqsubseteq \text{Conf}_1$.

2.5 Hyperproperties & Self-composition

The formal characterisations of software doping we present in this thesis reason about multiple execution traces simultaneously. That is, we cannot in general express these characterisations as predicates over single traces, but we can express them as predicates over sets of traces. The former are typically called *trace properties*, while the latter are *hyperproperties* [35].

In this section we consider a fragment of hyperproperties that can be characterised by predicates that reason over two traces. Such properties can be checked

using a self-composition technique by Barthe et al. [12]. We recapitulate this technique in a simplified form.

To model programs, let a function $\eta : \text{Var} \rightarrow \text{Val}$, mapping variables to values, be a *state*². We denote by P a *program* – without putting any constraints on how this program is represented. We assume that the semantics of P is represented by a *state transformer*, i.e., a function \Downarrow , where $(P, \eta) \Downarrow \eta'$ denotes that a program P initially takes values according to η and terminates in state η' . $(P, \eta) \Downarrow \perp$ indicates that the program P does not terminate for initial state η . For convenience, we denote by $\eta \models \phi$ that a predicate ϕ holds on a state η . Further, we assume that the behaviour of programs is deterministic. That is, whenever $(P, \eta) \Downarrow \eta_1$ and $(P, \eta) \Downarrow \eta_2$, then $\eta_1 = \eta_2$ (where η_1 and η_2 may be either a state or \perp).

Barthe et al. assume *indistinguishability criteria*, which are binary relations of states. In this thesis, we consider a binary relation $\mathcal{I} \in (\text{Var} \rightarrow \text{Val}) \times (\text{Var} \rightarrow \text{Val})$ as an indistinguishability criterion; $(\eta_1, \eta_2) \in \mathcal{I}$ corresponds to the notion $\eta_1 \sim_{id}^{\mathcal{I}} \eta_2$ in the original work. Two states $\eta_1, \eta_2 : \text{Var} \rightarrow \text{Val}$ are \mathcal{I} -*indistinguishable* if and only if $(\eta_1, \eta_2) \in \mathcal{I}$. For two indistinguishability criteria \mathcal{I} (for initial states) and \mathcal{I}' (for terminal states), a program P is *termination-sensitive (TS)* $(\mathcal{I}, \mathcal{I}')$ -*secure* if and only if for all states η_1, η_2 with $(\eta_1, \eta_2) \in \mathcal{I}$, and every state η'_1 it holds that if $(P, \eta_1) \Downarrow \eta'_1$, then there is some state η'_2 , such that $(P, \eta_2) \Downarrow \eta'_2$ and $(\eta'_1, \eta'_2) \in \mathcal{I}'$ [12, Definition 1]. This characterisation reasons along two independent executions of the program (thus, it belongs to the class of hyperproperties). Barthe et al. reformulate this characterisation by means of composition of P with a copy of itself where all variables are renamed to fresh variables. With this characterisation it becomes possible to reason along the execution of a *single* program. Concretely, let \vec{x} denote the vector of all variables in P . Then, we denote by $P[\vec{x}/\vec{x}']$ that every occurrence of a variable x in P is substituted by a fresh variable x' (that w.l.o.g. is not in \vec{x}). Thus, we denote the above mentioned self-composition by $P; P[\vec{x}/\vec{x}']$. Similarly, the states on which the program operates must be composed. We define the composition of two states η_1 and η_2 , denoted $\eta = \eta_1 \oplus \eta_2$, as $\eta(x) = \eta_1(x)$ if $x \in \vec{x}$ and $\eta(x') = \eta_2(x')$ if $x' \in \vec{x}'$. With this, P is TS $(\mathcal{I}, \mathcal{I}')$ -secure if and only if for all states η_1, η_2 with $(\eta_1, \eta_2) \in \mathcal{I}$, and every state η'_1 it holds that if $(P, \eta_1) \Downarrow \eta'_1$, then there is some state η'_2 , such that $(P; P[\vec{x}/\vec{x}'], \eta_1 \oplus \eta_2) \Downarrow \eta'_1 \oplus \eta'_2$ and $(\eta'_1, \eta'_2) \in \mathcal{I}'$.

Termination-sensitive $(\mathcal{I}, \mathcal{I}')$ -security of a concrete program P can be proven using Dijkstra’s weakest (conservative) precondition. The term $\text{wp}(P, Q)$ denotes the weakest precondition [45] for a predicate Q and program P . The equations to deduce the weakest precondition are shown in Figure 2.1. wp is sound and complete in the sense that some predicate R implies the weakest precondition

²We remark that Barthe et al. model states as tuples and call it ‘memory’.

$$\begin{aligned}
\text{wp}(x := e, Q) &= Q[e/x] \\
\text{wp}(\text{if } b \text{ then } P_1 \text{ else } P_2 \text{ end if}, Q) &= b \Rightarrow \text{wp}(P_1, Q) \wedge \neg b \Rightarrow \text{wp}(P_2, Q) \\
\text{wp}(P_1; P_2, Q) &= \text{wp}(P_1, \text{wp}(P_2, Q)) \\
\text{wp}(\text{while } b \text{ do } S \text{ end do}, Q) &= \exists k : k \geq 0 : H_k(Q) \\
\text{where } H_0(Q) &= \neg b \wedge Q \text{ and } H_{k+1}(Q) = (b \wedge \text{wp}(P, H_k(Q))) \vee H_0(Q)
\end{aligned}$$

Figure 2.1: Equations for the wp calculus

$\text{wp}(P, Q)$ if and only if for any initial state η that satisfies R (i.e., $\eta \models R$), there exists some state η' , such that $(P, \eta) \Downarrow \eta'$ and $\eta' \models Q$ [12, Equation (4)]. In particular, $\text{wp}(P, \text{true})$ denotes the weakest precondition for program P to terminate. To apply wp reasoning to $(\mathcal{I}, \mathcal{I}')$ -security, the indistinguishability criteria must be transformed into first-order predicates. Barthe et al. denote by $\mathbf{I}(\mathcal{I})$ the first-order predicate that represents \mathcal{I} using the variables occurring in the (self-composition of the) program P . Thus, for two states η_1 and η_2 , it must hold that $\eta_1 \oplus \eta_2 \models \mathbf{I}(\mathcal{I})$ if and only if $(\eta_1, \eta_2) \in \mathcal{I}$. Finally, we get the following proposition [12, Proposition 3]:

Proposition 2.6. Let P be a program, \vec{x} be a vector of all variables occurring in P , and $\mathcal{I}, \mathcal{I}'$ two indistinguishability criteria. Then P is termination-sensitive $(\mathcal{I}, \mathcal{I}')$ -secure if and only if $\mathbf{I}(\mathcal{I}) \wedge \text{wp}(P, \text{true}) \Rightarrow \text{wp}(P; P[\vec{x}/\vec{x}'], \mathbf{I}(\mathcal{I}'))$.

2.6 Temporal Logics

We will characterise notions of cleanness using temporal logics. Depending on the concrete cleanness definition, we need one of the logics that we recap in this section.

2.6.1 HyperLTL

HyperLTL [34] is a temporal logic for the specification of hyperproperties of reactive systems. HyperLTL extends linear-time temporal logic (LTL) with trace quantifiers and trace variables, which allow the logic to refer to multiple traces at the same time. It is more expressive than the framework presented in Section 2.5. We refer to the work by Coenen et al. [37] for a comparison of the expressiveness of HyperLTL and other logics. The problem of model checking a HyperLTL formula over a finite-state model is decidable [57]. In the following,

we interpret a program as a set $C \subseteq (2^{\text{AP}})^\omega$ of infinite traces over a set AP of atomic propositions.

Let π be a *trace variable* from a set \mathcal{V} of trace variables. A *HyperLTL formula* is defined by the following grammar:

$$\begin{aligned} \psi &::= \exists\pi. \psi \mid \forall\pi. \psi \mid \phi \\ \phi &::= a_\pi \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{X}\phi \mid \phi \mathcal{U} \phi \end{aligned} \quad (2.1)$$

The quantifiers \exists and \forall quantify existentially and universally, respectively, over the set of traces. For example, the formula $\forall\pi. \exists\pi'. \phi$ means that for every trace π there exists another trace π' such that ϕ holds over the pair of traces. If no universal quantifier occurs in the scope of an existential quantifier, and no existential quantifier occurs in the scope of a universal quantifier, we call the formula *alternation-free*. In order to refer to the values of the atomic propositions in the different traces, the atomic propositions are indexed with trace variables: for some atomic proposition $a \in \text{AP}$ and some trace variable $\pi \in \mathcal{V}$, a_π states that a holds in the initial position of trace π . The temporal operators and Boolean connectives are interpreted as usual. In particular, $\mathsf{X}\phi$ means that ϕ holds in the next state of every trace under consideration. Likewise, $\phi \mathcal{U} \phi'$ means that ϕ' eventually holds in every trace under consideration at the same point in time, provided ϕ holds in every previous instant in all such traces. We also use the standard derived operators: $\diamond\phi \equiv \text{true} \mathcal{U} \phi$, $\square\phi \equiv \neg\diamond\neg\phi$, and $\phi \mathcal{W} \phi' \equiv \phi \mathcal{U} \phi' \vee \square\phi$.

A *trace assignment* is a partial function $\Pi : \mathcal{V} \rightarrow (2^{\text{AP}})^\omega$ that assigns traces to variables. Let $\Pi[\pi := t]$ denote the same function as Π except that π is mapped to the trace t . The trace assignment suffix $\Pi[k..]$ is defined by $\Pi[k..](\pi) = \Pi(\pi)[k..]$. By $\Pi \models_C \psi$ we mean that formula ψ is satisfied by the program C under the trace assignment Π . Satisfaction is recursively defined as follows.

$$\begin{aligned} \Pi \models_C \exists\pi. \psi &\quad \text{iff} \quad \Pi[\pi := t] \models_C \psi \text{ for some } t \in C \\ \Pi \models_C \forall\pi. \psi &\quad \text{iff} \quad \Pi[\pi := t] \models_C \psi \text{ for every } t \in C \\ \Pi \models_C a_\pi &\quad \text{iff} \quad a \in \Pi(\pi)[0] \\ \Pi \models_C \neg\phi &\quad \text{iff} \quad \Pi \not\models_C \phi \\ \Pi \models_C \phi_1 \vee \phi_2 &\quad \text{iff} \quad \Pi \models_C \phi_1 \text{ or } \Pi \models_C \phi_2 \\ \Pi \models_C \mathsf{X}\phi &\quad \text{iff} \quad \Pi[1..] \models_C \phi \\ \Pi \models_C \phi_1 \mathcal{U} \phi_2 &\quad \text{iff} \quad \text{there exists } k \geq 0 \text{ s.t. } \Pi[k..] \models_C \phi_2 \text{ and} \\ &\quad \text{for all } 0 \leq j < k, \Pi[j..] \models_C \phi_1 \end{aligned}$$

We say that a program C *satisfies* a HyperLTL formula ψ if it is satisfied under the empty trace assignment, that is, if $\emptyset \models_C \psi$.

2.6.2 STL

LTL enables reasoning over traces $\sigma \in (2^{\text{AP}})^\omega$ which are of discrete nature with respect to the time domain they represent. With each literal in the trace representing a time step, σ can equivalently be viewed as a function $\mathbb{N} \rightarrow 2^{\text{AP}}$. One variation of LTL is *Signal Temporal Logic* (STL) [48, 87], which instead is used for reasoning over real-valued signals that may change in value along an underlying time domain. A signal is a function $s : \mathcal{T} \rightarrow \mathbb{R}$ where \mathcal{T} is the time domain. Thus, a signal is a generalised timed trace with value domain \mathbb{R} . This can be lifted to multi-dimensional signals $w(t) = (s_1(t), \dots, s_n(t))$, mapping each time point to some element of \mathbb{R}^n . We refer to such a $w : \mathcal{T} \rightarrow \mathbb{R}^n$ as a (discrete-time or continuous-time) *trace* of *width* n in the sequel. Thus, a multi-dimensional signal is a GTT with value domain \mathbb{R}^n .

STL formulas can express properties of systems that are defined as sets $M \subseteq (\mathcal{T} \rightarrow \mathbb{R}^n)$ of traces of some fixed width n , basically by making the atomic properties refer to booleanizations of the signal values. The syntax of the variant of STL that we use in this paper is as follows, where $f \in \mathbb{R}^n \rightarrow \mathbb{R}$:

$$\phi ::= \top \mid f > 0 \mid \neg\phi \mid \phi \wedge \psi \mid \phi \mathcal{U} \psi .$$

STL replaces atomic propositions by *threshold predicates* of the form $f > 0$, which hold if and only if function f applied to the signal values at the current time returns a positive value. The Boolean operators and the Until operator \mathcal{U} are very similar to those of HyperLTL. The Next operator \mathcal{X} is not part of STL, because “next” is without precise meaning in continuous time. The definitions of the derived operators \diamond , \square and \mathcal{W} are the same as for HyperLTL. Formally, the *Boolean semantics* of an STL formula ϕ at time point $t \in \mathcal{T}$ for a trace $w = (s_1, \dots, s_n)$ is defined inductively:

$$\begin{aligned} w, t &\models \top \\ w, t &\models f > 0 && \text{iff } f(s_1(t), \dots, s_n(t)) > 0 \\ w, t &\models \neg\phi && \text{iff } w, t \not\models \phi \\ w, t &\models \phi \wedge \psi && \text{iff } w, t \models \phi \text{ and } w, t \models \psi \\ w, t &\models \phi \mathcal{U} \psi && \text{iff exists } t' \geq t \text{ s.t. } w, t' \models \psi \text{ and for all } t'' \in [t, t'), w, t'' \models \phi \end{aligned}$$

A system M satisfies a formula ϕ , denoted $M \models \phi$, if and only if for every $w \in M$ it holds that $w, 0 \models \phi$.

2.6.3 Probabilistic Falsification

Given a temporal logic formula, we will aim at finding a trace that violates the formula. This process is called *falsification* – the goal is to *falsify* a formula. A

technique that has been proved useful to falsify STL formulas is called *probabilistic falsification* [2, 95]. This technique assumes a *robustness estimate* function \mathcal{R} that guides the falsification process. \mathcal{R} must satisfy two important conditions: whenever, for some trace w , $\mathcal{R}(w) > 0$ the property that is supposed to be falsified must be satisfied for w , and whenever $\mathcal{R}(w) < 0$, it must be violated.

To this end, STL has been extended by a *quantitative semantics* [48, 2, 54]:

$$\begin{aligned} \rho(\top, w, t) &= \infty \\ \rho(f > 0, w, t) &= f(s_1(t), \dots, s_n(t)) \\ \rho(\neg\phi, w, t) &= -\rho(\phi, w, t) \\ \rho(\phi \wedge \psi, w, t) &= \min(\rho(\phi, w, t), \rho(\psi, w, t)) \\ \rho(\phi \mathcal{U} \psi, w, t) &= \sup_{t' \geq t} \min\{\rho(\psi, w, t'), \inf_{t'' \in [t, t']} \rho(\phi, w, t'')\} \end{aligned}$$

Indeed, this semantics induces a robustness estimate; whenever $\rho(\phi, w, t) \neq 0$, its sign indicates whether $w, t \models \phi$ holds in the Boolean semantics. For any STL formula ϕ , trace w and time t , if $\rho(\phi, w, t) > 0$, then $w, t \models \phi$ holds, and if $\rho(\phi, w, t) < 0$, then $w, t \models \phi$ does not hold.

We remark that in this thesis, we restrict the Until operator to be untimed, i.e., instead of allowing $\mathcal{U}_{[a,b]}$ for arbitrary bounds $a, b \in \mathbb{R}_{\geq 0}$, we enforce $a = 0$ and $b = \infty$. With only the untimed Until operator, the continuous and Boolean semantics coincide [54].

The robustness of an STL formula ϕ is its quantitative value at time 0, that is, $\mathcal{R}_\phi(w) := \rho(\phi, w, 0)$. Falsifying a formula ϕ for a system M boils down to a search problem with the goal condition $\mathcal{R}_\phi(w) < 0$. Successful falsification algorithms solve this problem by understanding it as the optimisation problem $\text{minimise}_{w \in M} \mathcal{R}_\phi(w)$. Algorithm 2.1 [2, 95] sketches an algorithm for Monte-Carlo Markov Chain falsification, which is based on acceptance-rejection

sampling [32]. We depict a version of the algorithm that works on system traces instead of an input space. An input to the algorithm is an initial trace w together with a computable robustness function \mathcal{R} . Robustness computation for finite timed traces of simulations of a system has been discussed in the literature [48, 54]; we omit this discussion here. The third input PS is a proposal

Algorithm 2.1 Monte-Carlo falsification

Input: w : Initial trace, \mathcal{R} : Robustness function, PS: Proposal Scheme

Output: $w \in M$

- 1: **while** $\mathcal{R}(w) > 0$ **do**
 - 2: $w' \leftarrow \text{PS}(w)$
 - 3: $\alpha \leftarrow \exp(-\beta(\mathcal{R}(w') - \mathcal{R}(w)))$
 - 4: $r \leftarrow \text{UniformRandomReal}(0, 1)$
 - 5: **if** $r \leq \alpha$ **then**
 - 6: $w \leftarrow w'$
 - 7: **end if**
 - 8: **end while**
-

scheme that proposes a new trace to the algorithm based on the previous one (line 2). The parameter β (used in line 3) can be adjusted during the search and is a means to avoid being trapped in local minima (and, thus, preventing to find a global minimum). Any two traces w and $w' \in \mathbb{M}$ with robustness values $\mathcal{R}(w)$ and $\mathcal{R}(w')$ are sampled with probability proportional to $\frac{e^{-\beta\mathcal{R}(w)}}{e^{-\beta\mathcal{R}(w')}}$ (lines 3-6). The algorithm seeks to minimise \mathcal{R} over the system's traces \mathbb{M} , and terminates when it finds a trace with a negative robustness value, i.e., a trace that violates the STL property from which \mathcal{R} is derived.

2.6.4 HyperSTL*

An extension of STL is STL* [27], which enables reasoning about signal values at different time points. To this end, STL* introduces a new operator $*_i\varphi$ that *freezes* the signal values at the current time and makes them accessible to predicates in φ so that they can compare the current signal values to the frozen signal values. For example, the formula $\diamond_{[0,10]} *_1[\Box(x^{*1} > x)]$ is satisfied for traces, where in the initial ten time units signal x has its global maximum, i.e., all future values of x are smaller than the frozen value x^{*1} .

Recently, STL* was extended to HyperSTL* to express hyperproperties. The extension follows conceptually the extension of LTL to HyperLTL, i.e., trace quantifiers are added to the syntax and trace variables are added to the index of signal variables. This extension was proposed in our LMCS paper [20]; HyperSTL* was primarily developed by co-author Rayna Dimitrova.

The syntax of HyperSTL* formulas is defined by the following grammar.

$$\begin{aligned} \psi &::= \exists\pi.\psi \mid \forall\pi.\psi \mid \phi, \\ \varphi &::= \alpha \mid \top \mid \neg\phi \mid \phi \vee \phi \mid \phi \mathcal{U}_J \phi \mid \phi \mathcal{S}_J \phi \mid *_i \phi \end{aligned}$$

There are several differences to HyperLTL in Section 2.6.1. The freeze operator $*_i$ has the same meaning as in STL*. The until operator \mathcal{U} has an index J that allows to specify in which time frame the right operand of \mathcal{U} must hold. HyperLTL and our (simplified) syntax of STL implicitly assume $J = [0, \infty)$. The since operator \mathcal{S} is the counterpart of \mathcal{U} to reason about the past. That is, the right operand was satisfied previously (in time window J) and after that, the left operand has always been satisfied until now. Lastly, α is an atomic predicate over the current and all frozen signals.

The derived operators \diamond and \Box inherit the time window J from the until operator; $\diamond_J \phi$ is defined as $\top \mathcal{U}_J \phi$ and $\Box_J \phi$ is defined as $\neg \diamond_J \neg \phi$. Similar derived operators exist for the past. $\diamond_J \phi$ is defined as $\top \mathcal{S}_J \phi$ and $\Box_J \phi$ as $\neg \diamond_J \neg \phi$. We refer to the original papers for the full definitions of STL* [27] and HyperSTL* [20, Section 6].

Hybrid Conformance Dimitrova developed HyperSTL* characterisations for hybrid conformance, trace conformance and Skorokhod conformance (cf. Definitions 2.4 and 2.5)³ [20].

We will later use her characterisation of hybrid conformance. Let π_1 and π_2 be two trace variables, and let τ and ϵ be non-negative rational constants. We can express hybrid conformance for the signals in vector \vec{x} w.r.t. a distance function d and thresholds τ and ϵ , i.e., $\text{HybridConf}_{d,\tau,\epsilon}$, as follows:

$$\varphi_{d,\tau,\epsilon}^{\text{HybridConf}} = \left(\begin{aligned} & \left(\Box * _1 \left(\Diamond_{[0,\tau]} d(\vec{x}_{\pi_2}, \vec{x}_{\pi_1}^{*1}) \leq \epsilon \vee \Diamond_{[0,\tau]} d(\vec{x}_{\pi_2}, \vec{x}_{\pi_1}^{*1}) \leq \epsilon \right) \right) \wedge \\ & \left(\Box * _2 \left(\Diamond_{[0,\tau]} d(\vec{x}_{\pi_1}, \vec{x}_{\pi_2}^{*2}) \leq \epsilon \vee \Diamond_{[0,\tau]} d(\vec{x}_{\pi_1}, \vec{x}_{\pi_2}^{*2}) \leq \epsilon \right) \right) \end{aligned} \right) \quad (2.2)$$

where $d(\vec{x}, \vec{x}')$ is an arithmetic expression characterizing the distance function d . Note that the trace variables π_1 and π_2 are not quantified; the formula is meant to be a sub-formula in a larger formula.

Intuitively, the formula states that for every time point on the trace described by π_1 it holds that within τ time units in the past or in the future, there exists a point on the trace described by π_2 where the value is ϵ -close to the value of π_1 at the current time point, and symmetrically for the other direction with traces π_1 and π_2 swapped.

2.7 Diesel Emissions

New car models must pass a homologation process before they are legally allowed to be sold to customers. This process has two purposes. First, it is meant to evaluate key figures of the car model that allow customers to estimate the ecological and economic footprint of the car. In particular, these numbers provide reference values for the emissions and the fuel consumption of the car. Obviously, it is important that these numbers for different cars are *comparable*, so that the consumer is able to compare two cars w.r.t. to ecological and economic footprints. To ensure this comparability, all cars are tested under equal, precisely defined conditions.

Part of these conditions is a test cycle that the car has to drive on a chassis dynamometer while emissions and fuel consumption are measured. For example, the *New European Driving Cycle* (NEDC) [124] is one of these cycles; it is depicted in Figure 2.2. The NEDC consists of four repetitions of an elementary *urban driving cycle* (UDC) followed by one *extra urban driving cycle* (EUDC).

³She also developed HyperSTL* characterisations for the conformance-based cleanness notions that will be introduced in Section 3.4 of this thesis.

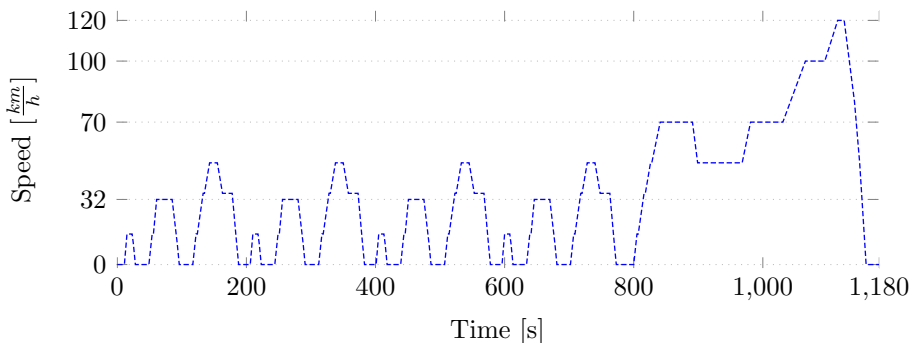


Figure 2.2: NEDC speed profile.

Each test run is preceded by a preconditioning phase (PreCon), in which three EUDCs are driven consecutively. Between PreCon and the test, the vehicle has to cool down for 6 to 36 hours at an ambient temperature between 20 and 30 degrees Celsius. In the EU, the NEDC test cycle has recently been superseded by the improved *Worldwide harmonized Light-duty vehicles Test Cycles* (WLTC) [118].

The second purpose of the homologation process is to enforce ecological requirements that are necessary to protect the environment and the health of humans and animals. These requirements are tightened regularly. Typically, a requirement is defined by a threshold for the amount of a certain substance that is emitted by the car; tightening of a requirement typically manifests in the threshold being decreased.

To easily comprehend under which requirements a car passed the homologation process, the EU uses terms like “Euro 5” or “Euro 6b” to refer to a particular test cycle or a combination of thresholds. Of particular importance in this thesis are the norms EURO 6B, EURO 6D-TEMP and EURO 6D. EURO 6B entails the NEDC, while EURO 6D-TEMP and EURO 6D use the newer WLTC. Also, there are subtle differences regarding the thresholds for nitric oxides (NO and NO₂; we will use NO_x to refer to both simultaneously). Under lab conditions, i.e., when the car is tested on a chassis dynamometer with the standard test cycle, all of the three norms define a threshold of 80 mg/km. The thresholds differ under real driving conditions: EURO 6B does not define a threshold for real-driving conditions; EURO 6D-TEMP defines a threshold of 168 mg/km and EURO 6D defines a threshold of 120 mg/km.

The EU regulation [118] also defines the precise conditions for real-world tests,

	Urban	Rural	Motorway
Ratio Range [%]	[29, 44]	[23, 43]	[23, 43]
Speed Range [km/h]	[0, 60]]60, 90]]90, 160]
Distance [km]	≥ 16	≥ 16	≥ 16
Additional Constraints	stop percentage between 6% and 30% of urban time; average velocity in range [15, 40]km/h		> 100km/h for at least 5mins
Temperature [K]	moderate: [273, 303]; extended: [266, 273[or]303, 308]		
Relative Altitude [m]	start and end point altitudes must not differ by more than 100		
Absolute Altitude [m]	moderate: < 700; extended:]700, 1300]		
Speed Limit [km/h]	145 (]145,160] for at most 3% of motorway time)		

Table 2.1: Some constraints for the three modes of RDE tests [70].

which must be driven on public roads, under usual traffic conditions, and for which the above thresholds apply. In the regulation, these tests are called *Real-Driving Emissions* (RDE) tests. The RDE requirements put constraints on the allowed routes, speeds and altitudes, and also the trip dynamics, that is, how sharp the car must be accelerated and decelerated. For example, RDE tests must comprise three modes: the urban, the rural, and the motorway mode covering different speed ranges and each making up approximately one third of the total trip distance. Table 2.1 provides an overview of the constraints for all three modes. We provide more details for these requirements in Section 2.8.2.

Also part of governmental regulations is the obligation to equip every car (powered by a combustion engine) with an *On-Board Diagnostics* (OBD) interface [117] to obtain diagnostic real-time data. OBD is based on the the CAN-based protocol, which is query-answer based. Every sensor has a unique parameter id (PID) and the current value for a PID can be queried any time. The amount of data offered through OBD depends on the type of engine, emission cleaning system and other components of the car, and company policies defining the degree of transparency a manufacturer is willing to offer.

In an official RDE test, a calibrated portable emissions measurement system (PEMS) is connected to the car’s exhaust pipe and to the OBD interface. It measures the amount of several gases and particles emitted by the car, and combines this information with the information received from the OBD interface. The costs

of a PEMS are in the order of €250,000. However, there are several minimal combinations of OBD data which can be combined to get a good approximation of emitted gases. Köhl et al. [82] successfully performed RDE tests with an Audi A7 solely using OBD data.

The Diesel Scandal In 2015, the Diesel Emissions Scandal unveiled an uncomfortable truth about the work of many car manufacturers. Millions of diesel powered passenger cars were equipped with tampered emission cleaning systems; during official test situations they performed commendably, but in real driving situations they excessively polluted the environment most of the time. Among the early discoveries and most severe cases is Volkswagen. Their cars were equipped with engine control units provably [3, 40] containing software components to detect whether the car is undergoing an official emissions test according to the, then effective, admission regulations. These regulations [124] were still using the NEDC test cycle, which has the weakness of enforcing a very unnatural driving pattern largely containing constant speed phases and repeating patterns. Thus, it was easily possible to be detected.

One of the first who understood what happened inside the 2015 affected Volkswagen cars was Felix Domke [47], who is associated with the Chaos Computer Club. His findings have also been published in a scientific contribution [40]. To defeat the regulations, the VW systems contain (as part of obfuscated code) pairs of piecewise linear functions that delineate certain regions in the time-distance domain. Figure 2.3 shows one of these pairs. The region of interest is the white region enclosed by the grey areas, confined by the function pair. The dark line inside this region represents the distance over time a car travels according to the NEDC test cycle. The logic of the emissions cleaning system is set up such that whenever the distance travelled stays within the white region (as it is the case for the NEDC itself) the emission cleaning is carried out as efficiently as possible. However, once a grey area is touched or entered, the effectiveness of the cleaning system is reduced significantly [40], and stays like that until engine restart.

2.8 Runtime Monitoring for Real Driving Emissions

As explained in Section 2.7, real driving emissions test became part of the official emissions regulation in the European Union. A specification document [118] spells out precise preconditions a trip, i.e., a trajectory driven with a car, has to satisfy in order to count as a valid RDE test (some of the preconditions are shown in Table 2.1). This specification document consists largely of prose, so, from an informatics perspective, the specification is informal. Köhl et al. [82]

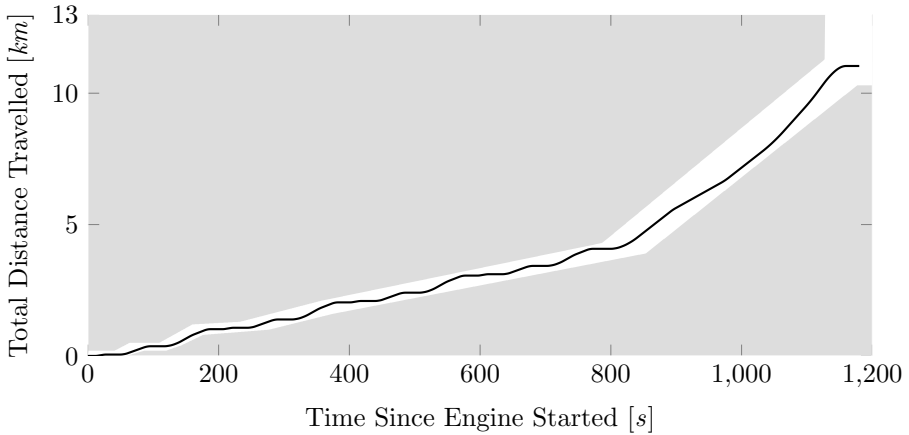


Figure 2.3: White region is encoded by a pair of piecewise linear functions found in several Volkswagen ECUs. Black line represents the distance travelled when following the NEDC speed profile on a chassis dynamometer.

formalised this specification using the specification language RTLOLA (named after the eponymous monitoring framework). The subsequent content in this section is largely taken word-by-word from [22].

2.8.1 RTLOLA

RTLOLA [55, 15] is a stream-based specification language for real-time properties. An RTLOLA specification is a collection of input stream, output stream, and trigger declarations. Input streams represent data sources such as sensors or information retrieved over the OBD interface. Each output stream declaration details how to filter and refine input data to obtain relevant statistical information. Trigger declarations use this information to indicate when the system under observation reaches an undesired state such as a violation of a safety margin or a transgression of the permitted NO_x emission.

The RTLOLA toolkit generates a monitor for a given specification. The monitor receives input data from the system. This data may be asynchronous, i.e., different input sources can produce data at different points in time. Upon reception, the monitor computes output stream values and checks for satisfaction

of trigger conditions. The result can then be fed back to the system or displayed to the user.

The RTLOLA language can most easily be understood by example. Consider the following specification:

```
input velocity: Float32
output is_urban: Bool := velocity ≤ 60
```

The specification consists of one input stream and one output stream. The input stream carries the velocity of the system in km/h as a 32-bit wide floating point number. Let us consider the system to operate in an urban environment when its velocity is below 60 km/h. To this end, the boolean output stream `is_urban` indicates exactly this condition.

To extend the specification, suppose the system is not allowed to travel more than 10 km in an urban environment within 20 min. In this case, the specification can be extended by two output streams and a trigger:

```
output urban_velo :=
  if is_urban then velocity else 0.0
output urban_dist @10Hz :=
  urban_velo.aggregate(over: 20min, using: integral)
trigger urban_dist > 10
  "Travelled more than 10km in urban env."
```

The first stream, `urban_velo`, carries the velocity of the system provided it operates in an urban environment, or zero otherwise. The second one integrates the urban velocity for 20 min to compute the distance travelled as a sliding window aggregation. Notice the annotation `@10Hz`, which transforms the stream into a *periodic* one. This prompts the monitor to compute the output stream only 10 times a second and is mandatory for streams with a sliding window aggregation operation such as the integral. This mandate allows the monitor to employ an efficient algorithm for aggregation [84]. In particular, if the aggregation additionally is a list homomorphism, the monitor does not have to store input values received within the time frame. This reduces the memory footprint of the monitor drastically. Details can be found in earlier work on RTLOLA [15, 109].

Other output streams are *event-based*, i.e., they are computed when the monitor receives new input values. However, some streams depend on more than one input stream. When the monitor receives only an update for a subset of input streams due to asynchrony, it re-computes the output streams for which all relevant inputs were updated. Since the `urban_velo` stream only depends on `velocity`, it will be evaluated upon every reception of this input.

To finalize the example, the trigger declaration states that the `urban_dist` should remain below 10. Specifiers may provide a human-readable explanation

after the condition. This string can then be displayed to users.

2.8.2 From Regulation to Specification

The RDE regulation has been issued by the European Commission [118] to make exhaust emissions tests more realistic. To this end, it meticulously describes conditions a trip driven on public roads has to satisfy in order to count as a valid RDE test. The regulation itself mostly relies on natural language, however, as we shall demonstrate, the individual conditions translate naturally into a stream-based specification language such as RTLOLA. We give an overview over how the informal regulation can be specified using RTLOLA. The full specification is available online.⁴ It was first proposed by Köhl et al. [82] and was further improved in our STTT paper [22].

Some of the RDE conditions apply universally, e.g., the ambient temperature must range between 273 K and 303 K throughout the whole trip. For others, the RDE regulation differentiates between three modes characterized by the speed of the car: *urban*, *rural*, and *motorway*. Table 2.1 shows an overview over both the universal conditions and the conditions for the individual modes. Driving in each mode can be interrupted by short periods of driving in another mode, e.g., when changing the motorway the data collected by virtue of the driving speed may count towards the rural or urban environment. While modes may be interrupted, each one needs to occupy a specific share of the total distance.

The Three Modes. To identify the mode a given data record belongs to, we introduce a boolean stream for each mode being true if and only if the speed of the car is in the respective range as required for the given mode. For instance, for the rural segment:

```
| output is_rural := (60.0 < v) && (v <= 90.0)
```

This directly reflects §6.3 of the regulation [118, ANNEX IIIA] which says: “Rural operation is characterised by vehicle speeds higher than 60 and lower than or equal to 90 km/h.” Note that in contrast to other EU regulations, such as domestic market trade laws, the RDE regulation lends itself well to formalization as it clearly defines RDE tests in terms of mathematical concepts. The whole regulation assumes that records of test data come in synchronously at a fixed frequency of at least 1 Hz. Hence, the stream-based specification language RTLOLA is a perfect fit for the formalization.

For each of the modes it is required to determine the distance driven in that mode. The ratio d_m/d of the distance d_m driven in a given mode m and the

⁴<https://www.loladrives.app/scientific-background/>

distance driven overall d needs to be within a specific interval for each mode (see Table 2.1 and §6.6 of the regulation [118, ANNEX IIIA]): “The trip shall consist of approximately 34 % urban, 33 % rural and 33 % motorway driving classified by speed as described in points 6.3 to 6.5 above. ‘Approximately’ shall mean the interval of ± 10 percentage points around the stated percentages. The urban driving shall however never be less than 29 % of the total trip distance.” So, we define a stream for the total distance d :

```
output Dd := v / 3.6 * 1.0
output d @1Hz := Dd.aggregate(over: 2h, using: sum)
```

Here Dd is the distance driven since the last data record. Assuming that data is provided with a fixed frequency of 1 Hz, we calculate the distance in m from the velocity v in km/h by dividing by 3.6 to obtain m/s and then multiplying the result with 1.0 s. To obtain the total distance d , we use RTLOLA’s aggregation functions and simply take the sum of Dd over the last 2 h, i.e., the maximal duration of a test.

Obtaining the distance travelled in a specific mode is also straightforward: Remember that we have a boolean stream for each of the modes. We first define an auxiliary stream for each mode whose value is Dd when the car is in the respective mode and 0 otherwise. For instance, for the rural mode, we define r_d_a :

```
output r_d_a := if is_rural then Dd else 0.0
```

Using aggregation functions again, we obtain the distance travelled in the rural mode with:

```
output r_d @1Hz := r_d_a.aggregate(
  over: 2h, using: sum
)
```

So, §6.6 of the RDE regulation translates in part to the following condition for the rural mode:

```
0.23 <= (if d > 0.0 then r_d / d else 0.0)
&& (if d > 0.0 then r_d / d else 0.0) <= 0.43
```

Doing this for all the other modes enables us to compute whether the ratio condition defined in §6.6 is satisfied or not. Analogously to the distance ratios, one defines streams for the remaining conditions.

Driving Dynamics. A more complex part of the RDE regulation⁵ concerns the driving dynamics. The intuition is simple: Too aggressive driving leads to highly

⁵See Appendix 7a of ANNEX IIIA [118].

increased emissions, hence, at least for testing, it would not be fair for the manufacturer to have their car evaluated based on unrealistically aggressive driving. Likewise, too restrained driving is unrealistic as well. Hence, the RDE regulation specifies lower and upper bounds on the driving dynamics. The driving dynamics is defined as the product of speed and acceleration:

```
|   output dyn := v * a / 3.6
```

v is the speed in km/h and a the acceleration in m/s^2 . The resulting stream dyn captures the dynamics in m^2/s^3 . For an analysis of the dynamics in a rural environment, the r_dyn stream mirrors dyn besides discarding values not acquired in a rural environment.

The requirements for the lower bound of the dynamics consider the relative positive acceleration (RPA). The RPA is the sum of the positive values of the dynamics. The regulation defines a dynamics $\text{dyn} = \frac{va}{3.6}$ as *positive*, if the acceleration a is at least to 0.1 m/s^2 . As shown below, our RTLOLA specifications computes the RPA by first generating an output stream rpa_va that copies dyn but replaces all values computed with a non-positive acceleration by zero. An output stream rpa_agg computes the sum of these values, and output stream rpa computes the RPA by dividing this sum by the length of the trip. The specification snippet below shows the computation of the RPA for the rural mode (hence, each stream name is prefixed with r_):

```
|   output r_rpa_va :=
      if a >= 0.1 && is_rural then dyn else 0.0
  output r_rpa_agg := r_rpa_va.aggregate(
    over_discrete: 7200, using: sum
  ).defaults(to: 0.0)
  output r_rpa := if r_d > 0 then r_rpa_agg
                 / r_d else 0.0
```

The threshold for the RPA depends on the average velocity. If v_{avg} is the average velocity, then the regulation requires that the RPA is above $-0.0016 \cdot v_{avg} + 0.1755$ if $v_{avg} \leq 94.05$ and above 0.025 otherwise.

The requirements for the upper bound for the dynamics consider the 95th percentile of the dynamics values. The following specification snippet shows the computation of the 95th percentile for the parts of the RDE test that belong to the rural mode:

```
|   output r_pctl_dyn @1Hz := r_dyn.aggregate(
    over_discrete: 7200, using: pctl95
  ).defaults(to: 0.0)
```

The percentile aggregation is computed for up to 7200 time steps, i.e., for a duration of 2h given the (regulation enforced) fixed frequency of 1Hz. The

computation uses the values of the rural dynamics output stream `r_dyn`. Analogous streams `u_pctl_dyn` and `m_pctl_dyn` are constructed for the urban and, respectively, motorway mode. As for the RPA, the concrete threshold depends on the average velocity v_{avg} : the 95th percentile of the dynamics must not be greater than $0.136 \cdot v_{avg} + 14.44$ if $v_{avg} \leq 74.6$ km/h and no greater than $0.0742 \cdot v_{avg} + 18.966$ otherwise.

We can now encode the validity of the (rural) dynamics as a boolean stream:

```
output r_is_dynamics_valid :=
  (if r_avg_v <= 94.05 then
    r_rpa > (-0.0016 * r_avg_v + 0.1755)
  else
    r_rpa > 0.025 )
  &&
  (if r_avg_v <= 74.6 then
    r_pctl_dyn <= (0.136 * r_avg_v + 14.44)
  else
    r_pctl_dyn <= (0.0742 * r_avg_v + 18.966) )
```

Checking Emissions. After establishing that a trip is indeed valid according to the conditions of the RDE regulation, the exhaust emissions have to be checked. If a trip is not valid, the emissions are irrelevant and the test has to be repeated. To calculate the emissions, one first needs the *Exhaust Mass Flow* (EMF), i.e., the mass of exhaust gas emitted per time unit. Based on the EMF in g/s and the measured particles per million one can then compute the emissions in g/s. From this, the amount of NO_x emission in g/km can be computed by dividing the total amount of NO_x (in gram) emitted during the test by the total trip length. For NO_x and diesel fuel the respective equations are:

```
output nox_mass_flow :=
  exhaust_mass_flowp * 0.001586 * nox_ppmp
output nox_mass_aggregated :=
  nox_mass_flow.aggregate(over: 2h, using: sum)
output nox_per_kilometer :=
  if d > 0.0 then
    nox_mass_aggregated / (d / 1000.0)
  else
    0.0
```

Appendix 4 of the RDE regulation provides a table of factors for computing the emissions. For NO_x and diesel fuel the necessary factor is 0.001586.

It remains to sum up all the emitted gases over the whole trip in order to calculate the amount of gases emitted per kilometre. For each of the gases, we

introduce a stream indicating whether the limit for the respective gas has been exceeded:

```
|   output nox_exceeded :=  
      nox_per_kilometer > 0.168
```

The exact threshold, here 168 mg/km, depends on the emission class of the vehicle.

Finally, we use a trigger to indicate when the trip is valid but the emission limits have been exceeded constituting a violation of the RDE regulation, i.e., that the vehicle did not pass the test. The boolean output stream `is_valid_test` is a conjunction of boolean streams such as `r_is_dynamics_valid` that must necessarily be true for a regulation conforming RDE test.

```
|   trigger is_valid_test && nox_exceeded
```

As soon as the condition is violated, the trigger goes of notifying the user of the violation.

3 Notions of Software Doping

Barthe et al. [11] proposed an informal characterisation of doped software systems: a system is doped, if it intentionally “has included a hidden functionality [...] against the interest of society or of the software licensee” [11]. This definition contains three requirements that are sufficient for software doping: 1) the software exhibits a behaviour that is not in the interest of a user or society, 2) the existence of this behaviour is hidden or concealed, and 3) this was done intentionally by the manufacturer. While this seems to be a reasonable characterisation for the doping of software, it remains unclear how it can be formally analysed.

Reasoning about the intentions of a manufacturer is effectively reasoning about the intentions of a legal or actual person from whom originates the intention, which is a task that is located in law enforcement rather than computer science. Whether a functionality is hidden or concealed depends on the *documentation* of the software. The line between documentations that document a functionality and those that do not is fuzzy. Unless the functionality is doubtlessly not mentioned in the documentation, determining on which side of this line a documentation is, falls into research fields of other disciplines, too. Fortunately, regarding the first of the above requirements, formally defining what a software is supposed to do is a standard task in computer science. It only remains the question how to generally express, in a formal way, that a software is working in the interest of a user or society. We slightly rephrase this requirement and will ask instead for a definition that ensures that a system meets the *expectations* of a user or society regarding a software’s behaviour. This is also in accordance with the definition of trust in Chapter 1. If a software behaves as expected, we will say that it is *clean*. Deliberately ignoring the intentionality and the concealment requirement above, we will say that a program or system is *doped* whenever it is not clean. This chapter proposes several cleanness definitions that represent different, general types of expectations of how a trustworthy program or system should behave.

A program or a system must be representable by a suitable type of computation model. The computation models introduced in Chapter 2 are relatively specific, because they are accompanied by a suitable analysis technique. In this chapter, we take a more general view on computation models that takes

an observation-based perspective on programs rather than a computation-based one. That is, we consider programs and systems as functions that receive inputs and produce outputs; how an output is computed is irrelevant. For example, the state transformers in Section 2.5 come with certain technical details, such as the existence of states and variables that can occur in programs. Essentially, state transformers represent *sequential programs*, i.e., programs that receive single inputs and produce single outputs. A simpler representation of such programs are plain functions from inputs to outputs. Due to the absence of details about how the program computes outputs, cleanness definitions for this representation become more comprehensible. Yet, they are also applicable to more concrete modelling formalisms for sequential programs.

Although the representation of sequential programs as functions is very general, it is insufficient to define cleanness for *reactive systems*, i.e., systems that continually receive inputs and produce outputs. Hence, we will dedicate a separate part of this chapter to reactive systems. In fact, we will distinguish between three conceptually different types of reactive systems. One very common representation of reactive systems considers a discrete time domain where at every step one input is consumed and one output is produced. Sometimes, though, this strong coupling of inputs and outputs is not desirable; in such cases a model that we call *mixed-IO system* may be more suitable, which allows an arbitrary alternation between inputs and outputs. Finally, if the exact time at which interactions with the system take place is relevant to decide whether the system behaves cleanly, then it is more suitable to assume a *hybrid systems* computation model. A simple representation of hybrid systems are functions that map generalised timed traces of input values to generalised timed traces of output values (cf. Section 2.2). We provide additional details for the different computation models in the subsequent sections.

Another detail that is relevant for the choice of a cleanness notion is whether a program is *parametrised*. By using parameters it is possible to distinguish different configurations of a program [36]. We will propose cleanness definitions for parametrised and non-parametrised programs and systems.

3.1 Sequential Programs

A *nondeterministic sequential program* $P : \text{In} \rightarrow 2^{\text{Out}}$ is a function P mapping inputs from a set In to outputs from a set Out . Because P may be nondeterministic, it returns a set of outputs. A program P is deterministic, if for every input it returns exactly one output, i.e., for all $i \in \text{In}$, $|P(i)| = 1$. In this case, we may also write $P : \text{In} \rightarrow \text{Out}$. A *parametrised nondeterministic sequential program* is a function $P_+ : \text{Param} \rightarrow \text{In} \rightarrow 2^{\text{Out}}$, where Param is a set of parameters each

enabling a particular instance (or configuration) of the program P_+ . For some parameter $p \in \text{Param}$, the program instance $P_+(p)$ maps inputs from a set In to outputs from a set Out . As for non-parametrised programs we call P_+ deterministic if and only if for all $p \in \text{Param}$ and $i \in \text{In}$, $|P_+(p)(i)| = 1$, in which case we also write $P_+ : \text{Param} \rightarrow \text{In} \rightarrow \text{Out}$.

The choice of whether a program is modelled as a parametrised or non-parametrised program is only a matter of style; both notions are equally expressive. To allow switching between both notions, we explicate how they can encode each other. Let $P_+ : \text{Param} \rightarrow \text{In} \rightarrow 2^{\text{Out}}$ be a parametrised nondeterministic sequential program. Then, with the input set $\text{In}' := \text{Param} \times \text{In}$ combining the parameters and inputs of P_+ in pairs, the non-parametrised program $P : \text{In}' \rightarrow 2^{\text{Out}}$ with $P(p, i) := P_+(p)(i)$ *encodes* P_+ . We denote this by $P_+ \hookrightarrow P$. The converse direction – the encoding of a non-parametrised program with a parametrised program – can be achieved in two ways. Let $P : \text{In} \rightarrow 2^{\text{Out}}$ be a non-parametrised nondeterministic sequential program. The most intuitive approach to encode P is to use a single default parameter *def*. Taking $\text{Param} = \{\text{def}\}$ eliminates the option to use a variant of a program other than the “default” one (that is implicitly given by the non-parametrised program). In this case, the inputs to the encoding program $P_+ : \text{Param} \rightarrow \text{In} \rightarrow 2^{\text{Out}}$ are passed unalteredly to the encoded program, i.e., $P_+(\text{def})(i) := P(i)$. We say that P_+ *encodes* P and denote this by $P \hookrightarrow P_+$. Alternatively, the roles of parameters and inputs may be switched: The set of parameters is defined as the inputs to P , i.e., $\text{Param} = \text{In}$, and there is only a single default input *def* that P_+ accepts, i.e., $\text{In}_+ = \{\text{def}\}$. P is then encoded by a parametrised program $P_+ : \text{Param} \rightarrow \text{In}_+ \rightarrow 2^{\text{Out}}$, where the inputs are passed as parameters to P_+ and forwarded to the original non-parametrised program: $P_+(i)(\text{def}) := P(i)$. We denote this encoding by $P \hookrightarrow_{\text{Param}} P_+$ explicitly indicating that the inputs are encoded by parameters. While the former kind of encoding appears to be the natural choice for encodings, we will later see that only the latter encoding is able to preserve cleanness properties.

In the remainder of this section, if not stated otherwise, we use the names Param , In and Out for the sets of parameters, inputs and outputs, respectively. Further, we use P to denote non-parametrised sequential programs and P_+ for parametrised sequential programs. If not stated otherwise, we assume that P is a function of type $\text{In} \rightarrow 2^{\text{Out}}$ and that P_+ is a function of type $\text{Param} \rightarrow \text{In} \rightarrow 2^{\text{Out}}$.

3.1.1 Strict cleanness

The abstract implementations of a printer shown in Figures 3.1 and 3.2 are examples for parametrised programs.

The program `PRINTER` in Figure 3.1 is parametrised by the information on the

```

procedure PRINTER(cartridge)
  if TYPE(cartridge) ∈ Compatible
  then
    READ(document)
    PRINT(stdout,document)
  else
    TURNON(alert_led)
  end if
end procedure

```

Figure 3.1: A simple printer

```

procedure PRINTER(cartridge)
  if BRAND(cartridge) = my-brand
  then
    READ(document)
    PRINT(stdout,document)
  else
    TURNON(alert_led)
  end if
end procedure

```

Figure 3.2: A doped printer

ink or toner cartridge with which it is being used. Before printing a document, it checks if the cartridge is compatible. For incompatible cartridges, the printer turns on an error LED, and for compatible cartridges, the printer receives a document as an input and prints it. The printed pages of the document are the output. Figure 3.2 shows a doped variant of the printer. Instead of checking if the cartridge is compatible, it prints documents only if the cartridge is a cartridge sold by the printer manufacturer. The manufacturer of the printer apparently has manipulated the software to favour its own cartridges.

Intuitively, we would expect from a clean printer that when comparing two compatible cartridges p_1 and p_2 and printing with both the same document i , that the output is the same for both, i.e., $\text{PRINTER}(p_1)(i) = \text{PRINTER}(p_2)(i)$. There are a few deliberations regarding the definition of cleanness that are particularly worth to mention:

1. A cleanness definition shall be based on events accessible to an external observer, i.e., it is based on the parameter-input-output behaviour of a program. From the program itself it is only known that it can be modelled as a function. Internals of the function are unknown.
2. The definition shall consider that not every pair of parameters is comparable. In the example of the printer, only cartridge parameters that are compatible with the printer shall be compared with each other. Assuming a set $\text{Compatible} \subseteq \text{Param}$ of cartridge parameters compatible with the printer, the definition must be flexible enough to restrict p_1 and p_2 to be parameters from Compatible . In a more general case, it may also be necessary to compare compatible black-cartridges with each other and compatible colour-cartridges with each other, but without comparing a compatible black- and a compatible colour-cartridge. Hence, to reason about cleanness we will work with a binary relation \approx_P that relates all combinations of parameters that are supposed to be compared with each other. This relation must be reflexive and symmetric. For the printer example, we would hence define

the comparability relation of cartridge parameters such that $p_1 \approx_P p_2$ if and only if $p_1 \in \text{Compatible}$ and $p_2 \in \text{Compatible}$. The distinction between black- and colour-cartridges can be formulated as $p_1 \approx_P p_2$ if and only if $p_1 \in \text{BlackCompatible}$ and $p_2 \in \text{BlackCompatible}$ or $p_1 \in \text{ColourCompatible}$ and $p_2 \in \text{ColourCompatible}$ (assuming that $\text{BlackCompatible} \subseteq \text{Param}$ represents all compatible black-cartridges and $\text{ColourCompatible} \subseteq \text{Param}$ represents all compatible colour-cartridges). With this definition, pairs consisting of a compatible black-cartridge and a compatible colour-cartridge would indeed not occur in \approx_P .

3. Finally, it is important that the cleanness definition shall not hinder the manufacturer to be innovative. For example, assume that the printer manufacturer provides an extra functionality that requires specialised hardware in the printer and the cartridge. Documents using the new feature are encoded in a new data format (we could compare this to the introduction of the postscript language when standard printing was based on dots or ASCII code). Figure 3.3 sketches the software of such a printer. While the printer prints all documents in the standard format with all compatible cartridges, documents in the new format require the specialised cartridge with the additional hardware. For such a printer, the requirement that compatible cartridges produce the same output for identical inputs would be violated for inputs that represent documents in the new file format. Hence, reasoning about cleanness shall concentrate only on a meaningful set of *standard inputs* $\text{StdIn} \subseteq \text{In}$. In case of the printer example, only documents with the standard file format are relevant, documents in the new file format should not be considered.

As the above list of thoughts suggests, the concrete application of a cleanness property is influenced by contextual information – e.g., a characterisation of which pairs of parameters should be considered, or the set of standard inputs StdIn . All cleanness notions in this chapter are defined w.r.t. a formal *contract* that specifies a context-dependent cleanness configuration. Definition 3.1 shows the first cleanness definition for sequential programs, called strict cleanness. It is similar to the idea presented above: For all pairs of parameters that shall be checked, every standard input $i \in \text{StdIn}$ must result in the same outputs. Hence, the contract w.r.t. which strict cleanness is checked contains a reflexive and symmetric binary relation \approx_P determining the pairs of comparable parameters, and the set StdIn of standard inputs. We represent contracts as tuples; a strict cleanness contract is a tuple of the form $\langle \approx_P, \text{StdIn} \rangle$.

Definition 3.1. A parametrised nondeterministic sequential program P_+ is *strictly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn} \rangle$, if for all pairs of parameters $p, p' \in$

Param with $\mathbf{p} \approx_P \mathbf{p}'$ and input $i \in \text{StdIn}$, it holds that $P_+(\mathbf{p})(i) = P_+(\mathbf{p}')(i)$.¹

The comparability relation for parameters in the contract is a powerful tool to restrict the combination of parameters that is supposed to be checked for cleanliness. For example, we imagine that it is a common use-case to check cleanness within pairwise disjoint sets of parameters $P_1, \dots, P_n \subseteq \text{Param}$. This can be encoded by the parameter comparability relation $\mathbf{p} \approx_P \mathbf{p}' := \mathbf{p} \in P_1 \wedge \mathbf{p}' \in P_1$, or \dots , or $\mathbf{p} \in P_n \wedge \mathbf{p}' \in P_n$. We remark that with this definition of \approx_P ,

any combination of parameters where both parameters are not part of any of the groups is not considered (i.e., there is no implicit group for parameters that are not contained in any of the groups P_1 to P_n). The parameter comparability relation is also powerful enough to encode that all variants of a program shall be compared to a reference implementation that serves as a specification. If the reference implementation is represented by a parameter \mathbf{p}_{ref} , then the parameter comparability relation defined as $\mathbf{p}_1 \approx_P \mathbf{p}_2 := \mathbf{p}_1 = \mathbf{p}_{\text{ref}} \vee \mathbf{p}_2 = \mathbf{p}_{\text{ref}}$ enforces that all program variants $P_+(\mathbf{p})$ (i.e., for all $\mathbf{p} \in \text{Param}$) are compared to the reference implementation $P_+(\mathbf{p}_{\text{ref}})$.

In many cases, program parametrisation is not needed and the parameter notation causes unnecessary clutter during reasoning about cleanliness. For such cases, Definition 3.2 defines an alternative of strict cleanliness for non-parametrised programs. Instead of using the comparability relation \approx_P , the definition weakens the constraints for inputs. For non-parametrised programs, the outputs considered for comparison are not necessarily the outputs for *identical* inputs, but instead it suffices that the inputs are *comparable* according to a \approx_I relation over inputs. As \approx_P, \approx_I must be a binary relation that is reflexive and symmetric. Thus, strict cleanliness is defined w.r.t. a contract $\langle \text{StdIn}, \approx_I \rangle$ that contains the input comparability relation instead of the parameter comparability relation.

Definition 3.2. A non-parametrised nondeterministic sequential program P is *strictly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, \approx_I \rangle$, if for all pairs of standard inputs $i, i' \in \text{StdIn}$ with $i \approx_I i'$, it holds that $P(i) = P(i')$.

¹I use a more general notion than D'Argenio et al. [42]; for the original definition with parameters of interest PIntrs , define $\mathbf{p} \approx_P \mathbf{p}'$ if and only if $\mathbf{p} \in \text{PIntrs} \wedge \mathbf{p}' \in \text{PIntrs}$.

```

procedure PRINTER(cartridge)
  if TYPE(cartridge)  $\in$  Compatible then
    READ(document)
    if ( $\neg$ NEWTYPE(document)
       $\vee$  SUPPORTSNEWTYPE(cartridge))
    then
      PRINT(stdout, document)
    else
      TURNON(alert_led)
    end if
  else
    TURNON(alert_signal)
  end if
end procedure

```

Figure 3.3: An innovative printer.

As mentioned above, choosing between parametrised and non-parametrised programs is a matter of style. Although the two variants of strict cleanness allow different kinds of configuration (via their contracts), they still are equally expressive, as Propositions 3.3 and 3.4 show.

Proposition 3.3. Let $P_+ : \text{Param} \rightarrow \text{In}_+ \rightarrow 2^{\text{Out}}$ be a parametrised nondeterministic sequential program and $P : \text{In}_\# \rightarrow 2^{\text{Out}}$ the non-parametrised nondeterministic sequential program that encodes P_+ , i.e., $P_+ \hookrightarrow P$ and $\text{In}_\# = \text{Param} \times \text{In}_+$. Further, let $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}_+ \rangle$ and $\mathcal{C} = \langle \text{StdIn}_\#, \approx_I \rangle$ be contracts with $\text{StdIn}_+ \subseteq \text{In}_+$, $\text{StdIn}_\# = \text{Param} \times \text{StdIn}_+$ and $(p, i) \approx_I (p', i')$ if and only if $p \approx_P p'$ and $i = i'$. Then, P_+ is strictly clean w.r.t. \mathcal{C}_+ if and only if P is strictly clean w.r.t. \mathcal{C} .

Proof. By definition of strict cleanness, P is strictly clean w.r.t. \mathcal{C} if and only if for all $(p_1, i_1), (p_2, i_2) \in \text{Param} \times \text{StdIn}$ it holds that $(p_1, i_1) \approx_I (p_2, i_2)$ implies $P(p_1, i_1) = P(p_2, i_2)$. From the definition of \approx_I and from $P_+ \hookrightarrow P$ we can replace the implication with the equivalent implication that $p_1 \approx_P p_2$ and $i_1 = i_2$ imply $P_+(p_1)(i_1) = P_+(p_2)(i_2)$. By unwinding the pair notation and syntactically applying the equality of i_1 and i_2 , it is equivalent to say that for all $p_1, p_2 \in \text{Param}$ and for all $i \in \text{StdIn}$, $p_1 \approx_P p_2$ implies $P_+(p_1)(i) = P_+(p_2)(i)$, which is the definition of P_+ being strictly clean w.r.t. \mathcal{C}_+ . \square

A similar statement for an encoding $P \hookrightarrow P_+$ (where inputs to P are modelled as inputs to P_+) does not hold. Strict cleanness of non-parametrised programs relates inputs as comparable by means of the \approx_I relation. Strict cleanness of parametrised programs, however, only relates parameters using a \approx_P relation, while it requires inputs to be identical. If, however, P_+ accepts inputs to P as parameters for encodings (i.e., when using the encoding $P \hookrightarrow_{\text{Param}} P_+$), every strict cleanness contract for non-parametrised programs can be translated to an equivalent contract for the parametrised encoding. This is what the following proposition shows.

Proposition 3.4. Let $P : \text{In} \rightarrow 2^{\text{Out}}$ be a non-parametrised nondeterministic sequential program and let $P_+ : \text{In} \rightarrow \{\text{def}\} \rightarrow 2^{\text{Out}}$ be the parametrised nondeterministic sequential program that encodes P using parameters as inputs, i.e., $P \hookrightarrow_{\text{Param}} P_+$. Further, let $\mathcal{C} = \langle \text{StdIn}, \approx_I \rangle$ and $\mathcal{C}_+ = \langle \approx_P, \{\text{def}\} \rangle$ be contracts with $\text{StdIn} \subseteq \text{In}$ and such that for every $i_1, i_2 \in \text{In}$, $i_1 \approx_P i_2$ if and only if $i_1 \in \text{StdIn}$ and $i_2 \in \text{StdIn}$ and $i_1 \approx_I i_2$. Then, P is strictly clean w.r.t. \mathcal{C} if and only if P_+ is strictly clean w.r.t. \mathcal{C}_+ .

Proof. By definition of strict cleanness, P_+ is strictly clean w.r.t. \mathcal{C}_+ if and only if for all $i, i' \in \text{In}$ and for all $i'' \in \{\text{def}\}$ it holds that $i \approx_P i'$ implies $P_+(i)(i'') = P_+(i')(i'')$. Applying the definition of \approx_P and replacing i'' by the only value def over which i'' is quantified, we get the equivalent statement that for all

```

procedure EMISSIONCONTROL()
  READ(throttle)
  def_dose := SCRMODEL(throttle)
  NOx := throttle3 / (2 · def_dose)
end procedure

```

Figure 3.4: Simple emission control

```

procedure EMISSIONCONTROL()
  READ(throttle)
  if throttle ∈ ThrottleTestValues then
    def_dose := SCRMODEL(throttle)
  else
    def_dose := ALTSCRMODEL(throttle)
  end if
  NOx := throttle3 / (2 · def_dose)
end procedure

```

Figure 3.5: Doped emission control

$i, i' \in \text{In}$, if $i \in \text{StdIn}$ and $i' \in \text{StdIn}$ and $i \approx_I i'$, then $P_+(i)(def) = P_+(i')(def)$. With $P \hookrightarrow_{\text{Param}} P_+$ and notational changes, it is equivalent to say that for all $i, i' \in \text{StdIn}$, $i \approx_I i'$ implies $P(i) = P(i')$, which is the definition of P being strictly clean w.r.t. \mathcal{C} . \square

3.1.2 Robust cleanness

Both variants of strict cleanness put restrictions on a program only for parameter combinations and standard inputs that are defined by the contract w.r.t. which cleanness is checked. For pairs of parameters p_1, p_2 with $p_1 \not\approx_P p_2$ or for inputs $i \notin \text{StdIn}$, the program is immediately deemed clean. This view results too mild in some cases where the change of behaviour of a program between a standard input and a non-standard but yet not-so-different input is extreme. The definition of *robust cleanness* overcomes this problem. While strict cleanness requires identical outputs for identical inputs and comparable parameters, robust cleanness expands the idea of cleanness beyond standard inputs. Conceptually, robust cleanness requires that for all pairs of comparable parameters and for every combination of a standard input and a similar (possibly non-standard) input, also the sets of outputs must be similar.

Example 3.5. Consider the electronic control unit (ECU) of a diesel vehicle, in particular its exhaust emission control module. Modern diesel cars use a selective catalytic reduction (SCR) system to reduce the amount of nitric oxides. Such systems inject a certain amount of an aqueous urea solution (also called diesel exhaust fluid or DEF) into the exhaust stream in order to lower nitric oxides (NO_x) emissions. We simplify this control problem to a minimal toy example. Figure 3.4 shows a program that reads the *throttle* position as an input and calculates the dose of DEF (stored in *def_dose*) that should be injected into the exhaust stream to reduce the NO_x emission. The last line of the program precisely models the NO_x emission by storing it in the output variable *NO_x* after a (made up) calculation directly depending on the *throttle* value and inversely

depending on the *def_dose*.

The Diesel Emissions Scandal arose precisely because emission cleaning software was instrumented so that it works as expected [47] *only if* operating in or very close to the lab testing conditions. For our simplified example, this behaviour is exemplified by the algorithm in Figure 3.5. There, the lab conditions are represented by the set `ThrottleTestValues`. If we define `StdIn` to be the set `ThrottleTestValues`, then the doped emission control system trivially meets the characterisation of *clean* given in Definition 3.2. However, this unit is intentionally programmed to defy the regulations when being unobserved and hence it falls directly within our intuition of what a doped software is. Let $\text{SCRMODEL}(x) = x^2$, $\text{ALTSCRMODEL}(x) = x$ and $\text{ThrottleTestValues} = \text{StdIn} = (0, 1]$. Then the doped program would at input 1 switch from a linear dependency between throttle and NO_x value to a quadratic dependency (i.e., from $\text{NO}_x := \text{throttle} / 2$ to $\text{NO}_x := \text{throttle}^2 / 2$). Thus, by allowing to compare the behaviour for standard inputs with the behaviour for non-standard inputs it would be possible to detect the drastic increase of emissions beyond input 1, and thus the doping in Figure 3.5.

Instead of reasoning via an abstract concept of “similarity” we use distance functions and distance thresholds to concretise similarity. A cleanness contract for robust cleanness contains a distance function $d_{\text{In}} : \text{In} \times \text{In} \rightarrow \overline{\mathbb{R}}_{\geq 0}$ for inputs and $d_{\text{Out}} : \text{Out} \times \text{Out} \rightarrow \overline{\mathbb{R}}_{\geq 0}$ for outputs. These distance functions must assign zero distance to identical values and must be symmetric. That is, $d : X \times X \rightarrow \overline{\mathbb{R}}_{\geq 0}$ is a distance function if and only if for every $a, b \in X$ it holds that $d(a, a) = 0$ and $d(a, b) = d(b, a)$. The cleanness contract further contains distance thresholds $\kappa_i \in \overline{\mathbb{R}}_{\geq 0}$ and $\kappa_o \in \overline{\mathbb{R}}_{\geq 0}$. Then, two inputs i_1 and i_2 are “similar” if and only if $d_{\text{In}}(i_1, i_2) \leq \kappa_i$, and analogously, two outputs o_1 and o_2 are “similar” if and only if $d_{\text{Out}}(o_1, o_2) \leq \kappa_o$. For nondeterministic programs we measure the distance between sets of outputs using the Hausdorff distance (cf. Section 2.1) based on d_{Out} .

Definition 3.6. A parametrised nondeterministic sequential program P_+ is *robustly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for all pairs of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$, and input $i' \in \text{In}$, if $d_{\text{In}}(i, i') \leq \kappa_i$, then $\mathcal{H}(d_{\text{Out}})(P_+(\mathbf{p})(i), P_+(\mathbf{p}')(i')) \leq \kappa_o$.

In contrast to strict cleanness, robust cleanness does not require i' to be a standard input from `StdIn`; all inputs that are at most κ_i away from a standard input i are considered. In such a case, the outputs produced by i' need to be within a κ_o distance of the outputs produced by i . Nevertheless, robust cleanness is a generalisation of strict cleanness. Strict cleanness w.r.t. $\langle \approx_P, \text{StdIn} \rangle$ is equivalent to robust cleanness w.r.t. $\langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if $d_{\text{In}}(i, i) = 0$

and $d_{\text{In}}(i, i') = \infty$ for $i \neq i'$, $d_{\text{Out}}(o, o) = 0$ and $d_{\text{Out}}(o, o') = \infty$ for $o \neq o'$ and $\kappa_i = \kappa_o = 0$.

Notice that for deterministic programs, the Hausdorff distance in Definition 3.6 can be safely replaced by the distance d_{Out} on single outputs, i.e., by $d_{\text{Out}}(P_+(\mathbf{p})(i), P_+(\mathbf{p}')(i')) \leq \kappa_o$. In the nondeterministic case, if the Hausdorff distance exceeds the κ_o threshold, then this can be fixed by either removing an output from one of the output sets such that this set becomes smaller and all outputs in this set are at most κ_o away from the other set. Or by adding an output to the other set – and thus by making this set larger – such that the distance between the two sets shrinks. This becomes particularly interesting in situations where it can be assumed that the standard behaviour is correct; that is, we consider robust cleanness as a specification for inputs that are not in StdIn and assume that the outputs for inputs in StdIn are correct. This may be the case, for example, because the correctness for inputs in StdIn has been carefully validated or is constrained by external specifications or legal regulations. Under this assumption, and thus with a focus on the non-standard behaviour, a violation of the Hausdorff distance requirement must be fixed by either adding or removing outputs from $P_+(\mathbf{p}')(i')$, because we assume that $P_+(\mathbf{p})(i)$ is correct. We identify two “sub-constraints” that are induced by the Hausdorff distance requirement and that relate to how a cleanness violation must be fixed. One of these sub-constraints defines a minimum set of outputs that must be contained in $P_+(\mathbf{p}')(i')$, while the other sub-constraint defines a maximum set of outputs that may be contained in $P_+(\mathbf{p}')(i')$. We will call the former a *lower-bound* property on $P_+(\mathbf{p}')(i')$ and the other one an *upper-bound* property. They are defined below as l-robust cleanness and u-robust cleanness.

Definition 3.7. A parametrised nondeterministic sequential program P_+ is *l-robustly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for all pairs of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$, and input $i' \in \text{In}$, if $d_{\text{In}}(i, i') \leq \kappa_i$, then for all $o \in P_+(\mathbf{p})(i)$, there exists $o' \in P_+(\mathbf{p}')(i')$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$.

Assuming that $P_+(\mathbf{p})(i)$ is fixed, l-robust cleanness enforces that $P_+(\mathbf{p}')(i')$ contains enough outputs, such that for every “standard output” $o \in P_+(\mathbf{p})(i)$, there is a counter-part $o' \in P_+(\mathbf{p}')(i')$ that is at most κ_o away from o . Effectively, l-robust cleanness enforces that no program behaviour observable for a standard input disappears under execution of a similar non-standard input.

Definition 3.8. A parametrised nondeterministic sequential program P_+ is *u-robustly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for all pairs of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$, and input $i' \in \text{In}$, if $d_{\text{In}}(i, i') \leq \kappa_i$, then for all $o' \in P_+(\mathbf{p}')(i')$, there exists $o \in P_+(\mathbf{p})(i)$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$.

Again assuming that $P_+(\mathbf{p})(i)$ is fixed, u-robust cleanliness enforces that the existence of every non-standard output $\mathfrak{o}' \in P_+(\mathbf{p}')(i')$ can be justified by a standard output $\mathfrak{o} \in P_+(\mathbf{p})(i)$ that is at most $\kappa_{\mathfrak{o}}$ away from \mathfrak{o}' . Thus, a u-robustly clean program cannot introduce a behaviour that is far away from the standard behaviour.

The conjunction of l-robust cleanliness and u-robust cleanliness is in many cases (but not for certain corner cases, as we will see) equivalent to robust cleanliness. In the following, we denote the (Boolean) conjunction of l-robust cleanliness and u-robust cleanliness as *quantifier-based robust cleanliness* and Definition 3.6 as *Hausdorff-based robust cleanliness*. While quantifier-based robust cleanliness always implies Hausdorff-based robust cleanliness, there are corner cases in which the inverse implication does not hold.

Proposition 3.9. Let P_+ be a parametrised nondeterministic sequential program and $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract for robust cleanliness. If P_+ is l-robustly clean w.r.t. \mathcal{C} and P_+ is u-robustly clean w.r.t. \mathcal{C} , then P_+ is robustly clean w.r.t. \mathcal{C} .

The proof of this proposition requires to establish that all values in a set being smaller or equal to κ is equivalent to the supremum of this set being smaller or equal to κ and, similarly, that a set containing a value that is smaller or equal to κ is equivalent to the infimum of that set being smaller or equal to κ . This is what Lemmas 3.10 to 3.12 show.

Lemma 3.10. For every threshold $\kappa \in \overline{\mathbb{R}}$, set A and function $f : A \rightarrow \overline{\mathbb{R}}$, if $f(a) \leq \kappa$ holds for all $a \in A$, then $\sup\{f(a) \mid a \in A\} \leq \kappa$.

Proof. Let S denote the set $\{f(a) \mid a \in A\}$. If A is empty, then S is empty; hence $\sup S = -\infty$, which is less or equal to every $\kappa \in \overline{\mathbb{R}}$. If A is non-empty, S is a non-empty subset of $\overline{\mathbb{R}}$. Then, the supremum of S exists and is the least upper bound of S [102]; let $t \in \overline{\mathbb{R}}$ be the least upper bound of S . t is an upper bound of S if for all $e \in S$, $t \geq e$. For S , this is equivalent to $t \geq f(a)$ for all $a \in A$. To be the least upper bound, all upper bounds of S (i.e., all $z \in \overline{\mathbb{R}}$ with $\forall a' \in A. z \geq f(a')$) must be greater or equal to t . From $\forall a \in A. \kappa \geq f(a)$ we know that κ is an upper bound of S . Because t is the least upper bound, it must be that $t \leq \kappa$. \square

Lemma 3.11. For every threshold $\kappa \in \overline{\mathbb{R}}$, set B and function $f : B \rightarrow \overline{\mathbb{R}}$, if there exists some $b \in B$ with $f(b) \leq \kappa$, then $\inf\{f(b) \mid b \in B\} \leq \kappa$.

Proof. Let S denote the set $\{f(b) \mid b \in B\}$. Since there exists some $b \in B$, B is not empty – and neither is S . S is a non-empty subset of the extended real numbers $\overline{\mathbb{R}}$, hence, by definition, the infimum of S exists and is the greatest lower

bound of S ; let $t \in \overline{\mathbb{R}}$ be the greatest lower bound of S . t is a lower bound of S if for all $e \in S$, $t \leq e$. For S , this is equivalent to $t \leq f(b')$ for all $b' \in B$. Hence, we now that in particular $f(b)$ is greater than or equal to t . With $\kappa \geq f(b)$, we get that $\kappa \geq t$. \square

In the next Lemma, we combine the two Lemmas above to reason about the composition of supremum and infimum.

Lemma 3.12. For every threshold $\kappa \in \overline{\mathbb{R}}$, all sets A and B and every function $f : A \times B \rightarrow \overline{\mathbb{R}}$, if for all $a \in A$, there exists some $b \in B$ such that $f(a, b) \leq \kappa$, then $\sup_{a \in A} \inf_{b \in B} f(a, b) \leq \kappa$.

Proof. Assume $\forall a \in A. \exists b \in B. f(a, b) \leq \kappa$. With Lemma 3.11 we get that $\forall a \in A. \inf_{b \in B} f(a, b) \leq \kappa$. We rewrite this to $\forall a \in A. f_1(a) \leq \kappa$, where f_1 is the function defined as $f_1(a) = \inf_{b \in B} f(a, b)$. Applying Lemma 3.10, we get that $\sup_{a \in A} f_1(a) \leq \kappa$. Finally, by replacing f_1 with its definition, we get that $\sup_{a \in A} \inf_{b \in B} f(a, b) \leq \kappa$. \square

With these Lemmas, we are able to prove Proposition 3.9.

Proof of Proposition 3.9. Let $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, $i \in \text{StdIn}$, $i' \in \text{In}$, and $d_{\text{In}}(i, i') \leq \kappa_i$. It suffices to show that $\forall \mathbf{o} \in P_+(\mathbf{p})(i). \exists \mathbf{o}' \in P_+(\mathbf{p}')(i'). d_{\text{Out}}(\mathbf{o}, \mathbf{o}') \leq \kappa_{\mathbf{o}}$ (l-robust cleanness) and $\forall \mathbf{o}' \in P_+(\mathbf{p}')(i'). \exists \mathbf{o} \in P_+(\mathbf{p})(i). d_{\text{Out}}(\mathbf{o}, \mathbf{o}') \leq \kappa_{\mathbf{o}}$ (u-robust cleanness) implies $\mathcal{H}(d_{\text{Out}})(P_+(\mathbf{p})(i), P_+(\mathbf{p}')(i')) \leq \kappa_{\mathbf{o}}$. For l-robust cleanness we get with Lemma 3.12 that $\sup_{\mathbf{o} \in P_+(\mathbf{p})(i)} \inf_{\mathbf{o}' \in P_+(\mathbf{p}')(i')} d_{\text{Out}}(\mathbf{o}, \mathbf{o}') \leq \kappa_{\mathbf{o}}$, and for u-robust cleanness that $\sup_{\mathbf{o}' \in P_+(\mathbf{p}')(i')} \inf_{\mathbf{o} \in P_+(\mathbf{p})(i)} d_{\text{Out}}(\mathbf{o}, \mathbf{o}') \leq \kappa_{\mathbf{o}}$. As both sup-inf-combinations are less or equal to $\kappa_{\mathbf{o}}$, also the maximum of both is less or equal to $\kappa_{\mathbf{o}}$. Hence, $\mathcal{H}(d_{\text{Out}})(P_+(\mathbf{p})(i), P_+(\mathbf{p}')(i')) \leq \kappa_{\mathbf{o}}$. \square

A proposition complementary to Proposition 3.9 where Hausdorff-based robust cleanness implies quantifier-based robust cleanness does not hold in general.² To illustrate that, assume that the inverse of Lemma 3.12 holds, i.e., that for any two sets $A, B \subseteq \mathbb{R}$ and function f it holds that $\sup_{a \in A} \inf_{b \in B} f(a, b) \leq \kappa$ implies that for every $a \in A$, there exists some $b \in B$, such that $f(a, b) \leq \kappa$. A counter-example to this claim is the following. Let $A = (-1, 0] \subseteq \mathbb{R}$ be the interval between -1 and 0 , where 0 is in A , but -1 is not in A . Further, let $B = (-2, -1) \subseteq \mathbb{R}$ be the open interval between -1 and -2 , let $f(a, b) = |a - b|$ be a distance function between values from A and B , and let $\kappa = 1$. Obviously, the value in A that has the largest distance to any value in B is 0 . It is easy to see that $\inf_{b \in B} f(0, b)$ is 1 . Thus, $\sup_{a \in A} \inf_{b \in B} f(a, b) \leq 1$ holds. Now, assume $a = 0 \in A$. According to the claim above, there must exist some $b \in B$, such

²Notably, the opposite has been claimed in previous work [17, 18], hence I elaborate on this in more detail.

that $f(0, b) \leq 1$. However, whatever value in B we pick, $f(0, b)$ is always greater than 1.

Still, an implication such as the one in the counter-example does hold for stronger versions of robust cleanness where the “smaller-or-equal” comparison is replaced by a strict “smaller” comparison. Hence, we propose the following proposition as a complement to Proposition 3.9, where output distances must be strictly smaller than κ_o .

Proposition 3.13. Let P_+ be a parametrised nondeterministic sequential program and $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract for robust cleanness. For all pairs of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$, and input $i' \in \text{In}$ with $d_{\text{In}}(i, i') \leq \kappa_i$, it holds that if $\mathcal{H}(d_{\text{Out}})(P_+(\mathbf{p})(i), P_+(\mathbf{p}')(i')) < \kappa_o$, then the following two propositions hold:

1. for all $o \in P_+(\mathbf{p})(i)$, there exists $o' \in P_+(\mathbf{p}')(i')$ such that $d_{\text{Out}}(o, o') < \kappa_o$
2. for all $o' \in P_+(\mathbf{p}')(i')$, there exists $o \in P_+(\mathbf{p})(i)$ such that $d_{\text{Out}}(o, o') < \kappa_o$.

Before we prove Proposition 3.13, we first give Lemmas similar to Lemmas 3.10 to 3.12 that establish a relationship between universal quantification and supremum and between existential quantification and infimum.

Lemma 3.14. For every threshold $\kappa \in \overline{\mathbb{R}}$, set A and function $f : A \rightarrow \overline{\mathbb{R}}$, if $\sup\{f(a) \mid a \in A\} < \kappa$, then it holds for all $a \in A$ that $f(a) < \kappa$.

Proof. If A is empty, then the Lemma trivially holds. Otherwise, let S denote the set $\{f(a) \mid a \in A\}$. S is a non-empty subset of the extend real numbers $\overline{\mathbb{R}}$. By definition, the supremum of S exists and is the least upper bound of S ; let $t \in \overline{\mathbb{R}}$ be the least upper bound of S . t is an upper bound of S if for all $e \in S$, $e \leq t$. For S , this is equivalent to $f(a) \leq t$ for all $a \in A$. Since $t < \kappa$, it also holds that $f(a) < \kappa$ for all $a \in A$. \square

Lemma 3.15. For every threshold $\kappa \in \overline{\mathbb{R}}$, set B and function $f : B \rightarrow \overline{\mathbb{R}}$, if $\inf\{f(b) \mid b \in B\} < \kappa$, then there exists some $b \in B$ with $f(b) < \kappa$.

Proof. Let S denote the set $\{f(b) \mid b \in B\}$. First, assume that B is the empty set. Then S is empty and $\inf S = \infty$. However, this violates the assumption that $\inf S < \kappa$, because $\infty \not< \kappa$ for all $\kappa \in \overline{\mathbb{R}}$. Hence, B must be non-empty, and S is non-empty, too. Thus, the infimum of S exists and is the greatest lower bound of S ; let $t \in \overline{\mathbb{R}}$ be this greatest lower bound. t is a lower bound of S if for all $e \in S$, $t \leq e$. For S , this is equivalent to $t \leq f(b')$ for all $b' \in B$. To be the greatest lower bound, all lower bounds of S (i.e., all $z \in \overline{\mathbb{R}}$ with $\forall b' \in B. z \leq f(b')$) must be smaller or equal to t . We continue by proving the contraposition of the proposition, i.e., we show that $t \geq \kappa$ under the assumption that $f(b) \geq \kappa$ for all

$b \in B$. From this assumption, we get that κ is a lower bound of S . Since t is the greatest lower bound of S , it must be that $t \geq \kappa$. \square

Lemma 3.16. For every $\kappa \in \overline{\mathbb{R}}$, all sets A and B and function $f : A \times B \rightarrow \overline{\mathbb{R}}$, if $\sup_{a \in A} \inf_{b \in B} f(a, b) < \kappa$, then for all $a \in A$, there exists some $b \in B$ such that $f(a, b) < \kappa$.

Proof. Let $f'(a) := \inf_{b \in B} f(a, b)$ be the function mapping elements in A to the infimum of $\{f(a, b) \mid b \in B\}$. Hence, $\sup_{a \in A} f'(a) < \kappa$. With Lemma 3.14 we get that $f'(a) < \kappa$ for all $a \in A$. Next, let $f_a(b) := f(a, b)$ be the family of functions, where a concrete $a \in A$ is fixed and passed to f with a value $b \in B$ provided as an argument to f_a . By using the definitions of f' and f_a , we get that for all $a \in A$, $\inf_{b \in B} f_a(b) < \kappa$. With Lemma 3.15 we get that for all $a \in A$, there exists some $b \in B$ such that $f_a(b) < \kappa$. Replacing $f_a(b)$ with $f(a, b)$ concludes the proof. \square

Proof of Proposition 3.13. It suffices to show that $\mathcal{H}(d_{\text{Out}})(\mathbb{P}_+(\mathfrak{p})(i), \mathbb{P}_+(\mathfrak{p}')(i')) < \kappa_{\mathfrak{o}}$ implies that $\forall \mathfrak{o} \in \mathbb{P}_+(\mathfrak{p})(i). \exists \mathfrak{o}' \in \mathbb{P}_+(\mathfrak{p}')(i'). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_{\mathfrak{o}}$ (l-robust cleanness) and $\forall \mathfrak{o}' \in \mathbb{P}_+(\mathfrak{p}')(i'). \exists \mathfrak{o} \in \mathbb{P}_+(\mathfrak{p})(i). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_{\mathfrak{o}}$ (u-robust cleanness). By unrolling the definition of the Hausdorff distance and maximum function, we get that $\sup_{\mathfrak{o} \in \mathbb{P}_+(\mathfrak{p})(i)} \inf_{\mathfrak{o}' \in \mathbb{P}_+(\mathfrak{p}')(i')} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_{\mathfrak{o}}$ holds and that $\sup_{\mathfrak{o}' \in \mathbb{P}_+(\mathfrak{p}')(i')} \inf_{\mathfrak{o} \in \mathbb{P}_+(\mathfrak{p})(i)} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_{\mathfrak{o}}$ holds. From the former, we get l-robust cleanness with Lemma 3.16, and from the latter and Lemma 3.16 we obtain u-robust cleanness. \square

As Propositions 3.9 and 3.13 demonstrate, Hausdorff-based robust cleanness and quantifier-based robust cleanness are conceptually equivalent, but when infinite sets are involved, it matters whether distances are compared using the \leq -operator or the $<$ -operator. In the remainder of this thesis, we will call such mutual implications as obtained from Propositions 3.9 and 3.13 *almost equivalence*; i.e., we say that Hausdorff-based robust cleanness and quantifier-based robust cleanness are *almost equivalent*. The peculiarity of the two notions of robust cleanness being only almost, but not exactly equivalent is a theoretical artefact; if the set of outputs of \mathbb{P}_+ is finite, then also the set of distances between all combinations of outputs is finite and the supremum and infimum are identical to the maximum and, respectively, minimum element of the set of output distances.

We will now turn to robust cleanness of non-parametrised programs and first provide a non-parametrised variant of Hausdorff-based robust cleanness.

Definition 3.17. A non-parametrised nondeterministic sequential program P is *robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every standard input $i \in \text{StdIn}$ and input $i' \in \text{In}$, if $d_{\text{In}}(i, i') \leq \kappa_i$, then $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) \leq \kappa_o$.

Robust cleanness contracts for non-parametrised programs do not contain the parameter comparability relation \approx_ρ . While for strict cleanness we compensated for that by adding an input comparability relation \approx_I , for robust cleanness it is already possible to define input comparability by means of d_{In} and κ_i . As we will see next, this suffices to show that robust cleanness for parametrised and robust cleanness for non-parametrised programs are equally expressive.

The encoding of robust cleanness of non-parametrised programs with its parametrised counterpart is straightforward:

Proposition 3.18. Let $P : \text{In} \rightarrow 2^{\text{Out}}$ be a non-parametrised nondeterministic sequential program and let $P_+ : \{\text{def}\} \rightarrow \text{In} \rightarrow 2^{\text{Out}}$ be the parametrised nondeterministic sequential program that encodes P , i.e., $P \hookrightarrow P_+$. Further, let $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ and $\mathcal{C}_+ = \langle \approx_\rho, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be contracts, where $\text{StdIn} \subseteq \text{In}$, $\text{def} \approx_\rho \text{def}$ and $\text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o$ are identical in both contracts. Then, P is robustly clean w.r.t. \mathcal{C} if and only if P_+ is robustly clean w.r.t. \mathcal{C}_+ .

Proof. We first assume that P is robustly clean w.r.t. \mathcal{C} to show that P_+ is robustly clean w.r.t. \mathcal{C}_+ . For this, let $\mathfrak{p} = \mathfrak{p}' = \text{def}$, $i \in \text{StdIn}$ and $i' \in \text{In}$ with $d_{\text{In}}(i, i') \leq \kappa_i$. With these assumptions, we get from robust cleanness of P that $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) \leq \kappa_o$. By definition of P_+ , this is equivalent to $\mathcal{H}(d_{\text{Out}})(P_+(\mathfrak{p})(i), P_+(\mathfrak{p})(i')) \leq \kappa_o$, so P_+ is robustly clean w.r.t. \mathcal{C}_+ .

Conversely, we assume that P_+ is robustly clean w.r.t. \mathcal{C}_+ to show that P is robustly clean w.r.t. \mathcal{C} . For this, let $i \in \text{StdIn}$ and $i' \in \text{In}$ with $d_{\text{In}}(i, i') \leq \kappa_i$. With these assumptions, we get from $\text{def} \approx_\rho \text{def}$ and robust cleanness of P_+ that $\mathcal{H}(d_{\text{Out}})(P_+(\text{def})(i), P_+(\text{def})(i')) \leq \kappa_o$. By definition of P_+ , this is equivalent to $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) \leq \kappa_o$, which concludes the proof. \square

Encoding robust cleanness for a parametrised program $P_+ : \text{Param} \rightarrow \text{In}_+ \rightarrow 2^{\text{Out}}$ with robust cleanness for the non-parametrised program $P : \text{In}_\times \rightarrow 2^{\text{Out}}$ that encodes P_+ requires a non-trivial construction of the contract for cleanness of P . As for strict cleanness, we encode parameters Param and inputs In_+ to P_+ as inputs $\text{In}_\times = \text{Param} \times \text{In}_+$ to P that are pairs containing a parameter and an input. The parameter comparability relation \approx_ρ is encoded using the input distance function and threshold. Equation (3.1) shows the input distance function $\sharp d_{\text{In}}^{(\kappa_i, \approx_\rho)}$ that is parametrised by κ_i and \approx_ρ , and the corresponding distance threshold $\flat \kappa_i$ that is parametrised by κ_i . If the parameters encoded by the P -inputs passed to $\sharp d_{\text{In}}^{(\kappa_i, \approx_\rho)}$ are not in the \approx_ρ relation, then the distance

between the P-inputs is infinite. Notice that $b\kappa_i$ is never infinite; hence, in this case, the input distance is always greater than the distance threshold, so for this combination of parameters, robust cleanness is trivially satisfied. If the encoded parameters are in the \approx_P relation, then it is necessary to distinguish between the case whether κ_i is infinity or not. If it is infinity, then the original robust cleanness of P_+ considered every combination of inputs as relevant for the cleanness evaluation, as the distance between every combination of inputs is less or equal to infinity. Hence, we reflect this case by assigning distance zero to all combinations of inputs in $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$ and by defining zero as the distance threshold $b\kappa_i$. Finally, in the case that κ_i is not infinity, $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$ is identical to the original d_{In} for the P_+ -inputs and $b\kappa_i$ is identical to κ_i . We remark that $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$ is a valid distance function; it is easy to see that for all $i, i' \in \text{In}_\neq$, $\#d_{\text{In}}^{(\kappa_i, \approx_P)}(i, i) = 0$ and $\#d_{\text{In}}^{(\kappa_i, \approx_P)}(i, i') = \#d_{\text{In}}^{(\kappa_i, \approx_P)}(i', i)$.

$$\begin{aligned} \#d_{\text{In}}^{(\kappa_i, \approx_P)}((p, i), (p', i')) &:= \begin{cases} d_{\text{In}}(i, i') & \text{if } \kappa_i \neq \infty \text{ and } p \approx_P p' \\ 0 & \text{if } \kappa_i = \infty \text{ and } p \approx_P p' \\ \infty & \text{otherwise,} \end{cases} \quad \text{and} \\ b\kappa_i &:= \begin{cases} \kappa_i & \text{if } \kappa_i \neq \infty \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \quad (3.1)$$

Proposition 3.19. Let $P_+ : \text{Param} \rightarrow \text{In}_+ \rightarrow 2^{\text{Out}}$ be a parametrised non-deterministic sequential program and let $P : \text{In}_\neq \rightarrow 2^{\text{Out}}$ be the non-parametrised program that encodes P_+ , i.e., $P_+ \hookrightarrow P$ and $\text{In}_\neq = \text{Param} \times \text{In}_+$. Further, let $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}_+, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ and $\mathcal{C} = \langle \text{StdIn}_\neq, d'_{\text{In}}, d_{\text{Out}}, b\kappa_i, \kappa_o \rangle$ be contracts with $\text{StdIn}_\neq = \text{Param} \times \text{StdIn}_+$ and $d'_{\text{In}} = \#d_{\text{In}}^{(\kappa_i, \approx_P)}$. Then, P_+ is robustly clean w.r.t. \mathcal{C}_+ if and only if P is robustly clean w.r.t. \mathcal{C} .

Proof. We prove both directions of the if-and-only-if separately. First, we assume that P is robustly clean w.r.t. \mathcal{C} to show that P_+ is robustly clean w.r.t. \mathcal{C}_+ . Hence, let p and p' be parameters with $p \approx_P p'$, $i \in \text{StdIn}_+$ and $i' \in \text{In}_+$ with $d_{\text{In}}(i, i') \leq \kappa_i$. Now, let $\hat{i} = (p, i)$ and $\hat{i}' = (p', i')$ be pairs of the given parameters and inputs. \hat{i} is a standard input in StdIn_\neq , because i is a standard input in StdIn_+ . We first consider the case that $\kappa_i \neq \infty$. Then, $b\kappa_i = \kappa_i$ and we get from the definition of d'_{In} , $\kappa_i \neq \infty$ and $p \approx_P p'$ that $d'_{\text{In}}(\hat{i}, \hat{i}') = d_{\text{In}}(i, i')$. Since $d_{\text{In}}(i, i') \leq \kappa_i$, it is also that $d'_{\text{In}}(\hat{i}, \hat{i}') \leq b\kappa_i$. In the case that $\kappa_i = \infty$, we know that $b\kappa_i = 0$ and, from the definition of d'_{In} , $\kappa_i = \infty$ and $p \approx_P p'$, that $d'_{\text{In}}(\hat{i}, \hat{i}') = 0$ and hence, $d'_{\text{In}}(\hat{i}, \hat{i}') \leq b\kappa_i$. Since in both cases $d'_{\text{In}}(\hat{i}, \hat{i}') \leq b\kappa_i$, we get from robust cleanness

of P that $\mathcal{H}(d_{\text{Out}})\left(P((p, i)), P((p', i'))\right) \leq \kappa_o$. From the definition of P_+ , we get that $\mathcal{H}(d_{\text{Out}})(P_+(p)(i), P(p')(i')) \leq \kappa_o$, which proves that P_+ is robustly clean.

Next, we assume that P_+ is robustly clean w.r.t. \mathcal{C}_+ to show that P is robustly clean w.r.t. \mathcal{C} . For this, let $\hat{i} = (p, i) \in \text{StdIn}_\times$ be a standard input of P , i.e., $p \in \text{Param}$ and $i \in \text{StdIn}_+$, and let $\hat{i}' = (p', i') \in \text{In}_\times$ be an input of P . We may assume that $d'_{\text{In}}(\hat{i}, \hat{i}') \leq b\kappa_i$. In the case that $\kappa_i \neq \infty$, $b\kappa_i = \kappa_i$ and we can infer that $d'_{\text{In}}(\hat{i}, \hat{i}') < \infty$. From this and the definition of d'_{In} , we conclude that $p \approx_P p'$ and $d'_{\text{In}}(\hat{i}, \hat{i}') = d_{\text{In}}(i, i')$. Since $d'_{\text{In}}(\hat{i}, \hat{i}') \leq b\kappa_i$, it is that $d_{\text{In}}(i, i') \leq \kappa_i$. If it is the case that $\kappa_i = \infty$, then $b\kappa_i = 0$; from $d'_{\text{In}}(\hat{i}, \hat{i}') \leq b\kappa_i$ we can infer that $d'_{\text{In}}(\hat{i}, \hat{i}') = 0$, which, for $\kappa_i = \infty$, is the case only if $p \approx_P p'$. As $\kappa_i = \infty$, $d_{\text{In}}(i, i') \leq \kappa_i$ is trivially satisfied. Since in both cases, we have that $p \approx_P p'$ and $d_{\text{In}}(i, i') \leq \kappa_i$, we get from robust cleanliness of P_+ that $\mathcal{H}(d_{\text{Out}})(P_+(p)(i), P(p')(i')) \leq \kappa_o$. Rewriting this with the definition of P , we get $\mathcal{H}(d_{\text{Out}})\left(P((p, i)), P((p', i'))\right) \leq \kappa_o$, proving that P is robustly clean. \square

Definitions 3.20 and 3.21 define l-robust cleanliness and u-robust cleanliness for non-parametrised programs.

Definition 3.20. A non-parametrised nondeterministic sequential program P is *l-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every standard input $i \in \text{StdIn}$ and input $i' \in \text{In}$, if $d_{\text{In}}(i, i') \leq \kappa_i$, then for all $o \in P(i)$, there exists $o' \in P(i')$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$.

Definition 3.21. A non-parametrised nondeterministic sequential program P is *u-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every standard input $i \in \text{StdIn}$ and input $i' \in \text{In}$, if $d_{\text{In}}(i, i') \leq \kappa_i$, then for all $o' \in P(i')$, there exists $o \in P(i)$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$.

The definitions provide the same expressiveness as their corresponding counterpart from Definitions 3.7 and 3.8 for parametrised programs. For l-robust cleanliness, this is shown in Propositions 3.22 and 3.23, in a way analogue to Propositions 3.18 and 3.19.

Proposition 3.22. Let $P : \text{In} \rightarrow 2^{\text{Out}}$ be a non-parametrised nondeterministic sequential program and let $P_+ : \{\text{def}\} \rightarrow \text{In} \rightarrow 2^{\text{Out}}$ be the parametrised nondeterministic sequential program that encodes P , i.e., $P \hookrightarrow P_+$. Further, let $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ and $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be contracts, where $\text{def} \approx_P \text{def}$ and $\text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o$ are identical in both contracts. Then, P is l-robustly clean w.r.t. \mathcal{C} if and only if P_+ is l-robustly clean w.r.t. \mathcal{C}_+ .

Proof. We first assume that P is l-robustly clean w.r.t. \mathcal{C} to show that P_+ is l-robustly clean w.r.t. \mathcal{C}_+ . For this, let $p = p' = \text{def}$, $i \in \text{StdIn}$ and $i' \in \text{In}$

with $d_{\text{In}}(i, i') \leq \kappa_i$. With these assumptions, we get from l-robust cleanness of P that for all $o \in P(i)$ there exists some $o' \in P(i')$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$. By definition of P_+ , it is equivalent to say that for all $o \in P_+(\mathbf{p})(i)$ there exists some $o' \in P_+(\mathbf{p}')(i')$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$, so P_+ is l-robustly clean w.r.t. \mathcal{C}_+ .

Conversely, we assume that P_+ is l-robustly clean w.r.t. \mathcal{C}_+ to show that P is l-robustly clean w.r.t. \mathcal{C} . For this, let $i \in \text{StdIn}$ and $i' \in \text{In}$ with $d_{\text{In}}(i, i') \leq \kappa_i$. With these assumptions and with $\text{def} \approx_P \text{def}$, we get from l-robust cleanness of P_+ that for all $o \in P_+(\mathbf{p})(i)$ there exists some $o' \in P_+(\mathbf{p}')(i')$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$. By definition of P_+ , this is equivalent to saying that for all $o \in P(i)$ there exists some $o' \in P(i')$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$, which concludes the proof. \square

Proposition 3.23. Let $P_+ : \text{Param} \rightarrow \text{In} \rightarrow 2^{\text{Out}}$ be a parametrised nondeterministic sequential program and let $P : \text{Param} \times \text{In} \rightarrow 2^{\text{Out}}$ be the non-parametrised program that encodes P_+ , i.e., $P_+ \hookrightarrow P$. Further, let the tuples $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ and $\mathcal{C} = \langle \text{Param} \times \text{StdIn}, d'_{\text{In}}, d_{\text{Out}}, \flat\kappa_i, \kappa_o \rangle$ be contracts with $d'_{\text{In}} = \sharp d_{\text{In}}^{(\kappa_i, \approx_P)}$. Then, P_+ is l-robustly clean w.r.t. \mathcal{C}_+ if and only if P is l-robustly clean w.r.t. \mathcal{C} .

Proof. The proof is conceptually the same as for Proposition 3.19. The difference is only at the end of each of the subproofs for the implications making up the if-and-only-if. For the implication from l-robust cleanness of P to l-robust cleanness of P_+ , we get from the l-robust cleanness of P that for all $o \in P((\mathbf{p}, i))$, there exists some $o' \in P((\mathbf{p}', i'))$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$. From the definition of P , we get that for all $o \in P_+(\mathbf{p})(i)$, there exists some $o' \in P_+(\mathbf{p}')(i')$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$, which proves that P_+ is l-robustly clean w.r.t. \mathcal{C}_+ . For the inverse implication, in the last proof step we can infer from l-robust cleanness of P_+ that for all $o \in P_+(\mathbf{p})(i)$, there exists some $o' \in P_+(\mathbf{p}')(i')$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$. Rewriting this with the definition of P , we get that for all $o \in P((\mathbf{p}, i))$, there exists some $o' \in P((\mathbf{p}', i'))$, such that $d_{\text{Out}}(o, o') \leq \kappa_o$, proving that P is l-robustly clean w.r.t. \mathcal{C} . \square

Likewise, there are propositions for u-robust cleanness, which we omit here; the changes are mostly of syntactic nature.

The relationship between Hausdorff-based robust cleanness and quantifier-based cleanness that Propositions 3.9 and 3.13 establish for parametrised programs also holds for non-parametrised programs, as the following propositions show.

Proposition 3.24. Let P be a non-parametrised nondeterministic sequential program and $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract for robust cleanness. If P is l-robustly clean w.r.t. \mathcal{C} and P is u-robustly clean w.r.t. \mathcal{C} , then P is robustly clean w.r.t. \mathcal{C} .

Proof. Let $i \in \text{StdIn}$, $i' \in \text{In}$, and $d_{\text{In}}(i, i') \leq \kappa_i$. Then, it suffices to show that $\forall \mathfrak{o} \in \mathcal{P}(i). \exists \mathfrak{o}' \in \mathcal{P}(i'). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa_o$ (l-robust cleanness) and $\forall \mathfrak{o}' \in \mathcal{P}(i'). \exists \mathfrak{o} \in \mathcal{P}(i). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa_o$ (u-robust cleanness) implies $\mathcal{H}(d_{\text{Out}})(\mathcal{P}(i), \mathcal{P}(i')) \leq \kappa_o$. For l-robust cleanness we get with Lemma 3.12 that $\sup_{\mathfrak{o} \in \mathcal{P}(i)} \inf_{\mathfrak{o}' \in \mathcal{P}(i')} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa_o$, and for u-robust cleanness that $\sup_{\mathfrak{o}' \in \mathcal{P}(i')} \inf_{\mathfrak{o} \in \mathcal{P}(i)} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa_o$. As both sup-inf-combinations are less or equal to κ_o , also the maximum of both is less or equal to κ_o . Hence, $\mathcal{H}(d_{\text{Out}})(\mathcal{P}(i), \mathcal{P}(i')) \leq \kappa_o$. \square

Proposition 3.25. Let P be a non-parametrised nondeterministic sequential program and $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract for robust cleanness. Then, for every standard input $i \in \text{StdIn}$, and input $i' \in \text{In}$ with $d_{\text{In}}(i, i') \leq \kappa_i$, it holds that if $\mathcal{H}(d_{\text{Out}})(\mathcal{P}(i), \mathcal{P}(i')) < \kappa_o$, then

1. for all $\mathfrak{o} \in \mathcal{P}(i)$, there exists $\mathfrak{o}' \in \mathcal{P}(i')$, such that $d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_o$ and
2. for all $\mathfrak{o}' \in \mathcal{P}(i')$, there exists $\mathfrak{o} \in \mathcal{P}(i)$, such that $d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_o$.

Proof. To prove the proposition, it suffices to show that $\mathcal{H}(d_{\text{Out}})(\mathcal{P}(i), \mathcal{P}(i')) < \kappa_o$ implies that $\forall \mathfrak{o} \in \mathcal{P}(i). \exists \mathfrak{o}' \in \mathcal{P}(i'). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_o$ (l-robust cleanness) and $\forall \mathfrak{o}' \in \mathcal{P}(i'). \exists \mathfrak{o} \in \mathcal{P}(i). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_o$ (u-robust cleanness). By unrolling the definition of the Hausdorff distance and maximum function, we get that $\sup_{\mathfrak{o} \in \mathcal{P}(i)} \inf_{\mathfrak{o}' \in \mathcal{P}(i')} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_o$ holds and that $\sup_{\mathfrak{o}' \in \mathcal{P}(i')} \inf_{\mathfrak{o} \in \mathcal{P}(i)} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa_o$ holds. From the former, we get l-robust cleanness with Lemma 3.16, and from the latter and Lemma 3.16 we obtain u-robust cleanness. \square

Example 3.26. The doped cleaning emission in Figure 3.5 can be shown to be doped using robust cleanness. Since this program is not parametrised, we use Definition 3.17 for non-parametrised programs. Assume that the input set, i.e., the values that can be assigned to variable *throttle*, is $\text{In} = (0, 2]$. For the contract w.r.t. which we check robust cleanness, we take $\text{StdIn} = (0, 1]$ (equal to $\text{ThrottleTestValues}$), $\kappa_i = 2$ and $\kappa_o = 1$. As distance functions d_{In} and d_{Out} we take the absolute values of the differences of the values of *throttle* and *NOx*, respectively. The program in Figure 3.4 is robustly clean w.r.t. this contract, while the program in Figure 3.5 is not.

Robust cleanness requires programs to behave in a reasonable way for all inputs that are in a κ_i distance from any of the standard inputs in StdIn . For all other inputs, there are no requirements. This is useful for programs where inputs beyond the κ_i threshold describe situations where cleanness of the program is sacrificed in favour of other, for example safety, properties, when there is a consensus that such sacrifices are justifiable. For instance, a smart battery may stop accepting charge if the current emitted by a standardised but foreign charger is higher than “reasonable” (i.e. than the tolerance values); however, it may still

proceed in case it is dealing with a charger of the same brand for which it may know that it can resort to a customised protocol allowing ultra-fast charging in a safe manner.

To capture cleanness over the whole input domain, we introduce a third notion of cleanness called *func-cleanness*.

3.1.3 Func-cleanness

Similar to robust cleanness, contracts for func-cleanness specify comparability of parameters by a binary relation \approx_P , they specify a set of standard inputs StdIn , and distance functions for inputs d_{In} and outputs d_{Out} . Instead of using constant thresholds κ_i to restrict the input space to be considered for output analysis and κ_o to enforce reasonable amount of variation between outputs, contracts for func-cleanness contain a function $f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ that constrains the amount of variation between outputs dynamically under consideration of the distance between the inputs producing the outputs. For two inputs i and i' , function f defines an output threshold given the distance between i and i' :

Definition 3.27. A parametrised nondeterministic sequential program P_+ is *func-clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for all pairs of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$ and input $i' \in \text{In}$, $\mathcal{H}(d_{\text{Out}})(P_+(\mathbf{p})(i), P_+(\mathbf{p}')(i')) \leq f(d_{\text{In}}(i, i'))$.

Like for Definition 3.6 of robust cleanness, the definition of func-cleanness does not require the second i' to be in StdIn . Since f can map into ∞ , it is possible to allow outputs to be arbitrarily far away from each other if the corresponding inputs are a certain distance away from each other. In particular, this makes func-cleanness strictly more general than robust cleanness: a robust cleanness contract $\langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ is equivalent to a the func-cleanness contract $\langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ with $f(x) = \kappa_o$ whenever $x \leq \kappa_i$ and $f(x) = \infty$ otherwise. Notably, if P_+ is deterministic, we may replace $\mathcal{H}(d_{\text{Out}})(P_+(\mathbf{p})(i), P_+(\mathbf{p}')(i')) \leq f(d_{\text{In}}(i, i'))$ with $d_{\text{Out}}(P_+(\mathbf{p})(i), P_+(\mathbf{p}')(i')) \leq f(d_{\text{In}}(i, i'))$ in Definition 3.27.

As discussed on page 44, the Hausdorff distance entails two sub-constraints that can be related to a lower-bound and upper-bound requirement for non-standard behaviour. Accordingly, Definitions 3.28 and 3.29 explicitly define these two constraints as l-func-cleanness and u-func-cleanness.

Definition 3.28. A parametrised nondeterministic sequential program P_+ is *l-func-clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for all pairs of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$, and input $i' \in \text{In}$, for all $o \in P_+(\mathbf{p})(i)$, there exists $o' \in P_+(\mathbf{p}')(i')$, such that $d_{\text{Out}}(o, o') \leq f(d_{\text{In}}(i, i'))$.

Definition 3.29. A parametrised nondeterministic sequential program P_+ is *u-func-clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for all pairs of parameters $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, standard input $i \in \text{StdIn}$, and input $i' \in \text{In}$, for all $\mathfrak{o}' \in P_+(\mathfrak{p}')(i')$, there exists $\mathfrak{o} \in P_+(\mathfrak{p})(i)$, such that $d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq f(d_{\text{In}}(i, i'))$.

As we did for robust cleanness, we denote the conjunction of Definitions 3.28 and 3.29 as *quantifier-based func-cleanness* and Definition 3.27 as *Hausdorff-based func-cleanness*. Also in the case of func-cleanness, the quantifier-based and the Hausdorff-based definitions are almost equivalent, i.e., they coincide except for some corner cases. This is established formally in Propositions 3.30 and 3.31.

Proposition 3.30. Let P_+ be a parametrised nondeterministic sequential program and $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract for func-cleanness. If P_+ is l-func-clean w.r.t. \mathcal{C} and P_+ is u-func-clean w.r.t. \mathcal{C} , then P_+ is func-clean w.r.t. \mathcal{C} .

Proof. Let $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, $i \in \text{StdIn}$ and $i' \in \text{In}$. In the following, let $\kappa = f(d_{\text{In}}(i, i'))$. Then, it suffices to show that the conjunction of $\forall \mathfrak{o} \in P_+(\mathfrak{p})(i). \exists \mathfrak{o}' \in P_+(\mathfrak{p}')(i'). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa$ (l-func-cleanness) and $\forall \mathfrak{o}' \in P_+(\mathfrak{p}')(i'). \exists \mathfrak{o} \in P_+(\mathfrak{p})(i). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa$ (u-func-cleanness) implies the inequality $\mathcal{H}(d_{\text{Out}})(P_+(\mathfrak{p})(i), P_+(\mathfrak{p}')(i')) \leq \kappa$. For l-func-cleanness we get with Lemma 3.12 that $\sup_{\mathfrak{o} \in P_+(\mathfrak{p})(i)} \inf_{\mathfrak{o}' \in P_+(\mathfrak{p}')(i')} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa$. Similarly, we use Lemma 3.12 to get for u-func-cleanness that $\sup_{\mathfrak{o}' \in P_+(\mathfrak{p}')(i')} \inf_{\mathfrak{o} \in P_+(\mathfrak{p})(i)} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa$. As both sup-inf-combinations are less or equal to κ , also the maximum of both is less or equal to κ . Hence, $\mathcal{H}(d_{\text{Out}})(P_+(\mathfrak{p})(i), P_+(\mathfrak{p}')(i')) \leq \kappa = f(d_{\text{In}}(i, i'))$. \square

Proposition 3.31 shows the inverse direction of the implication in Proposition 3.30, expect that all smaller-equal comparisons are replaced by strict smaller comparisons.

Proposition 3.31. Let P_+ be a parametrised nondeterministic sequential program and $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract for func-cleanness. Then, for all pairs of parameters $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, standard input $i \in \text{StdIn}$ and input $i' \in \text{In}$, $\mathcal{H}(d_{\text{Out}})(P_+(\mathfrak{p})(i), P_+(\mathfrak{p}')(i')) < f(d_{\text{In}}(i, i'))$ implies that

1. for all $\mathfrak{o} \in P_+(\mathfrak{p})(i)$, there exists $\mathfrak{o}' \in P_+(\mathfrak{p}')(i')$, such that $d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < f(d_{\text{In}}(i, i'))$ and
2. for all $\mathfrak{o}' \in P_+(\mathfrak{p}')(i')$, there exists $\mathfrak{o} \in P_+(\mathfrak{p})(i)$, such that $d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < f(d_{\text{In}}(i, i'))$.

Proof. Let $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, $i \in \text{StdIn}$ and $i' \in \text{In}$. In the following, let $\kappa = f(d_{\text{In}}(i, i'))$. Then, it suffices to show that $\mathcal{H}(d_{\text{Out}})(P_+(\mathfrak{p})(i), P_+(\mathfrak{p}')(i')) < \kappa$

implies that $\forall \mathfrak{o} \in P_+(\mathfrak{p})(i). \exists \mathfrak{o}' \in P_+(\mathfrak{p}')(i'). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa$ (l-func-cleanness) and $\forall \mathfrak{o}' \in P_+(\mathfrak{p}')(i'). \exists \mathfrak{o} \in P_+(\mathfrak{p})(i). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa$ (u-func-cleanness). By unrolling the definition of the Hausdorff distance and the definition of the maximum function, we get that $\sup_{\mathfrak{o} \in P_+(\mathfrak{p})(i)} \inf_{\mathfrak{o}' \in P_+(\mathfrak{p}')(i')} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa$ holds and that $\sup_{\mathfrak{o}' \in P_+(\mathfrak{p}')(i')} \inf_{\mathfrak{o} \in P_+(\mathfrak{p})(i)} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa$ holds. From the former, we get l-func-cleanness with Lemma 3.16, and from the latter and Lemma 3.16 we obtain u-func-cleanness. \square

Analogue to strict cleanness and robust cleanness, we provide func-cleanness definitions for non-parametrised programs.

Definition 3.32. A non-parametrised nondeterministic sequential program P is *func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for every standard input $i \in \text{StdIn}$ and input $i' \in \text{In}$, $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) \leq f(d_{\text{In}}(i, i'))$.

As explained earlier, non-parametrised programs can be encoded by parametrised ones. The following proposition shows that func-cleanness for parametrised programs is able to encode func-cleanness for non-parametrised programs.

Proposition 3.33. Let $P : \text{In} \rightarrow 2^{\text{Out}}$ be a non-parametrised nondeterministic sequential program and let $P_+ : \{def\} \rightarrow \text{In} \rightarrow 2^{\text{Out}}$ be the parametrised nondeterministic sequential program that encodes P , i.e., $P \hookrightarrow P_+$. Further, let $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ and $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ be contracts, where $def \approx_P def$ and $\text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f$ are identical in both contracts. Then, P is func-clean w.r.t. \mathcal{C} if and only if P_+ is func-clean w.r.t. \mathcal{C}_+ .

Proof. We first assume that P is func-clean w.r.t. \mathcal{C} to show that P_+ is func-clean w.r.t. \mathcal{C}_+ . For this, let $\mathfrak{p} = \mathfrak{p}' = def$, $i \in \text{StdIn}$ and $i' \in \text{In}$. With these assumptions, we get from func-cleanness of P that $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) \leq f(d_{\text{In}}(i, i'))$. By definition of P_+ , this is equivalent to $\mathcal{H}(d_{\text{Out}})(P_+(\mathfrak{p})(i), P_+(\mathfrak{p})(i')) \leq f(d_{\text{In}}(i, i'))$, so P_+ is robustly clean w.r.t. \mathcal{C}_+ .

Conversely, we assume that P_+ is func-clean w.r.t. \mathcal{C}_+ to show that P is func-clean w.r.t. \mathcal{C} . For this, let $i \in \text{StdIn}$ and $i' \in \text{In}$. With these assumptions, we get from $def \approx_P def$ and func-cleanness of P_+ that $\mathcal{H}(d_{\text{Out}})(P_+(def)(i), P_+(def)(i')) \leq f(d_{\text{In}}(i, i'))$. By definition of P_+ , this is equivalent to $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) \leq f(d_{\text{In}}(i, i'))$, which concludes the proof. \square

Conversely, func-cleanness for non-parametrised programs is able to encode func-cleanness for parametrised programs. As it was the case for robust cleanness, the cleanness contract for this encoding is not straightforward. Again, the contract requires a special input distance function and a customised function f . Equation (3.2) shows both. The input distance function $\#d_{\text{In}}^{(f, \approx_P)}$ is parametrised

with the function f and the comparability relation \approx_P from the cleanness contract for P_+ . As it is the case for $\#d_{\text{In}}^{(f, \approx_P)}$ in Equation (3.1), the inputs to the distance function are pairs (\mathbf{p}, i) , (\mathbf{p}', i') , each combining a parameter and an input to the parametrised program P_+ . The distance function distinguishes whether the two parameters \mathbf{p} and \mathbf{p}' are in the \approx_P -relation. If they are not in the relation, the distance between the corresponding outputs may be arbitrary large. Thus, in this case $\#d_{\text{In}}^{(f, \approx_P)}$ defines the input distance to be infinite and function $\flat f$ passes this value through to make it the upper bound for the output distance. So, the distance between outputs must be smaller or equal to infinity; i.e., it may be arbitrarily large. If \mathbf{p} and \mathbf{p}' are in the \approx_P -relation, the case distinction continues on whether the inputs i and i' are equal. If they are, $\#d_{\text{In}}^{(f, \approx_P)}$ returns zero, and $\flat f(0)$ returns $f(0)$. If i and i' are unequal, then $\#d_{\text{In}}^{(f, \approx_P)}$ returns $f(d_{\text{In}}(i, i')) + 1$. Since this is always larger than zero, $\flat f(\#d_{\text{In}}^{(f, \approx_P)}((\mathbf{p}, i), (\mathbf{p}', i')))$ is equal to $f(d_{\text{In}}(i, i'))$ and, hence, forwarding the output threshold originally intended by the contract for P_+ . Notice that the distinction between the first and the second case in $\#d_{\text{In}}^{(f, \approx_P)}$ is necessary, because $\flat f(\#d_{\text{In}}^{(f, \approx_P)}((\mathbf{p}, i), (\mathbf{p}, i)))$ must be equal to zero for $\#d_{\text{In}}^{(f, \approx_P)}$ to qualify as a distance function. We can also not use $d_{\text{In}}(i, i')$ in the second case, because this may return infinity, in which case $\flat f$ is not able to distinguish between an infinite input distance and infinity caused by non-matching parameters in case three of $\#d_{\text{In}}^{(f, \approx_P)}$. This would cause an incorrect cleanness verdict if f does not return infinity for infinite input distances. Also notice, that the symmetry property for $\#d_{\text{In}}^{(f, \approx_P)}$ is satisfied; thus, $\#d_{\text{In}}^{(f, \approx_P)}$ is a valid distance function.

$$\#d_{\text{In}}^{(f, \approx_P)}((\mathbf{p}, i), (\mathbf{p}', i')) := \begin{cases} 0 & \text{if } \mathbf{p} \approx_P \mathbf{p}' \text{ and } i = i' \\ f(d_{\text{In}}(i, i')) + 1 & \text{if } \mathbf{p} \approx_P \mathbf{p}' \text{ and } i \neq i' \\ \infty & \text{otherwise,} \end{cases} \quad \text{and}$$

$$\flat f(x) := \begin{cases} f(0) & \text{if } x = 0 \\ x - 1 & \text{otherwise.} \end{cases} \quad (3.2)$$

The following proposition shows how $\#d_{\text{In}}^{(f, \approx_P)}$ and $\flat f$ can be used in a contract for func-cleanness of a non-parametrised program to encode func-cleanness of a parametrised program.

Proposition 3.34. Let $P_+ : \text{Param} \rightarrow \text{In}_+ \rightarrow 2^{\text{Out}}$ be a parametrised program and $P : \text{In}_\# \rightarrow 2^{\text{Out}}$ the non-parametrised program with $\text{In}_\# = \text{Param} \times \text{In}_+$ and $P_+ \hookrightarrow P$, and let $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}_+, d_{\text{In}}, d_{\text{Out}}, f \rangle$ be a contract for parametrised programs and $\mathcal{C} = \langle \text{StdIn}_\#, d'_{\text{In}}, d_{\text{Out}}, \flat f \rangle$ a contract for non-parametrised programs, with

$\text{StdIn}_\# = \text{Param} \times \text{StdIn}_+$ and $d'_{\text{In}} = \#d_{\text{In}}^{(f, \approx_P)}$. Then, P_+ is func-clean w.r.t. C_+ if and only if P is func-clean w.r.t. C .

Proof. We first prove that func-cleanness of P_+ w.r.t. C_+ implies func-cleanness of P w.r.t. C . For this let $(p, i) \in \text{StdIn}_\#$ be a standard input of P and $(p', i') \in \text{In}_\#$ a regular input. Hence, $p, p' \in \text{Param}$, $i \in \text{StdIn}_+$ and $i' \in \text{In}_+$. We distinguish three cases.

- For the first case, assume $p \approx_P p'$ and $i = i'$. Then, we get from func-cleanness of P_+ that $\mathcal{H}(d_{\text{Out}})(P_+(p)(i), P_+(p')(i')) \leq f(d_{\text{In}}(i, i'))$. Since $i = i'$, $d_{\text{In}}(i, i') = 0$. By definition of d'_{In} , $d'_{\text{In}}((p, i), (p', i')) = 0$ and hence, by definition of $\flat f$, $\flat f(d'_{\text{In}}((p, i), (p', i'))) = f(0) = f(d_{\text{In}}(i, i'))$. Finally, by using the definition of P , we get $\mathcal{H}(d_{\text{Out}})(P(p, i), P(p', i')) \leq \flat f(d'_{\text{In}}((p, i), (p', i')))$, proving that P is func-clean w.r.t. C .
- Next, assume $p \approx_P p'$ and $i \neq i'$. Then, we get from func-cleanness of P_+ that $\mathcal{H}(d_{\text{Out}})(P_+(p)(i), P_+(p')(i')) \leq f(d_{\text{In}}(i, i'))$. By definition of P , this is equivalent to $\mathcal{H}(d_{\text{Out}})(P(p, i), P(p', i')) \leq f(d_{\text{In}}(i, i'))$. Using arithmetic operations, we get $f(d_{\text{In}}(i, i')) = f(d_{\text{In}}(i, i')) + 1 - 1 = d'_{\text{In}}((p, i), (p', i')) - 1$. Finally, since $d'_{\text{In}}((p, i), (p', i'))$ is always greater than zero, this is equal to $\flat f(d'_{\text{In}}((p, i), (p', i')))$, which proves that P is func-clean w.r.t. C .
- If we, conversely, assume that $p \not\approx_P p'$, then, by definition of d'_{In} , it suffices to show that $\mathcal{H}(d_{\text{Out}})(P(p, i), P(p', i')) \leq \infty$, which is always satisfied regardless of the arguments of the Hausdorff distance.

To prove the inverse implication, that func-cleanness of P_+ w.r.t. C_+ follows from func-cleanness of P w.r.t. C , let $p, p' \in \text{Param}$ be parameters with $p \approx_P p'$, $i \in \text{StdIn}_+$ a standard input and $i' \in \text{In}_+$ a regular input of P_+ . Noticing that $(p, i) \in \text{StdIn}_\#$ is a standard input of P and that $(p', i') \in \text{In}_\#$ is a regular input of P , it follows from func-cleanness of P that $\mathcal{H}(d_{\text{Out}})(P(p, i), P(p', i')) \leq \flat f(d'_{\text{In}}((p, i), (p', i')))$. Because we know that $p \approx_P p'$, we can infer either, in case $i = i'$, that $\flat f(d'_{\text{In}}((p, i), (p', i'))) = \flat f(0) = f(0) = f(d_{\text{In}}(i, i'))$, or, in case $i \neq i'$, that $\flat f(d'_{\text{In}}((p, i), (p', i'))) = \flat f(f(d_{\text{In}}(i, i')) + 1)$, which is equal to $f(d_{\text{In}}(i, i'))$, because $d_{\text{In}}(i, i') + 1$ is always greater than 0. By using this and by applying the definition of P , we get that $\mathcal{H}(d_{\text{Out}})(P_+(p)(i), P_+(p')(i')) \leq f(d_{\text{In}}(i, i'))$, which concludes the proof. \square

Analogue to l-func-cleanness and u-func-cleanness for parametrised programs, there are quantifier-based concepts of lower and upper bounds of non-standard behaviour for non-parametrised programs.

Definition 3.35. A non-parametrised nondeterministic sequential program P is *l-func-clean* w.r.t. contract $C = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for every standard input

$i \in \text{StdIn}$ and input $i' \in \text{In}$, it holds that for all $\mathfrak{o} \in P(i)$, there exists $\mathfrak{o}' \in P(i')$, such that $d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq f(d_{\text{In}}(i, i'))$.

Definition 3.36. A non-parametrised nondeterministic sequential program P is *u-func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for every standard input $i \in \text{StdIn}$ and input $i' \in \text{In}$, it holds that for all $\mathfrak{o}' \in P(i')$, there exists $\mathfrak{o} \in P(i)$, such that $d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq f(d_{\text{In}}(i, i'))$.

The following Propositions 3.37 and 3.38 show that also for non-parametrised programs the quantifier-based notion of cleanness given by the conjunction of Definitions 3.35 and 3.36 is almost equivalent to the Hausdorff-based func-cleanness.

Proposition 3.37. Let $P : \text{In} \rightarrow 2^{\text{Out}}$ be a non-parametrised nondeterministic sequential program and $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract for func-cleanness. If P is l-func-clean w.r.t. \mathcal{C} and P is u-func-clean w.r.t. \mathcal{C} , then P is func-clean w.r.t. \mathcal{C} .

Proof. Let $i \in \text{StdIn}$, $i' \in \text{In}$, and $\kappa = f(d_{\text{In}}(i, i'))$. Then, it suffices to show that $\forall \mathfrak{o} \in P(i). \exists \mathfrak{o}' \in P(i'). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa$ (l-func-cleanness) and $\forall \mathfrak{o}' \in P(i'). \exists \mathfrak{o} \in P(i). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa$ (u-func-cleanness) imply $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) \leq \kappa$. For l-func-cleanness we get with Lemma 3.12 that $\sup_{\mathfrak{o} \in P(i)} \inf_{\mathfrak{o}' \in P(i')} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa$, and for u-func-cleanness that $\sup_{\mathfrak{o}' \in P(i')} \inf_{\mathfrak{o} \in P(i)} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') \leq \kappa$. As both sup-inf-combinations are less or equal to κ , also the maximum of both is less or equal to κ . Hence, $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) \leq \kappa = f(d_{\text{In}}(i, i'))$. \square

Proposition 3.38. Let P be a non-parametrised nondeterministic sequential program and $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract for func-cleanness. Then, for every standard input $i \in \text{StdIn}$ and input $i' \in \text{In}$, $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) < f(d_{\text{In}}(i, i'))$ implies that the following two proposition holds:

1. for all $\mathfrak{o} \in P(i)$, there exists $\mathfrak{o}' \in P(i')$ such that $d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < f(d_{\text{In}}(i, i'))$,
2. for all $\mathfrak{o}' \in P(i')$, there exists $\mathfrak{o} \in P(i)$ such that $d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < f(d_{\text{In}}(i, i'))$.

Proof. Let $i \in \text{StdIn}$, $i' \in \text{In}$ and $\kappa = f(d_{\text{In}}(i, i'))$. Then, it suffices to show that $\mathcal{H}(d_{\text{Out}})(P(i), P(i')) < \kappa$ implies that $\forall \mathfrak{o} \in P(i). \exists \mathfrak{o}' \in P(i'). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa$ (l-func-cleanness) and $\forall \mathfrak{o}' \in P(i'). \exists \mathfrak{o} \in P(i). d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa$ (u-func-cleanness). By unrolling the definition of the Hausdorff distance and the definition of the maximum function, we get that $\sup_{\mathfrak{o} \in P(i)} \inf_{\mathfrak{o}' \in P(i')} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa$ holds and that $\sup_{\mathfrak{o}' \in P(i')} \inf_{\mathfrak{o} \in P(i)} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}') < \kappa$ holds. From the former, we get l-func-cleanness with Lemma 3.16, and from the latter and Lemma 3.16 we obtain u-func-cleanness. \square

Example 3.39. For the example of the emission control system in Figures 3.4 and 3.5 take the same cleanness contract as in Example 3.26, but where κ_i and κ_o are replaced by $f(x) = x/2$. Then the program in Figure 3.4 is func-clean w.r.t. this adapted contract, while the program in Figure 3.5 is not.

3.2 Reactive Systems

We illustrated in Examples 3.26 and 3.39 for sequential programs how robust cleanness and func-cleanness can put requirements on a car beyond a well-tested set of inputs (which we call standard inputs). In these examples, the input is the position of the throttle and the set of standard inputs a set `ThrottleTestValues`. The output is the amount of NO_x emitted by the car. Obviously, these examples are toy examples – the real NO_x result is a complex chemical reaction that depends on more inputs than just the position of the throttle. Another aspect that is currently off reality is the assumption that the engine and the emission cleaning system are sequential programs. In fact, these systems are inherently reactive. A proper modelling of the car would reflect that the car is in a certain state containing the current speed, the engine temperature, the position of the camshafts, etc. In particular, the selective catalytic reduction emission cleaning system uses a combination of the amount of NO_x measured before the emission cleaning, the amount of DEF injected in the exhaust stream in a previous time instant and the amount of remaining NO_x in the exhaust stream after this injection. Such reactive behaviour cannot be captured by the definitions of strict cleanness, robust cleanness or func-cleanness in the previous section. Hence, in this section we will extend the definitions to make these cleanness concepts applicable to reactive systems.

We consider a parameterised reactive system as a function $S_+ : \text{Param} \rightarrow \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ and a non-parametrised reactive system as $S : \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$, so that any instance of the system reacts to the k -th input in the input sequence producing the k -th output in each respective output sequence. Thus, each instance of the system can be seen, for instance, as a (non-deterministic) Mealy or Moore machine. Similar to sequential programs, parametrised and non-parametrised reactive systems can encode each other. Let $S_+ : \text{Param} \rightarrow \text{In}_+^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a parametrised system. Then, with $\text{In}_x = \text{Param} \times \text{In}_+$, the non-parametrised system $S : \text{In}_x^\omega \rightarrow 2^{(\text{Out}^\omega)}$ with $S((p, i)) := S_+(p[0])(i)$ encodes S_+ , denoted $S_+ \hookrightarrow S$. Conversely, let $S : \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a non-parametrised reactive system. S can be encoded by the parametrised reactive system $S_+ : \{\text{def}\} \rightarrow \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ with $S_+(\text{def})(i) := S(i)$, denoted $S \hookrightarrow S_+$.

In the remainder of this section, we use S to denote non-parametrised reactive systems and S_+ for parametrised reactive systems. If not stated otherwise, we

assume that S is a function of type $\text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ and that S_+ is a function of type $\text{Param} \rightarrow \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$.

3.2.1 Strict cleanness

For cleanness contracts, we require that $\text{StdIn} \subseteq \text{In}^\omega$. The definitions of strict cleanness for reactive systems strongly resemble strict cleanness for sequential programs defined in Definitions 3.1 and 3.2.

Definition 3.40. A parametrised nondeterministic reactive system S_+ is *strictly clean* w.r.t. contract $\mathcal{C} = \langle \approx_\rho, \text{StdIn} \rangle$, if for all pairs of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_\rho \mathbf{p}'$ and every standard input $i \in \text{StdIn}$, it holds that $S_+(\mathbf{p})(i) = S_+(\mathbf{p}')(i)$.

As for sequential programs, strict cleanness considers pairs of parameters that are in a parameter comparability relation \approx_ρ . For an arbitrary infinitely long input sequence $i \in \text{StdIn}$ that is considered a standard input, both instantiations $S_+(\mathbf{p})$ and $S_+(\mathbf{p}')$ of the system must produce the same set of outputs for i . The relation \approx_ρ and set StdIn make up the cleanness contract.

In cases where the behaviour of a system can be better modelled without the distinction between inputs and parameters, we provide a definition for non-parametrised systems.

Definition 3.41. A non-parametrised nondeterministic reactive system S is *strictly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, \approx_I \rangle$, if for all pairs of inputs $i, i' \in \text{StdIn}$ with $i \approx_I i'$, it holds that $S(i) = S(i')$.

This definition is also very similar to its sequential-program counterpart. Instead of a parameter relation \approx_ρ , contracts for the non-parametrised strict cleanness define comparability for inputs by means of a relation \approx_I . Strict cleanness enforces for two arbitrary standard inputs (i.e., infinitely long sequences of input symbols, where both sequences come from set StdIn) in this relation that the system produces the same set of outputs.

The parametrised and the non-parametrised variant of strict cleanness are equally expressive. We show in Propositions 3.42 and 3.43 contracts that encode each other assuming that the systems encode each other according to the \hookrightarrow -definition. We remark that we will use the operator \times^ω , which builds the component-wise cross product of infinite traces. Also, we will in the following silently convert pairs of traces into traces of pairs and vice versa. We refer to Section 2.1 for the formal details.

Proposition 3.42. Let $S_+ : \text{Param} \rightarrow \text{In}_+^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a parametrised nondeterministic reactive system and $S : \text{In}_\#^\omega \rightarrow 2^{(\text{Out}^\omega)}$ the non-parametrised nondeterministic reactive system that encodes S_+ , i.e., $S_+ \hookrightarrow S$ and $\text{In}_\# = \text{Param} \times \text{In}_+$. Further, let $\mathcal{C}_+ = \langle \approx_\rho, \text{StdIn}_+ \rangle$ and $\mathcal{C} = \langle \text{StdIn}_\#, \approx_I \rangle$ be contracts with $\text{StdIn}_+ \subseteq \text{In}_+^\omega$,

$\text{Stdln}_\# = \text{Param}^\omega \times^\omega \text{Stdln}_+$ and $(\mathbf{p}, i) \approx_1 (\mathbf{p}', i')$ if and only if $\mathbf{p}[0] \approx_P \mathbf{p}'[0]$ and $i = i'$. Then, \mathbf{S}_+ is strictly clean w.r.t. \mathcal{C}_+ if and only if \mathbf{S} is strictly clean w.r.t. \mathcal{C} .

Proof. We prove the two directions of the equivalence separately. First, assume strict cleanness of \mathbf{S}_+ . To prove strict cleanness of \mathbf{S} , let $(\mathbf{p}_1, i_1), (\mathbf{p}_2, i_2) \in \text{Stdln}_\#$. Hence $\mathbf{p}_1, \mathbf{p}_2 \in \text{Param}^\omega$ and $i_1, i_2 \in \text{Stdln}_+$. Further, we may assume that $(\mathbf{p}_1, i_1) \approx_1 (\mathbf{p}_2, i_2)$. We must show that $\mathbf{S}(\mathbf{p}_1, i_1) = \mathbf{S}(\mathbf{p}_2, i_2)$. Since $\mathbf{S}_+ \hookrightarrow \mathbf{S}$, this is equivalent to showing that $\mathbf{S}_+(\mathbf{p}_1[0])(i_1) = \mathbf{S}_+(\mathbf{p}_2[0])(i_2)$. As \mathbf{S}_+ is strictly clean, it suffices to show that $\mathbf{p}_1[0] \approx_P \mathbf{p}_2[0]$ and $i_1 = i_2$ which follows immediately from $(\mathbf{p}_1, i_1) \approx_1 (\mathbf{p}_2, i_2)$ with the definition of \approx_1 .

For the inverse implication, assume strict cleanness of \mathbf{S} to prove strict cleanness of \mathbf{S}_+ . Let $\mathbf{p}_1, \mathbf{p}_2 \in \text{Param}$ with $\mathbf{p}_1 \approx_P \mathbf{p}_2$, and $i \in \text{Stdln}_+$. Then, we must show that $\mathbf{S}_+(\mathbf{p}_1)(i) = \mathbf{S}_+(\mathbf{p}_2)(i)$, or equivalently that $\mathbf{S}_+(\mathbf{p}_1^\omega[0])(i) = \mathbf{S}_+(\mathbf{p}_2^\omega[0])(i)$ which, by definition of $\mathbf{S}_+ \hookrightarrow \mathbf{S}$, is equivalent to $\mathbf{S}(\mathbf{p}_1^\omega, i) = \mathbf{S}(\mathbf{p}_2^\omega, i)$. Since \mathbf{S} is strictly clean, it suffices to show that $(\mathbf{p}_1^\omega, i) \approx_1 (\mathbf{p}_2^\omega, i)$, i.e., according to the definition of \approx_1 , that $i = i$ (which is obviously true) and that $\mathbf{p}_1^\omega[0] \approx_P \mathbf{p}_2^\omega[0]$. The latter is equivalent to $\mathbf{p}_1 \approx_P \mathbf{p}_2$ for which we know from the assumptions of strict cleanness of \mathbf{S}_+ that it holds. \square

Next, we show that every non-parametrised system \mathbf{S} can be encoded by some parametrised system \mathbf{S}_+ , and at the same time that strict cleanness for a contract for non-parametrised systems can be encoded by some contract for parametrised systems. To this end, we use an encoding technique similar to $\hookrightarrow_{\text{Param}}$ for sequential programs (cf. Section 3.1). The inputs that would be passed to \mathbf{S} will be passed as parameters to the encoding system \mathbf{S}_+ . The inputs to \mathbf{S}_+ are irrelevant for the behaviour of \mathbf{S}_+ , so the set of inputs contains only a single “default” trace. The details are given in the following proposition.

Proposition 3.43. Let $\mathbf{S} : \text{In}_\#^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a non-parametrised nondeterministic reactive system. To encode \mathbf{S} with a parametrised system, let $\text{Param} = \text{In}_\#^\omega$ and $\mathbf{S}_+ : \text{Param} \rightarrow \{\text{def}\}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ the parametrised nondeterministic reactive system with $\mathbf{S}_+(i)(\text{def}^\omega) := \mathbf{S}(i)$. Further, let $\mathcal{C} = \langle \text{Stdln}_\#, \approx_1 \rangle$ and $\mathcal{C}_+ = \langle \approx_P, \text{Stdln}_+ \rangle$ be contracts with $\text{Stdln}_\# \subseteq \text{In}_\#^\omega$, $\text{Stdln}_+ = \{\text{def}\}^\omega$ and $i_1 \approx_P i_2$ if and only if $i_1 \in \text{Stdln}_\#$ and $i_2 \in \text{Stdln}_\#$ and $i_1 \approx_1 i_2$. Then, \mathbf{S} is strictly clean w.r.t. \mathcal{C} if and only if \mathbf{S}_+ is strictly clean w.r.t. \mathcal{C}_+ .

Proof. We prove the two implications underneath the equivalence separately. First, we assume strict cleanness of \mathbf{S} to show strict cleanness of \mathbf{S}_+ . Hence, let $i_1, i_2 \in \text{Param} (= \text{In}_\#)$ with $i_1 \approx_P i_2$. Further, let $i_3 \in \text{Stdln}_+ = \{\text{def}\}^\omega$, i.e., $i_3 = \text{def}^\omega$. We must show that $\mathbf{S}_+(i_1)(\text{def}^\omega) = \mathbf{S}_+(i_2)(\text{def}^\omega)$. By definition of \mathbf{S}_+ , this is equivalent to $\mathbf{S}(i_1) = \mathbf{S}(i_2)$. Using strict cleanness of \mathbf{S} , it suffices to show

that $i_1, i_2 \in \text{StdIn}_\neq$ and $i_1 \approx_1 i_2$. This follows from the assumption that $i_1 \approx_P i_2$ and the definition of \approx_P .

For the inverse implication, we must show that under the assumption that S_+ is strictly clean, also S is strictly clean. For this, let $i_1, i_2 \in \text{StdIn}_\neq$ with $i_1 \approx_1 i_2$. We must show $S(i_1) = S(i_2)$, which is, according to the definition of S_+ , equivalent to $S_+(i_1)(\text{def}^\omega) = S_+(i_2)(\text{def}^\omega)$. From strict cleanliness of S_+ , we know that this is the case if $i_1, i_2 \in \text{Param} = \text{In}^\omega$ (which is obviously true) and if $i_1 \approx_P i_2$. By definition of \approx_P , $i_1 \approx_P i_2$ holds if $i_1, i_2 \in \text{StdIn}_\neq$ and $i_1 \approx_1 i_2$. Both conditions hold by assumption. \square

3.2.2 Robust cleanliness

The strict cleanliness definitions adapted for reactive systems are conceptually identical to their counterparts for sequential programs; roughly, the only changes are “type changes” to replace single values with infinite sequences of values. For robust cleanliness, such minimal changes would cause a very undesired side effect. Suppose two input sequences in In^ω that only differ by a single input in some late k -th position. Assume that this difference causes the inputs to have a quantified distance of more than κ_i . Now the program under study may become clean even if the respective outputs differ enormously at an early k' -th position ($k' < k$). Notice that there is no justification for such early difference on the output, since the input sequences are the same up to position k' .

In fact, we notice that the property of being clean should be of a safety nature: if there is a point in a pair of executions in which the system is detected to be doped, there should be no extension of such executions that can correct it and make the system clean. In the observation above, the k' -th prefix of the trace should be considered the bad prefix and the system deemed as doped.

Therefore, we consider distances on finite traces: $d_{\text{In}} : (\text{In}^* \times \text{In}^*) \rightarrow \overline{\mathbb{R}}_{\geq 0}$ and $d_{\text{Out}} : (\text{Out}^* \times \text{Out}^*) \rightarrow \overline{\mathbb{R}}_{\geq 0}$. With these, we provide definitions of robust cleanliness on reactive systems that ensure for any $k \in \mathbb{N}$ that, as long as all j -th prefixes of a given input sequence, with $j \leq k$, are within κ_i distance, the k -th prefixes of the output sequence are within κ_o distance.

Definition 3.44. A parametrised nondeterministic reactive system S_+ is *robustly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for all pairs of parameters $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and index $k \in \mathbb{N}$, it holds whenever $d_{\text{In}}(i[.j], i'[.j]) \leq \kappa_i$ for all $j \leq k$, that $\mathcal{H}(d_{\text{Out}})(S_+(\mathfrak{p})(i)[.k], S_+(\mathfrak{p}')(i')[.k]) \leq \kappa_o$.

By having as precondition that $d_{\text{In}}(i[.j], i'[.j]) \leq \kappa_i$ for all $j \leq k$, this definition considers the fact that once one instance of the system deviates too much from the normal behaviour (i.e., beyond κ_i distance at the input), this instance

is not obliged any longer to meet (within κ_o distance) the output, even if later inputs get closer again. This enables robustly clean systems to stop if an input outside the standard domain may result harmful for the system. Also, notice that, by considering the conditions through all k -th prefixes the definition encompasses the safety nature of robust cleanness.

The conceptual idea of lower and upper bounds for non-standard behaviour (cf. page 44) can also be applied to reactive systems. The following definitions define l-robust cleanness and u-robust cleanness for reactive systems.

Definition 3.45. A parametrised nondeterministic reactive system S_+ is *l-robustly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every pair of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$, and index $k \in \mathbb{N}$, it holds whenever $d_{\text{In}}(i[..j], i'[..j]) \leq \kappa_i$ for all $j \leq k$, that for all $o \in S_+(\mathbf{p})(i)$, there exists $o' \in S_+(\mathbf{p}')(i')$, such that $d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$.

Definition 3.46. A parametrised nondeterministic reactive system S_+ is *u-robustly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every pair of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$, and index $k \in \mathbb{N}$, it holds whenever $d_{\text{In}}(i[..j], i'[..j]) \leq \kappa_i$ for all $j \leq k$, that for all $o' \in S_+(\mathbf{p}')(i')$, there exists $o \in S_+(\mathbf{p})(i)$, such that $d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$.

As we did for sequential programs, we assemble a quantifier-based variant of robust cleanness by taking the conjunction of l-robust cleanness and u-robust cleanness. The following propositions show that this variant is almost equivalent (in the same spirit as in Section 3.1) to the Hausdorff-based variant.

Proposition 3.47. Let S_+ be a parametrised nondeterministic reactive system and $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract for robust cleanness. If S_+ is l-robustly clean w.r.t. \mathcal{C} and S_+ is u-robustly clean w.r.t. \mathcal{C} , then S_+ is robustly clean w.r.t. \mathcal{C} .

Proof. Let $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. Assume that for all $j \leq k$, $d_{\text{In}}(i[..j], i'[..j]) \leq \kappa_i$. To prove robust cleanness, we must show that $\mathcal{H}(d_{\text{Out}})(S_+(\mathbf{p})(i)[..k], S_+(\mathbf{p}')(i')[..k]) \leq \kappa_o$. After unrolling the definition of the Hausdorff distance, we need to show that the maximum of $\sup_{o \in S_+(\mathbf{p})(i)[..k]} \inf_{o' \in S_+(\mathbf{p}')(i')[..k]} d_{\text{Out}}(o, o')$ and $\sup_{o' \in S_+(\mathbf{p}')(i')[..k]} \inf_{o \in S_+(\mathbf{p})(i)[..k]} d_{\text{Out}}(o, o')$ is less or equal than κ_o . Simple logical operations transform the proof goal into the conjunction of $\sup_{o \in S_+(\mathbf{p})(i)} \inf_{o' \in S_+(\mathbf{p}')(i')} d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$ and $\sup_{o' \in S_+(\mathbf{p}')(i')} \inf_{o \in S_+(\mathbf{p})(i)} d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$. With Lemma 3.12, it suffices to show that $\forall o \in S_+(\mathbf{p})(i). \exists o' \in S_+(\mathbf{p}')(i'). d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$ and $\forall o' \in S_+(\mathbf{p}')(i'). \exists o \in S_+(\mathbf{p})(i). d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$. The first conjunct follows immediately from l-robust cleanness of S_+ for \mathbf{p}, \mathbf{p}' , i and i' , and similarly, the second conjunct follows from u-robust cleanness. \square

Proposition 3.48. Let S_+ be a parametrised nondeterministic reactive system and $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract for robust cleanness. Then, for every pair of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$, and index $k \in \mathbb{N}$, $\mathcal{H}(d_{\text{Out}})(S_+(\mathbf{p})(i)[..k], S_+(\mathbf{p}')(i')[..k]) < \kappa_o$ implies that

1. for all $o \in S_+(\mathbf{p})(i)$, there exists $o' \in S_+(\mathbf{p}')(i')$,
such that $d_{\text{Out}}(o[..k], o'[..k]) < \kappa_o$ and
2. for all $o' \in S_+(\mathbf{p}')(i')$, there exists $o \in S_+(\mathbf{p})(i)$,
such that $d_{\text{Out}}(o[..k], o'[..k]) < \kappa_o$.

Proof. Let $\mathbf{p}, \mathbf{p}' \in \text{Param}$, $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. We may assume that $\mathcal{H}(d_{\text{Out}})(S_+(\mathbf{p})(i)[..k], S_+(\mathbf{p}')(i')[..k]) < \kappa_o$. We must show that the inequalities $\forall o \in S_+(\mathbf{p})(i). \exists o' \in S_+(\mathbf{p}')(i'). d_{\text{Out}}(o[..k], o'[..k]) < \kappa_o$ and $\forall o' \in S_+(\mathbf{p}')(i'). \exists o \in S_+(\mathbf{p})(i). d_{\text{Out}}(o[..k], o'[..k]) < \kappa_o$ hold. Using Lemma 3.16, we may equivalently show that $\sup_{o \in S_+(\mathbf{p})(i)} \inf_{o' \in S_+(\mathbf{p}')(i')} d_{\text{Out}}(o[..k], o'[..k]) < \kappa_o$ and $\sup_{o' \in S_+(\mathbf{p}')(i')} \inf_{o \in S_+(\mathbf{p})(i)} d_{\text{Out}}(o[..k], o'[..k]) < \kappa_o$. With simple logical operation, this can be transformed into the equivalent proposition constraining the maximum of $\sup_{o \in S_+(\mathbf{p})(i)[..k]} \inf_{o' \in S_+(\mathbf{p}')(i')[..k]} d_{\text{Out}}(o, o')$ and $\sup_{o' \in S_+(\mathbf{p}')(i')[..k]} \inf_{o \in S_+(\mathbf{p})(i)[..k]} d_{\text{Out}}(o, o')$ to be smaller than κ_o . By the definition of the Hausdorff distance, this is equivalent to $\mathcal{H}(d_{\text{Out}})(S_+(\mathbf{p})(i)[..k], S_+(\mathbf{p}')(i')[..k]) < \kappa_o$, which we assumed at the beginning of the proof. \square

For systems in which parameters do not occur we provide alternative robust cleanness definitions for non-parametrised systems.

Definition 3.49. A non-parametrised nondeterministic reactive system S is *robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and index $k \in \mathbb{N}$, if $d_{\text{In}}(i[..j], i'[..j]) \leq \kappa_i$ for all $j \leq k$, then $\mathcal{H}(d_{\text{Out}})(S(i)[..k], S(i')[..k]) \leq \kappa_o$.

Example 3.50. A slightly more realistic version of the emission control system than the one in Figure 3.4 is given in Figure 3.6. It is a closed loop where the calculation of the DEF dosage also depends on the previous reading of NO_x . Moreover, the DEF dosage affects the amount of NO_x emissions nondeterministically. The program models a margin of error on the NO_x emission, which is represented by the factor λ in the nondeterministic assignment of variable NO_x in the penultimate line within the loop. This nondeterministic assignment is an (admittedly unrealistic) abstraction of the chemical reaction between the exhaust gases and the DEF dosage.

The doped version of the emission control system instrumenting the cheating hack is shown in Figure 3.7. Both emission cleaning systems use constant $\lambda = 0.1$

```

procedure EMISSIONCONTROL()
   $NOx := 0$ 
  loop
    READ(throttle)
     $def\_dose := SCRMODEL(throttle, NOx)$ 
     $NOx := \left[ (1 - \lambda) \frac{throttle^3}{2 \cdot def\_dose}, (1 + \lambda) \frac{throttle^3}{2 \cdot def\_dose} \right]$ 
    OUTPUT( $NOx$ )
  end loop
end procedure

```

Figure 3.6: Emission control (reactive).

and a function `SCRMODEL` that models the amount of DEF that the SCR injects into the exhaust stream. The cheating version of the emission cleaning system switches to an “alternative” SCR system if the throttle inputs are not in the throttle test values set. We use the same `ThrottleTestValues = (0, 1]` as in Section 3.1. The SCR models are given by the functions

$$SCRMODEL(x, n) = \begin{cases} x^2 & \text{if } 2 \cdot n \leq x \\ (1 + \lambda) \cdot x^2 & \text{otherwise} \end{cases} \quad \text{and}$$

$$ALTSCRMODEL(x, n) = x.$$

`ALTSCRMODEL` ignores the feedback of the NO_x emission resulting in the same `ALTSCRMODEL` as in Example 3.5. We also take `ln = (0, 2]` (recall that these are the values that variable *throttle* takes). The idea of the feedback in `SCRMODEL` is that if the previous emission was higher than expected with the planned current dosage, then the actual current dosage is an extra λ portion above the planned dosage. For the contract w.r.t. which we analyse robust cleanness, let `Stdln = (0, 1]ω` and define $d_{\text{In}}(i, i') = |\text{last}(i) - \text{last}(i')|$ and similarly $d_{\text{Out}}(o, o') = |\text{last}(o) - \text{last}(o')|$, where `last(t)` is the last element of the finite trace *t*. We take $\kappa_i = 2$ and $\kappa_o = 1.1$. (κ_o needs to be a little larger than in Example 3.26 due to the nondeterministic assignment to *NOx*.) With this contract, the system in Figure 3.6 satisfies robust cleanness and the system in Figure 3.7 violates it.

As it was the case for sequential programs, robust cleanness for non-parametrised systems is equally expressive as for parametrised systems. To show this, Propositions 3.51 and 3.52 provide a contract for a parametrised system that encodes a non-parametrised one, and vice versa.


```

procedure EMISSIONCONTROL()
   $NOx := 0$ 
  loop
    READ( $throttle$ )
    if  $throttle \in \text{ThrottleTestValues}$  then
       $def\_dose := \text{SCRMODEL}(throttle, NOx)$ 
    else
       $def\_dose := \text{ALTSCRMODEL}(throttle, NOx)$ 
    end if
     $NOx := \left[ (1 - \lambda) \frac{throttle^3}{2 \cdot def\_dose}, (1 + \lambda) \frac{throttle^3}{2 \cdot def\_dose} \right]$ 
    OUTPUT( $NOx$ )
  end loop
end procedure

```

Figure 3.7: Doped emission control (reactive)

Proposition 3.51. Let $S : \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a non-parametrised nondeterministic reactive system and let $S_+ : \{def\} \rightarrow \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be the parametrised nondeterministic reactive system that encodes S , i.e., $S \hookrightarrow S_+$. Further, let $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ and $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be contracts, where $def \approx_P def$ and $\text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o$ are identical in both contracts. Then, S is robustly clean w.r.t. \mathcal{C} if and only if S_+ is robustly clean w.r.t. \mathcal{C}_+ .

Proof. We prove the equivalence by separately proving the two underlying implications. First, let S be robustly clean; we must show that S_+ is robustly clean. For this, let $\mathfrak{p}, \mathfrak{p}' \in \{def\}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, i.e., $\mathfrak{p} = \mathfrak{p}' = def$, let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. We may assume that $\forall j \leq k. d_{\text{In}}(i[..\cdot j], i'[..\cdot j]) \leq \kappa_i$ and must show that $\mathcal{H}(d_{\text{Out}})(S_+(def)(i)[..k], S_+(def)(i')[..k]) \leq \kappa_o$. From $S \hookrightarrow S_+$, we know that this is equivalent to showing that $\mathcal{H}(d_{\text{Out}})(S(i)[..k], S(i')[..k]) \leq \kappa_o$. With the above assumptions, this follows immediately from robust cleanliness of S .

For the inverse implication, assume robust cleanliness of S_+ , to show that S is robustly clean. Let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. We may assume that $\forall j \leq k. d_{\text{In}}(i[..\cdot j], i'[..\cdot j]) \leq \kappa_i$ and need to show that $\mathcal{H}(d_{\text{Out}})(S(i)[..k], S(i')[..k]) \leq \kappa_o$. This is equivalent to $\mathcal{H}(d_{\text{Out}})(S_+(def)(i)[..k], S_+(def)(i')[..k]) \leq \kappa_o$, since $S \hookrightarrow S_+$. With all the assumptions above, and that by definition $def \approx_P def$, the inequality follows from S_+ 's robust cleanliness. \square

The inverse encoding (again) requires a more elaborate contract for the non-parametrised system encoding the parametrised one. We adjust the input distance function $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$ from Equation (3.1) to reactive systems (and leave the

definition of $b\kappa_i$ unchanged). The main difference is that the conditions for cases 1 and 2 use $\mathbf{p}[0]$ (and $\mathbf{p}'[0]$) instead of \mathbf{p} (and \mathbf{p}'). We remark that this change keeps the reflexivity and symmetry of $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$ intact. Thus, $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$ is still a valid distance function.

$$\#d_{\text{In}}^{(\kappa_i, \approx_P)}((\mathbf{p}, i), (\mathbf{p}', i')) := \begin{cases} d_{\text{In}}(i, i') & \text{if } \kappa_i \neq \infty \text{ and } \mathbf{p}[0] \approx_P \mathbf{p}'[0] \\ 0 & \text{if } \kappa_i = \infty \text{ and } \mathbf{p}[0] \approx_P \mathbf{p}'[0] \\ \infty & \text{otherwise,} \end{cases} \quad (3.3)$$

The following proposition specifies for a parametrised system and a robust cleanness contract, the cleanness contract for the non-parametrised system that encodes the parametrised on.

Proposition 3.52. Let $S_+ : \text{Param} \rightarrow \text{In}_+^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a parametrised non-deterministic reactive system and let $S : \text{In}_\times^\omega \rightarrow 2^{(\text{Out}^\omega)}$, with $\text{In}_\times = \text{Param} \times \text{In}_+$, be the non-parametrised system that encodes S_+ , i.e., $S_+ \hookrightarrow S$. Further, let $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}_+, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ and $\mathcal{C} = \langle \text{StdIn}_\times, d'_{\text{In}}, d_{\text{Out}}, b\kappa_i, \kappa_o \rangle$ be contracts with $\text{StdIn}_\times = \text{Param}^\omega \times^\omega \text{StdIn}_+$ and $d'_{\text{In}} = \#d_{\text{In}}^{(\kappa_i, \approx_P)}$. Then, S_+ is robustly clean w.r.t. \mathcal{C}_+ if and only if S is robustly clean w.r.t. \mathcal{C} .

The proof of this proposition involves many steps that are shared by similar proofs, e.g., proofs that show that l-robust cleanness for parametrised and non-parametrised systems are equally expressive. These shared proof steps are captured in the proof of the following lemma.

Lemma 3.53. Let Param be a set of parameters, In_+ a set of input symbols and $\text{StdIn}_+ \subseteq \text{In}_+^\omega$ a set of standard inputs. Derived from these, let $\text{In}_\times := \text{Param} \times \text{In}_+$ be another set of input symbols and $\text{StdIn}_\times := \text{Param}^\omega \times^\omega \text{StdIn}_+$ a set of standard inputs. Further, let \approx_P be a parameter comparability relation, $d_{\text{In}} : \text{In}_+^* \times \text{In}_+^* \rightarrow \overline{\mathbb{R}}_{\geq 0}$ an input distance function and $\kappa_i \in \overline{\mathbb{R}}_{\geq 0}$ an input distance threshold. Finally, let V and W be two predicates for which it holds for all $\mathbf{p}_1, \mathbf{p}_2 \in \text{Param}^\omega$, $i_1 \in \text{StdIn}_+$, $i_2 \in \text{In}_+^\omega$ and index $k \in \mathbb{N}$ that $V(\mathbf{p}_1[0], \mathbf{p}_2[0], i_1, i_2, k) \Leftrightarrow W(\mathbf{p}_1, \mathbf{p}_2, i_1, i_2, k)$. Then, the following statements are equivalent:

1. For all $\mathbf{p}_1, \mathbf{p}_2 \in \text{Param}$ with $\mathbf{p}_1 \approx_P \mathbf{p}_2$, $i_1 \in \text{StdIn}_+$, $i_2 \in \text{In}_+^\omega$, and index $k \in \mathbb{N}$, if $d_{\text{In}}(i_1[.j], i_2[.j]) \leq \kappa_i$ for all $j \leq k$, then $V(\mathbf{p}_1, \mathbf{p}_2, i_1, i_2, k)$.
2. For all $\hat{i}_1 \in \text{StdIn}_\times$, $\hat{i}_2 \in \text{In}_\times^\omega$, and index $k \in \mathbb{N}$, if for all $j \leq k$ it holds that $\#d_{\text{In}}^{(\kappa_i, \approx_P)}(\hat{i}_1[.j], \hat{i}_2[.j]) \leq b\kappa_i$, then there exist $\mathbf{p}_1, \mathbf{p}_2 \in \text{Param}^\omega$, $i_1 \in \text{StdIn}_+$, and $i_2 \in \text{In}_+^\omega$, such that $\hat{i}_1 = (\mathbf{p}_1, i_1)$, $\hat{i}_2 = (\mathbf{p}_2, i_2)$, and $W(\mathbf{p}_1, \mathbf{p}_2, i_1, i_2, k)$.

Proof. We prove both directions of the equivalence separately. First, we show that statement 1 implies statement 2. For this, let $\hat{i}_1 \in \text{Stdln}_x$, $\hat{i}_2 \in \text{ln}_x^\omega$ and $k \in \mathbb{N}$. We may assume that $\forall j \leq k. \#d_{\text{In}}^{(\kappa_i, \approx_P)}(\hat{i}_1[..j], \hat{i}_2[..j]) \leq b\kappa_i$. Then, we first conclude from the definition of Stdln_x and ln_x , that there exist $\mathbf{p}_1, \mathbf{p}_2 \in \text{Param}^\omega$, $i_1 \in \text{Stdln}_+$ and $i_2 \in \text{ln}_+^\omega$, such that $\hat{i}_1 = (\mathbf{p}_1, i_1)$, $\hat{i}_2 = (\mathbf{p}_2, i_2)$. It remains to show that $W(\mathbf{p}_1, \mathbf{p}_2, i_1, i_2, k)$. With the equivalence of W and V from the assumptions, we may equivalently show that $V(\mathbf{p}_1[0], \mathbf{p}_2[0], i_1, i_2, k)$. Applying statement 1, it suffices to show that i) $\mathbf{p}_1[0] \approx_P \mathbf{p}_2[0]$ and ii) $\forall j \leq k. d_{\text{In}}(i_1[..j], i_2[..j]) \leq \kappa_i$. We continue by case distinction on whether κ_i is ∞ . First, assume $\kappa_i \neq \infty$. Then, by definition of b , $b\kappa_i = \kappa_i$. As $b\kappa_i < \infty$, we know that $\forall j \leq k. \#d_{\text{In}}^{(\kappa_i, \approx_P)}(\hat{i}_1[..j], \hat{i}_2[..j]) < \infty$. Using the trace-pair-equality (cf. Section 2.2) we get that $\forall j \leq k. \#d_{\text{In}}^{(\kappa_i, \approx_P)}((\mathbf{p}_1[..j], i_1[..j]), (\mathbf{p}_2[..j], i_2[..j])) < \infty$. We can infer that the input distance is given by the first case of the definition of $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$: it cannot be the second case, because $\kappa_i \neq \infty$ and cannot be the third case, because the input distance is smaller than ∞ . Hence, we get that $\forall j \leq k. \#d_{\text{In}}^{(\kappa_i, \approx_P)}((\mathbf{p}_1[..j], i_1[..j]), (\mathbf{p}_2[..j], i_2[..j])) = d_{\text{In}}(i_1[..j], i_2[..j])$ and $\mathbf{p}_1[..j][0] \approx_P \mathbf{p}_2[..j][0]$, i.e., $\mathbf{p}_1[0] \approx_P \mathbf{p}_2[0]$, as we were required to show. In the other case in which $\kappa_i = \infty$, precondition ii) is immediately satisfied, because every value in $\mathbb{R}_{\geq 0}$ is less or equal to ∞ . To prove condition i), observe that $b\kappa_i = 0$. Hence, it is the case that $\forall j \leq k. \#d_{\text{In}}^{(\kappa_i, \approx_P)}(\hat{i}_1[..j], \hat{i}_2[..j]) \leq 0$, i.e., $\forall j \leq k. \#d_{\text{In}}^{(\kappa_i, \approx_P)}((\mathbf{p}_1[..j], i_1[..j]), (\mathbf{p}_2[..j], i_2[..j])) \leq 0$. Similar to before, we can infer that in this case the input distance is determined by the second case of $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$. Hence, we also infer that $\forall j \leq k. \mathbf{p}_1[..j][0] \approx_P \mathbf{p}_2[..j][0]$ and hence $\mathbf{p}_1[0] \approx_P \mathbf{p}_2[0]$.

To show the inverse implication, that statement 2 implies statement 1, let $\mathbf{p}_1, \mathbf{p}_2 \in \text{Param}$ with $\mathbf{p}_1 \approx_P \mathbf{p}_2$, $i_1 \in \text{Stdln}_+$, $i_2 \in \text{ln}_+^\omega$ and $k \in \mathbb{N}$. We may assume that $\forall j \leq k. d_{\text{In}}(i_1[..j], i_2[..j]) \leq \kappa_i$ and must show $V(\mathbf{p}_1, \mathbf{p}_2, i_1, i_2, k)$. This is equivalent to $V(\mathbf{p}_1^\omega[0], \mathbf{p}_2^\omega[0], i_1, i_2, k)$, because $\mathbf{p}_1^\omega[0] = \mathbf{p}_1$ and $\mathbf{p}_2^\omega[0] = \mathbf{p}_2$. With the equivalence of W and V from the assumptions, we may equivalently show that $W(\mathbf{p}_1^\omega, \mathbf{p}_2^\omega, i_1, i_2, k)$. Using statement 2, it suffices to show that i) $(\mathbf{p}_1^\omega, i_1) \in \text{Stdln}_x$, ii) $(\mathbf{p}_2^\omega, i_2) \in \text{ln}_x^\omega$, and iii) $\forall j \leq k. \#d_{\text{In}}^{(\kappa_i, \approx_P)}((\mathbf{p}_1^\omega, i_1)[..j], (\mathbf{p}_2^\omega, i_2)[..j]) \leq b\kappa_i$. The first premise follows from the definition of Stdln_x , $\mathbf{p}_1^\omega \in \text{Param}^\omega$ and $i_1 \in \text{Stdln}_+$. Similarly, the second premise follows from the definition of ln_x^ω , $\mathbf{p}_2^\omega \in \text{Param}^\omega$ and $i_2 \in \text{ln}_+^\omega$. To prove the third premise, we must identify which of the three cases of $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$ is responsible for the input distance. For this, first observe that since $\mathbf{p}_1 \approx_P \mathbf{p}_2$ it also holds that $\mathbf{p}_1^\omega[0] \approx_P \mathbf{p}_2^\omega[0]$. We continue depending on whether κ_i is infinity. First, let $\kappa_i \neq \infty$. Then, $b\kappa_i = \kappa_i$ and the first case of $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$ applies, i.e., $\forall j \leq k. \#d_{\text{In}}^{(\kappa_i, \approx_P)}((\mathbf{p}_1^\omega[..j], i_1[..j]), (\mathbf{p}_2^\omega[..j], i_2[..j])) \leq \kappa_i$ if and only if $\forall j \leq k. d_{\text{In}}(i_1[..j], i_2[..j]) \leq \kappa_i$, which holds by assumption. In the

other case that $\kappa_i = \infty$, we get that $b\kappa_i = 0$ and the second case of $\#d_{\text{In}}^{(\kappa_i, \approx_P)}$ applies, i.e., by definition, $\forall j \leq k. \#d_{\text{In}}^{(\kappa_i, \approx_P)}((\mathbf{p}_1^\omega[.j], i_1[.j]), (\mathbf{p}_2^\omega[.j], i_2[.j])) = 0$. We conclude the proof by acknowledging that $0 \leq 0$ is a tautology. \square

With the above lemma, only a single step of reasoning remains for the proof of Proposition 3.52.

Proof of Proposition 3.52. With Lemma 3.53, it suffices to show for all $\mathbf{p}_1, \mathbf{p}_2 \in \text{Param}^\omega$, $i_1 \in \text{StdIn}_+$, $i_2 \in \text{In}_+^\omega$ and index $k \in \mathbb{N}$ that

$$\begin{aligned} \mathcal{H}(d_{\text{Out}})(S_+(\mathbf{p}_1[0])(i_1)[.k], S_+(\mathbf{p}_2[0])(i_2)[.k]) &\leq \kappa_o \text{ if and only if} \\ \mathcal{H}(d_{\text{Out}})(S(\mathbf{p}_1, i_1)[.k], S(\mathbf{p}_2, i_2)[.k]) &\leq \kappa_o. \end{aligned}$$

This follows directly from $S_+ \leftrightarrow S$. \square

Next, we provide for non-parametrised systems the quantifier-based definitions for the lower and upper bound interpretation of robust cleanness.

Definition 3.54. A non-parametrised nondeterministic reactive system S is *l-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$, if $d_{\text{In}}(i[.j], i'[.j]) \leq \kappa_i$ for all $j \leq k$, then for all $o \in S(i)$, there exists $o' \in S(i')$, such that $d_{\text{Out}}(o[.k], o'[.k]) \leq \kappa_o$.

Definition 3.55. A non-parametrised nondeterministic reactive system S is *u-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$, if $d_{\text{In}}(i[.j], i'[.j]) \leq \kappa_i$ for all $j \leq k$, then for all $o' \in S(i')$, there exists $o \in S(i)$, such that $d_{\text{Out}}(o[.k], o'[.k]) \leq \kappa_o$.

We show that l-robust cleanness for parametrised and non-parametrised systems are equally expressive; the proofs follow the same structure as those for robust cleanness.

Proposition 3.56. Let $S : \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a non-parametrised nondeterministic reactive system and let $S_+ : \{\text{def}\} \rightarrow \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be the parametrised nondeterministic reactive system that encodes S , i.e., $S \leftrightarrow S_+$. Further, let $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ and $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be contracts, where $\text{def} \approx_P \text{def}$ and $\text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o$ are identical in both contracts. Then, S is l-robustly clean w.r.t. \mathcal{C} if and only if S_+ is l-robustly clean w.r.t. \mathcal{C}_+ .

Proof. We prove the equivalence by separately proving the two underlying implications. First, let S be l-robustly clean; we must show that S_+ is l-robustly clean. For this, let $\mathbf{p}, \mathbf{p}' \in \{\text{def}\}$ with $\mathbf{p} \approx_P \mathbf{p}'$, i.e., $\mathbf{p} = \mathbf{p}' = \text{def}$, let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. We may assume that $\forall j \leq k. d_{\text{In}}(i[.j], i'[.j]) \leq \kappa_i$ and

must show that $\forall o \in S_+(def)(i). \exists o' \in S_+(def)(i'). d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$. From $S \hookrightarrow S_+$, we know that this is equivalent to showing that $\forall o \in S(i). \exists o' \in S(i'). d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$. With the above assumptions, this follows from l-robust cleanness of S .

For the inverse implication, assume l-robust cleanness of S_+ , to show that S is l-robustly clean. Let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. We may assume that $\forall j \leq k. d_{\text{In}}(i[..j], i'[..j]) \leq \kappa_i$ and need to show that $\forall o \in S(i). \exists o' \in S(i'). d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$. This is equivalent to $\forall o \in S_+(def)(i). \exists o' \in S_+(def)(i'). d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$, since $S \hookrightarrow S_+$. With the all the assumptions above, and that by definition $def \approx_\rho def$, the inequality follows from S_+ 's l-robust cleanness. \square

Proposition 3.57. Let $S_+ : \text{Param} \rightarrow \text{In}_+^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a parametrised nondeterministic reactive system and let $S : \text{In}_x^\omega \rightarrow 2^{(\text{Out}^\omega)}$, with $\text{In}_x = \text{Param} \times \text{In}_+$, be the non-parametrised system that encodes S_+ , i.e., $S_+ \hookrightarrow S$. Further, let $\mathcal{C}_+ = \langle \approx_\rho, \text{StdIn}_+, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ and $\mathcal{C} = \langle \text{StdIn}_x, d'_{\text{In}}, d_{\text{Out}}, \flat\kappa_i, \kappa_o \rangle$ be contracts with $\text{StdIn}_x = \text{Param}^\omega \times \text{StdIn}_+$ and $d'_{\text{In}} = \sharp d_{\text{In}}^{(\kappa_i, \approx_\rho)}$. Then, S_+ is l-robustly clean w.r.t. \mathcal{C}_+ if and only if S is l-robustly clean w.r.t. \mathcal{C} .

Proof. With Lemma 3.53, it suffices to show that for all $p_1, p_2 \in \text{Param}^\omega$, $i_1 \in \text{StdIn}_+$, $i_1 \in \text{In}_+^\omega$ and index $k \in \mathbb{N}$ that

$$\begin{aligned} \forall o_1 \in S_+(p_1[0])(i_1). \exists o_2 \in S_+(p_2[0])(i_2). d_{\text{Out}}(o_1[..k], o_2[..k]) \leq \kappa_o \text{ if and only if} \\ \forall o_1 \in S(p_1, i_1). \exists o_2 \in S(p_2, i_2). d_{\text{Out}}(o_1[..k], o_2[..k]) \leq \kappa_o. \end{aligned}$$

This follows from $S_+ \hookrightarrow S$. \square

A similar proposition holds for u-robust cleanness; we omit the propositions and proofs here.

Unsurprisingly, the quantifier-based and the Hausdorff-based variants of robust cleanness are almost equivalent also for non-parametrised reactive systems, as the following two propositions show.

Proposition 3.58. Let $S : \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a non-parametrised nondeterministic reactive system and $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract for robust cleanness. If S is l-robustly clean w.r.t. \mathcal{C} and S is u-robustly clean w.r.t. \mathcal{C} , then S is robustly clean w.r.t. \mathcal{C} .

Proof. To show robust cleanness for S , let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$ and assume that for all $j \leq k$, $d_{\text{In}}(i[..j], i'[..j]) \leq \kappa_i$. We must show that $\mathcal{H}(d_{\text{Out}}(S(i)[..k], S(i')[..k])) \leq \kappa_o$. After unrolling the definition of the Hausdorff distance, we need to show that the maximum of $\sup_{o \in S(i)[..k]} \inf_{o' \in S(i')[..k]} d_{\text{Out}}(o, o')$ and $\sup_{o' \in S(i')[..k]} \inf_{o \in S(i)[..k]} d_{\text{Out}}(o, o')$ is less or equal than κ_o . Simple

logical operations transform the proof obligation to the conjunction of $\sup_{\mathfrak{o} \in \mathcal{S}(i)} \inf_{\mathfrak{o}' \in \mathcal{S}(i')} d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) \leq \kappa_{\mathfrak{o}}$ and $\sup_{\mathfrak{o}' \in \mathcal{S}(i')} \inf_{\mathfrak{o} \in \mathcal{S}(i)} d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) \leq \kappa_{\mathfrak{o}}$. With Lemma 3.12, it suffices to show that the conjunction of $\forall \mathfrak{o} \in \mathcal{S}(i). \exists \mathfrak{o}' \in \mathcal{S}(i'). d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) \leq \kappa_{\mathfrak{o}}$ and $\forall \mathfrak{o}' \in \mathcal{S}(i'). \exists \mathfrak{o} \in \mathcal{S}(i). d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) \leq \kappa_{\mathfrak{o}}$ holds. The first conjunct follows immediately from l-robust cleanness of \mathcal{S} for i and i' , and similarly, the second conjunct follows from u-robust cleanness of \mathcal{S} . \square

Proposition 3.59. Let $\mathcal{S} : \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a non-parametrised nondeterministic reactive system and $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract for robust cleanness. For every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and index $k \in \mathbb{N}$, if $d_{\text{In}}(i[\dots j], i'[\dots j]) \leq \kappa_i$ for all $j \leq k$, then $\mathcal{H}(d_{\text{Out}})(\mathcal{S}(i)[\dots k], \mathcal{S}(i')[\dots k]) < \kappa_o$ implies that

1. for all $\mathfrak{o} \in \mathcal{S}(i)$, there exists $\mathfrak{o}' \in \mathcal{S}(i')$ such that $d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < \kappa_o$ and
2. for all $\mathfrak{o}' \in \mathcal{S}(i')$, there exists $\mathfrak{o} \in \mathcal{S}(i)$ such that $d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < \kappa_o$.

Proof. Let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. Assume that $d_{\text{In}}(i[\dots j], i'[\dots j]) \leq \kappa_i$ for all $j \leq k$, and that $\mathcal{H}(d_{\text{Out}})(\mathcal{S}(i)[\dots k], \mathcal{S}(i')[\dots k]) < \kappa_o$. We must show that $\forall \mathfrak{o} \in \mathcal{S}(i). \exists \mathfrak{o}' \in \mathcal{S}(i'). d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < \kappa_o$ and $\forall \mathfrak{o}' \in \mathcal{S}(i'). \exists \mathfrak{o} \in \mathcal{S}(i). d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < \kappa_o$. Using Lemma 3.16, we may equivalently show that the conjunction of $\sup_{\mathfrak{o} \in \mathcal{S}(i)} \inf_{\mathfrak{o}' \in \mathcal{S}(i')} d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < \kappa_o$ and $\sup_{\mathfrak{o}' \in \mathcal{S}(i')} \inf_{\mathfrak{o} \in \mathcal{S}(i)} d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < \kappa_o$ holds. With simple logical operations, this conjunction can be transformed into the equivalent proposition constraining the maximum of $\sup_{\mathfrak{o} \in \mathcal{S}(i)[\dots k]} \inf_{\mathfrak{o}' \in \mathcal{S}(i')[\dots k]} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}')$ and $\sup_{\mathfrak{o}' \in \mathcal{S}(i')[\dots k]} \inf_{\mathfrak{o} \in \mathcal{S}(i)[\dots k]} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}')$ to be smaller than κ_o . This is exactly what the assumed Hausdorff distance defines, hence, the proposition holds. \square

The flavours of definitions of robust cleanness presented in this section – be it for parametrised or non-parametrised systems, Hausdorff-based or quantifier-based – develop further the definitions from Section 3.1 for sequential programs to definitions for reactive systems. While the structure of this section is very similar to that in Section 3.1, the definitions have one particular noteworthy distinguishing peculiarity: the (infinitely long) input and output sequences of the system are not handled as opaque values, but the reasoning is based on a reasoning about finite prefixes of these sequences. This gives the definitions their safety character – once robust cleanness is violated at some index k , this violation cannot be recovered at a later index. For inputs, this concept has to be taken one step further. Once two input prefixes cross the κ_i threshold, the cleanness constraint on the corresponding outputs is dropped, and, in fact, it is dropped also for all continuations of these prefixes.

3.2.3 Func-cleanness

As we did for sequential programs, we will now further generalise robust cleanness to func-cleanness. Behind func-cleanness is a function $f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ that, given a distance between inputs, provides a threshold for the distance of the outputs belonging to these inputs. For reactive systems this remains unchanged.

Definition 3.60. A parametrised nondeterministic reactive system S_+ is *func-clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for all pairs of parameters $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$, $\mathcal{H}(d_{\text{Out}})(S_+(\mathfrak{p})(i)[..k], S_+(\mathfrak{p}')(i')[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

Like for robust cleanness, the definition of func-cleanness also considers distances on prefixes to ensure that major differences in late inputs do not impact differences of early outputs, capturing also the safety nature of the property.

To work towards a quantifier-based variant of func-cleanness, the following definitions spell out the lower and upper bound interpretations related to func-cleanness.

Definition 3.61. A parametrised nondeterministic reactive system S_+ is *l-func-clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for all pairs of parameters $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$, for all $o \in S_+(\mathfrak{p})(i)$, there exists $o' \in S_+(\mathfrak{p}')(i')$, such that $d_{\text{Out}}(o[..k], o'[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

Definition 3.62. A parametrised nondeterministic reactive system S_+ is *u-func-clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for all pairs of parameters $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$, for all $o' \in S_+(\mathfrak{p}')(i')$, there exists $o \in S_+(\mathfrak{p})(i)$, such that $d_{\text{Out}}(o[..k], o'[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

We define the quantifier-based variant of func-cleanness as the conjunction of l-func-cleanness and u-func-cleanness and show in the following propositions its almost-equivalence to the Hausdorff-based definition.

Proposition 3.63. Let $S_+ : \text{Param} \rightarrow \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a parametrised nondeterministic reactive system and $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract for func-cleanness. If S_+ is l-func-clean w.r.t. \mathcal{C} and S_+ is u-func-clean w.r.t. \mathcal{C} , then S_+ is func-clean w.r.t. \mathcal{C} .

Proof. Let $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. In the following, let $\kappa = f(d_{\text{In}}(i[..k], i'[..k]))$. Then, it suffices to show that $\forall o \in S_+(\mathfrak{p})(i)$. $\exists o' \in S_+(\mathfrak{p}')(i')$. $d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa$ (l-func-cleanness) and $\forall o' \in S_+(\mathfrak{p}')(i')$. $\exists o \in S_+(\mathfrak{p})(i)$. $d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa$ (u-func-cleanness) implies that

$\mathcal{H}(d_{\text{Out}})(S_+(\mathbf{p})(i)[..k], S_+(\mathbf{p}')(i')[..k]) \leq \kappa$. We use Lemma 3.12 to get from l-func-cleanness that $\sup_{\mathbf{o} \in S_+(\mathbf{p})(i)} \inf_{\mathbf{o}' \in S_+(\mathbf{p}')(i')} d_{\text{Out}}(\mathbf{o}[..k], \mathbf{o}'[..k]) \leq \kappa$, and from u-func-cleanness that $\sup_{\mathbf{o}' \in S_+(\mathbf{p}')(i')} \inf_{\mathbf{o} \in S_+(\mathbf{p})(i)} d_{\text{Out}}(\mathbf{o}[..k], \mathbf{o}'[..k]) \leq \kappa$. It is easy to see that it is equivalent to enforce the maximum of $\sup_{\mathbf{o} \in S_+(\mathbf{p})(i)[..k]} \inf_{\mathbf{o}' \in S_+(\mathbf{p}')(i')[..k]} d_{\text{Out}}(\mathbf{o}, \mathbf{o}')$ and $\sup_{\mathbf{o}' \in S_+(\mathbf{p}')(i')[..k]} \inf_{\mathbf{o} \in S_+(\mathbf{p})(i)[..k]} d_{\text{Out}}(\mathbf{o}, \mathbf{o}')$ to be less or equal to κ . Hence, $\mathcal{H}(d_{\text{Out}})(S_+(\mathbf{p})(i)[..k], S_+(\mathbf{p}')(i')[..k]) \leq \kappa = f(d_{\text{In}}(i[..k], i'[..k]))$. \square

Proposition 3.64. Let $S_+ : \text{Param} \rightarrow \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a parametrised nondeterministic reactive system and $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract for func-cleanness. Then, for all pairs of parameters $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$, $\mathcal{H}(d_{\text{Out}})(S_+(\mathbf{p})(i)[..k], S_+(\mathbf{p}')(i')[..k]) < f(d_{\text{In}}(i[..k], i'[..k]))$ implies that

1. for all $\mathbf{o} \in S_+(\mathbf{p})(i)$, there exists $\mathbf{o}' \in S_+(\mathbf{p}')(i')$, such that $d_{\text{Out}}(\mathbf{o}[..k], \mathbf{o}'[..k]) < f(d_{\text{In}}(i[..k], i'[..k]))$ and
2. for all $\mathbf{o}' \in S_+(\mathbf{p}')(i')$, there exists $\mathbf{o} \in S_+(\mathbf{p})(i)$, such that $d_{\text{Out}}(\mathbf{o}[..k], \mathbf{o}'[..k]) < f(d_{\text{In}}(i[..k], i'[..k]))$.

Proof. Let $\mathbf{p}, \mathbf{p}' \in \text{Param}$ with $\mathbf{p} \approx_P \mathbf{p}'$, $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. In the following, let $\kappa = f(d_{\text{In}}(i[..k], i'[..k]))$. Then, it suffices to show that the inequality $\mathcal{H}(d_{\text{Out}})(S_+(\mathbf{p})(i)[..k], S_+(\mathbf{p}')(i')[..k]) < \kappa$ implies that $\forall \mathbf{o} \in S_+(\mathbf{p})(i). \exists \mathbf{o}' \in S_+(\mathbf{p}')(i'). d_{\text{Out}}(\mathbf{o}[..k], \mathbf{o}'[..k]) < \kappa$ (l-func-cleanness) and $\forall \mathbf{o}' \in S_+(\mathbf{p}')(i'). \exists \mathbf{o} \in S_+(\mathbf{p})(i). d_{\text{Out}}(\mathbf{o}[..k], \mathbf{o}'[..k]) < \kappa$ (u-func-cleanness). By definition of the Hausdorff distance, the maximum of $\sup_{\mathbf{o} \in S_+(\mathbf{p})(i)[..k]} \inf_{\mathbf{o}' \in S_+(\mathbf{p}')(i')[..k]} d_{\text{Out}}(\mathbf{o}, \mathbf{o}')$ and $\sup_{\mathbf{o}' \in S_+(\mathbf{p}')(i')[..k]} \inf_{\mathbf{o} \in S_+(\mathbf{p})(i)[..k]} d_{\text{Out}}(\mathbf{o}, \mathbf{o}')$ is smaller than κ . This is equivalent to the conjunction that $\sup_{\mathbf{o} \in S_+(\mathbf{p})(i)} \inf_{\mathbf{o}' \in S_+(\mathbf{p}')(i')} d_{\text{Out}}(\mathbf{o}[..k], \mathbf{o}'[..k]) < \kappa$ and $\sup_{\mathbf{o}' \in S_+(\mathbf{p}')(i')} \inf_{\mathbf{o} \in S_+(\mathbf{p})(i)} d_{\text{Out}}(\mathbf{o}[..k], \mathbf{o}'[..k]) < \kappa$. From the former, we get l-func-cleanness with Lemma 3.16, and from the latter and Lemma 3.16 we obtain u-func-cleanness. \square

For reactive systems that can better be modelled without parameters, we define func-cleanness for non-parametrised systems below.

Definition 3.65. A non-parametrised nondeterministic reactive system S is *func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$, $\mathcal{H}(d_{\text{Out}})(S(i)[..k], S(i')[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

Example 3.66. The emissions control systems from Figures 3.6 and 3.7 are non-parametrised systems. Hence, func-cleanness from Definition 3.65 can be applied to these examples. For a contract, we can take the same StdIn , d_{In} and d_{Out} as proposed in Example 3.50 for the robust cleanness contract. For f in the contract, one suitable choice is $f(x) = x/2 + 0.3$. Notice that using the same

function f as in Example 3.39 for sequential programs would not be a good choice here, because of the nondeterminism in the output in the reactive variant of the emission control system. As expected, the system in Figure 3.6 is func-clean w.r.t. this contract; the system in Figure 3.7 violates func-cleanness.

In the following two propositions, we will prove that func-cleanness for parametrised systems and non-parametrised ones are equally expressive. We start with the easier direction showing how non-parametrised func-cleanness maps to parametrised func-cleanness.

Proposition 3.67. Let $S : \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a non-parametrised nondeterministic reactive system and let $S_+ : \{def\} \rightarrow \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be the parametrised nondeterministic reactive system that encodes S , i.e., $S \hookrightarrow S_+$. Further, let $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ and $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ be contracts, where $def \approx_P def$ and $\text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f$ are identical in both contracts. Then, S is func-clean w.r.t. \mathcal{C} if and only if S_+ is func-clean w.r.t. \mathcal{C}_+ .

Proof. We first assume that S is func-clean w.r.t. \mathcal{C} to show that S_+ is func-clean w.r.t. \mathcal{C}_+ . For this, let $\mathbf{p} = \mathbf{p}' = def$, $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. With these assumptions, we get from func-cleanness of S that $\mathcal{H}(d_{\text{Out}})(S(i)[..k], S(i')[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$. By definition of S_+ as given by $S \hookrightarrow S_+$, this is equivalent to the inequality $\mathcal{H}(d_{\text{Out}})(S_+(def)(i)[..k], S_+(def)(i')[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$, so S_+ is robustly clean w.r.t. \mathcal{C}_+ .

Conversely, we assume that S_+ is func-clean w.r.t. \mathcal{C}_+ to show that S is func-clean w.r.t. \mathcal{C} . Let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. By definition, $def \approx_P def$. We get from func-cleanness of S_+ that $\mathcal{H}(d_{\text{Out}})(S_+(def)(i)[..k], S_+(def)(i')[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$. By definition of S_+ as given by $S \hookrightarrow S_+$, this is equivalent to $\mathcal{H}(d_{\text{Out}})(S(i)[..k], S(i')[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$, which concludes the proof. \square

As for sequential programs, the inverse encoding step requires a more sophisticated contract for the cleanness of the non-parametrised system. Below is the input distance function $\sharp d_{\text{In}}^{(f, \approx_P)}$, which is very similar to its counterpart for sequential programs in Equation (3.2). The only difference between the two definitions is that for reactive systems we must access the first symbol from the parameter traces \mathbf{p} and \mathbf{p}' . We remark that this change keeps the reflexivity and symmetry of $\sharp d_{\text{In}}^{(f, \approx_P)}$ intact. Thus, $\sharp d_{\text{In}}^{(f, \approx_P)}$ is still a valid distance function.

$$\sharp d_{\text{In}}^{(f, \approx_P)}((\mathbf{p}, i), (\mathbf{p}', i')) := \begin{cases} 0 & \text{if } \mathbf{p}[0] \approx_P \mathbf{p}'[0] \text{ and } i = i' \\ f(d_{\text{In}}(i, i')) + 1 & \text{if } \mathbf{p}[0] \approx_P \mathbf{p}'[0] \text{ and } i \neq i' \\ \infty & \text{otherwise,} \end{cases} \quad (3.4)$$

Leaving the definition of $\flat f$ unchanged, we can now construct func-cleanness contracts for non-parametrised systems that encode func-cleanness contracts for parametrised systems.

Proposition 3.68. Let $S_+ : \text{Param} \rightarrow \text{In}_+^\omega \rightarrow 2^{\text{Out}^\omega}$ be a parametrised system and $S : \text{In}_\times^\omega \rightarrow 2^{\text{Out}^\omega}$, with $\text{In}_\times = \text{Param} \times \text{In}_+$, the non-parametrised system such that $S_+ \hookrightarrow S$. Further, let $\mathcal{C}_+ = \langle \approx_P, \text{StdIn}_+, d_{\text{In}}, d_{\text{Out}}, f \rangle$ be a contract for parametrised systems and $\mathcal{C} = \langle \text{StdIn}_\times, d'_{\text{In}}, d_{\text{Out}}, \flat f \rangle$ a contract for non-parametrised systems, where $\text{StdIn}_\times = \text{Param}^\omega \times^\omega \text{StdIn}_+$ and $d'_{\text{In}} = \sharp d_{\text{In}}^{(f, \approx_P)}$. Then, S_+ is func-clean w.r.t. \mathcal{C}_+ if and only if S is func-clean w.r.t. \mathcal{C} .

Proof. We first show that if S_+ is func-clean w.r.t. \mathcal{C}_+ , then S is func-clean w.r.t. \mathcal{C} . Hence, let $(p_1, i_1) \in \text{StdIn}_\times$, $(p_2, i_2) \in \text{In}_\times^\omega$ and $k \in \mathbb{N}$. From the definitions of StdIn_\times and In_\times we get that $p_1, p_2 \in \text{Param}^\omega$, $i_1 \in \text{StdIn}_+$ and $i_2 \in \text{In}_+^\omega$. We must show that $\mathcal{H}(d_{\text{Out}})(S(p_1, i_1)[..k], S(p_2, i_2)[..k]) \leq \flat f(d'_{\text{In}}((p_1, i_1)[..k], (p_2, i_2)[..k]))$. Applying the definition of S as given by $S_+ \hookrightarrow S$, it suffices to show that $\mathcal{H}(d_{\text{Out}})(S_+(p_1[0])(i_1)[..k], S_+(p_2[0])(i_2)[..k]) \leq \flat f(d'_{\text{In}}((p_1, i_1)[..k], (p_2, i_2)[..k]))$. We distinguish three cases, analogue to the cases of the definition of $\sharp d_{\text{In}}^{(f, \approx_P)}$:

- Case $p_1[..k][0] \approx_P p_2[..k][0]$ and $i_1[..k] = i_2[..k]$: Then, $d_{\text{In}}(i_1[..k], i_2[..k]) = 0$ and, by definition of $\sharp d_{\text{In}}^{(f, \approx_P)}$, $d'_{\text{In}}((p_1, i_1)[..k], (p_2, i_2)[..k]) = 0$. Since by definition $\flat f(0) = f(0)$, $\flat f(d'_{\text{In}}((p_1, i_1)[..k], (p_2, i_2)[..k])) = f(d_{\text{In}}(i_1[..k], i_2[..k]))$ and it suffices to show that $\mathcal{H}(d_{\text{Out}})(S_+(p_1[0])(i_1)[..k], S_+(p_2[0])(i_2)[..k]) \leq f(d_{\text{In}}(i_1[..k], i_2[..k]))$. This follows from func-cleanness of S_+ .
- Case $p_1[..k][0] \approx_P p_2[..k][0]$ and $i_1[..k] \neq i_2[..k]$: Then, by definition of $\sharp d_{\text{In}}^{(f, \approx_P)}$, $d'_{\text{In}}((p_1, i_1)[..k], (p_2, i_2)[..k]) = f(d_{\text{In}}(i_1[..k], i_2[..k])) + 1$, which is always larger than 0. Since d'_{In} is always larger than 0 in this case, we get from the definition of $\flat f$ that $\flat f(d'_{\text{In}}((p_1, i_1)[..k], (p_2, i_2)[..k])) = f(d_{\text{In}}(i_1[..k], i_2[..k])) + 1 - 1$, which is equal to $f(d_{\text{In}}(i_1[..k], i_2[..k]))$. Hence, it suffices to show that $\mathcal{H}(d_{\text{Out}})(S_+(p_1[0])(i_1)[..k], S_+(p_2[0])(i_2)[..k])$ is smaller or equal to $f(d_{\text{In}}(i_1[..k], i_2[..k]))$, which follows from func-cleanness of S_+ .
- Case $p_1[..k][0] \not\approx_P p_2[..k][0]$: The, we get from the definition of $\sharp d_{\text{In}}^{(f, \approx_P)}$ that $d'_{\text{In}}((p_1, i_1)[..k], (p_2, i_2)[..k]) = \infty$ and, by definition of $\flat f$ and since $\infty > 0$, $\flat f(d'_{\text{In}}((p_1, i_1)[..k], (p_2, i_2)[..k])) = \infty$. Hence, it suffices to show that $\mathcal{H}(d_{\text{Out}})(S_+(p_1[0])(i_1)[..k], S_+(p_2[0])(i_2)[..k]) \leq \infty$, which is always the case.

It remains to show that if S is func-clean w.r.t. \mathcal{C} , then S_+ is func-clean w.r.t. \mathcal{C}_+ . Let $p_1, p_2 \in \text{Param}$ with $p_1 \approx_P p_2$, $i_1 \in \text{StdIn}_+$, $i_2 \in \text{In}_+^\omega$ and $k \in \mathbb{N}$. Then, we must show that $\mathcal{H}(d_{\text{Out}})(S_+(p_1)(i_1)[..k], S_+(p_2)(i_2)[..k]) \leq$

$f(d_{\text{In}}(i_1[..k], i_2[..k]))$. Notice that $p_1^\omega[0] = p_1$ and $p_2^\omega[0] = p_2$, so, with the definition of S as given by the encoding $S_+ \hookrightarrow S$, it suffices to show that $\mathcal{H}(d_{\text{Out}}(S(p_1^\omega, i_1)[..k], S(p_2^\omega, i_2)[..k])) \leq f(d_{\text{In}}(i_1[..k], i_2[..k]))$. From $p_1 \approx_P p_2$ it follows that $p_1^\omega[0] \approx_P p_2^\omega[0]$. We distinguish on whether $i_1[..k] = i_2[..k]$. If $i_1[..k]$ and $i_2[..k]$ are equal, then, by definition of d_{In} , $d(d_{\text{In}}(i_1[..k], i_2[..k])) = 0$, and, by definition of $\sharp d_{\text{In}}^{(f, \approx_P)}$, $d'_{\text{In}}((p_1^\omega, i_1)[..k], (p_2^\omega, i_2)[..k]) = 0$. Moreover, using the definition of $\flat f$, we get that $f(d_{\text{In}}(i_1[..k], i_2[..k])) = f(0) = \flat f(0) = \flat f(d'_{\text{In}}((p_1^\omega, i_1)[..k], (p_2^\omega, i_2)[..k]))$. In the other case that $i_1[..k] \neq i_2[..k]$, we similarly use the definitions of $\sharp d_{\text{In}}^{(f, \approx_P)}$ and $\flat f$ to conclude that $f(d_{\text{In}}(i_1[..k], i_2[..k])) = f(d_{\text{In}}(i_1[..k], i_2[..k])) + 1 - 1 = d'_{\text{In}}((p_1^\omega, i_1)[..k], (p_2^\omega, i_2)[..k]) - 1$, which is equal to $\flat f(d'_{\text{In}}((p_1^\omega, i_1)[..k], (p_2^\omega, i_2)[..k]))$. Hence, in both cases, it suffices to show that $\mathcal{H}(d_{\text{Out}}(S(p_1^\omega, i_1)[..k], S(p_2^\omega, i_2)[..k])) \leq \flat f(d'_{\text{In}}((p_1^\omega, i_1)[..k], (p_2^\omega, i_2)[..k]))$, which follows from func-cleanness of S for standard input (p_1^ω, i_1) and input (p_2^ω, i_2) . \square

Below we provide func-cleanness-related quantifier-based definitions for lower and upper bound constraints on non-standard behaviour.

Definition 3.69. A non-parametrised nondeterministic reactive system S is *l-func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and index $k \in \mathbb{N}$, it holds that for all $o \in S(i)$, there exists $o' \in S(i')$, such that $d_{\text{Out}}(o[..k], o'[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

Definition 3.70. A non-parametrised nondeterministic reactive system S is *u-func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and index $k \in \mathbb{N}$, it holds that for all $o' \in S(i')$, there exists $o \in S(i)$, such that $d_{\text{Out}}(o[..k], o'[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

We remark that l-func-cleanness and u-func-cleanness for non-parametrised systems is as expressive as their counterparts for parametrised systems. This can be shown formally in a way similar to Propositions 3.67 and 3.68. We omit the propositions and proofs.

As expected, composing the above two definitions into a conjunction yields a quantifier-based formulation of func-cleanness that is almost equivalent to the Hausdorff-based one, as shown in the following two propositions.

Proposition 3.71. Let $S : \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a non-parametrised nondeterministic reactive system and $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract for func-cleanness. If S is l-func-clean w.r.t. \mathcal{C} and S is u-func-clean w.r.t. \mathcal{C} , then S is func-clean w.r.t. \mathcal{C} .

Proof. Let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. Also, let $\kappa = f(d_{\text{In}}(i[..k], i'[..k]))$. Then, it suffices to show that $\forall o \in S(i). \exists o' \in S(i'). d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa$

(l-func-cleanness) and $\forall \mathfrak{o}' \in \mathcal{S}(i')$. $\exists \mathfrak{o} \in \mathcal{S}(i)$. $d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) \leq \kappa$ (u-func-cleanness) implies that $\mathcal{H}(d_{\text{Out}})(\mathcal{S}(i)[\dots k], \mathcal{S}(i')[\dots k]) \leq \kappa$. With Lemma 3.12 we get from l-func-cleanness that $\sup_{\mathfrak{o} \in \mathcal{S}(i)} \inf_{\mathfrak{o}' \in \mathcal{S}(i')} d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) \leq \kappa$, and from u-func-cleanness that $\sup_{\mathfrak{o}' \in \mathcal{S}(i')} \inf_{\mathfrak{o} \in \mathcal{S}(i)} d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) \leq \kappa$. It is easy to see that this is equivalent to the maximum of $\sup_{\mathfrak{o} \in \mathcal{S}(i)[\dots k]} \inf_{\mathfrak{o}' \in \mathcal{S}(i')[\dots k]} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}')$ and $\sup_{\mathfrak{o}' \in \mathcal{S}(i')[\dots k]} \inf_{\mathfrak{o} \in \mathcal{S}(i)[\dots k]} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}')$ being less or equal to κ . Hence, we can conclude that $\mathcal{H}(d_{\text{Out}})(\mathcal{S}(i)[\dots k], \mathcal{S}(i')[\dots k]) \leq \kappa = f(d_{\text{In}}(i[\dots k], i'[\dots k]))$. \square

Proposition 3.72. Let $\mathcal{S} : \text{In}^\omega \rightarrow \mathcal{2}^{\text{Out}^\omega}$ be a non-parametrised nondeterministic reactive system and $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract for func-cleanness. Then, for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and index $k \in \mathbb{N}$, $\mathcal{H}(d_{\text{Out}})(\mathcal{S}(i)[\dots k], \mathcal{S}(i')[\dots k]) < f(d_{\text{In}}(i[\dots k], i'[\dots k]))$ implies that

1. for all $\mathfrak{o} \in \mathcal{S}(i)$, there exists $\mathfrak{o}' \in \mathcal{S}(i')$ such that $d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < f(d_{\text{In}}(i[\dots k], i'[\dots k]))$ and
2. for all $\mathfrak{o}' \in \mathcal{S}(i')$, there exists $\mathfrak{o} \in \mathcal{S}(i)$ such that $d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < f(d_{\text{In}}(i[\dots k], i'[\dots k]))$.

Proof. Let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$ and $k \in \mathbb{N}$. Also, let $\kappa = f(d_{\text{In}}(i[\dots k], i'[\dots k]))$. Then, it suffices to show that $\mathcal{H}(d_{\text{Out}})(\mathcal{S}(i)[\dots k], \mathcal{S}(i')[\dots k]) < \kappa$ implies that $\forall \mathfrak{o} \in \mathcal{S}(i)$. $\exists \mathfrak{o}' \in \mathcal{S}(i')$. $d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < \kappa$ (l-func-cleanness) and $\forall \mathfrak{o}' \in \mathcal{S}(i')$. $\exists \mathfrak{o} \in \mathcal{S}(i)$. $d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) < \kappa$ (u-func-cleanness). From the definition of the Hausdorff distance, we get that the maximum of $\sup_{\mathfrak{o} \in \mathcal{S}(i)[\dots k]} \inf_{\mathfrak{o}' \in \mathcal{S}(i')[\dots k]} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}')$ and $\sup_{\mathfrak{o}' \in \mathcal{S}(i')[\dots k]} \inf_{\mathfrak{o} \in \mathcal{S}(i)[\dots k]} d_{\text{Out}}(\mathfrak{o}, \mathfrak{o}')$ is smaller than κ . This is equivalent to the conjunction of the inequalities $\sup_{\mathfrak{o} \in \mathcal{S}(i)} \inf_{\mathfrak{o}' \in \mathcal{S}(i')} d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) \leq \kappa$ and $\sup_{\mathfrak{o}' \in \mathcal{S}(i')} \inf_{\mathfrak{o} \in \mathcal{S}(i)} d_{\text{Out}}(\mathfrak{o}[\dots k], \mathfrak{o}'[\dots k]) \leq \kappa$. From the former, we get l-func-cleanness with Lemma 3.16, and from the latter and Lemma 3.16 we obtain u-func-cleanness. \square

In Section 3.1.3 about func-cleanness for sequential programs we remarked that func-cleanness is strictly more general than robust cleanness. We substantiated this claim by providing a function f (parametrised by κ_i and κ_o) that, with unchanged \approx_P , StdIn , d_{In} and d_{Out} , forms a contract that implements robust cleanness in the framework of func-cleanness. With respect to reactive systems, it is still the case that func-cleanness is strictly more general than robust cleanness. To model a contract for func-cleanness that proves this, it is, however, necessary to replace the input distance function by a function that, instead of

producing actual distance values, encodes “messages” for the function f :

$$d_{\text{in}}^{\text{new}}(i[..k], i'[..k]) = \begin{cases} 0 & \text{if } i[..k] = i'[..k] \\ 1 & \text{if } i \in \text{StdIn} \text{ or } i' \in \text{StdIn}, i[..k] \neq i'[..k] \\ & \text{and } d_{\text{in}}(i[..j], i'[..j]) \leq \kappa_i \text{ for all } 0 \leq j \leq k \\ 2 & \text{otherwise} \end{cases}$$

Roughly speaking, if two input prefixes passed to this new input distance function are supposed to be considered for an output check, then the result is zero or one. Otherwise, the result is two. Hence, all the function f must do is to return κ_o if $x \leq 1$ and to return ∞ (to effectively disable the output check) otherwise.

3.2.4 Past-Forgetful Distance Functions & Trace Integrity

The distance functions used for robust cleanness and func-cleanness above define distances between finite prefixes of traces. In some situations it is preferable to use distance functions that do only consider the distance between the last symbols of two traces instead of considering the full traces. We call this property *past-forgetful*. A distance function $d : X^* \times X^* \rightarrow \overline{\mathbb{R}}_{\geq 0}$ is past-forgetful if and only if for every non-empty traces $x_1, x_2 \in X^*$ it holds that $d(x_1, x_2) = d(\text{last}(x_1), \text{last}(x_2))$. When such distance functions are used with the cleanness definitions proposed in Section 3.2 this opens an opportunity to effectively circumvent the restrictions imposed by these definitions.

Example 3.73. Consider a reactive system $S : \mathbb{R}^\omega \rightarrow 2^{(\mathbb{R}^\omega)}$ and two inputs i_1 and i_2 from \mathbb{R}^ω . Assume that $S(i_1)$ produces two outputs $o_1 = 1\ 100\ 1\ 100\ 1\ 100\ \dots$ and $o'_1 = 100\ 1\ 100\ 1\ 100\ 1\ \dots$. and that $S(i_2)$ produces three outputs $o_2 = 1\ 1\ 100\ 1\ 1\ 100\ 1\ 1\ 100\ \dots$, $o'_2 = 1\ 100\ 1\ 1\ 100\ 1\ \dots$ and $o''_2 = 100\ 1\ 1\ 100\ 1\ 1\ \dots$. That is, the outputs for i_1 are all possible traces for which the output symbols 1 and 100 alternate in every step, and the outputs to i_2 are all possibilities to repeatedly have twice the symbol 1 followed by one symbol 100. Consider a contract with $\text{StdIn} = \{i_1\}$, past-forgetful output distance function $d_{\text{Out}}(t_1, t_2) = |\text{last}(t_1) - \text{last}(t_2)|$ and $\kappa_o = 0$. Further assume that d_{In} and κ_i are defined such that $d_{\text{In}}(i_1, i_2) \leq \kappa_i$. Then, for inputs i_1 and i_2 S is l-robustly clean: We have to show for some index $k \in \mathbb{N}$ and non-standard output $o \in S(i_2)$ that there exists some $o' \in S(i_1)$ such that $|o[k] - o'[k]| = 0$. Assume $o = o_2$ (the argument is analogue for o'_2 and o''_2). Depending on the concrete k , $o_2[k]$ is either 1 or 100. Observe that $o_1[k] = 1 \vee o'_1[k] = 1$ and $o_1[k] = 100 \vee o'_1[k] = 100$, i.e., exactly one of the two outputs in $S(i_1)$ exhibits 1 and exactly one exhibits 100 at position k . Thus, $|o_2[k] - o'[k]| = 0$ holds for either $o' = o_1$ or $o' = o'_1$. More precisely, l-robust cleanness can be satisfied for S , i_1 , i_2 and o_2 by choosing alternately o_1

and \mathfrak{o}'_1 for \mathfrak{o}' . However, there does not exist a single output \mathfrak{o}' in $\mathbb{S}(i_1)$ such that $|\mathfrak{o}[k] - \mathfrak{o}'[k]| = 0$ holds at all positions k .

The above example demonstrates that l-robust cleanness does not have a characteristic that we call *trace integrity*. Intuitively, from a clean program we expect that for every $\mathfrak{o} \in \mathbb{S}(i)$ there exists an output $\mathfrak{o}' \in \mathbb{S}(i')$ such that at every position k , the distance between \mathfrak{o} and \mathfrak{o}' is at most $\kappa_{\mathfrak{o}}$. Example 3.73 shows an extreme case in which trace integrity is violated. This problem occurs if the distance functions are past-forgetful. For history-aware distance functions it can sometimes be circumvented. Definition 3.74 proposes a variant of l-robust cleanness (Definition 3.45) that satisfies trace integrity (i.e., which is trace integral).

Definition 3.74. A parametrised nondeterministic reactive system \mathbb{S}_+ is *trace integral l-robustly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every pair of parameters $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$, and output $\mathfrak{o} \in \mathbb{S}_+(\mathfrak{p})(i)$, there exists $\mathfrak{o}' \in \mathbb{S}_+(\mathfrak{p}')(i')$, such that for every index $k \in \mathbb{N}$, if $d_{\text{In}}(i[.j], i'[.j]) \leq \kappa_i$ for all $j \leq k$, then $d_{\text{Out}}(\mathfrak{o}[.k], \mathfrak{o}'[.k]) \leq \kappa_o$.

The change to Definition 3.45 is subtle: effectively, we switched the order of the $\forall \mathfrak{o} \in \mathbb{P}_+(\mathfrak{p})(i)$, $\exists \mathfrak{o}' \in \mathbb{P}_+(\mathfrak{p}')(i')$ and the $\forall k \in \mathbb{N}$ quantifications.

The problem depicted in Example 3.73 for l-robust cleanness can easily be adapted to show that u-robust cleanness suffers from the same problem. Hence, Definition 3.75 shows a trace integral variant of u-robust cleanness (as defined in Definition 3.46).

Definition 3.75. A parametrised nondeterministic reactive system \mathbb{S}_+ is *trace integral u-robustly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every pair of parameters $\mathfrak{p}, \mathfrak{p}' \in \text{Param}$ with $\mathfrak{p} \approx_P \mathfrak{p}'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$, and output $\mathfrak{o}' \in \mathbb{S}_+(\mathfrak{p}')(i')$, there exists $\mathfrak{o} \in \mathbb{S}_+(\mathfrak{p})(i)$, such that for every index $k \in \mathbb{N}$, if $d_{\text{In}}(i[.j], i'[.j]) \leq \kappa_i$ for all $j \leq k$, then $d_{\text{Out}}(\mathfrak{o}[.k], \mathfrak{o}'[.k]) \leq \kappa_o$.

Despite the potential lack of trace integrity, we favour robust cleanness as defined in Definitions 3.44 and 3.49 over their trace integral variants, because the latter can not be expressed using the Hausdorff distance. Consequently, we consider the conjunction of Definitions 3.74 and 3.75 as the default (and only) definition for trace integral robust cleanness.

Definition 3.76. A parametrised nondeterministic reactive system \mathbb{S}_+ is *trace integral robustly clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if \mathbb{S}_+ is trace integral l-robustly clean w.r.t. \mathcal{C} and \mathbb{S}_+ is trace integral u-robustly clean w.r.t. \mathcal{C} .

For completeness, we also provide the the non-parametrised variants of the trace integral definitions.

Definition 3.77. A non-parametrised nondeterministic reactive system S is *trace integral l-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$, and output $o \in S(i)$, there exists $o' \in S(i')$, such that for every index $k \in \mathbb{N}$, if $d_{\text{In}}(i[..j], i'[..j]) \leq \kappa_i$ for all $j \leq k$, then $d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$.

Definition 3.78. A non-parametrised nondeterministic reactive system S is *trace integral u-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$, and output $o' \in S(i')$, there exists $o \in S(i)$, such that for every index $k \in \mathbb{N}$, if $d_{\text{In}}(i[..j], i'[..j]) \leq \kappa_i$ for all $j \leq k$, then $d_{\text{Out}}(o[..k], o'[..k]) \leq \kappa_o$.

Definition 3.79. A non-parametrised nondeterministic reactive system S is *trace integral robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if S is trace integral l-robustly clean w.r.t. \mathcal{C} and S is trace integral u-robustly clean w.r.t. \mathcal{C} .

Func-cleanness suffers from a loss of trace integrity when past-forgetful distance functions are being used, too. Hence, below we define trace integral variants of Definitions 3.60 to 3.62, 3.65, 3.69 and 3.70.

Definition 3.80. A parametrised nondeterministic reactive system S_+ is *trace integral l-func-clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for all pairs of parameters $p, p' \in \text{Param}$ with $p \approx_P p'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and output $o \in S_+(p)(i)$, there exists $o' \in S_+(p')(i')$, such that for every $k \in \mathbb{N}$, $d_{\text{Out}}(o[..k], o'[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

Definition 3.81. A parametrised nondeterministic reactive system S_+ is *trace integral u-func-clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for all pairs of parameters $p, p' \in \text{Param}$ with $p \approx_P p'$, standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and output $o' \in S_+(p')(i')$, there exists $o \in S_+(p)(i)$, such that for every $k \in \mathbb{N}$, $d_{\text{Out}}(o[..k], o'[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

Definition 3.82. A parametrised nondeterministic reactive system S_+ is *trace integral func-clean* w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if S_+ is trace integral l-func-clean w.r.t. \mathcal{C} and S_+ is trace integral u-func-clean w.r.t. \mathcal{C} .

Finally, trace integral func-cleanness for non-parametrised programs is defined below.

Definition 3.83. A non-parametrised nondeterministic reactive system S is *trace integral l-func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and output $o \in S(i)$, there exists $o' \in S(i')$, such that for every $k \in \mathbb{N}$, $d_{\text{Out}}(o[..k], o'[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

Definition 3.84. A non-parametrised nondeterministic reactive system S is *trace integral u-func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if for every standard input $i \in \text{StdIn}$, input $i' \in \text{In}^\omega$ and output $o' \in S(i')$, there exists $o \in S(i)$, such that for every $k \in \mathbb{N}$, $d_{\text{Out}}(o[..k], o'[..k]) \leq f(d_{\text{In}}(i[..k], i'[..k]))$.

Definition 3.85. A non-parametrised nondeterministic reactive system S is *trace integral func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if S is trace integral l-func-clean w.r.t. \mathcal{C} and S is trace integral u-func-clean w.r.t. \mathcal{C} .

3.3 Mixed Input-Output Systems

The cleanness analysis of reactive systems in Section 3.2 assumes that the system is modelled as a function that transforms an infinite sequence of inputs i into an infinite sequence of outputs o , where at every index $k \in \mathbb{N}_+$, $o[k]$ is the output for input $i[k]$ (possibly considering also inputs and outputs at indices smaller than k for the computation of $o[k]$). In other words, the events “receiving an input” and “producing an output” are in fact considered as a single event where receiving an input and producing the output is an atomic step. While this view is a simplification of a real system (e.g., during the execution of a program such as the one in Figure 3.6 there is a delay between reading the input and finishing the computation of the output), it is very suitable for many different types of verification and analysis techniques. In particular, popular modelling formalisms, like finite state transducers, encourage to think in terms of this abstraction. Most verification or analysis results based on this abstraction are valid, because the properties that are being checked are time insensitive. That is, for the analysis it is irrelevant whether an output was produced a few milliseconds earlier or later – what is important is that the output produced is a reaction to the input received.

In this section we propose a generalisation of this strict requirement that every input must have a corresponding output. We propose *mixed input-output systems* (or short mixed-IO systems) that are flexible enough to describe that at any time the system may receive an input or produce an output without putting any constraints on the order in which inputs and outputs occur. For an input domain In and an output domain Out , a mixed-IO system L is defined by the set of all (infinitely long) input-output sequences that can be generated by the system. Hence, formally, $L \subseteq (\text{In} \cup \text{Out})^\omega$. Mixed-IO systems can, for example, be modelled by any kind of labelled automata – where the traces of (accepting) runs of the automaton determine the set of input-output sequences of the system being modelled. We call the input-output sequences in L also *traces*. Concretely, in Section 5.1 we will use labelled transition systems to model the observable behaviour of cyber-physical systems.

The programs and systems in the previous sections are modelled as (total) functions. Hence, these models implicitly required that the systems respond to any possible input. This property is called *input enabled*. A mixed-IO system L is input enabled, if for every $\sigma \in L$, every $k \in \mathbb{N}$, and every input symbol $i \in \text{In}$, there exists a trace $\sigma' \in L$ such that $\sigma[..k] = \sigma'[..k]$ and $\sigma'[k+1] = i$, i.e., at any time, L must accept *any possible* input symbol from its environment. Notice that in contrast to the function-based definitions of reactive systems in Section 3.2, it is possible to define mixed-IO systems that are not input enabled.

Similar to input-enabledness is output-enabledness. L is output enabled, if for every $\sigma \in L$ and every $k \in \mathbb{N}$, there exists a trace $\sigma' \in L$ such that $\sigma[..k] = \sigma'[..k]$ and $\sigma'[k+1] \in \text{Out}$, i.e., at any time, L must be able to produce *at least one* output symbol. In the remainder of this section, we will only consider mixed-IO systems that are input and output enabled.

The mixed-IO system modelling approach is meant to represent black-box systems based on observations of the system behaviour.

Example 3.86. To illustrate mixed-IO systems, we consider the behaviour of a printer. In particular, we consider its behaviour not only for a single printing job, but for all jobs it receives after it turns on. Thus, we consider it as a reactive system. Our observations about the printer are as follows. The inputs to the printer are documents that shall be printed. We can send arbitrary many documents to the printer. After receiving a document, it is printed and appears in the output tray. The “output” of the printer is the printed document we remove from the output tray. The printed documents appear in the output tray in the same order in which the documents were sent to the printer. As long as a document appears in the output tray after it was sent to the printer, there are no restrictions on the order of sending documents to the printer and removing print-outs from the tray. Thus, if we consider five documents i_1 to i_5 and their print-outs o_1 to o_5 , a valid trace of the above printer is $i_1 o_1 i_2 o_2 i_3 o_3 i_4 o_4 i_5 o_5$, i.e., after a document is printed, we remove it from the output tray before the next document is sent to the printer. Alternatively, a valid trace of the printer is $i_1 i_2 i_3 i_4 i_5 o_1 o_2 o_3 o_4 o_5$, i.e., first, all documents are sent to the printer and then all print-outs are removed from the output tray. Obviously, this printer is input-enabled, because new documents can be sent to the printer at any time. To make it output enabled, we introduce the additional output symbol δ (inspired by quiescence as used for model-based conformance tests, cf. Section 2.3) to denote that the output tray is empty and we cannot remove a print-out. So, if we attempt to remove two print-outs from the tray after only one input was sent to the printer, we would observe the trace $i_1 o_1 \delta i_2 o_2 \delta i_3 o_3 \delta i_4 o_4 \delta i_5 o_5 \delta$.

The example highlights a few traces that a mixed-IO model of the printer can generate, but it does not provide a full model of the printer, i.e., it does not

provide the set of *all* traces the printer can generate. Motivated by the above mentioned observer perspective we take on mixed-IO systems, we will operate under the assumption that a system *can* be modelled as a (input- and output-enabled) mixed-IO system without necessarily knowing the (full) model. We will explain in Section 5.3 how mixed-IO system models are used to facilitate testing-based cleanness analyses. On the theoretical level, we will use the assumption that the model is fully known; this is necessary to reason about the correctness of the testing approach. In practice, it suffices to know only those parts of the system that are relevant to conduct a particular test case.

Example 3.87. We take up the example of manipulated diesel emission cleaning systems from the previous sections. From an external observer perspective it is common to consider the speed of a car as the input symbols to the system. Hence, an input to the system is a speed trajectory. Such a speed trajectory is called *test cycle*. As a standard input, we consider the NEDC test cycle (cf. Section 2.7). While driving the car, there are complex physical and chemical processes happening that influence the amount of NO_x in the exhaust stream of the car. Thus, when comparing an NEDC drive and a second test drive, checking the emissions with a high frequency (e.g., every second) could easily lead to false accusations of doping. Convincing test results are achieved if the (average of the) emissions are compared only at the very end of the test. In this case, the standard behaviour would be a trace of 1180 inputs (representing the NEDC) followed by a single output (representing the average amount of NO_x). This is one trace of the car we know and that is also necessary to know to evaluate a cleanness test. When conducting a cleanness test, we obtain a second trace consisting of 1180 inputs (different to the NEDC) and one output value at the end. To evaluate the outcome of the test, only these two traces must be compared; the behaviour of the car beyond these two traces is not relevant and does not need to be known.³

In general, every reactive system modelled as a function $S : \text{In}^\omega \rightarrow 2^{\text{Out}^\omega}$ can also be modelled as a mixed-IO system. Given a reactive system S , an input $i \in \text{In}^\omega$ and the output $o \in S(i)$, the mixed-IO trace that best describes this behaviour of S is the strictly alternating trace $i[0] \ o[0] \ i[1] \ o[1] \ \dots$. This trace reflects S 's property that the k th output symbol in o corresponds to the k th input symbol in i . We say that a trace $\sigma \in (\text{In} \cup \text{Out})^\omega$ is *strictly alternating (for) input i and output o* if and only if $\sigma[2k-1] = i[k]$ and $\sigma[2k] = o[k]$ for all $k \in \mathbb{N}_+$. A mixed-IO system $L \subseteq (\text{In} \cup \text{Out})^\omega$ is *consistent with* a reactive system S if for all $i \in \text{In}^\omega$, $o \in \text{Out}^\omega$ and every $\sigma \in (\text{In} \cup \text{Out})^\omega$ that is strictly alternating input i and output o , it is the case that $o \in S(i)$ if and only if $\sigma \in L$.

³This is the case, if the car is a deterministic system. For nondeterministic systems, we refer to Chapter 6 for more details.

To capture the notion of cleanness in the context of mixed-IO systems, we provide two projections of a trace that transform it into a trace that only contains input information or, respectively, output information. To do this, we extend the set of labels by adding the input \neg_i that indicates that in the respective step some output was produced (but masking the precise output), and the output \neg_o that indicates that in this step some (masked) input was given. *Projection on inputs* $\downarrow_i: (\text{In} \cup \text{Out})^\omega \rightarrow (\text{In} \cup \{\neg_i\})^\omega$ and *projection on outputs* $\downarrow_o: (\text{In} \cup \text{Out})^\omega \rightarrow (\text{Out} \cup \{\neg_o\})^\omega$ are defined for all traces $\sigma \in (\text{In} \cup \text{Out})^\omega$ and $k \in \mathbb{N}_+$ as follows: $\sigma \downarrow_i[k] := \text{if } \sigma[k] \in \text{In} \text{ then } \sigma[k] \text{ else } \neg_i$ and similarly $\sigma \downarrow_o[k] := \text{if } \sigma[k] \in \text{Out} \text{ then } \sigma[k] \text{ else } \neg_o$. They are lifted to sets of traces in the usual elementwise way.

We call the set $\text{In} \cup \{\neg_i\}$ *extended input set* and $\text{Out} \cup \{\neg_o\}$ *extended output set*. We consider strict alternation also for traces over extended input, respectively, output sets. A trace $\sigma \in (\text{In} \cup \{\neg_i\})^\omega$ is *strictly alternating for input i* if and only if $\sigma[2k-1] = i[k]$ and $\sigma[2k] = \neg_i$ for all $k \in \mathbb{N}_+$. Similarly, a trace $\sigma \in (\text{Out} \cup \{\neg_o\})^\omega$ is *strictly alternating for output o* if and only if $\sigma[2k-1] = \neg_o$ and $\sigma[2k] = o[k]$ for all $k \in \mathbb{N}_+$.

For the cleanness contracts, the distance functions d_{In} and d_{Out} apply on the extended input, respectively, output set, i.e. they are distance functions $(\text{In} \cup \{\neg_i\})^* \times (\text{In} \cup \{\neg_i\})^* \rightarrow \mathbb{R}_{\geq 0}$ and, respectively, $(\text{Out} \cup \{\neg_o\})^* \times (\text{Out} \cup \{\neg_o\})^* \rightarrow \mathbb{R}_{\geq 0}$. The set of standard inputs is a subset of the set of infinite traces over the extended input set, i.e., $\text{StdIn} \subseteq (\text{In} \cup \{\neg_i\})^\omega$. For the definition of robust cleanness, it is necessary to get for a mixed-IO system $L \subseteq (\text{In} \cup \text{Out})^\omega$ and a set of standard inputs $\text{StdIn} \subseteq (\text{In} \cup \{\neg_i\})^\omega$ the set $L(\text{StdIn}) := \{\sigma \in L \mid \sigma \downarrow_i \in \text{StdIn}\}$ of all traces in L with the same input as any trace in StdIn . In other words, $L(\text{StdIn})$ replaces all occurrences of \neg_i in a trace $\sigma \in \text{StdIn}$ with a real (unmasked) output such that the resulting trace after the replacements is a trace in L .

3.3.1 Robust cleanness

The notions of l-robust cleanness, u-robust cleanness and robust cleanness for (non-parametrised) reactive systems from Section 3.2.2 can be adapted for mixed-IO systems. We first give a definition for l-robust cleanness.

Definition 3.88. A mixed-IO system $L \subseteq (\text{In} \cup \text{Out})^\omega$ is *l-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if for all $\sigma \in L(\text{StdIn})$, $\sigma' \in L$ and $k \in \mathbb{N}$, if $d_{\text{In}}(\sigma[..j] \downarrow_i, \sigma'[..j] \downarrow_i) \leq \kappa_i$ for all $j \leq k$, then there exists $\sigma'' \in L$, such that $\sigma' \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq \kappa_o$.

The main difference to Definition 3.54 is that the quantifications of inputs and outputs is replaced by quantifications of traces. In the definition above, the universal quantification of σ implicitly universally quantifies a standard input i and

a corresponding output \mathfrak{o} , trace σ' implicitly quantifies a non-standard input i' , and the existential quantification of σ'' implicitly quantifies an output \mathfrak{o}' corresponding to input i' (because $\sigma' \downarrow_i = \sigma'' \downarrow_i$). To compute distances between inputs and outputs, the respective projections of the traces are given to the distance functions d_{In} and d_{Out} . Effectively, d_{In} compares i and i' and d_{Out} compares \mathfrak{o} and \mathfrak{o}' – analogue to Definition 3.54.

The following proposition shows that this definition of l-robust cleanness is at least as expressive as l-robust cleanness for reactive systems.

Proposition 3.89. Let $S : \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ be a reactive system, $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract for robust cleanness, and $L \subseteq (\text{In} \cup \text{Out})^\omega$ a mixed-IO system that is consistent with S . Further, let $\mathcal{C}' = \langle \text{StdIn}', d'_{\text{In}}, d'_{\text{Out}}, 0, 0 \rangle$ be a robust cleanness contract with

$$\begin{aligned} \text{StdIn}' &= \{ \sigma \in (\text{In} \cup \{-i\}) \mid \exists i \in \text{StdIn}. \sigma \text{ is strictly alternating for } i \}, \\ d'_{\text{In}}(\sigma_1, \sigma_2) &= \begin{cases} 0 & \text{if } \sigma_1 \text{ is strictly alternating for input } i_1 \text{ and} \\ & \sigma_2 \text{ is strictly alternating for input } i_2 \text{ and} \\ & d_{\text{In}}(i_1, i_2) \leq \kappa_i \\ 1 & \text{otherwise, and} \end{cases} \\ d'_{\text{Out}}(\sigma_1, \sigma_2) &= \begin{cases} 0 & \text{if } \sigma_1 \text{ is strictly alternating for output } \mathfrak{o}_1 \text{ and} \\ & \sigma_2 \text{ is strictly alternating for output } \mathfrak{o}_2 \\ & d_{\text{Out}}(\mathfrak{o}_1, \mathfrak{o}_2) \leq \kappa_o \\ 1 & \text{otherwise.} \end{cases} \end{aligned}$$

Then, S is l-robustly clean w.r.t. \mathcal{C} if and only if L is l-robustly clean w.r.t. \mathcal{C}' .

Proof. We use the following facts about strictly alternating traces (each easy to prove; hence we omit the proofs):

- (i) For every (finite or infinite) trace $\sigma \in (\text{In} \cup \text{Out})^* \cup (\text{In} \cup \text{Out})^\omega$, if $\sigma \downarrow_i$ is strictly alternating for some input i , then σ is strictly alternating for i and some output \mathfrak{o} .
- (ii) For every mixed-IO system $L \subseteq (\text{In} \cup \text{Out})^\omega$ and finite trace $\sigma \in (\text{In} \cup \text{Out})^*$, if σ is strictly alternating for some i and \mathfrak{o} , then there exists some infinite suffix $\sigma' \in (\text{In} \cup \text{Out})^\omega$, such that $\sigma \cdot \sigma'$ is strictly alternating for some i' and \mathfrak{o}' , and $\sigma \cdot \sigma' \in L$. (Follows from input- and output-enabledness of mixed-IO systems.)
- (iii) For all infinite traces $\sigma_1, \sigma_2 \in (\text{In} \cup \text{Out})^\omega$, inputs $i_1, i_2 \in \text{In}^\omega$, outputs $\mathfrak{o}_1, \mathfrak{o}_2 \in \text{Out}^\omega$, and indices $k, k_1, k_2, k_3, k_4 \in \mathbb{N}$, if $\sigma_1[..k]$ is strictly alternating for $i_1[..k_1]$ and $\mathfrak{o}_1[..k_2]$, and $\sigma_2[..k]$ is strictly alternating for $i_2[..k_3]$ and $\mathfrak{o}_2[..k_4]$, then $k_1 = k_3$ and $k_2 = k_4$.

- (iv) For any two infinite traces $\sigma_1, \sigma_2 \in (\text{In} \cup \text{Out})^\omega$ and inputs $i_1, i_2 \in \text{In}^\omega$, if $\sigma_1 \downarrow_{i_1}$ is strictly alternating for i_1 and $\sigma_2 \downarrow_{i_2}$ is strictly alternating for i_2 , then $i_1 = i_2$ if and only if $\sigma_1 \downarrow_{i_1} = \sigma_2 \downarrow_{i_1}$.
- (v) For every infinite input projection $\sigma \in (\text{In} \cup \{-i\})^\omega$, input $i \in \text{In}^\omega$ and $k \in \mathbb{N}$, if σ is strictly alternating for i , then $\sigma[..k]$ is strictly alternating for $i[..\lfloor \frac{k}{2} \rfloor]$.

To prove the equivalence of the proposition, we show the two underlying implications separately. We first show that l-robust cleanness of S implies l-robust cleanness of L . Let $\sigma_1 \in L(\text{StdIn}')$, $\sigma_2 \in L$, $k \in \mathbb{N}$, and assume that $d'_{\text{In}}(\sigma_1[..j] \downarrow_{i_1}, \sigma_2[..j] \downarrow_{i_2}) \leq 0$ for all $j \leq k$. By definition of $L(\text{StdIn})$, we know that $\sigma_1 \in L$ and that $\sigma_1 \downarrow_{i_1} \in \text{StdIn}'$ constitutes a standard input. From $d'_{\text{In}}(\sigma_1[..k] \downarrow_{i_1}, \sigma_2[..k] \downarrow_{i_2}) = 0$ we can infer that $\sigma_1[..k] \downarrow_{i_1}$ is strictly alternating for some input i'_1 , that $\sigma_2[..k] \downarrow_{i_2}$ is strictly alternating for some input i'_2 , and that $d_{\text{In}}(i'_1[..j], i'_2[..j]) \leq \kappa_i$ for all $j \leq |i'_2|$ (*). Using Fact (i), we get an output o'_2 such that $\sigma_2[..k]$ is strictly alternating for i'_2 and o'_2 ; using Fact (ii), we get a trace $\sigma'_2 \in L$, such that $\sigma'_2[..k] = \sigma_2[..k]$ and σ'_2 is strictly alternating for some i_2 and o_2 . (Note that σ_2 is not necessarily strictly alternating beyond index k .) Since $\sigma_1 \downarrow_{i_1} \in \text{StdIn}'$, we know that $\sigma_1 \downarrow_{i_1}$ is strictly alternating for some input $i_1 \in \text{StdIn}$, and thus, according to Fact (i), σ_1 is strictly alternating for i_1 and some output o_1 . Since L is consistent with S , and σ_1 and σ'_2 are strictly alternating and traces in L , we can infer that $o_1 \in S(i_1)$ and that $o_2 \in S(i_2)$. Let $k_i = |i'_2|$ and $k_o = |o'_2|$. It is easy to verify that $i_1[..k_i] = i'_1$ and $i_2[..k_i] = i'_2$. Notice that strictly alternating traces have at least as many input symbols as output symbols, hence, $k_o \leq k_i$. Thus, (*) and the definition of d'_{In} imply that $d_{\text{In}}(i_1[..j], i_2[..j]) \leq \kappa_i$ for all $j \leq k_o$. From l-robust cleanness of S , we get for i_1, i_2, k_o and the above inequality, that for $o_1 \in S(i_1)$, there exists some $o_3 \in S(i_2)$, such that $d_{\text{Out}}(o_1[..k_o], o_3[..k_o]) \leq \kappa_o$. Let σ_3 be a trace such that it is strictly alternating for i_2 and o_3 . Since $o_3 \in S(i_2)$ and L is consistent with S , $\sigma_3 \in L$. Using Fact (iii), we can infer that $\sigma_3[..k]$ is strictly alternating for $i_2[..k_i]$ and $o_3[..k_o]$. Hence, $\sigma_3[..k] \downarrow_{o_3}$ is strictly alternating for $o_3[..k_o]$. By the definition of d'_{Out} , $d'_{\text{Out}}(\sigma_1[..k] \downarrow_{o_1}, \sigma_3[..k] \downarrow_{o_3}) = 0$. We argue using Fact (iv) that $\sigma_2 \downarrow_{i_2} = \sigma_3 \downarrow_{i_2}$. This proves that L is l-robustly clean w.r.t. C' .

To prove that l-robust cleanness of L implies l-robust cleanness of S , let $i \in \text{StdIn}$, $i' \in \text{In}^\omega$, $k \in \mathbb{N}$, $o \in S(i)$, and assume that $d_{\text{In}}(i[.. for all $j \leq k$ (**). Let $\sigma_1 \in (\text{In} \cup \text{Out})^\omega$ be the trace that is strictly alternating for i and o . Since $o \in S(i)$ and L consistent with S , $\sigma_1 \in L$. Moreover, $\sigma_1 \downarrow_{i_1} \in \text{StdIn}'$, because $\sigma_1 \downarrow_{i_1}$ is strictly alternating for i and $i \in \text{StdIn}$. Let $\sigma_2 \in L$ be a second trace such that σ_2 is strictly alternating for i' and some \hat{o} . Notice that such a trace exists, because L is input- and output-enabled. Next, we want to use l-robust cleanness of L with σ_1, σ_2 and $2k$. Thus, we have to show that for every $j \leq 2k$, $d'_{\text{In}}(\sigma_1[..j] \downarrow_{i_1}, \sigma_2[..j] \downarrow_{i_2}) \leq 0$; we do this by showing that all conditions of$

the first case of the definition of d'_{In} are satisfied. Using Fact (v), we get that $\sigma_1[..j]\downarrow_i$ is strictly alternating for $i[..\lfloor \frac{j}{2} \rfloor]$ and that $\sigma_2[..j]\downarrow_i$ is strictly alternating for $i'[..\lfloor \frac{j}{2} \rfloor]$. Hence, to satisfy the first case of the definition of d'_{In} , it must be that $d_{\text{In}}(i[..\lfloor \frac{j}{2} \rfloor], i'[..\lfloor \frac{j}{2} \rfloor]) \leq \kappa_i$. Since $j \leq 2k$, and thus $\lfloor \frac{j}{2} \rfloor \leq k$, this follows from (**). Now, we get from l-robust cleanness of \mathbf{L} a trace $\sigma_3 \in \mathbf{L}$, such that $\sigma_2\downarrow_i = \sigma_3\downarrow_i$ and $d'_{\text{Out}}(\sigma_1[..2k]\downarrow_{\circ}, \sigma_2[..2k]\downarrow_{\circ}) \leq 0$. From $\sigma_2\downarrow_i = \sigma_3\downarrow_i$ follows with Facts (iv) and (i) that σ_3 is strictly alternating for i' and some output \circ' . Since $\sigma_3 \in \mathbf{L}$ and since \mathbf{L} is consistent with \mathbf{S} , $\sigma' \in \mathbf{S}(i')$. To prove l-robust cleanness of \mathbf{S} it remains to show that $d_{\text{Out}}(\sigma[..k], \sigma'[..k]) \leq \kappa_{\circ}$. It is easy to verify that, since σ_1 is strictly alternating for i and \circ , $\sigma_1[..2k]\downarrow_{\circ}$ is strictly alternating for $\circ[..k]$. Similarly, $\sigma_3[..2k]\downarrow_{\circ}$ is strictly alternating for $\circ'[..k]$. Hence, it follows from $d'_{\text{Out}}(\sigma_1[..2k]\downarrow_{\circ}, \sigma_3[..2k]\downarrow_{\circ}) = 0$ and the definition of d'_{Out} that indeed $d_{\text{Out}}(\sigma[..k], \sigma'[..k]) \leq \kappa_{\circ}$. \square

Notably, \mathbf{L} in Proposition 3.89 is required to be consistent with \mathbf{S} , but beyond that, the traces in \mathbf{L} are not specified. This demonstrates the strength of mixed-IO systems to operate with only partial knowledge about the system. In Proposition 3.89 we only need those traces of \mathbf{L} that are strictly alternating, because these are the only traces that are relevant for robust cleanness w.r.t. \mathcal{C}' .

The definition of u-robust cleanness is analogue to the definition of l-robust cleanness:

Definition 3.90. A mixed-IO system $\mathbf{L} \subseteq (\text{In} \cup \text{Out})^{\omega}$ is *u-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_{\circ} \rangle$ if and only if for all $\sigma \in \mathbf{L}(\text{StdIn})$, $\sigma' \in \mathbf{L}$ and $k \in \mathbb{N}$, if $d_{\text{In}}(\sigma[..j]\downarrow_i, \sigma'[..j]\downarrow_i) \leq \kappa_i$ for all $j \leq k$, then there exists $\sigma'' \in \mathbf{L}$, such that $\sigma\downarrow_i = \sigma''\downarrow_i$ and $d_{\text{Out}}(\sigma'[..k]\downarrow_{\circ}, \sigma''[..k]\downarrow_{\circ}) \leq \kappa_{\circ}$.

A proposition for u-robust cleanness analogue to Proposition 3.89 can be stated and proven very similar. We omit this proposition.

In Section 3.2, we defined robust cleanness using the Hausdorff distance and showed that a quantifier-based definition of robust cleanness (using the conjunction of l-robust cleanness and u-robust cleanness) is almost equivalent to it. The relation between robust cleanness, l-robust cleanness and u-robust cleanness for mixed-IO systems will become more important later (in Chapter 5). To have a “full” equivalence between robust cleanness and the conjunction of l-robust cleanness and u-robust cleanness instead of only an almost equivalence, we use the quantifier-based notion of robust cleanness as the default one for mixed-IO systems.

Definition 3.91. A mixed-IO system $\mathbf{L} \subseteq (\text{In} \cup \text{Out})^{\omega}$ is *robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_{\circ} \rangle$ if and only if for all $\sigma \in \mathbf{L}(\text{StdIn})$, $\sigma' \in \mathbf{L}$ and $k \in \mathbb{N}$, if $d_{\text{In}}(\sigma[..j]\downarrow_i, \sigma'[..j]\downarrow_i) \leq \kappa_i$ for all $j \leq k$, then

1. there exists $\sigma'' \in \mathbf{L}$, such that $\sigma' \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq \kappa_o$,
2. there exists $\sigma'' \in \mathbf{L}$, such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}}(\sigma'[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq \kappa_o$.

3.3.2 Func-cleanness

The adaptations of the family of func-cleanness definitions from Section 3.2.3 for mixed-IO systems are analogue to those of robust cleanness.

Definition 3.92. A mixed-IO system $\mathbf{L} \subseteq (\text{In} \cup \text{Out})^\omega$ is *l-func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if for all $\sigma \in \mathbf{L}(\text{StdIn})$, $\sigma' \in \mathbf{L}$ and $k \in \mathbb{N}$, there exists $\sigma'' \in \mathbf{L}$, such that $\sigma' \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i))$.

Definition 3.93. A mixed-IO system $\mathbf{L} \subseteq (\text{In} \cup \text{Out})^\omega$ is *u-func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if for all $\sigma \in \mathbf{L}(\text{StdIn})$, $\sigma' \in \mathbf{L}$ and $k \in \mathbb{N}$, there exists $\sigma'' \in \mathbf{L}$, such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}}(\sigma'[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i))$.

Definition 3.94. A mixed-IO system $\mathbf{L} \subseteq (\text{In} \cup \text{Out})^\omega$ is *func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if for all $\sigma \in \mathbf{L}(\text{StdIn})$, $\sigma' \in \mathbf{L}$ and $k \in \mathbb{N}$,

1. there exists $\sigma'' \in \mathbf{L}$, such that $\sigma' \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i))$, and
2. there exists $\sigma'' \in \mathbf{L}$, such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}}(\sigma'[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i))$.

3.3.3 Trace Integrity

As discussed in Section 3.2.4, it is sometimes recommended to use trace integral variants of robust or func-cleanness, in particular for contracts that entail past-forgetful distance functions. Below, we enumerate the trace integral cleanness variants for mixed-IO systems.

Definition 3.95. A mixed-IO system $\mathbf{L} \subseteq (\text{In} \cup \text{Out})^\omega$ is *trace integral l-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if for all $\sigma \in \mathbf{L}(\text{StdIn})$ and $\sigma' \in \mathbf{L}$, there exists $\sigma'' \in \mathbf{L}$, such that $\sigma' \downarrow_i = \sigma'' \downarrow_i$ and for every $k \in \mathbb{N}$, if $d_{\text{In}}(\sigma[..j] \downarrow_i, \sigma'[..j] \downarrow_i) \leq \kappa_i$ for all $j \leq k$, then $d_{\text{Out}}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq \kappa_o$.

Definition 3.96. A mixed-IO system $\mathbf{L} \subseteq (\text{In} \cup \text{Out})^\omega$ is *trace integral u-robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if for all $\sigma \in \mathbf{L}(\text{StdIn})$ and $\sigma' \in \mathbf{L}$ there exists $\sigma'' \in \mathbf{L}$, such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and for every $k \in \mathbb{N}$, if $d_{\text{In}}(\sigma[..j] \downarrow_i, \sigma'[..j] \downarrow_i) \leq \kappa_i$ for all $j \leq k$, then $d_{\text{Out}}(\sigma'[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq \kappa_o$.

Definition 3.97. A mixed-IO system $L \subseteq (\text{In} \cup \text{Out})^\omega$ is *trace integral robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if L is trace integral l-robust cleanness w.r.t. \mathcal{C} and L is trace integral u-robustly clean w.r.t. \mathcal{C} .

The trace integral variants of func-cleanness for mixed-IO systems are given below.

Definition 3.98. A mixed-IO system $L \subseteq (\text{In} \cup \text{Out})^\omega$ is *trace integral l-func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if for all $\sigma \in L(\text{StdIn})$ and $\sigma' \in L$, there exists $\sigma'' \in L$ with $\sigma' \downarrow_i = \sigma'' \downarrow_i$, such that for every $k \in \mathbb{N}$, it holds that $d_{\text{Out}}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i))$.

Definition 3.99. A mixed-IO system $L \subseteq (\text{In} \cup \text{Out})^\omega$ is *trace integral u-func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if for all $\sigma \in L(\text{StdIn})$ and $\sigma' \in L$, there exists $\sigma'' \in L$ with $\sigma \downarrow_i = \sigma'' \downarrow_i$, such that for every index $k \in \mathbb{N}$, it holds that $d_{\text{Out}}(\sigma'[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i))$.

Definition 3.100. A mixed-IO system $L \subseteq (\text{In} \cup \text{Out})^\omega$ is *trace integral func-clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if L is trace integral l-func-clean w.r.t. \mathcal{C} and L is trace integral u-func-clean w.r.t. \mathcal{C} .

3.4 Hybrid Systems

Sequential programs, reactive systems and mixed-IO systems presented in Sections 3.1 to 3.3 describe discrete systems; the exact time at which an input or output occurs is not explicitly modelled. Still, these models encode implicit timing information about the relative times when inputs or outputs occur. For reactive systems it is a requirement that after receiving the k th input symbol and before receiving the $k + 1$ th input symbol the system produces the k th output. This was relaxed for mixed-IO system models, which provide the freedom to specify the order in which inputs are received and outputs are produced.

For some systems it is indispensable to explicitly model the time when some interaction with the system takes place. For example, systems that are typically called hybrid systems have dynamic components, where the time at which an input is received or an output is computed plays an important role for the output value. To explicitly consider time in execution traces of a system we will use generalised timed traces (cf. Section 2.2). The GTTs provide an elegant way to model hybrid systems from a black-box perspective. In this thesis, a *hybrid system* is a function $H : \text{In}^\vartheta \rightarrow 2^{(\text{Out}^\vartheta)}$ such that for all $\mu \in \text{In}^\vartheta$ and all $\mu' \in H(\mu)$ it holds that $\text{dom}(\mu') = \text{dom}(\mu)$. H gets a GTT with value domain In and non-deterministically produces a GTT with value domain Out and with the

same time domain as the input GTT. If H is deterministic, we may also write $H : \text{In}^\vartheta \rightarrow \text{Out}^\vartheta$.

Recall that the time domain of GTTs may be discrete or continuous. Hence, hybrid systems are a generalisation of reactive systems. An infinite sequence of inputs $i \in \text{In}^\omega$ can be represented by a GTT $\mu_i : \mathbb{N} \rightarrow \text{In}$ with $\mu_i(k) := i[k + 1]$ for all $k \in \mathbb{N}$. That is, the time domain of the GTT is the set of natural numbers and every natural number represents a position in trace i . A reactive (non-parametrised) system S can be encoded by a hybrid system H that maps every input GTT constructed from $i \in \text{In}^\omega$ to the set of output GTTs that are constructed from $S(i)$. Formally, such an encoding hybrid system must satisfy $H(\mu_i) = \{\mu : \mathbb{N} \rightarrow \text{Out} \mid \exists o \in S(i). \forall k \in \mathbb{N}. \mu(k) = o[k]\}$ if μ_i is constructed from $i \in \text{In}^\omega$.

The cleanness definitions for non-parametrised reactive systems can be adapted for hybrid systems. We show for robust cleanness the major changes for such an adaptation. The cleanness contract for robust cleanness of hybrid systems consists of a set of standard inputs $\text{StdIn} \subseteq \text{In}^\vartheta$. As for reactive systems, this set is a subset of input traces. In light of potentially continuous time domains and possible retimings of traces (which we will explore later), we consider distance functions that are *past-forgetful*, i.e., distances function $d_{\text{In}} : \text{In} \times \text{In} \rightarrow \mathbb{R}_{\geq 0}$ and $d_{\text{Out}} : \text{Out} \times \text{Out} \rightarrow \mathbb{R}_{\geq 0}$ that compute distances only between single input symbols, respectively output symbols. The threshold values κ_i and κ_o are values from the extended reals. For such cleanness contracts, we propose the following quantifier-based definition.

Definition 3.101. A hybrid system $H : \text{In}^\vartheta \rightarrow 2^{(\text{Out}^\vartheta)}$ is *robustly clean* w.r.t. contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$, if for every standard input $i \in \text{StdIn}$, $i' \in \text{In}^\vartheta$ and every time $t \in \text{dom}(i) \cup \text{dom}(i')$, if $\text{TraceConf}_{d_{\text{In}}, \kappa_i}(i[..t'], i'[..t'])$ for all $t' \leq t$, then the following two statements hold:

1. for every $o \in H(i)$, there exists $o' \in H(i')$, s.t. $\text{TraceConf}_{d_{\text{Out}}, \kappa_o}(o[..t], o'[..t])$,
2. for every $o' \in H(i')$, there exists $o \in H(i)$, s.t. $\text{TraceConf}_{d_{\text{Out}}, \kappa_o}(o[..t], o'[..t])$.

The definition compares every standard input $i \in \text{StdIn}$ with every, potentially non-standard, input $i' \in \text{In}^\vartheta$. As for reactive systems, robust cleanness checks all prefixes of the traces by universally quantifying t to satisfy the safety characteristic as discussed in Section 3.2. For generalised timed traces with a potentially continuous time domain, robust cleanness quantifies over all points of time that occur in the time domain of i or of i' (and hence o and o'). Instead of applying d_{In} and κ_i or d_{Out} and κ_o directly to the trace prefixes, robust cleanness uses the trace conformance predicate TraceConf (cf. Section 2.4). The trace conformance enforces that the time domains of two traces are identical and that at every

time the distance of the input (or output) symbols does not exceed the input (or output) threshold κ_i (or κ_o). Hence, although robust cleanness does not require that $\text{dom}(i) = \text{dom}(i')$, the trace conformance condition on the inputs ensures that traces where $\text{dom}(i) \neq \text{dom}(i')$ are immediately deemed clean. If $\text{dom}(i) = \text{dom}(i')$ and if at all times in the inputs' time domain the distance between the input symbols is less or equal to κ_i , then the outputs for these inputs must be trace conformant, too. Since the definition shall be applicable to nondeterministic systems, the output trace conformance check is resolved by a quantification that is analogue to that of l-robust cleanness and u-robust cleanness for reactive systems. Notice that using a Hausdorff-based comparison is not possible, because the output distance constraint is hidden inside the `TraceConf` predicate.

Although hybrid systems can encode reactive systems, robust cleanness of hybrid systems can, in general, not encode robust cleanness of reactive systems, because the former requires past-forgetful distance functions and the latter does not. For reactive systems cleanness contracts with past-forgetful distance functions, robust cleanness of hybrid systems is equivalent to the quantifier-based definition of robust cleanness for reactive systems. Hence, in this case, robust cleanness of hybrid systems is almost equivalent to robust cleanness of reactive systems.

The definition of hybrid systems above is based on the traces that the system can generate rather than a model that describes the internals of the system. Thus, the system is considered to be a black-box and its analysis can rely only on the observed traces. Observing a system comes with the disadvantage that the time information of a trace is typically imprecise. Reasons for this imprecision are numerous. For example, the start of a test and the start of a recording may be off sync if the tester and the recorder are independent systems. Also, some systems require a human to pass the inputs to the system, but the human passes it with a certain delay or ahead of the scheduled time, et cetera. We make this problem more concrete in the following example.

Example 3.102. We take up the example of manipulated diesel emission cleaning systems from the previous sections using speed trajectories as inputs to the system. Figure 3.8.(a) shows two speed trajectories. The black solid line depicts an input trajectory i_{st} that we assume to be a standard input. The red dashed line represents a trajectory i_{dev} that is defined for the purpose of doing a robust cleanness test: the distance between the individual speed values of i_{st} and i_{dev} is always at most κ_i large. During the actual test execution a human driver tries to follow the trajectory i_{dev} , but they can only manage to drive it with a small delay. In subfigure (b) the red dashed line shows a recording of this drive; it is slightly

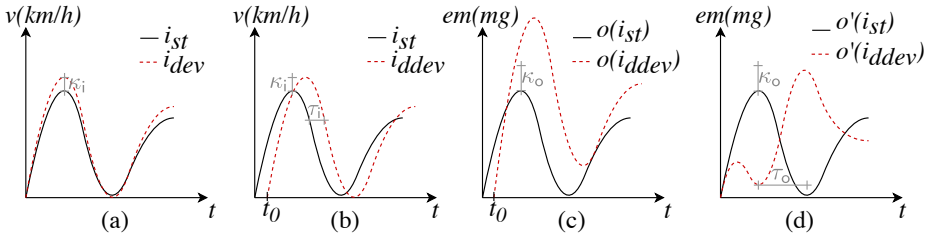


Figure 3.8: Specified (a) and actual (b) speed trajectories and emission footprints obtained from different (fictitious) vehicles (c) and (d).

shifted to the right. We denote this shifted trajectory as i_{ddev} . Subfigures (c) and (d) show how the outputs for i_{st} and i_{ddev} could look like (for two distinct fictitious cars). The output symbols represent the absolute amount of emissions that a car emits. The (identical) black solid lines show the emissions for i_{st} and the red dashed lines show the emissions for i_{ddev} . The problem that this example shall make aware of is the following. Intuitively, the difference between i_{st} and i_{ddev} is small. However, the input comparison inside the definition of robust cleanness dismisses the pair of inputs, because the value difference is often above the κ_i threshold. For example, $|i_{st}(t_0) - i_{ddev}(t_0)| > \kappa_i$. The outputs for i_{st} and i_{ddev} deviate vastly from each other, so intuitively, we would say that the system producing these outputs is doped. Assuming that our intuitions are correct, this is an example for a false negative verdict of doping analysis, because robust cleanness would label this case as clean behaviour although it is doped.

The example demonstrates that not accounting for timing disturbances when relating input trajectories can result in false negatives in doping detection. Dually, using the traditional comparison for output traces can result in false positives by requiring overly strict matching of outputs.

The above example motivates the need to account for timing deviations in trajectories. Intuitively, for input trajectories this relaxation results in considering more traces as conforming, and thus enforcing more comparisons when checking if a system is clean. For output trajectories this means relaxing the conformance requirement by considering two output sequences as conforming even if their values are not perfectly aligned in time.

A relaxation that solves the problem explained in Example 3.102 is a replacement of trace conformance in Definition 3.101 by hybrid conformance.

Example 3.103. In Example 3.102 the input trajectories i_{st} and i_{dev} are trace conformant for $\epsilon = \kappa_i$ and $d_{in}(v_1, v_2) = |v_1 - v_2|$. For i_{st} and i_{ddev} this is not the case, because of the time shift by time τ_i to the right. However, i_{st} and i_{ddev}

are hybrid conformant for d_{in} , κ_i and τ_i , because hybrid conformance allows to shift i_{ddev} back to the left such that it is identical to i_{dev} . Hence, when replacing $\text{TraceConf}_{d_{in}, \kappa_i}$ by $\text{HybridConf}_{d_{in}, \kappa_i, \tau_i}$, the instance of software doping manifesting in Figure 3.8.(c) is correctly detected.

Still, hybrid conformance is not in every situation the best conformance notion. For example, hybrid conformance does not preserve the order in which events occur, which can turn it into a too lax requirement, as the following examples demonstrates.

Example 3.104. Based on Example 3.103, assume a cleanness definition that is like Definition 3.101, but where $\text{HybridConf}_{d_{in}, \kappa_i, \tau_i}$ replaces $\text{TraceConf}_{d_{in}, \kappa_i}$ for inputs, and similarly, where hybrid conformance $\text{HybridConf}_{d_{out}, \kappa_o, \tau_o}$ replaces trace conformance $\text{TraceConf}_{d_{out}, \kappa_o}$ for outputs. Observe that now the outputs obtained from the second fictitious car in Figure 3.8.(d) are hybrid conformant and that this behaviour is classified as clean. However, the output for i_{ddev} is clearly suspicious, as the outputs for i_{st} and i_{ddev} are reversed. This motivates considering conformance notions that require retimings to be order-preserving. Indeed, using Skorokhod conformance we can detect that the system is doped. That is, when replacing $\text{TraceConf}_{d_{in}, \kappa_i}$ with $\text{SkorConf}_{d_{in}, \kappa_i, \tau_i}$ and $\text{TraceConf}_{d_{out}, \kappa_o}$ with $\text{SkorConf}_{d_{out}, \kappa_o, \tau_o}$, the doping in Figure 3.8.(d) is detected.

The above examples show that in order to be useful in a diverse set of applications, a software cleanness theory should allow for using a variety of conformance notions.

3.4.1 Conformance-Based Cleanness

To support more than hybrid conformance or Skorokhod conformance in cleanness definitions, we provide a generic framework to express conformance notions. A conformance notion may allow variations in time and variations in values in a controlled way. To control value variations we keep the established combination of a distance function and a distance threshold up to which two values may deviate from each other. For time variations we introduce the concept of retimings. In its general form a retiming is a pair of functions between two time domains. Intuitively, given two GTTs, a retiming will define a mapping from time points in each of the traces to time points in the other trace. Note that in general the mappings are not required to be injective; this way we can cater for notions of conformance allowing for the so-called local disorder phenomenon (in particular hybrid conformance – see Proposition 3.108).

Definition 3.105. Let $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathbb{R}_{\geq 0}$ be two time domains. A *retiming* is a pair (r_1, r_2) , where $r_1 : \mathcal{T}_1 \rightarrow \mathcal{T}_2$ and $r_2 : \mathcal{T}_2 \rightarrow \mathcal{T}_1$. We denote the set of all retimings between \mathcal{T}_1 and \mathcal{T}_2 with $\mathfrak{R}(\mathcal{T}_1, \mathcal{T}_2)$.

Retiming is explicitly present in the definition of Skorokhod conformance. There, each Skorokhod retiming is required to be a strictly increasing continuous bijection. We can express a Skorokhod retiming r as an instance of our definition as the pair (r, r^{-1}) . The concept of retimings is also general enough to define hybrid conformance. When working with retimings, it may be convenient to work with whole classes of retimings (e.g., the class of all Skorokhod retimings). Hence, we will use the term *family of retimings* to denote a set of retimings that are related.

For convenience, we define two standard operations to filter the retimings in a family of retimings Ret . One operation constraints Ret by applying a time threshold τ ; the subset Ret_τ of Ret contains only functions that shift time by at most τ time units. The second operation reduces a family of retimings to those retimings that are applicable to two concrete time domains \mathcal{T}_1 and \mathcal{T}_2 : $\text{Ret}(\mathcal{T}_1, \mathcal{T}_2)$ is a subset of Ret that contains only retimings that map between \mathcal{T}_1 and \mathcal{T}_2 .

Definition 3.106. Let Ret be a family of retimings. Then,

$$\begin{aligned} \text{Ret}_\tau &:= \{(r_1, r_2) \in \text{Ret} \mid \forall t \in \text{dom}(r_i) : |r_i(t) - t| \leq \tau \ (i = 1, 2)\}, \\ \text{Ret}(\mathcal{T}_1, \mathcal{T}_2) &:= \text{Ret} \cap \mathfrak{R}(\mathcal{T}_1, \mathcal{T}_2). \end{aligned}$$

This leads to a generic notion of conformance associated with a family of retimings Ret , a distance function d , a time threshold τ and a value threshold ϵ .

Definition 3.107. A *conformance notion* with distance function $d : X \times X \rightarrow \overline{\mathbb{R}}_{\geq 0}$, time threshold $\tau \in \overline{\mathbb{R}}_{\geq 0}$ and value threshold $\epsilon \in \overline{\mathbb{R}}_{\geq 0}$ induced by the family of retimings Ret is a predicate $\text{Conf}_{d, \tau, \epsilon}^{\text{Ret}}$ on pairs of GTTs such that, for $\mu_1 : \mathcal{T}_1 \rightarrow X$, $\mu_2 : \mathcal{T}_2 \rightarrow X$:

$$\begin{aligned} \text{Conf}_{d, \tau, \epsilon}^{\text{Ret}}(\mu_1, \mu_2) &:\Leftrightarrow \exists (r_1, r_2) \in \text{Ret}_\tau(\mathcal{T}_1, \mathcal{T}_2). \quad \forall t \in \mathcal{T}_1. d(\mu_1(t), \mu_2 \circ r_1(t)) \leq \epsilon \\ &\quad \wedge \quad \forall t \in \mathcal{T}_2. d(\mu_2(t), \mu_1 \circ r_2(t)) \leq \epsilon. \end{aligned}$$

Notice that if $\tau = \infty$, then $\text{Conf}_{d, \infty, \epsilon}^{\text{Ret}}$ is a conformance notion with unbounded⁴ time deviation.

Using the above definition, we can easily express the specific notions of conformance we used in the previous examples by selecting a suitable family of retimings.

Proposition 3.108. Let $d : X \times X \rightarrow \overline{\mathbb{R}}_{\geq 0}$ be a distance function, $\tau \in \overline{\mathbb{R}}_{\geq 0}$ be a time threshold, and $\epsilon \in \overline{\mathbb{R}}_{\geq 0}$ a value threshold. Then, the conformance notions induced by the retimings below are equivalent to the corresponding conformance notions from Definitions 2.4 and 2.5 in Section 2.4.

⁴It is not bounded explicitly; Ret may, of course, define arbitrary bounds on the time deviation.

- Let $\text{Ret}_{\text{id}} = \{(\text{id}, \text{id}) \mid \text{id} : \mathcal{T} \rightarrow \mathcal{T} \text{ is the identity on some } \mathcal{T} \subseteq \mathbb{R}_{\geq 0}\}$ be the family of retimings containing only identity functions. Then, for every $\mu_1 : \mathcal{T}_1 \rightarrow X$ and $\mu_2 : \mathcal{T}_2 \rightarrow X$, it holds that $\text{Conf}_{d,0,\epsilon}^{\text{Ret}_{\text{id}}}(\mu_1, \mu_2)$ if and only if $\text{TraceConf}_{d,\epsilon}(\mu_1, \mu_2)$.
- Let $\text{Ret}_{\text{sk}} = \{(r, r^{-1}) \mid r \text{ is a strictly increasing continuous bijection}\}$. For every $\mu_1 : \mathcal{T}_1 \rightarrow X$ and $\mu_2 : \mathcal{T}_2 \rightarrow X$, it holds that $\text{Conf}_{d,\tau,\epsilon}^{\text{Ret}_{\text{sk}}}(\mu_1, \mu_2)$ if and only if $\text{SkorConf}_{d,\tau,\epsilon}(\mu_1, \mu_2)$.
- Let $\text{Ret}_{\text{hy}} = \{(r_1, r_2) \in (\mathcal{T}_1 \rightarrow \mathcal{T}_2) \times (\mathcal{T}_2 \rightarrow \mathcal{T}_1) \mid \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathbb{R}_{\geq 0}\}$ be the family of arbitrary retimings. Then, for every $\mu_1 : \mathcal{T}_1 \rightarrow X$ and $\mu_2 : \mathcal{T}_2 \rightarrow X$, it holds that $\text{Conf}_{d,\tau,\epsilon}^{\text{Ret}_{\text{hy}}}(\mu_1, \mu_2)$ if and only if $\text{HybridConf}_{d,\tau,\epsilon}(\mu_1, \mu_2)$.

Definition 3.107 also enables us to define other notions of conformance, such as, for instance a “shift conformance”, which, intuitively, shifts all time points by a given constant $c \in \mathbb{R}$. Such a conformance notion would be constructed from the family of retimings $\text{Ret}_c = \{(r, r^{-1}) \mid r(t) = t + c\}$.

Next, we define a generic notion of cleanness, parametrised by conformance predicates for the input and for the output traces. Instantiating these predicates with existing or new conformance notions, yields different conformance-based notions of cleanness that can capture a variety of cleanness specifications.

As already sketched in Examples 3.103 and 3.104, we can obtain a new type of cleanness definition by replacing trace conformance in Definition 3.101 by different conformance notions. Intuitively, we will extend robust cleanness from “small input changes imply reasonable output changes” to *conf-cleanness* that enforces that “small input changes under small time variations imply reasonable output changes under reasonable time variations”. *Conf-cleanness* operates w.r.t. a formal contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{In}}, \text{Ret}_{\text{Out}}, \tau_i, \tau_o \rangle$ containing all information necessary to construct conformance notions for inputs and outputs.

To accommodate for distance in time, we (1) compare prefixes using a conformance relation, and (2) allow for variation in the length of the compared prefixes that is within the corresponding time-distance threshold. More precisely, when comparing two prefixes, we allow for discarding start and end segments of length at most τ_i , respectively τ_o . This idea is formally implemented by the predicate $\text{Pref}_{\tau}^{\text{Conf}}$, which is parametrised by a conformance notion Conf and a time threshold τ . The predicate $\text{Pref}_{\tau}^{\text{Conf}}$ compares the prefixes of two traces μ_1 and μ_2 by requiring that there exist infixes $\mu_1[t_1^s \dots t_1^e]$ and $\mu_2[t_2^s \dots t_2^e]$ of these traces that are conformant with respect to Conf . The infixes are obtained from μ_1 and μ_2 by taking a prefix of them that has a “normative” length of t but where prefixes

or suffixes of length up to τ are added or removed at the beginning or the end of the t -prefix.

Definition 3.109. Let Conf be a notion of conformance on GTTs with tolerance threshold $\tau \in \mathbb{R}_{\geq 0}$ for time disturbance. For any pair of GTTs $\mu_1 : \mathcal{T}_1 \rightarrow \mathsf{X}$ and $\mu_2 : \mathcal{T}_2 \rightarrow \mathsf{X}$, and time $t \in \mathcal{T}_1 \cup \mathcal{T}_2$, the predicate $\text{Pref}_\tau^{\text{Conf}}$ is defined as:

$$\begin{aligned} \text{Pref}_\tau^{\text{Conf}}(\mu_1, \mu_2, t) &: \Leftrightarrow \exists t_1^s \in [0, \tau] \cap \mathcal{T}_1. \exists t_1^e \in [t - \tau, t + \tau] \cap \mathcal{T}_1. \\ &\quad \exists t_2^s \in [0, \tau] \cap \mathcal{T}_2. \exists t_2^e \in [t - \tau, t + \tau] \cap \mathcal{T}_2. \\ &\quad \text{Conf}(\mu_1[t_1^s..t_1^e], \mu_2[t_2^s..t_2^e]). \end{aligned}$$

For conformance notions with unbounded timing deviation $\text{Pref}_\tau^{\text{Conf}}$ coincides with Conf , i.e., $\text{Pref}_\infty^{\text{Conf}} := \text{Conf}$.

The predicate $\text{Pref}_\tau^{\text{Conf}}$ provides a generic notion of prefix-conformance. By instantiating it with conformance relations $\text{Conf}_{d_{\text{In}}, \tau_i, \kappa_i}^{\text{Ret}_{\text{In}}}$ and $\text{Conf}_{d_{\text{Out}}, \tau_o, \kappa_o}^{\text{Ret}_{\text{Out}}}$ for input and output traces respectively, we can define *conf-cleanness*.

Definition 3.110. Let $H : \text{In}^\vartheta \rightarrow 2^{\text{Out}^\vartheta}$ be a hybrid system and consider the cleanness contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{In}}, \text{Ret}_{\text{Out}}, \tau_i, \tau_o \rangle$. Let further be $\text{Conf}_{\text{In}} = \text{Conf}_{d_{\text{In}}, \tau_i, \kappa_i}^{\text{Ret}_{\text{In}}}$ and $\text{Conf}_{\text{Out}} = \text{Conf}_{d_{\text{Out}}, \tau_o, \kappa_o}^{\text{Ret}_{\text{Out}}}$ the conformance notions for inputs and, respectively, outputs obtained from \mathcal{C} . Then, H is *conf-clean* w.r.t. \mathcal{C} , if and only if

$$\begin{aligned} \forall i_1 \in \text{StdIn}, i_2 \in \text{In}^\vartheta. \forall t \in \text{dom}(i_1) \cup \text{dom}(i_2). \\ (\forall t' \leq t. \text{Pref}_{\tau_i}^{\text{Conf}_{\text{In}}}(i_1, i_2, t')) \Rightarrow \\ ((\forall o_1 \in H(i_1). \exists o_2 \in H(i_2). \text{Pref}_{\tau_o}^{\text{Conf}_{\text{Out}}}(o_1, o_2, t)) \wedge \\ (\forall o_2 \in H(i_2). \exists o_1 \in H(i_1). \text{Pref}_{\tau_o}^{\text{Conf}_{\text{Out}}}(o_1, o_2, t))). \end{aligned}$$

As for robust cleanness, we consider two inputs, one of which is a standard input. To not miss any possible prefix of the two inputs, the definition quantifies over all time values that are in at least one of the two inputs' time domains. Also, the adequacy of the combination of inputs is determined analogously to previous robust cleanness definitions, except that *conf-cleanness* uses the prefix conformance w.r.t. Conf_{In} as the predicate deciding the adequacy. For an adequate pair of inputs, the remaining part of the definition is analogue to the quantifier-based definition of robust cleanness and ensures that the sets of outputs satisfy prefix conformance w.r.t. Conf_{Out} .

The following propositions shows that *conf-cleanness* is a generalisation of robust cleanness.

Proposition 3.111. Let $H : \text{In}^\vartheta \rightarrow 2^{\text{Out}^\vartheta}$ be a hybrid system and assume a robust cleanness contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$. Then, for conf-cleanness contract $\mathcal{C}' = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{id}}, \text{Ret}_{\text{id}}, 0, 0 \rangle$ it holds that H is robustly clean w.r.t. \mathcal{C} if and only if H is conf-clean w.r.t. \mathcal{C}' .

Proof. Let $i \in \text{StdIn}$, $i' \in \text{In}^\vartheta$, and $t \in \text{dom}(i_1) \cup \text{dom}(i_2)$. For conf-cleanness, we may assume the premise that restricts the combination of inputs to be checked. Hence, we assume that $\forall t' \leq t$, $\text{Pref}_0^{\text{ConfIn}}(i_1, i_2, t')$ for $\text{ConfIn} = \text{Conf}_{d_{\text{In}}, 0, \kappa_i}^{\text{Ret}_{\text{id}}}$. By Definition 3.109, $\text{Pref}_0^{\text{ConfIn}}(i_1, i_2, t')$ if and only if $\text{ConfIn}(i_1[0..t'], i_2[0..t'])$. We know from Prop. 3.108 that $\text{Conf}_{d_{\text{In}}, 0, \kappa_i}^{\text{Ret}_{\text{id}}} = \text{TraceConf}_{d_{\text{In}}, \kappa_i}$. Thus, $\text{Pref}_0^{\text{ConfIn}}(i_1, i_2, t')$ if and only if $\text{TraceConf}_{d_{\text{In}}, \kappa_i}(i_1[.t'], i_2[.t'])$, which is exactly the premise of robust cleanness that restricts the combination of inputs to be checked. This proves that conf-cleanness and robust cleanness consider the same combinations of inputs for the output conformance check. Using the same arguments, we get for any two outputs o_1 and o_2 that $\text{Pref}_0^{\text{ConfOut}}(o_1, o_2, t)$ if and only if $\text{Conf}_{d_{\text{Out}}, 0, \kappa_i}^{\text{Ret}_{\text{id}}}(o_1[0..t], o_2[0..t])$ (using Definition 3.109) and that this is equivalent to $\text{TraceConf}_{d_{\text{Out}}, \kappa_i}(o_1[.t'], o_2[.t'])$ (using Proposition 3.108). Hence, we can conclude that any H is robustly clean w.r.t. \mathcal{C} if and only if H is conf-clean w.r.t. \mathcal{C}' . \square

In this section, we introduced hybrid conformance and Skorokhod conformance as concrete examples for suitable conformance notions. Derived from conf-cleanness we will explicitly define cleanness notions using hybrid and, respectively, Skorokhod conformance. Both definitions define cleanness w.r.t. a contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \tau_i, \tau_o \rangle$ containing the set of standard inputs, value distance functions, and thresholds for value distances and time distances.

Definition 3.112. Let $H : \text{In}^\vartheta \rightarrow 2^{\text{Out}^\vartheta}$ be a hybrid system and consider a cleanness contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \tau_i, \tau_o \rangle$. Then,

1. H is *hybrid-conformance clean* w.r.t. \mathcal{C} if and only if H is conf-clean w.r.t. contract $\langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{hy}}, \text{Ret}_{\text{hy}}, \tau_i, \tau_o \rangle$,
2. H is *Skorokhod-conformance clean* w.r.t. \mathcal{C} if and only if conf-clean w.r.t. contract $\langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{sk}}, \text{Ret}_{\text{sk}}, \tau_i, \tau_o \rangle$.

Example 3.113. Using the new cleanness definitions, we can re-examine the behaviour depicted in Figure 3.8. For the inputs i_{st} and i_{ddev} (in subfigure (b)) the retiming function $r(t) := t - \tau_i$ shifts i_{ddev} to the left so that it is identical to i_{ddev} . Hence, it is obvious that i_{st} and i_{ddev} are hybrid conformant. Since r is a strictly increasing bijection between the time domains of i_{st} and i_{ddev} , the retiming (r, r^{-1}) induces Skorokhod conformance; hence, i_{st} and i_{ddev} are also Skorokhod conformant. As discussed in Example 3.102 i_{st} and i_{ddev} are not

trace conformant. The outputs for i_{st} and for i_{ddev} in subfigure (c) are neither trace, hybrid nor Skorokhod conformant. Indeed, whatever retiming one might choose, the peak value of i_{ddev} 's output has a distance of more than κ_o from all values in i_{st} 's output. In subfigure (d), the two outputs are hybrid conformant for κ_o and τ_o . However, they are not trace or Skorokhod conformant. Based on these conformance evaluations, we can conclude that both the outputs in (c) and those in (d) are robustly clean, because the inputs for which these outputs were produced did not meet the adequacy criterion in the first place. Hybrid cleanness is violated for the outputs in (c), but satisfied for the outputs in (d). Finally, both the outputs in (c) and (d) violate Skorokhod cleanness. We remark that these verdicts relate only to the inputs i_{st} and i_{ddev} and under the assumption, that the outputs shown in (c) and (d) are the only outputs that can be produced for these inputs by the fictitious cars we assumed in this running example. Notably, the car that produced the outputs in subfigure (d) is not necessarily hybrid-conformance clean. To get a (positive) cleanness verdict for the system as a whole, the full input space of the system must be considered, i.e., the system must satisfy hybrid-conformance cleanness for every combination of inputs.

The contracts for conf-cleanness provide a variety of parameters to modify when hybrid systems are deemed clean or doped. The following proposition shows, that, in general, less tight input conformances or tighter output conformances lead to stricter cleanness definitions.

Proposition 3.114. Let H be a hybrid system, StdIn be a set of standard inputs, d_{In} and d_{Out} distance functions, and $\kappa_i^1, \kappa_i^2, \kappa_o^1, \kappa_o^2, \tau_i^1, \tau_i^2, \tau_o^1$ and τ_o^2 value and time thresholds. Further, let $\text{Ret}_{\text{In}}^1, \text{Ret}_{\text{Out}}^1, \text{Ret}_{\text{In}}^2$ and $\text{Ret}_{\text{Out}}^2$ be families of retimings and let $\text{Conf}_{\text{In}}^1 = \text{Conf}_{d_{\text{In}}, \tau_i^1, \kappa_i^1}^{\text{Ret}_{\text{In}}^1}$, $\text{Conf}_{\text{Out}}^1 = \text{Conf}_{d_{\text{Out}}, \tau_o^1, \kappa_o^1}^{\text{Ret}_{\text{Out}}^1}$, $\text{Conf}_{\text{In}}^2 = \text{Conf}_{d_{\text{In}}, \tau_i^2, \kappa_i^2}^{\text{Ret}_{\text{In}}^2}$ and $\text{Conf}_{\text{Out}}^2 = \text{Conf}_{d_{\text{Out}}, \tau_o^2, \kappa_o^2}^{\text{Ret}_{\text{Out}}^2}$ be conformance notions induced by these families of retimings.

Then, whenever $\text{Conf}_{\text{In}}^1 \supseteq \text{Conf}_{\text{In}}^2$ and $\text{Conf}_{\text{Out}}^1 \sqsubseteq \text{Conf}_{\text{Out}}^2$, it holds that H being conf-clean w.r.t. contract $\langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i^1, \kappa_o^1, \text{Ret}_{\text{In}}^1, \text{Ret}_{\text{Out}}^1, \tau_i^1, \tau_o^1 \rangle$ implies that H is conf-clean w.r.t. contract $\langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i^2, \kappa_o^2, \text{Ret}_{\text{In}}^2, \text{Ret}_{\text{Out}}^2, \tau_i^2, \tau_o^2 \rangle$.

The result of Proposition 3.114 becomes particularly interesting in combination with the following proposition, which establishes relations between trace, hybrid and Skorokhod conformance.

Proposition 3.115. For any distance function d and thresholds $\tau, \epsilon \in \overline{\mathbb{R}}_{\geq 0}$, the following relations hold:

$$\text{TraceConf}_{d, \epsilon} \sqsubseteq \text{SkorConf}_{d, \tau, \epsilon} \sqsubseteq \text{HybridConf}_{d, \tau, \epsilon}$$

Propositions 3.114 and 3.115 have two important corollaries. The first one explains the relationships between robust cleanness, and notions of cleanness based on Skorokhod conformance and hybrid conformance, in particular stating the conservative generalisation property for the latter notions. The second corollary compares cleanness notions with different distance thresholds.

In the remainder of this section we assume some arbitrary, but fixed StdIn , d_{In} and d_{Out} and write $\text{RobustClean}(\kappa_i, \kappa_o)$ to mean “robustly clean w.r.t. contract $\langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ ”, $\text{HybridClean}(\tau_i, \kappa_i, \tau_o, \kappa_o)$ to mean “hybrid-conformance clean w.r.t. $\langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \tau_i, \tau_o \rangle$ ”, and $\text{SkorClean}(\tau_i, \kappa_i, \tau_o, \kappa_o)$ to mean “Skorokhod-conformance clean w.r.t. $\langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \tau_i, \tau_o \rangle$ ”.

Corollary 3.116. For all $\tau_i, \tau_o, \kappa_i, \kappa_o \in \overline{\mathbb{R}}_{\geq 0}$, the following implications hold:

1. $\text{RobustClean}(\kappa_i, \kappa_o) \Rightarrow \text{SkorClean}(0, \kappa_i, \tau_o, \kappa_o) \Rightarrow \text{HybridClean}(0, \kappa_i, \tau_o, \kappa_o)$,
2. $\text{HybridClean}(\tau_i, \kappa_i, 0, \kappa_o) \Rightarrow \text{SkorClean}(\tau_i, \kappa_i, 0, \kappa_o) \Rightarrow \text{RobustClean}(\kappa_i, \kappa_o)$.

Also, $\text{RobustClean}(\kappa_i, \kappa_o) = \text{SkorClean}(0, \kappa_i, 0, \kappa_o) = \text{HybridClean}(0, \kappa_i, 0, \kappa_o)$ and hence SkorClean and HybridClean are conservative extensions of robust cleanness.

Corollary 3.117. For all $\kappa_i, \kappa'_i, \kappa_o, \kappa'_o, \tau_i, \tau'_i, \tau_o, \tau'_o$ that satisfy the inequalities $\kappa'_i \leq \kappa_i, \tau'_i \leq \tau_i, \kappa'_o \geq \kappa_o, \tau'_o \geq \tau_o$ the following implications hold:

1. $\text{RobustClean}(\kappa_i, \kappa_o) \Rightarrow \text{RobustClean}(\kappa'_i, \kappa'_o)$,
2. $\text{HybridClean}(\kappa_i, \tau_i, \kappa_o, \tau_o) \Rightarrow \text{HybridClean}(\kappa'_i, \tau'_i, \kappa'_o, \tau'_o)$, and
3. $\text{SkorClean}(\kappa_i, \tau_i, \kappa_o, \tau_o) \Rightarrow \text{SkorClean}(\kappa'_i, \tau'_i, \kappa'_o, \tau'_o)$.

3.4.2 Synchronised Retiming

An intuitive and useful notion of doping cleanness should capture precisely what we expect from a clean system subject to disturbances in time and value. In this regard, the very strict Skorokhod cleanness has certain drawbacks, as the following example demonstrates.

Example 3.118. Figure 3.9 shows another example of inputs and outputs related to the emission cleaning system of a fictitious car. The left plot shows a standard input i_{st} and a non-standard input i_{dev} with $i_{dev}(t) = i_{st}(t - \tau_i)$, i.e., the two inputs are identical up to time shift τ_i to the right. Fictitious outputs for i_{st} and i_{dev} are shown in the right plot. Assume that for i_{st} , the vehicle shows the output (emission) profile o_{st} . For i_{dev} , consider two possible output trajectories: one output is $o_{dev}(t) = o_{st}(t - \tau_i)$, i.e., it is shifted in the same manner as the input; this is assumed to be the best response to i_{dev} . The other output is of

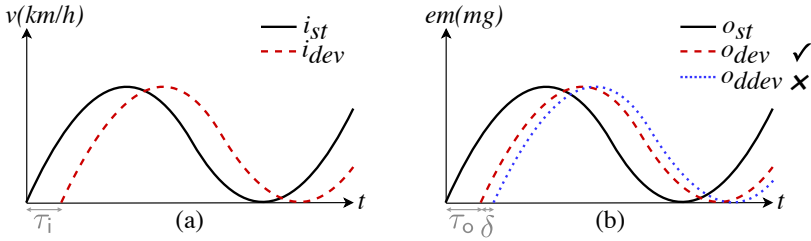


Figure 3.9: Imprecision problem of Skorokhod-conformance cleanliness without synchronisation of retimings. Inputs are depicted in (a), outputs are shown in (b). For $\tau_i = \tau_o$, Skorokhod-conformance cleanliness considers the combination of o_{st} and o_{dev} as clean, but the combination of o_{st} and o_{ddev} as doped.

the form $o_{ddev}(t) = o_{st}(t - \tau_i - \delta)$, where $\delta > 0$ can be arbitrarily small; this output is the optimal output with an arbitrary small, additional shift to the right. Skorokhod-conformance cleanliness with $\tau_i = \tau_o$ would accept output o_{dev} , but it would reject o_{ddev} , because Skorokhod cleanliness does not allow outputs for i_{dev} to deviate any further to the right than the τ_i margin that has already been used up by the input deviation. A potential solution could be to increase the value of τ_o so that it is significantly larger than τ_i , but this increases the imprecision by accepting also trajectories shifted far to the left from the optimal output o_{dev} .

Intuitively, when considering a pair of inputs that is shifted by some τ_i , we would like to adjust the corresponding output trajectories accordingly before comparing them. In the example above, one would therefore ideally like to check conformance of output o_{st} against τ_i -shifted o_{dev} and o_{ddev} , rather than the original, unshifted o_{dev} and o_{ddev} .

To couple input retimings and output retimings, we propose a cleanliness definition that checks for the two outputs involved in the cleanliness check, whether each of the outputs and the retimed other output satisfy a conformance predicate. The best retiming to adjust the output trajectories based on an input retiming may not necessarily be identical to the input retiming. We account for that by adding a *retiming synchronisation function* as an additional component to cleanliness contracts. This function specifies the output adjusting retiming for a concrete input retiming. The cleanliness definition then applies a retiming to the output trajectories twice: first, in reaction to the input retiming (e.g., the τ_i shift in the example) and second, the retiming that is applied by the output conformance predicate (e.g., the δ shift in the example).

The function $\text{WitConf}_{d,\tau,\epsilon}^{\text{Ret}}$ below is an extension of the predicate $\text{Conf}_{d,\tau,\epsilon}^{\text{Ret}}$ from Definition 3.107. Instead of just saying “yes” or “no” to whether two traces are conformant, it returns a set of retimings witnessing such conformance. This set is non-empty if and only if the two traces are conformant.

Definition 3.119. The *conformance witness function* for a family of retimings Ret , a distance function d , time distance threshold τ , value distance threshold ϵ , and two GTTs μ_1 and μ_2 is the function

$$\begin{aligned} \text{WitConf}_{d,\tau,\epsilon}^{\text{Ret}}(\mu_1, \mu_2) := & \{(r_1, r_2) \in \text{Ret}_\tau(\mathcal{T}_1, \mathcal{T}_2) \mid \\ & \forall t \in \mathcal{T}_1 : d(\mu_1(t) - \mu_2 \circ r_1(t)) \leq \epsilon \wedge \\ & \forall t \in \mathcal{T}_2 : d(\mu_1 \circ r_2(t) - \mu_2(t)) \leq \epsilon\}. \end{aligned}$$

Similarly, the function $\text{PrefWit}_\tau^{\text{WitConf}}$ returns the set of retimings witnessing that $\text{Pref}_\tau^{\text{Conf}}$ from Definition 3.109 holds.

Definition 3.120. Let WitConf be a conformance witness function, and τ a time distance threshold. Then, the set of witnesses for prefix-conformance of two GTTs μ_1 and μ_2 , and time t is defined by the function

$$\begin{aligned} \text{PrefWit}_\tau^{\text{WitConf}}(\mu_1, \mu_2, t) := & \{(r_1, r_2) \in \text{WitConf}(\mu_1[t_1^s..t_1^e], \mu_2[t_2^s..t_2^e]) \mid \\ & t_1^s \in [0, \tau] \cap \mathcal{T}_1, t_1^e \in [t - \tau, t + \tau] \cap \mathcal{T}_1, \\ & t_2^s \in [0, \tau] \cap \mathcal{T}_2, t_2^e \in [t - \tau, t + \tau] \cap \mathcal{T}_2\}. \end{aligned}$$

We remark that the domains of the retimings in $\text{PrefWit}_\tau^{\text{WitConf}}(\mu_1, \mu_2, t)$ can be smaller than the domains of μ_1 and μ_2 .

The adjustment of the output traces is realised through a synchronisation function Sync that specifies for an input retiming $r \in \mathfrak{R}(\mathcal{T}_1, \mathcal{T}_2)$ a set $\text{Ret} \subseteq \mathfrak{R}(\mathcal{T}_1, \mathcal{T}_2)$ of output retimings that may be used to adjust the output traces to appropriately react to the input retiming r . We add the function Sync to the cleanness contracts for conformance-based cleanness with synchronisation. Hence, contracts are of the form $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{In}}, \text{Ret}_{\text{Out}}, \tau_i, \tau_o, \text{Sync} \rangle$.

To expand conf-cleanness as defined in Definition 3.110 by synchronised retiming, the definition must change after the premise that enforces input conformance of two inputs i_1 and i_2 up to time t . For valid pairs of inputs the definition must extract a retiming (r_1, r_2) for i_1 , i_2 and t using $\text{PrefWit}_{\tau_i}^{\text{ConfIn}}$. Then, it passes this retiming to Sync to obtain a retiming (r'_1, r'_2) to adjust the output traces accordingly. Finally, the definition checks the conformance of the retimed outputs according to the output conformance predicate. Analogue to the definition of conformance notions in Definition 3.107, the output predicate check $\text{Pref}_{\tau_o}^{\text{ConfOut}}(o_1, o_2, t)$ expands to the conjunction of $\text{Pref}_{\tau_o}^{\text{ConfOut}}(o_1, o_2 \circ r'_1, t)$ and $\text{Pref}_{\tau_o}^{\text{ConfOut}}(o_1 \circ r'_2, o_2, t)$.

Definition 3.121. Let $H : \text{In}^\vartheta \rightarrow 2^{(\text{Out}^\vartheta)}$ be a hybrid system and assume that $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{In}}, \text{Ret}_{\text{Out}}, \tau_i, \tau_o, \text{Sync} \rangle$ is a cleanness contract. Let further be $\text{Conf}_{\text{In}} = \text{Conf}_{d_{\text{In}}, \tau_i, \kappa_i}^{\text{Ret}_{\text{In}}}$ and $\text{Conf}_{\text{Out}} = \text{Conf}_{d_{\text{Out}}, \tau_o, \kappa_o}^{\text{Ret}_{\text{Out}}}$ the conformance notions for inputs and, respectively, outputs obtained from \mathcal{C} . Then, H is *sync-conf-clean* w.r.t. \mathcal{C} , if and only if

$$\forall i_1 \in \text{StdIn}. \forall i_2 \in \text{In}^\vartheta. \forall t \in \text{dom}(i_1) \cup \text{dom}(i_2).$$

$$(\forall t' \leq t. \text{Pref}_{\tau_i}^{\text{Conf}_{\text{In}}}(i_1, i_2, t')) \Rightarrow$$

$$\exists (r_1, r_2) \in \text{PrefWit}_{\tau_i}^{\text{Conf}_{\text{In}}}(i_1, i_2, t). \exists (r'_1, r'_2) \in \text{Sync}(r_1, r_2).$$

$$\begin{aligned} & ((\forall o_1 \in H(i_1). \exists o_2 \in H(i_2). \text{Pref}_{\tau_o}^{\text{Conf}_{\text{Out}}}(o_1 \circ r'_2, o_2, t) \wedge \text{Pref}_{\tau_o}^{\text{Conf}_{\text{Out}}}(o_1, o_2 \circ r'_1, t)) \wedge \\ & (\forall o_2 \in H(i_2). \exists o_1 \in H(i_1). \text{Pref}_{\tau_o}^{\text{Conf}_{\text{Out}}}(o_1 \circ r'_2, o_2, t) \wedge \text{Pref}_{\tau_o}^{\text{Conf}_{\text{Out}}}(o_1, o_2 \circ r'_1, t))). \end{aligned}$$

An obvious default choice for Sync is to apply the same retiming for the output that has been used for the input, i.e., $\text{Sync}(r_1, r_2) := \{(r_1, r_2)\}$.

Example 3.122. With sync-conf-cleanness, there is an elegant solution to the problem explained in Example 3.118. When using Skorokhod conformance with time threshold τ_i for inputs, the synchronisation function $\text{Sync}(r_1, r_2) := \{(r_1, r_2)\}$ lets the outputs \mathbf{o}_{ddev} shifted by τ_i to the left and \mathbf{o}_{st} be Skorokhod conformant with time threshold δ . Obviously, with $r_1(t) = t + \tau$ and $r_2(t) = t - \tau$, (r_1, r_2) is a retiming that can be obtained from $\text{PrefWit}_{\tau_i}^{\text{Conf}_{\text{In}}}$. Because of the synchronisation, \mathbf{o}_{st} and \mathbf{o}_{ddev} are retimed before the conformance check, such that actually, $\mathbf{o}_{st}(t - \tau_i)$ and $\mathbf{o}_{ddev}(t)$ ($= \mathbf{o}_{st}(t - \tau_i - \delta)$) are compared, respectively, $\mathbf{o}_{st}(t)$ and $\mathbf{o}_{ddev}(t + \tau_i)$ ($= \mathbf{o}_{st}(t - \delta)$).

In certain scenarios, synchronisation functions that deviate from the above advertised default function are better suited. For example, although the emission cleaning system of a car reacts immediately to the inputs it receives and adjusts the physical or chemical processes inside the car in a reasonable way, the effects may be observable only with a certain delay. Hence, it would be reasonable to incorporate the average delay into the synchronisation function. Another use case of a non-default synchronisation function are scenarios in which the timelines for the inputs and the outputs have different scales. Then, Sync can ensure that the translation from the input timeline to the output timeline is done correctly. In general, the definition of Sync allows to incorporate any available knowledge regarding the expected output behaviours for conforming input trajectories.

It is worth to mention, that sync-conf-cleanness is a generalisation of conf-cleanness, as the following proposition shows.

Proposition 3.123. Let $H : \text{In}^\vartheta \rightarrow 2^{(\text{Out}^\vartheta)}$ be a hybrid system and consider the conf-cleanness contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{In}}, \text{Ret}_{\text{Out}}, \tau_i, \tau_o \rangle$. Let

further be $\text{Sync}(r_1, r_2) := (\text{id}, \text{id})$ be the synchronisation function that does not apply any retiming to output traces prior to the output conformance check, and let $\mathcal{C}' = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{In}}, \text{Ret}_{\text{Out}}, \tau_i, \tau_o, \text{Sync} \rangle$ be the cleanness contract for sync-conf-cleanness that is identical to \mathcal{C} up to the additional component Sync . Then, H is conf-clean w.r.t. \mathcal{C} if and only if H is sync-conf-clean w.r.t. \mathcal{C}' .

3.5 Summary

In this chapter, we provide cleanness definitions for sequential programs, reactive systems, mixed-IO systems and hybrid systems. For sequential programs in Section 3.1 and reactive systems in Section 3.2 we consider both parametrised and non-parametrised variants of strict cleanness, robust cleanness and func-cleanness. The constraints defined by robust cleanness and func-cleanness may be seen as a combination of two sub-constraints that define a *lower bound* and *upper bound* on the allowed (non-standard) software behaviour. These constraints are explicitly captured by l-cleanness, respectively u-cleanness definitions. In Section 3.2.4 we additionally provide for reactive systems trace integral variants of robust cleanness and func-cleanness, which are stricter than their original counterparts and are particularly useful for contracts that entail past-forgetful distance functions. Table 3.1 provides an overview of all cleanness notions for sequential programs and reactive systems.

In Section 3.3 we demonstrate how robust and func-cleanness can be adapted for mixed-IO systems. The resulting definitions are summarised in Table 3.2.

Finally, in Section 3.4 covering hybrid systems, we propose an appropriate variant of robust cleanness (Definition 3.101 on Page 91), and additionally cleanness notions based on conformance relations, namely conf-cleanness (Definition 3.110 on Page 97) and sync-conf-cleanness (Definition 3.121 on Page 103).

3.6 Related Work & Contributions

The term “software doping” was coined by the press in 2015 after the Volkswagen exhaust emissions scandal, when it surfaced that software developers introduced code intended to deceive [67]. Shortly after, a special session at ISOLA 2016 was devoted to this topic [88]. Baum [14] addresses the problem from a philosophical point of view and elaborates on the ethics of it. Barthe et al. [11] provided a first discussion of the problem and some informal characterisations that were very influential for some cleanness definitions in this thesis. Though all these works point out the need for a technical attack on the problem, none of them provide a formal proposal.

cleanness	sequential		reactive	
	parametrised	non-parametrised	parametrised	non-parametrised
strict	Def. 3.1 Page 39	Def. 3.2 Page 40	Def. 3.40 Page 61	Def. 3.41 Page 61
robust	Def. 3.6 Page 43	Def. 3.17 Page 48	Def. 3.44 Page 63	Def. 3.49 Page 65
l-robust	Def. 3.7 Page 44	Def. 3.20 Page 51	Def. 3.45 Page 64	Def. 3.54 Page 70
u-robust	Def. 3.8 Page 44	Def. 3.21 Page 51	Def. 3.46 Page 64	Def. 3.55 Page 70
robust trace integral			Def. 3.76 Page 80	Def. 3.79 Page 81
l-robust trace integral			Def. 3.74 Page 80	Def. 3.77 Page 81
u-robust trace integral			Def. 3.75 Page 80	Def. 3.78 Page 81
func	Def. 3.27 Page 54	Def. 3.32 Page 56	Def. 3.60 Page 73	Def. 3.65 Page 74
l-func	Def. 3.28 Page 54	Def. 3.35 Page 58	Def. 3.61 Page 73	Def. 3.69 Page 77
u-func	Def. 3.29 Page 54	Def. 3.36 Page 59	Def. 3.62 Page 73	Def. 3.70 Page 77
func trace integral			Def. 3.82 Page 81	Def. 3.85 Page 82
l-func trace integral			Def. 3.80 Page 81	Def. 3.83 Page 81
u-func trace integral			Def. 3.81 Page 81	Def. 3.84 Page 82

Table 3.1: Overview of cleanness notions for sequential and reactive programs.

cleanness	robust	robust (trace integral)	func	func (trace integral)
l-	Def. 3.88	Def. 3.95	Def. 3.92	Def. 3.98
	Page 85	Page 89	Page 89	Page 90
u-	Def. 3.90	Def. 3.96	Def. 3.93	Def. 3.99
	Page 88	Page 89	Page 89	Page 90
both	Def. 3.91	Def. 3.97	Def. 3.94	Def. 3.100
	Page 88	Page 90	Page 89	Page 90

Table 3.2: Overview of cleanness notions for mixed-IO systems.

Similar to software doping, backdoored software is a class of software that does not act in the best interest of users; see for instance the recent analysis in [106]. The primary emphasis of backdoored software is on leaking confidential information while guaranteeing functionality.

Cleanness in sequential programs is related to abstract non-interference [12, 62]. More generally, our notions of cleanness are hyperproperties [35], a general class that encompasses notions across different domains, in particular non-interference in security [72, 116].

Some of the definitions we will propose share conceptual similarities to continuity properties [30, 65] of programs, and stability and robustness notions for cyber-physical systems [115, 49, 86, 101]. Absence of discontinuities alone does not provide an adequate guarantee of cleanness. This is because physical outputs (e.g. the amount of nitric oxides in the exhaust stream of a car) usually do change continuously. For instance, a doped car may alter its emission cleaning in a discrete way, but that induces a (rapid but) continuous change of nitric oxides concentrations. Notions of stability and robustness assure the outputs to stabilise despite transient input disturbances. Our definitions will be based on comparison of two different inputs; the second input does not necessarily need to be a disturbance of the first input.

The contents in Sections 3.1 and 3.2 are based on [42]. The cleanness definitions for parametrised programs and systems in these sections are developed jointly by my co-authors and by myself. New in this thesis are the trace integral definitions, which become important in the remaining chapters. The definitions for non-parametrised strict cleanness of sequential programs and a quantifier-based definition of robust cleanness of reactive systems have been published in [17, 18]. All other (non-parametrised) cleanness definitions in Sections 3.1 and 3.2 are new and have not been published before. The same holds for all proofs in Sections 3.1 and 3.2.

The concept of mixed-IO computation models in Section 3.3 has its origins in the model-based testing theory by Tretmans [119] that assumes systems to be modelled as labelled transition systems. We used model-based testing in [17, 18] and the “mixed-IO model” has been first mentioned explicitly in [25]. There, we introduced the mixed-IO model to avoid talking about LTS and to reason instead directly over traces (irrespective of whether they are obtained from an LTS or not). The cleanness definitions in Section 3.3 are inspired by those for LTS [17, 18], but they are foundationally different (also to the definitions in [25]) in the sense that in these publications robust cleanness is defined for cleanness contexts (with a fixed standard behaviour; this will be explained in Chapter 5), while in this thesis, the cleanness notions are defined w.r.t. a cleanness contract in its general form (i.e., for standard inputs instead of standard inputs *and* outputs). Moreover, Section 3.3 proposes func-cleanness and trace integral variants of robust cleanness and func-cleanness for mixed-IO systems for the first time.

The content regarding hybrid systems in Section 3.4 is based on [46, 20]. The cleanness definitions in this section are a result of the cooperation with my co-authors of these publications. In this thesis I further developed these definitions so that they are consistent with the definitions in Sections 3.1 to 3.3. In particular, I introduce the distinction between input and output domain, I add the set of standard inputs to the contracts, I consistently make cleanness definitions suitable for nondeterministic systems, and more.

4 Model-Aware Software Doping Analysis

The definitions introduced in Chapter 3 offer a rich portfolio of cleanness notions for programs and systems. It depends on the concrete scenario, which of these cleanness notions is most suitable. In this chapter, we present techniques to verify that a program or system satisfies a certain cleanness notion. We start with the analysis of sequential programs using a self-composition technique. For reactive systems, we formalise cleanness in the logic HyperLTL and demonstrate how the toy examples from Examples 3.50 and 3.66 can be model-checked using this formalisation.

4.1 Analysis through self-composition

To analyse sequential programs, we consider a program P as a state transformer on states $\eta : \mathbf{Var} \rightarrow \mathbf{Val}$ mapping variables to values; see Section 2.5. Let $\vec{x}_p = (x_p^{(1)}, \dots, x_p^{(n)})$ be a vector of variables representing parameters, $\vec{x}_i = (x_i^{(1)}, \dots, x_i^{(m)})$ a vector of variables representing inputs, and $\vec{x}_o = (x_o^{(1)}, \dots, x_o^{(r)})$ a vector of variables representing outputs. We denote by \vec{x} the vector of all variables, i.e., $\vec{x} = (x_p^{(1)}, \dots, x_p^{(n)}, x_i^{(1)}, \dots, x_i^{(m)}, x_o^{(1)}, \dots, x_o^{(r)})$. Thus, in this section, \mathbf{Var} is the set of all variables in \vec{x} . We assume that all variables are pairwise different and, hence, that $|\mathbf{Var}| = n + m + r$. For any vector \vec{y} , we denote by $|\vec{y}|$ the number of variables in \vec{y} . In particular, $|\vec{x}_p| = n$, $|\vec{x}_i| = m$ and $|\vec{x}_o| = r$. To consistently use the notations **Param**, **In** and **Out**, we define $\mathbf{Param} = \mathbf{Val}^{|\vec{x}_p|}$, $\mathbf{In} = \mathbf{Val}^{|\vec{x}_i|}$ and $\mathbf{Out} = \mathbf{Val}^{|\vec{x}_o|}$ to be sets of \mathbf{Val} -vectors of lengths n , m and r , respectively.

For two states η_1 and η_2 we will use the equality predicate $\eta_1 =_V \eta_2$ that holds if and only if for every $v \in V$, $\eta_1(v) = \eta_2(v)$. For some state η and a vector of variables $V = (v_1, \dots, v_l)$, we define $\eta(V) := (\eta(v_1), \dots, \eta(v_l))$.

To define strict cleanness for a state transformer P , we consider cleanness contracts $\mathcal{C} = \langle \approx_P, \mathbf{StdIn} \rangle$, where the predicate \approx_P encodes a binary relation on parameter vectors from **Param** and **StdIn** is a predicate over input vectors from **In**. Strict cleanness for deterministic sequential programs can then be reformulated as follows.

Definition 4.1. A sequential and deterministic program P is strictly clean w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn} \rangle$ if and only if for all states η_1, η_2 and η'_1 such that $\eta_1(\vec{x}_p) \approx_P \eta_2(\vec{x}_p)$, $\text{StdIn}(\eta_1(\vec{x}_i))$, $\eta_1 =_{\vec{x}_i} \eta_2$ and $(P, \eta_1) \Downarrow \eta'_1$, there exists a state η'_2 , such that $(P, \eta_2) \Downarrow \eta'_2$ and $\eta'_1 =_{\vec{x}_o} \eta'_2$.

The definition is essentially a notation change of (a deterministic version of) Definition 3.1. In contrast to the sequential programs in Section 3.1, state transformers might not terminate. Due to its symmetry, strict cleanness ensures that P terminates for initial state η_1 if and only if it terminates for initial state η_2 . If P does not terminate for η_1 , then strict cleanness is satisfied for this combination of η_1 and η_2 . But it would be violated for the combination η_2 and η_1 (i.e., switching the roles of η_1 and η_2): Since $(P, \eta_2) \Downarrow \eta'_2$, there must exist a state η'_1 such that $(P, \eta_1) \Downarrow \eta'_1$, but since P does not terminate for η_1 , such η'_1 does not exist and, hence, P is not strictly clean.

For strict cleanness, we define the indistinguishable criteria (cf. Section 2.5 and [12, Section 3])

$$\begin{aligned} \mathcal{I} &= \{(\eta_1, \eta_2) \mid \eta_1(\vec{x}_p) \approx_P \eta_2(\vec{x}_p) \wedge \text{StdIn}(\eta_1(\vec{x}_i)) \wedge \eta_1 =_{\vec{x}_i} \eta_2\} \text{ and} \\ \mathcal{I}' &= \{(\eta_1, \eta_2) \mid \eta_1 =_{\vec{x}_o} \eta_2\}. \end{aligned}$$

Thus, Definition 4.1 characterises termination-sensitive $(\mathcal{I}, \mathcal{I}')$ -security. From Proposition 2.6 we get that strict cleanness of deterministic sequential programs can be analysed using Dijkstra's weakest precondition (wp) through self-composition.

Proposition 4.2. A deterministic sequential program P is strictly clean if and only if

$$\vec{x}_p \approx_P \vec{x}'_p \wedge \text{StdIn}(\vec{x}'_i) \wedge \vec{x}_i = \vec{x}'_i \wedge \text{wp}(P, \text{true}) \Rightarrow \text{wp}(P; P[\vec{x}/\vec{x}'], \vec{x}_o = \vec{x}'_o).$$

The term $\text{wp}(P, \text{true})$ in the antecedent of the implication is the weakest precondition that ensures that program P terminates. It is necessary in the predicate, otherwise it could become false only because program P does not terminate.

To analyse robust cleanness, let $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be a contract containing, in addition to the components that are already in the strict cleanness contract, distance functions $d_{\text{In}} : \text{In} \times \text{In} \rightarrow \overline{\mathbb{R}}_{\geq 0}$ and $d_{\text{Out}} : \text{Out} \times \text{Out} \rightarrow \overline{\mathbb{R}}_{\geq 0}$, and thresholds $\kappa_i \in \overline{\mathbb{R}}_{\geq 0}$ for inputs and $\kappa_o \in \overline{\mathbb{R}}_{\geq 0}$ for outputs. The following definition of robust cleanness for state transformers corresponds to the definition of robust cleanness for sequential programs (Definition 3.6):

Definition 4.3. A sequential and deterministic program P is robustly clean if and only if for all states η_1, η_2 , and η' such that $\eta_1(\vec{x}_p) \approx_P \eta_2(\vec{x}_p)$, $\text{StdIn}(\eta_1(\vec{x}_i))$, and $d_{\text{In}}(\eta_1(\vec{x}_i), \eta_2(\vec{x}_i)) \leq \kappa_i$, the following two conditions hold:

1. if $(P, \eta_1) \Downarrow \eta'$, then there exists a state η'_2 , such that $(P, \eta_2) \Downarrow \eta'_2$ and $d_{\text{Out}}(\eta'(\vec{x}_o), \eta'_2(\vec{x}_o)) \leq \kappa_o$; and
2. if $(P, \eta_2) \Downarrow \eta'$, then there exists a state η'_1 , such that $(P, \eta_1) \Downarrow \eta'_1$ and $d_{\text{Out}}(\eta'_1(\vec{x}_o), \eta'(\vec{x}_o)) \leq \kappa_o$.

In contrast to strict cleanness, this definition is not symmetric, because only the input of η_1 is required to be a standard input. Thus, to ensure that (P, η_1) terminates if and only if (P, η_2) terminates, there are two conditions in Definition 4.3.

For robust cleanness, we use the following indistinguishability criteria.

$$\begin{aligned} \mathcal{I} &= \{(\eta_1, \eta_2) \mid \eta_1(\vec{x}_p) \approx_P \eta_2(\vec{x}_p) \wedge \text{StdIn}(\eta_1(\vec{x}_i)) \wedge d_{\text{In}}(\eta_1(\vec{x}_i), \eta_2(\vec{x}_i)) \leq \kappa_i\} \\ \mathcal{I}' &= \{(\eta_1, \eta_2) \mid d_{\text{Out}}(\eta_1(\vec{x}_o), \eta_2(\vec{x}_o)) \leq \kappa_o\} \end{aligned}$$

As Definition 4.3, the relation \mathcal{I} is not symmetric, i.e., there exist η_1 and η_2 with $(\eta_1, \eta_2) \in \mathcal{I}$ and $(\eta_2, \eta_1) \notin \mathcal{I}$. In particular, it turns out that the first condition of Definition 4.3 defines termination-sensitive $(\mathcal{I}, \mathcal{I}')$ -security, and the second condition defines termination-sensitive $(\mathcal{I}^{-1}, \mathcal{I}')$ -security. Finally, we get from Proposition 2.6 that robust cleanness can be analysed using wp through self-composition.

Proposition 4.4. A deterministic program P is robustly clean if and only if

$$\begin{aligned} &\vec{x}_p \approx_P \vec{x}'_p \wedge \text{StdIn}(\vec{x}_i) \wedge d_{\text{In}}(\vec{x}_i, \vec{x}'_i) \leq \kappa_i \\ &\Rightarrow \left(\begin{array}{ll} \text{wp}(P, \text{true}) & \Rightarrow \text{wp}(P; P[\vec{x}/\vec{x}'], d_{\text{Out}}(\vec{x}_o, \vec{x}'_o) \leq \kappa_o) \\ \wedge \text{wp}(P[\vec{x}/\vec{x}'], \text{true}) & \Rightarrow \text{wp}(P[\vec{x}/\vec{x}']; P, d_{\text{Out}}(\vec{x}_o, \vec{x}'_o) \leq \kappa_o) \end{array} \right) \end{aligned}$$

Proceeding in a similar manner, we can also obtain a definition of func-cleanness for deterministic state transformers. For func-cleanness a contract is a tuple $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$; it is similar to contracts for robust cleanness, but the distance thresholds are replaced by the function $f : \overline{\mathbb{R}}_{\geq 0} \rightarrow \overline{\mathbb{R}}_{\geq 0}$ to relate input distances and output distances.

Definition 4.5. A sequential and deterministic program P is func-clean w.r.t. a contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if for all states η_1, η_2 , and η' such that $\eta_1(\vec{x}_p) \approx_P \eta_2(\vec{x}_p)$, $\text{StdIn}(\eta_1(\vec{x}_i))$, the following two conditions hold:

1. if $(P, \eta_1) \Downarrow \eta'$, then there exists a state η'_2 , such that $(P, \eta_2) \Downarrow \eta'_2$ and $d_{\text{Out}}(\eta'(\vec{x}_o), \eta'_2(\vec{x}_o)) \leq f(d_{\text{In}}(\eta_1(\vec{x}_i), \eta_2(\vec{x}_i)))$; and
2. if $(P, \eta_2) \Downarrow \eta'$, then there exists a state η'_1 , such that $(P, \eta_1) \Downarrow \eta'_1$ and $d_{\text{Out}}(\eta'_1(\vec{x}_o), \eta'(\vec{x}_o)) \leq f(d_{\text{In}}(\eta_1(\vec{x}_i), \eta_2(\vec{x}_i)))$.

Here, the term $f(d_{\text{In}}(\eta_1(\vec{x}_i), \eta_2(\vec{x}_i)))$ appears in the conclusion of the implications of both items. This may look unexpected since it seems to be related to the input requirements rather than the output requirements, because it refers to the input states. To overcome this situation, we introduce a parameter $Y \in \overline{\mathbb{R}}_{\geq 0}$ which we assume universally quantified. Using this, we define the following indistinguishability criteria

$$\begin{aligned} \mathcal{I}_Y &= \{(\eta_1, \eta_2) \mid \eta_1(\vec{x}_p) \approx_P \eta_2(\vec{x}_p) \wedge \text{StdIn}(\eta_1(\vec{x}_i)) \wedge f(d_{\text{In}}(\eta_1(\vec{x}_i), \eta_2(\vec{x}_i))) = Y\} \\ \mathcal{I}'_Y &= \{(\eta_1, \eta_2) \mid d_{\text{Out}}(\eta_1(\vec{x}_o), \eta_2(\vec{x}_o)) \leq Y\} \end{aligned}$$

We can reformulate Definition 4.5 using the indistinguishability criteria: P is func-clean w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if for every $Y \in \overline{\mathbb{R}}_{\geq 0}$, and for all states η_1, η_2 , and η' such that $(\eta_1, \eta_2) \in \mathcal{I}_Y$

1. if $(P, \eta_1) \Downarrow \eta'$, then there exists some state η'_2 , such that $(P, \eta_2) \Downarrow \eta'_2$ and $(\eta', \eta'_2) \in \mathcal{I}'_Y$; and
2. if $(P, \eta_2) \Downarrow \eta'$, then there exists some state η'_1 , such that $(P, \eta_1) \Downarrow \eta'_1$ and $(\eta'_1, \eta) \in \mathcal{I}'_Y$.

With this new definition, and taking into account again the non-symmetry of \mathcal{I}_Y , the first item characterises termination-sensitive $(\mathcal{I}_Y, \mathcal{I}'_Y)$ -security while the second one characterises termination-sensitive $(\mathcal{I}_Y^{-1}, \mathcal{I}'_Y)$ -security. From this and Proposition 2.6 it follows that the property of func-cleanness can be analysed using wp and self-composition.

Proposition 4.6. A deterministic program P is func-clean w.r.t. contract $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if for all $Y \in \overline{\mathbb{R}}_{\geq 0}$

$$\begin{aligned} \vec{x}_p \approx_P \vec{x}'_p \wedge \text{StdIn}(\vec{x}_i) \wedge f(d_{\text{In}}(\vec{x}_i, \vec{x}'_i)) = Y \\ \Rightarrow \left(\begin{array}{ll} \text{wp}(P, \text{true}) & \Rightarrow \text{wp}(P; P[\vec{x}/\vec{x}'], d_{\text{Out}}(\vec{x}_o, \vec{x}'_o) \leq Y) \\ \wedge \text{wp}(P[\vec{x}/\vec{x}'], \text{true}) & \Rightarrow \text{wp}(P[\vec{x}/\vec{x}']; P, d_{\text{Out}}(\vec{x}_o, \vec{x}'_o) \leq Y) \end{array} \right) \end{aligned}$$

Example 4.7. In this example, we use Proposition 4.6 to prove correct the statements in Example 3.39. The programs shown in Figures 4.1 and 4.2 are equivalent to the programs in Figures 3.4 and 3.5 and are syntactically adapted to match the language used for the wp calculus. The programs are not explicitly parametrised, hence we assume a single parameter $()$ that is equivalent to itself, i.e., $() \approx_P ()$. Using the same StdIn , d_{In} , d_{Out} and f as in Example 3.39, we get the predicates $\text{StdIn}(\text{thrtl}) \Leftrightarrow (\text{thrtl} \in (0, 1])$, $f(d_{\text{In}}(\text{thrtl}, \text{thrtl}')) = Y \Leftrightarrow \frac{|\text{thrtl} - \text{thrtl}'|}{2} = Y$, and $d_{\text{Out}}(\text{NOx}, \text{NOx}') \leq Y \Leftrightarrow |\text{NOx} - \text{NOx}'| \leq Y$.

```

def_dose := thrtl2
NOx := thrtl3 / (2 · def_dose)

```

Figure 4.1: Program EC

```

if thrtl ∈ ThrottleTestValues then
  def_dose := thrtl2
else
  def_dose := thrtl
end if
NOx := thrtl3 / (2 · def_dose)

```

Figure 4.2: Program AEC

To proof func-cleanness of EC, let EC' be another instance of EC with every program variable x renamed to x' . First observe that $\text{wp}(\text{EC}, \text{true}) = \text{true}$ and $\text{wp}(\text{EC}', \text{true}) = \text{true}$, i.e., EC and EC' always terminate. It remains to show that

$$\text{thrtl} \in (0, 1] \wedge \left(\frac{|\text{thrtl} - \text{thrtl}'|}{2} = Y \right) \Rightarrow \left(\begin{array}{l} \text{wp}(\text{EC}; \text{EC}', |\text{NOx} - \text{NOx}'| \leq Y) \\ \wedge \text{wp}(\text{EC}'; \text{EC}, |\text{NOx} - \text{NOx}'| \leq Y) \end{array} \right)$$

It is easy to verify that

$$\text{wp}(\text{EC}; \text{EC}', |\text{NOx} - \text{NOx}'| \leq Y) \equiv \left(\frac{|\text{thrtl} - \text{thrtl}'|}{2} \leq Y \right)$$

and

$$\text{wp}(\text{EC}'; \text{EC}, |\text{NOx} - \text{NOx}'| \leq Y) \equiv \left(\frac{|\text{thrtl}' - \text{thrtl}|}{2} \leq Y \right)$$

from which the implication follows and hence EC is func-clean.

Let AEC' be another instance of AEC with every program variable x renamed to x' . Again, $\text{wp}(\text{AEC}, \text{true}) = \text{true}$ and $\text{wp}(\text{AEC}', \text{true}) = \text{true}$. It is easy to verify that $\text{wp}(\text{AEC}; \text{AEC}', |\text{NOx} - \text{NOx}'| \leq Y)$ is equivalent to

$$\begin{aligned} (\text{thrtl} \in (0, 1] \wedge \text{thrtl}' \in (0, 1]) &\Rightarrow \frac{|\text{thrtl} - \text{thrtl}'|}{2} \leq Y \\ \wedge (\text{thrtl} \in (0, 1] \wedge \text{thrtl}' \notin (0, 1]) &\Rightarrow \frac{|\text{thrtl} - \text{thrtl}'^2|}{2} \leq Y \\ \wedge (\text{thrtl} \notin (0, 1] \wedge \text{thrtl}' \in (0, 1]) &\Rightarrow \frac{|\text{thrtl}^2 - \text{thrtl}'|}{2} \leq Y \\ \wedge (\text{thrtl} \notin (0, 1] \wedge \text{thrtl}' \notin (0, 1]) &\Rightarrow \frac{|\text{thrtl}^2 - \text{thrtl}'^2|}{2} \leq Y \end{aligned}$$

For $\text{wp}(\text{AEC}'; \text{AEC}, |\text{NOx} - \text{NOx}'| \leq Y)$, the predicate is semantically the same, since $|a - b| = |b - a|$. Then, the predicate

$$\text{thrtl} \in (0, 1] \wedge \left(\frac{|\text{thrtl} - \text{thrtl}'|}{2} = Y \right) \Rightarrow \left(\begin{array}{l} \text{wp}(\text{AEC}; \text{AEC}', |\text{NOx} - \text{NOx}'| \leq Y) \\ \wedge \text{wp}(\text{AEC}'; \text{AEC}, |\text{NOx} - \text{NOx}'| \leq Y) \end{array} \right)$$

is equivalent to

$$\left(\text{thrtl} \in (0, 1] \wedge \frac{|\text{thrtl} - \text{thrtl}'|}{2} = Y \right) \Rightarrow \left(\begin{array}{l} \text{thrtl}' \in (0, 1] \Rightarrow \frac{|\text{thrtl} - \text{thrtl}'|}{2} \leq Y \\ \wedge \text{thrtl}' \notin (0, 1] \Rightarrow \frac{|\text{thrtl} - \text{thrtl}'^2|}{2} \leq Y \end{array} \right)$$

which can be proved false if, e.g., $\text{thrtl} = 1$ and $\text{thrtl}' = 1.5$. Consequently, AEC is not func-clean.

Notwithstanding the simplicity of the previous example, the technique can be applied to complex programs including loops. We decided to keep it simple as it is not our intention to show the power of wp, but the applicability of our definition.

4.2 HyperLTL

In this section, we propose HyperLTL characterisations for strict cleanness, trace integral robust cleanness and trace integral func-cleanness of reactive systems.¹ HyperLTL reasons about sets of infinite traces over sets of atomic propositions. Thus, we assume that the state of a system can be represented by a Boolean combination of atomic propositions. Concretely, we consider the set $\text{AP} = \text{AP}_p \cup \text{AP}_i \cup \text{AP}_o$ of atomic propositions, where AP_p , AP_i , and AP_o are the atomic propositions that define the parameter values, the input values, and the output values, respectively. Thus, we take $\text{Param} = 2^{\text{AP}_p}$, $\text{In} = 2^{\text{AP}_i}$ and $\text{Out} = 2^{\text{AP}_o}$. The reactive behaviour of a system is represented as an infinite trace of sets of atomic propositions, i.e., a subset of $(2^{\text{AP}})^\omega$. To distinguish the AP encoded reactive systems from the function encoding in Section 3.2, we call the former *circuits* and denote them as $\mathcal{C} \subseteq (2^{\text{AP}})^\omega$. Still, there is a relation to function encoded reactive systems. \mathcal{C} can be interpreted as a function $\hat{S} : \text{Param} \rightarrow \text{In}^\omega \rightarrow 2^{(\text{Out}^\omega)}$ where

$$t \in \mathcal{C} \quad \text{if and only if} \quad (t \downarrow_{\text{AP}_o}) \in \hat{S}(t \downarrow_{\text{AP}_p}[0])(t \downarrow_{\text{AP}_i}), \quad (4.1)$$

with $t \downarrow_A$ defined by $(t \downarrow_A)[k] = t[k] \cap A$ for all $k \in \mathbb{N}$.

For the remainder of this chapter, we restrict contracts for robust cleanness and func-cleanness to provide distance functions that are past-forgetful and, therefore, consider the trace integral variants of the cleanness definitions (cf. Section 3.2.4).

¹The contents in this section deviate from previous work [42], which contains incorrect proofs claiming that the proposed HyperLTL formulas characterise non-trace-integral robust cleanness and func-cleanness.

In the HyperLTL formulas below occur, for convenience, non-atomic propositions. For the correctness of these formulas, we must assume that the translation of these non-atomic propositions satisfy the following properties:

Assumption 4.8. Let $\mathbf{C} \subseteq (2^{\text{AP}})^\omega$ be a circuit with $\text{AP} = \text{AP}_p \cup \text{AP}_i \cup \text{AP}_o$, let $t, t' \in \mathbf{C}$ be traces in this circuit, and let $\Pi = \{\pi := t, \pi' := t'\}$ be a trace assignment. Further, let $\mathbf{p} \in \text{Param}$, $i \in \text{In}$, and $o \in \text{Out}$. Then, the following equivalences must hold.

1. $\Pi \models_{\mathbf{C}} \mathbf{p}_\pi = \mathbf{p}_{\pi'}$ if and only if $t \downarrow_{\text{AP}_p}[0] = t' \downarrow_{\text{AP}_p}[0]$
2. $\Pi \models_{\mathbf{C}} i_\pi = i_{\pi'}$ if and only if $t \downarrow_{\text{AP}_i}[0] = t' \downarrow_{\text{AP}_i}[0]$
3. $\Pi \models_{\mathbf{C}} o_\pi = o_{\pi'}$ if and only if $t \downarrow_{\text{AP}_o}[0] = t' \downarrow_{\text{AP}_o}[0]$
4. $\Pi \models_{\mathbf{C}} \mathbf{p}_\pi = \mathbf{p}$ if and only if $t \downarrow_{\text{AP}_p}[0] = \mathbf{p}$
5. $\Pi \models_{\mathbf{C}} i_\pi = i$ if and only if $t \downarrow_{\text{AP}_i}[0] = i$
6. $\Pi \models_{\mathbf{C}} o_\pi = o$ if and only if $t \downarrow_{\text{AP}_o}[0] = o$
7. $\Pi \models_{\mathbf{C}} \mathbf{p}_\pi \approx_P \mathbf{p}_{\pi'}$ if and only if $t \downarrow_{\text{AP}_p}[0] \approx_P t' \downarrow_{\text{AP}_p}[0]$
8. $\Pi \models_{\mathbf{C}} d_{\text{In}}(i_\pi, i_{\pi'}) \leq \kappa_i$ if and only if $d_{\text{In}}(t \downarrow_{\text{AP}_i}[0], t' \downarrow_{\text{AP}_i}[0]) \leq \kappa_i$
9. $\Pi \models_{\mathbf{C}} d_{\text{Out}}(o_\pi, o_{\pi'}) \leq \kappa_o$ if and only if $d_{\text{Out}}(t \downarrow_{\text{AP}_o}[0], t' \downarrow_{\text{AP}_o}[0]) \leq \kappa_o$
10. $\Pi \models_{\mathbf{C}} d_{\text{Out}}(o_\pi, o_{\pi'}) \leq f(d_{\text{In}}(i_\pi, i_{\pi'}))$ if and only if $d_{\text{Out}}(t \downarrow_{\text{AP}_o}[0], t' \downarrow_{\text{AP}_o}[0]) \leq f(d_{\text{In}}(t \downarrow_{\text{AP}_i}[0], t' \downarrow_{\text{AP}_i}[0]))$

Figure 4.3 proposes a translations from the non-atomic propositions to valid quantifier-free HyperLTL subformulas that satisfy the assumption above. The proofs are straightforward and are omitted.

We assume that there is an LTL formula that can check whether a trace is in the set of standard inputs $\text{StdIn} \subseteq \text{In}^\omega$. We ambiguously call this LTL formula StdIn . The LTL formula StdIn contains only atomic propositions in AP_i . That is, the formula is obtained with the grammar in the second line of Equation (2.1) where atomic propositions have the form a (instead of a_π) with $a \in \text{AP}_i$. With StdIn_π we represent the HyperLTL formula that is exactly like StdIn but where each occurrence of $a \in \text{AP}_i$ has been replaced by a_π . Obviously, the LTL formula StdIn should be defined such that for every trace $t \in \mathbf{C}$ it holds that $\{\pi := t\} \models_{\mathbf{C}} \text{StdIn}_\pi$ if and only if $(t \downarrow_{\text{AP}_i}) \in \text{StdIn}$.

The following proposition shows the HyperLTL formula to analyse strict cleanliness.

$$\begin{array}{ll}
\mathbf{p}_\pi = \mathbf{p}_{\pi'} & \iff \bigwedge_{a \in \text{AP}_p} a_\pi \leftrightarrow a_{\pi'} & \mathbf{p}_\pi = \mathbf{p} & \iff \bigwedge_{a \in p} a_\pi \wedge \bigwedge_{a \in \text{AP}_p \setminus p} \neg a_\pi \\
\mathbf{i}_\pi = \mathbf{i}_{\pi'} & \iff \bigwedge_{a \in \text{AP}_i} a_\pi \leftrightarrow a_{\pi'} & \mathbf{i}_\pi = \mathbf{i} & \iff \bigwedge_{a \in i} a_\pi \wedge \bigwedge_{a \in \text{AP}_i \setminus i} \neg a_\pi \\
\mathbf{o}_\pi = \mathbf{o}_{\pi'} & \iff \bigwedge_{a \in \text{AP}_o} a_\pi \leftrightarrow a_{\pi'} & \mathbf{o}_\pi = \mathbf{o} & \iff \bigwedge_{a \in o} a_\pi \wedge \bigwedge_{a \in \text{AP}_o \setminus o} \neg a_\pi
\end{array}$$

$$\begin{array}{l}
\mathbf{p}_\pi \approx_P \mathbf{p}_{\pi'} \iff \bigvee_{\substack{p, p' \in \text{Param} \\ p \approx_P p'}} \mathbf{p}_\pi = p \wedge \mathbf{p}_{\pi'} = p' \\
d_{\text{In}}(\mathbf{i}_\pi, \mathbf{i}_{\pi'}) \leq \kappa_i \iff \bigvee_{\substack{i, i' \in \text{In} \\ d_{\text{In}}(i, i') \leq \kappa_i}} \mathbf{i}_\pi = i \wedge \mathbf{i}_{\pi'} = i' \\
d_{\text{Out}}(\mathbf{o}_\pi, \mathbf{o}_{\pi'}) \leq \kappa_o \iff \bigvee_{\substack{o, o' \in \text{Out} \\ d_{\text{Out}}(o, o') \leq \kappa_o}} \mathbf{o}_\pi = o \wedge \mathbf{o}_{\pi'} = o' \\
d_{\text{Out}}(\mathbf{o}_\pi, \mathbf{o}_{\pi'}) \leq f(d_{\text{In}}(\mathbf{i}_\pi, \mathbf{i}_{\pi'})) \iff \bigvee_{\substack{o, o' \in \text{Out}, i, i' \in \text{In} \\ d_{\text{Out}}(o, o') \leq f(d_{\text{In}}(i, i'))}} \mathbf{i}_\pi = i \wedge \mathbf{i}_{\pi'} = i' \wedge \mathbf{o}_\pi = o \wedge \mathbf{o}_{\pi'} = o'
\end{array}$$

Figure 4.3: Syntactic sugar for comparisons between traces

Proposition 4.9. Let $\mathbf{C} \subseteq (2^{\text{AP}})^\omega$ be a circuit, let $\hat{\mathbf{S}}$ be the reactive system constructed from \mathbf{C} according to Equation (4.1), let $\mathcal{C} = \langle \approx_P, \text{StdIn} \rangle$ be a contract for strict cleanness, and let StdIn_π be a HyperLTL subformula such that $\{\pi := t\} \models_{\mathbf{C}} \text{StdIn}_\pi$ if and only if $t \downarrow_{\text{AP}_i} \in \text{StdIn}$. Then, $\hat{\mathbf{S}}$ is strictly clean w.r.t. \mathcal{C} if and only if \mathbf{C} satisfies the HyperLTL formula

$$\forall \pi_1. \forall \pi_2. \exists \pi'_2. (\mathbf{p}_{\pi_1} \approx_P \mathbf{p}_{\pi_2} \wedge \text{StdIn}_{\pi_1}) \rightarrow (\mathbf{p}_{\pi_2} = \mathbf{p}_{\pi'_2} \wedge \square(\mathbf{i}_{\pi_1} = \mathbf{i}_{\pi'_2} \wedge \mathbf{o}_{\pi_1} = \mathbf{o}_{\pi'_2}))$$

The first quantifier (for π_1) in the HyperLTL formula in the proposition implicitly quantifies the first parameter \mathbf{p}_1 , the standard input i , and an output $\mathbf{o}_1 \in \hat{\mathbf{S}}(\mathbf{p}_1)(i)$. The second quantifier (for π_2) implicitly quantifies the second parameter \mathbf{p}_2 ; however, due to the potential nondeterminism in the behaviour of the system, the third, existential, quantifier for π'_2 is necessary. Trace π'_2 must encode parameter \mathbf{p}_2 and, according to the definition of strict cleanness, the inputs of this trace must be identical to i . Thus, it existentially quantifies an

output $\mathbf{o}_2 \in \hat{S}(\mathbf{p}_2)(i)$ and enforces that $\mathbf{o}_1 = \mathbf{o}_2$. In essence, the formula checks that for every $\mathbf{o}_1 \in \hat{S}(\mathbf{p}_1)(i)$, there exists $\mathbf{o}_2 \in \hat{S}(\mathbf{p}_2)(i)$, such that $\mathbf{o}_1 = \mathbf{o}_2$. Hence, it checks that $\hat{S}(\mathbf{p}_1)(i) \subseteq \hat{S}(\mathbf{p}_2)(i)$. Since the formula is symmetric, it also checks that $\hat{S}(\mathbf{p}_2)(i) \subseteq \hat{S}(\mathbf{p}_1)(i)$ and, ultimately, ensures that $\hat{S}(\mathbf{p}_1)(i) = \hat{S}(\mathbf{p}_2)(i)$.

To prove Proposition 4.9, we will use the following lemma, which follows almost directly from Equation (4.1).

Lemma 4.10. Let $\mathbf{C} \subseteq (2^{\text{AP}})^\omega$ be a circuit with $\text{AP} = \text{AP}_p \cup \text{AP}_i \cup \text{AP}_o$ and P be a predicate over $\text{Param} \times \text{In}^\omega \times \text{Out}^\omega$. Then,

1. $\forall t \in \mathbf{C}. P(t \downarrow_{\text{AP}_p}[0], t \downarrow_{\text{AP}_i}, t \downarrow_{\text{AP}_o})$ if and only if $\forall \mathbf{p} \in \text{Param}, i \in \text{In}^\omega. \forall \mathbf{o} \in \hat{S}(\mathbf{p})(i). P(\mathbf{p}, i, \mathbf{o})$, and
2. $\exists t \in \mathbf{C}. P(t \downarrow_{\text{AP}_p}[0], t \downarrow_{\text{AP}_i}, t \downarrow_{\text{AP}_o})$ if and only if $\exists \mathbf{p} \in \text{Param}, i \in \text{In}^\omega. \exists \mathbf{o} \in \hat{S}(\mathbf{p})(i). P(\mathbf{p}, i, \mathbf{o})$.

Furthermore, we need to make explicit the reasoning about time enforced by the HyperLTL operators globally (\square) and weak until (\mathcal{W}).

Lemma 4.11. Let Π be a trace assignment and let ϕ and ψ be quantifier-free HyperLTL formulas. Then the following equivalences hold.

1. $\Pi \models_{\mathbf{C}} \square \phi$ if and only if $\forall k > 0. \Pi[k..] \models_{\mathbf{C}} \phi$
2. $\Pi \models_{\mathbf{C}} \phi \mathcal{W} \psi$ if and only if $\forall k > 0. (\forall j \leq k. \Pi[j..] \models_{\mathbf{C}} \neg \psi) \rightarrow \Pi[k..] \models_{\mathbf{C}} \phi$.

Proof. We prove the two statements separately.

1. Using the definitions of \square and \diamond (from Section 2.6.1), we get that $\Pi \models_{\mathbf{C}} \square \phi$ holds if and only if $\Pi \models_{\mathbf{C}} \neg(\top \mathcal{U} \neg \phi)$ holds. Applying the semantics for \neg and \mathcal{U} yields the equivalent proposition that $\neg(\exists k > 0. \Pi[k..] \models_{\mathbf{C}} \neg \phi \wedge \forall j < k. \Pi[j..] \models_{\mathbf{C}} \top)$. By applying again the semantics for \neg and by logical simplifications, we know that the above is equivalent to $\forall k > 0. \Pi[k..] \models_{\mathbf{C}} \phi$.
2. According to the definition of \mathcal{W} , the semantics of \mathcal{U} and with 1, $\Pi \models_{\mathbf{C}} \phi \mathcal{W} \psi$ holds if and only if $(\exists k > 0. \Pi[k..] \models_{\mathbf{C}} \psi \wedge \forall j < k. \Pi[j..] \models_{\mathbf{C}} \phi) \vee \forall k > 0. \Pi[k..] \models_{\mathbf{C}} \phi$ holds. We denote this proposition as V . It is easy to see that the right operand of the equivalence to prove holds if and only if $\forall k > 0. (\exists j \leq k. \Pi[j..] \models_{\mathbf{C}} \psi) \vee \Pi[k..] \models_{\mathbf{C}} \phi$ holds. We denote this proposition as W . Thus, we must prove that $V \Rightarrow W$ and that $W \Rightarrow V$. To prove that V implies W , we distinguish two cases.

- For the first case, assume that the left operand of the disjunction in V holds, i.e., that there is some $k > 0$, such that $\Pi[k..] \models_C \psi \wedge \forall j < k. \Pi[j..] \models_C \phi$. To show W , let $k' > 0$ be arbitrary. If $k' \geq k$, then there exists $j \leq k'$ (namely $j = k$) such that $\Pi[j..] \models_C \psi$; hence, W holds. If $k' < k$, then we know from $\forall j < k. \Pi[j..] \models_C \phi$ that $\Pi[k'..] \models_C \phi$ is true; hence, W holds.
- For the second case, assume that the right operand of the disjunction in V holds, i.e., that $\forall k > 0. \Pi[k..] \models_C \phi$. Then, obviously W holds.

To prove that W implies V , let $PV = \{k \mid \Pi[k..] \models_C \psi\}$ be the set of all indices where ψ holds for the given trace assignment Π . If PV is the empty set, then it follows immediately from W that $\forall k > 0. \Pi[k..] \models_C \phi$ and that, hence, V holds. If PV is not empty, let $k = \min PV$ be the smallest index in PV . Then, obviously, $\exists k > 0. \Pi[k..] \models_C \psi$. To show that V holds, it suffices to show that $\forall j < k. \Pi[j..] \models_C \phi$ holds. This follows from W , because k is the smallest index for which $\Pi[k..] \models_C \psi$ holds and, therefore, for every $j < k$ it does not hold that $\Pi[j..] \models_C \psi$.

□

With these Lemmas, we are able to prove that the strict cleanness characterisation above is correct.

Proof of Proposition 4.9. From the semantics of HyperLTL (cf. Section 2.6.1) we get that

$$\begin{aligned} \emptyset \models_C \forall \pi_1. \forall \pi_2. \exists \pi'_2. (\mathbf{p}_{\pi_1} \approx_P \mathbf{p}_{\pi_2} \wedge \mathbf{StdIn}_{\pi_1}) \\ \rightarrow (\mathbf{p}_{\pi_2} = \mathbf{p}_{\pi'_2} \wedge \square(i_{\pi_1} = i_{\pi'_2} \wedge \mathbf{o}_{\pi_1} = \mathbf{o}_{\pi'_2})) \end{aligned}$$

if and only if

$$\begin{aligned} \forall t_1 \in \mathbf{C}. \forall t_2 \in \mathbf{C}. \exists t'_2 \in \mathbf{C}. (\Pi \models_C \mathbf{p}_{\pi_1} \approx_P \mathbf{p}_{\pi_2}) \wedge (\Pi \models_C \mathbf{StdIn}_{\pi_1}) \\ \rightarrow (\Pi \models_C \mathbf{p}_{\pi_2} = \mathbf{p}_{\pi'_2}) \wedge (\forall k > 0. \Pi[k..] \models_C i_{\pi_1} = i_{\pi'_2} \wedge \mathbf{o}_{\pi_1} = \mathbf{o}_{\pi'_2}), \end{aligned}$$

where $\Pi = \{\pi_1 := t_1, \pi_2 := t_2, \pi'_2 := t'_2\}$. From Assumption 4.8 and the the premise that $\{\pi := t\} \models_C \mathbf{StdIn}_{\pi}$ if and only if $t \downarrow_{\text{AP}_i} \in \mathbf{StdIn}$, we get that the above is equivalent to

$$\begin{aligned} \forall t_1 \in \mathbf{C}. \forall t_2 \in \mathbf{C}. \exists t'_2 \in \mathbf{C}. (t_1 \downarrow_{\text{AP}_p}[0] \approx_P t_2 \downarrow_{\text{AP}_p}[0]) \wedge (t_1 \downarrow_{\text{AP}_i} \in \mathbf{StdIn}) \\ \rightarrow (t_2 \downarrow_{\text{AP}_p}[0] = t'_2 \downarrow_{\text{AP}_p}[0]) \wedge \\ (\forall k > 0. t_1 \downarrow_{\text{AP}_i}[k] = t'_2 \downarrow_{\text{AP}_i}[k] \wedge t_1 \downarrow_{\text{AP}_o}[k] = t'_2 \downarrow_{\text{AP}_o}[k]). \end{aligned}$$

With Lemma 4.10 we get that this is equivalent to

$$\begin{aligned} \forall \mathbf{p}_1, \mathbf{p}_2 \in \text{Param}. \forall i_1, i_2 \in \text{In}^\omega. \forall \mathbf{o}_1 \in \hat{S}(\mathbf{p}_1)(i_1). \forall \mathbf{o}_2 \in \hat{S}(\mathbf{p}_2)(i_2). \\ \exists \mathbf{p}'_2 \in \text{Param}. \exists i'_2 \in \text{In}^\omega. \exists \mathbf{o}'_2 \in \hat{S}(\mathbf{p}'_2)(i'_2). \mathbf{p}_1 \approx_P \mathbf{p}_2 \wedge i_1 \in \text{StdIn} \\ \rightarrow \mathbf{p}_2 = \mathbf{p}'_2 \wedge \forall k > 0. i_1[k] = i'_2[k] \wedge \mathbf{o}_1[k] = \mathbf{o}'_2[k], \end{aligned}$$

which can be simplified to

$$\forall \mathbf{p}_1, \mathbf{p}_2 \in \text{Param}. \mathbf{p}_1 \approx_P \mathbf{p}_2 \rightarrow \forall i \in \text{StdIn}. \forall \mathbf{o}_1 \in \hat{S}(\mathbf{p}_1)(i). \exists \mathbf{o}_2 \in \hat{S}(\mathbf{p}_2)(i). \mathbf{o}_1 = \mathbf{o}_2.$$

This, in turn, is equivalent to

$$\forall \mathbf{p}_1, \mathbf{p}_2 \in \text{Param}. \mathbf{p}_1 \approx_P \mathbf{p}_2 \rightarrow \forall i \in \text{StdIn}. \hat{S}(\mathbf{p}_1)(i). \subseteq \hat{S}(\mathbf{p}_2)(i).$$

By symmetry, this is equivalent to Definition 3.40, hence proving the proposition. \square

A HyperLTL characterisation also exists for robust cleanness. Propositions 4.12 and 4.13 show HyperLTL formulas for trace integral l-robust cleanness and trace integral u-robust cleanness, respectively. Thus, the conjunction of the two formulas characterises trace integral robust cleanness.

Proposition 4.12. Let $C \subseteq (2^{\text{AP}})^\omega$ be a circuit, let \hat{S} be the reactive system constructed from C according to Equation (4.1), and let $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be a contract for robust cleanness, where d_{In} and d_{Out} are past-forgetful. Further, let StdIn_π be a HyperLTL subformula such that $\{\pi := t\} \models_C \text{StdIn}_\pi$ if and only if $t \downarrow_{\text{AP}_i} \in \text{StdIn}$. Then, \hat{S} is trace integral l-robustly clean w.r.t. \mathcal{C} if and only if C satisfies the HyperLTL formula

$$\begin{aligned} \forall \pi_1. \forall \pi_2. \exists \pi'_2. \\ (\mathbf{p}_{\pi_1} \approx_P \mathbf{p}_{\pi_2} \wedge \text{StdIn}_{\pi_1}) \\ \rightarrow \left(\mathbf{p}_{\pi_2} = \mathbf{p}_{\pi'_2} \wedge \square(i_{\pi_2} = i_{\pi'_2}) \wedge ((d_{\text{Out}}(\mathbf{o}_{\pi_1}, \mathbf{o}_{\pi'_2}) \leq \kappa_o) \mathcal{W}(d_{\text{In}}(i_{\pi_1}, i_{\pi'_2}) > \kappa_i)) \right) \end{aligned}$$

Proposition 4.13. Let $C \subseteq (2^{\text{AP}})^\omega$ be a circuit, let \hat{S} be the reactive system constructed from C according to Equation (4.1), and let $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be a contract for robust cleanness, where d_{In} and d_{Out} are past-forgetful. Further, let StdIn_π be a HyperLTL subformula such that $\{\pi := t\} \models_C \text{StdIn}_\pi$ if and only if $t \downarrow_{\text{AP}_i} \in \text{StdIn}$. Then \hat{S} is trace integral u-robustly clean w.r.t. \mathcal{C} if and only if C satisfies the HyperLTL formula

$$\begin{aligned} \forall \pi_1. \forall \pi_2. \exists \pi'_1. \\ (\mathbf{p}_{\pi_1} \approx_P \mathbf{p}_{\pi_2} \wedge \text{StdIn}_{\pi_1}) \\ \rightarrow \left(\mathbf{p}_{\pi_1} = \mathbf{p}_{\pi'_1} \wedge \square(i_{\pi_1} = i_{\pi'_1}) \wedge ((d_{\text{Out}}(\mathbf{o}_{\pi'_1}, \mathbf{o}_{\pi_2}) \leq \kappa_o) \mathcal{W}(d_{\text{In}}(i_{\pi'_1}, i_{\pi_2}) > \kappa_i)) \right) \end{aligned}$$

The difference between the first and second formula is subtle, but reflects the fact that, while the first formula has the universal quantification on the outputs of the program that takes the standard input and the existential quantification on the behaviour that may deviate, the second one works in the other way around. Thus, each of the formulas captures the $\forall\exists$ alternation in the conclusion of Definitions 3.74 and 3.75 of l-robust cleanness and u-robust cleanness. To notice this, follow the existentially quantified variable (π'_2 for the first formula and π'_1 for the second one). In both formulas we use the weak until operator \mathcal{W} . It has exactly the behaviour that we need to represent the interaction between the distances of inputs and the distances of outputs (cf. Lemma 4.11).

Below is the proof for the correctness of the characterisation of l-robust cleanness, i.e., for Proposition 4.12. For Proposition 4.13 the proof is analogue, hence we omit it.

Proof of Proposition 4.12. Using the semantics of HyperLTL and Lemma 4.11.2 we get that

$$\begin{aligned} \emptyset \models_C \forall \pi_1. \forall \pi_2. \exists \pi'_2. \\ (\mathbf{p}_{\pi_1} \approx_P \mathbf{p}_{\pi_2} \wedge \mathbf{StdIn}_{\pi_1}) \\ \rightarrow \left(\mathbf{p}_{\pi_2} = \mathbf{p}_{\pi'_2} \wedge \square(i_{\pi_2} = i_{\pi'_2}) \wedge ((d_{\text{Out}}(\mathbf{o}_{\pi_1}, \mathbf{o}_{\pi'_2}) \leq \kappa_o) \mathcal{W} (d_{\text{In}}(i_{\pi_1}, i_{\pi'_2}) > \kappa_i)) \right) \end{aligned}$$

is equivalent to

$$\begin{aligned} \forall t_1, t_2 \in C. \exists t'_2 \in C. (\Pi \models_C \mathbf{p}_{\pi_1} \approx_P \mathbf{p}_{\pi_2}) \wedge (\Pi \models_C \mathbf{StdIn}_{\pi_1}) \rightarrow \\ (\Pi \models_C \mathbf{p}_{\pi_2} = \mathbf{p}_{\pi'_2}) \wedge (\forall k > 0. \Pi[k..] \models_C i_{\pi_2} = i_{\pi'_2}) \wedge \\ (\forall k > 0. (\forall j \leq k. \Pi[j..] \models_C d_{\text{In}}(i_{\pi_1}, i_{\pi'_2}) \leq \kappa_i) \rightarrow \Pi[k..] \models_C d_{\text{Out}}(\mathbf{o}_{\pi_1}, \mathbf{o}_{\pi'_2}) \leq \kappa_o), \end{aligned}$$

where $\Pi = \{\pi_1 := t_1, \pi_2 := t_2, \pi'_2 := t'_2\}$. Further, with Assumption 4.8, and the assumption that $\{\pi := t\} \models_C \mathbf{StdIn}_{\pi}$ if and only if $t \downarrow_{\text{AP}_i} \in \mathbf{StdIn}$, we get that this is equivalent to

$$\begin{aligned} \forall t_1, t_2 \in C. \exists t'_2 \in C. (t_1 \downarrow_{\text{AP}_p}[0] \approx_P t_2 \downarrow_{\text{AP}_p}[0]) \wedge (t_1 \downarrow_{\text{AP}_i} \in \mathbf{StdIn}) \rightarrow \\ (t_2 \downarrow_{\text{AP}_p}[0] = t'_2 \downarrow_{\text{AP}_p}[0]) \wedge (\forall k > 0. t_2 \downarrow_{\text{AP}_i}[k] = t'_2 \downarrow_{\text{AP}_i}[k]) \wedge \\ (\forall k > 0. (\forall j \leq k. d_{\text{In}}(t_1 \downarrow_{\text{AP}_i}[j], t'_2 \downarrow_{\text{AP}_i}[j]) \leq \kappa_i) \rightarrow d_{\text{Out}}(t_1 \downarrow_{\text{AP}_o}[k], t'_2 \downarrow_{\text{AP}_o}[k]) \leq \kappa_o). \end{aligned}$$

We switch from trace quantification to explicit parameter, input and output quantification by using Lemma 4.10:

$$\begin{aligned} \forall \mathbf{p}_1, \mathbf{p}_2 \in \text{Param}. \forall i_1, i_2 \in \text{In}^\omega. \forall \mathbf{o}_1 \in \hat{S}(\mathbf{p}_1)(i_1). \forall \mathbf{o}_2 \in \hat{S}(\mathbf{p}_2)(i_2). \\ \exists \mathbf{p}'_2 \in \text{Param}. \exists i'_2 \in \text{In}^\omega. \exists \mathbf{o}'_2 \in \hat{S}(\mathbf{p}'_2)(i'_2). \mathbf{p}_1 \approx_P \mathbf{p}_2 \wedge i_1 \in \mathbf{StdIn} \\ \rightarrow \mathbf{p}_2 = \mathbf{p}'_2 \wedge (\forall k > 0. i_2[k] = i'_2[k]) \wedge \\ (\forall k > 0. (\forall j \leq k. d_{\text{In}}(i_1[j], i'_2[j]) \leq \kappa_i) \rightarrow d_{\text{Out}}(\mathbf{o}_1[k], \mathbf{o}'_2[k]) \leq \kappa_o). \end{aligned}$$

By further simplifying the above and by using that for past-forgetful distance functions $d(x[k], x'[k]) = d(x[..k], x'[..k])$ for $k > 0$, we get

$$\begin{aligned} \forall p_1, p_2 \in \text{Param. } p_1 \approx_P p_2 &\rightarrow \\ \forall i_1 \in \text{StdIn. } \forall i_2 \in \text{In}^\omega. \forall o_1 \in \hat{S}(p_1)(i_1). \exists o_2 \in \hat{S}(p_2)(i_2). \forall k > 0. \\ (\forall j \leq k. d_{\text{In}}(i_1[..j], i_2[..j]) \leq \kappa_i) &\rightarrow d_{\text{Out}}(o_1[..k], o_2[..k]) \leq \kappa_o. \end{aligned}$$

Since for $k = 0$ we have that $d_{\text{Out}}(o_1[..k], o_2[..k]) = d_{\text{Out}}(\epsilon, \epsilon) = 0$, we have in total that the above is equivalent to trace integral l-robust cleanness as defined in Definition 3.74. \square

Finally, we also give the characterisation of a func-clean program in terms of HyperLTL.

Proposition 4.14. Let $C \subseteq (2^{\text{AP}})^\omega$ be a circuit, let \hat{S} be the reactive system constructed from C according to Equation (4.1), and let $\mathcal{C} = \langle \approx_P, \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ be a contract for func-cleanness, where d_{In} and d_{Out} are past-forgetful. Further, let StdIn_π be a HyperLTL subformula such that $\{\pi := t\} \models_C \text{StdIn}_\pi$ if and only if $t \downarrow_{\text{AP}_i} \in \text{StdIn}$. Then \hat{S} is trace integral l-func-clean w.r.t. \mathcal{C} if and only if C satisfies the HyperLTL formula

$$\begin{aligned} \forall \pi_1. \forall \pi_2. \exists \pi'_2. \\ (p_{\pi_1} \approx_P p_{\pi_2} \wedge \text{StdIn}_{\pi_1}) \\ \rightarrow (p_{\pi_2} = p_{\pi'_2} \wedge \square(i_{\pi_2} = i_{\pi'_2}) \wedge \square(d_{\text{Out}}(o_{\pi_1}, o_{\pi'_2}) \leq f(d_{\text{In}}(i_{\pi_1}, i_{\pi'_2}))))). \end{aligned}$$

Proof. Using the semantics of HyperLTL and Lemma 4.11.1 we get that

$$\begin{aligned} \emptyset \models_C \forall \pi_1. \forall \pi_2. \exists \pi'_2. \\ (p_{\pi_1} \approx_P p_{\pi_2} \wedge \text{StdIn}_{\pi_1}) \\ \rightarrow (p_{\pi_2} = p_{\pi'_2} \wedge \square(i_{\pi_2} = i_{\pi'_2}) \wedge \square(d_{\text{Out}}(o_{\pi_1}, o_{\pi'_2}) \leq f(d_{\text{In}}(i_{\pi_1}, i_{\pi'_2}))))). \end{aligned}$$

holds if and only if

$$\begin{aligned} \forall t_1, t_2 \in C. \exists t'_2 \in C. (\Pi \models_C p_{\pi_1} \approx_P p_{\pi_2}) \wedge (\Pi \models_C \text{StdIn}_{\pi_1}) \\ \rightarrow (\Pi \models_C p_{\pi_2} = p_{\pi'_2}) \wedge (\forall k > 0. \Pi[k..] \models_C i_{\pi_2} = i_{\pi'_2}) \wedge \\ (\forall k > 0. \Pi[k..] \models_C d_{\text{Out}}(o_{\pi_1}, o_{\pi'_2}) \leq f(d_{\text{In}}(i_{\pi_1}, i_{\pi'_2}))) \end{aligned}$$

holds with $\Pi = \{\pi_1 := t_1, \pi_2 := t_2, \pi'_2 := t'_2\}$. Using Assumption 4.8 and the assumption that $\{\pi := t\} \models_C \text{StdIn}_\pi$ if and only if $t \downarrow_{\text{AP}_i} \in \text{StdIn}$, we get that the

above is equivalent to

$$\begin{aligned} \forall t_1, t_2 \in \mathbf{C}. \exists t'_2 \in \mathbf{C}. (t_1 \downarrow_{\text{AP}_p}[0] \approx_P t_2 \downarrow_{\text{AP}_p}[0]) \wedge (t_1 \downarrow_{\text{AP}_i} \in \mathbf{StdIn}) \rightarrow \\ (t_2 \downarrow_{\text{AP}_p}[0] = t'_2 \downarrow_{\text{AP}_p}[0]) \wedge (\forall k > 0. t_2 \downarrow_{\text{AP}_i}[k] = t'_2 \downarrow_{\text{AP}_i}[k]) \wedge \\ (\forall k > 0. d_{\text{Out}}(t_1 \downarrow_{\text{AP}_o}[k], t'_2 \downarrow_{\text{AP}_o}[k]) \leq f(d_{\text{In}}(t_1 \downarrow_{\text{AP}_i}[k], t'_2 \downarrow_{\text{AP}_i}[k]))). \end{aligned}$$

Using Lemma 4.10, we can replace the trace quantifiers by explicit quantifiers over parameters, inputs and outputs, and get

$$\begin{aligned} \forall \mathbf{p}_1, \mathbf{p}_2 \in \mathbf{Param}. \forall i_1, i_2 \in \text{In}^\omega. \forall \mathbf{o}_1 \in \hat{\mathbf{S}}(\mathbf{p}_1)(i_1). \forall \mathbf{o}_2 \in \hat{\mathbf{S}}(\mathbf{p}_2)(i_2). \\ \exists \mathbf{p}'_2 \in \mathbf{Param}. \exists i'_2 \in \text{In}^\omega. \exists \mathbf{o}'_2 \in \hat{\mathbf{S}}(\mathbf{p}'_2)(i'_2). \mathbf{p}_1 \approx_P \mathbf{p}_2 \wedge i_1 \in \mathbf{StdIn} \\ \rightarrow \mathbf{p}_2 = \mathbf{p}'_2 \wedge (\forall k > 0. i_2[k] = i'_2[k]) \wedge \\ (\forall k > 0. d_{\text{Out}}(\mathbf{o}_1[k], \mathbf{o}'_2[k]) \leq f(d_{\text{In}}(i_1[k], i'_2[k]))). \end{aligned}$$

By further simplifying the above and by using that for past-forgetful distance functions $d(x[k], x'[k]) = d(x[..k], x'[..k])$ for $k > 0$, we get

$$\begin{aligned} \forall \mathbf{p}_1, \mathbf{p}_2 \in \mathbf{Param}. \mathbf{p}_1 \approx_P \mathbf{p}_2 \rightarrow \forall i_1 \in \mathbf{StdIn}. \forall i_2 \in \text{In}^\omega. \\ \forall \mathbf{o}_1 \in \hat{\mathbf{S}}(\mathbf{p}_1)(i_1). \exists \mathbf{o}_2 \in \hat{\mathbf{S}}(\mathbf{p}_2)(i_2). \\ \forall k > 0. d_{\text{Out}}(\mathbf{o}_1[..k], \mathbf{o}_2[..k]) \leq f(d_{\text{In}}(i_1[..k], i_2[..k])). \end{aligned}$$

Since for $k = 0$ we have that $d_{\text{Out}}(\mathbf{o}_1[..k], \mathbf{o}_2[..k]) = d_{\text{Out}}(\epsilon, \epsilon) = 0$, we have in total that the above is equivalent to trace integral l-func-cleanness as defined in Definition 3.80. \square

Proposition 4.15. Let $\mathbf{C} \subseteq (2^{\text{AP}})^\omega$ be a circuit, let $\hat{\mathbf{S}}$ be the reactive system constructed from \mathbf{C} according to Equation (4.1), and let $\mathcal{C} = \langle \approx_P, \mathbf{StdIn}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ be a contract for func-cleanness, where d_{In} and d_{Out} are past-forgetful. Further, let \mathbf{StdIn}_π be a HyperLTL subformula such that $\{\pi := t\} \models_{\mathbf{C}} \mathbf{StdIn}_\pi$ if and only if $t \downarrow_{\text{AP}_i} \in \mathbf{StdIn}$. Then $\hat{\mathbf{S}}$ is trace integral u-func-clean w.r.t. \mathcal{C} if and only if \mathbf{C} satisfies the HyperLTL formula

$$\begin{aligned} \forall \pi_1. \forall \pi_2. \exists \pi'_1. \\ (\mathbf{p}_{\pi_1} \approx_P \mathbf{p}_{\pi_2} \wedge \mathbf{StdIn}_{\pi_1}) \\ \rightarrow (\mathbf{p}_{\pi_1} = \mathbf{p}_{\pi'_1} \wedge \square(i_{\pi_1} = i_{\pi'_1}) \wedge \square(d_{\text{Out}}(\mathbf{o}_{\pi'_1}, \mathbf{o}_{\pi_2}) \leq f(d_{\text{In}}(i_{\pi'_1}, i_{\pi_2}))))). \end{aligned}$$

The proof for Proposition 4.15 is analogue to the proof of Proposition 4.14, hence we omit it.

As before, the difference between the first and second formula is subtle and can be noticed again by following the existentially quantified variables in each of the formulas.

Example 4.16. We take up the emission cleaning examples from Examples 3.50 and 3.66. With the HyperLTL formulas presented in this section, we can now check whether the toy programs from Section 3.2 are robustly clean, respectively func-clean. For the concrete example, the property of l-robust cleanness reduces to checking formula

$$\forall \pi_1. \forall \pi_2. \exists \pi'_2. \quad (4.2)$$

$$\text{StdIn}_{\pi_1} \rightarrow \left(\square(t_{\pi_2} = t_{\pi'_2}) \wedge \left((d_{\text{Out}}(n_{\pi_1}, n_{\pi'_2}) \leq \kappa_o) \mathcal{W}(d_{\text{In}}(t_{\pi_1}, t_{\pi'_2}) > \kappa_i) \right) \right).$$

For u-robust cleanness it reduces to the obvious symmetric formula. For readability reasons, we shorthandedly write t for *thrtl* and n for *NOx*. Notice that any reference to parameters disappears since the emission control system does not have parameters. The set of standard inputs is characterised by the LTL formula $\text{StdIn} \equiv \square(t \in (0, 1])$. Likewise, we can verify that the model of the emission control system is l-func-clean through the formula

$$\forall \pi_1. \forall \pi_2. \exists \pi'_2. \quad (4.3)$$

$$\text{StdIn}_{\pi_1} \rightarrow \left(\square(t_{\pi_2} = t_{\pi'_2}) \wedge \square(d_{\text{Out}}(n_{\pi_1}, n_{\pi'_2}) \leq f(d_{\text{In}}(t_{\pi_1}, t_{\pi'_2}))) \right),$$

and that it is u-func-clean through the symmetric formula.

4.2.1 Experimental Results

We verified the cleanness of the toy emission cleaning system from Example 4.16 using the HyperLTL model checker MCHyper [57]. The input to the model checker is a description of the system as an Aiger circuit and a hyperproperty specified as a HyperLTL formula. At the time when we conducted the experiments outlined below, MCHyper could only check HyperLTL formulas that were alternation-free. Since the HyperLTL formulas from the previous section are of the form $\forall \pi_1 \forall \pi_2 \exists \pi'_2 \dots$, and are, hence, not alternation-free, MCHyper could not check these formulas directly. At the time when this thesis was submitted, MCHyper is able to check formulas with a single quantifier alternation [38]. For a $\forall \forall \exists$ quantified formula the tool replaces the existential quantifier by “strategic choice”. The strategy, though, must be passed as an additional input to MCHyper. Coenen et al. [38, Table 1] show that robust cleanness can be checked with this new version of MCHyper.

Without this new version of the tool, we proved, respectively disproved, our formulas by strengthening them and their negations such that they become alternation-free formulas.

To prove that program EC in Figure 4.1 is robustly clean, we strengthen formula (4.2) by substituting π_2 for the existentially quantified variable π'_2 . The

resulting formula is alternation-free :

$$\forall \pi_1. \forall \pi_2. \text{StdIn}_{\pi_1} \rightarrow ((d_{\text{Out}}(n_{\pi_1}, n_{\pi_2}) \leq \kappa_o) \mathcal{W}(d_{\text{In}}(t_{\pi_1}, t_{\pi_2}) > \kappa_i)) \quad (4.4)$$

MCHyper confirms that program EC satisfies formula (4.4). The program thus also satisfies formula (4.2). Notice that we had obtained the same formula if we would have started from the formula symmetric to formula (4.2).

To prove that program AEC in Figure 4.2 is doped with respect to Equation (4.2), we negate formula (4.2) and obtain

$$\begin{aligned} & \exists \pi_1. \exists \pi_2. \forall \pi'_2. \\ & \neg \left(\text{StdIn}_{\pi_1} \rightarrow \left(\square(t_{\pi_2} = t_{\pi'_2}) \wedge ((d_{\text{Out}}(n_{\pi_1}, n_{\pi'_2}) \leq \kappa_o) \mathcal{W}(d_{\text{In}}(t_{\pi_1}, t_{\pi'_2}) > \kappa_i)) \right) \right) \end{aligned}$$

This formula is of the form $\exists \pi_1. \exists \pi_2. \forall \pi'_2. \dots$ and, hence, again not alternation-free. We replace the two existential quantifiers with universal quantifiers and restrict the quantification to two specific throttle values, a for π_1 and b for π_2 (we will explain further down which concrete values for a and b we used for the analysis):

$$\begin{aligned} & \forall \pi_1. \forall \pi_2. \forall \pi'_2. \quad (4.5.a) \\ & \square(t_{\pi_1} = a \wedge t_{\pi_2} = b) \rightarrow \\ & \neg \left(\text{StdIn}_{\pi_1} \rightarrow \left(\square(t_{\pi_2} = t_{\pi'_2}) \wedge ((d_{\text{Out}}(n_{\pi_1}, n_{\pi'_2}) \leq \kappa_o) \mathcal{W}(d_{\text{In}}(t_{\pi_1}, t_{\pi'_2}) > \kappa_i)) \right) \right) \end{aligned}$$

This transformation is sound as long as there actually exist traces with throttle values a and b . We establish this by checking, separately, that the following existential formula is satisfied:

$$\exists \pi_1. \exists \pi_2. \square(t_{\pi_1} = a \wedge t_{\pi_2} = b) \quad (4.6)$$

MCHyper confirms the satisfaction of both formulas, which proves that formula (4.2) is violated by program AEC. Precisely, the counterexample that shows the violation of formula (4.2) is any pair of traces π_1 and π_2 that makes $\square(t_{\pi_1} = a \wedge t_{\pi_2} = b)$ true in formula (4.6). We proceed similarly for the formula symmetric to formula (4.2) obtaining two formulas just as before which are also satisfied by AEC and hence the original formula is not. Also, we follow a similar process to prove that EC is func-clean but AEC is not.

Table 4.1 shows experimental results obtained with MCHyper² version 0.91 for the verification of robust cleanness. The Aiger models were constructed by discretizing the values of the throttle and the NO_x . We show results from two

²<https://www.react.uni-saarland.de/tools/mchyper/>

Program	NO _x step	model size	circuit size		property	time (sec.)
		#transitions	#latches	#gates		
EC	0.05	1436	17	9749	eq. (4.4)	0.92
	0.00625	60648	23	505123	eq. (4.4)	22.19
AEC	0.05	3756	19	27574	(4.5.a) $a = 0.1$	1.62
					(4.5.b) $a = 0.1$	1.6
					(4.5.a) $a = 1$	1.68
					(4.5.b) $a = 1$	1.56
	0.00625	175944	25	1623679	(4.5.a) $a = 0.1$	102.07
					(4.5.b) $a = 0.1$	96.3
				(4.5.a) $a = 1$	97.67	
				(4.5.b) $a = 1$	92.8	

Table 4.1: Experimental results from the verification of robust cleanness of EC and AEC

different models, where the values of the throttle was discretised in steps of 0.1 units in both models and the values of the NO_x in steps of 0.05 and 0.00625. All experiments were run under OS X “El Capitan” (10.11.6) on a MacBook Air with a 1.7GHz Intel Core i5 and 4GB 1333MHz DDR3. In Table 4.1, the model size is given in terms of the number of transitions, while the size of the Aiger circuit encoding the model prepared for the property is given in terms of the number of latches and gates. The specification checked by MCHyper is the formula indicated in the property column. Formula (4.5.b) is the formula symmetric to (4.5.a). For the throttle values a and b in formulas (4.5.a) and (4.5.b), we chose $b = 2$ and let a vary as specified in the property column. Table 4.2 shows similar experimental results for the verification of func-cleanness. With (4.4’), (4.5.a’), and (4.5.b’) we indicate the similar variations to formulas (4.4), (4.5.a), and (4.5.b) required to verify formula (4.3). Model checking takes less than two seconds for the coarse discretisation and about two minutes for the fine discretisation.

4.3 Related Work & Contributions

In this chapter, we picked two out of many possible techniques to prove cleanness of sequential programs and reactive systems. The weakest precondition reasoning provides a convenient methodology to prove or disprove cleanness for sequential programs. For reactive systems, the HyperLTL characterisations allow for a tool-supported cleanness analysis. Other verification techniques include relational

Program	NO _x step	model size	circuit size		property	time (sec.)
		#transitions	#latches	#gates		
EC	0.05	1436	5	9869	(4.4')	1.08
	0.00625	60648	8	505285	(4.4')	21.74
AEC	0.05	3756	6	27708	(4.5.a') a = 0.1	1.71
					(4.5.b') a = 0.1	1.72
					(4.5.a') a = 1	1.72
					(4.5.b') a = 1	1.77
	0.00625	175944	9	1623855	(4.5.a') a = 0.1	95.29
					(4.5.b') a = 0.1	97.48
				(4.5.a') a = 1	95.57	
				(4.5.b') a = 1	95.5	

Table 4.2: Experimental results from the verification of func-cleanness of EC and AEC

and Cartesian Hoare logics [16, 137, 110], self-composition and product programs constructions [10], temporal logics [34, 57, 56], or games [92]. These techniques greatly vary in their completeness, efficiency, and scalability.

The adaptation of the parametrised cleanness definitions for weakest precondition reasoning has its origin in one of our previous works [42]. It is the result of a fruitful cooperation with my co-authors. The same is true for the HyperLTL characterisations. During my work on this thesis, though, I realised that parts of the encodings of the non-atomic propositions (cf. Figure 4.3) were incorrect, and that the propositions in [42] incorrectly claimed that the HyperLTL characterisations capture the non-trace-integral variants of robust cleanness and func-cleanness. I revised the complete proof structure and fixed the propositions in this thesis. As part of this restructuring I decoupled the proofs from the encoding in Figure 4.3 and instead summarised in Assumption 4.8 the properties that an encoding must satisfy for the proofs to be valid. This provides the flexibility to replace the encoding from Figure 4.3 with any other encoding satisfying Assumption 4.8. The experiment with MCHyper is primarily a contribution of myself.

5 Model-Agnostic Software Doping Analysis

A common reason to check whether a program satisfies (a suitable form of) cleanliness is to detect software doping, i.e., a behaviour of a software that is intended by the manufacturer, but that is not in the interest of the user or society. When analysing software for this reason, the manufacturer is typically not involved; they already know that their software is not clean. Instead, the analysis might be initiated by the user of the system, an NGO, or by researchers. In most of these cases, there is not sufficient information available to do a model-aware analysis and the system is merely a black-box. The analysis must, essentially, be based on observations of the system. In this chapter, we introduce formal foundations to conduct robust cleanliness tests using a model-based testing approach [119, 120, 121]. Furthermore, we will extend the HyperLTL characterisations of robust cleanliness and func-cleanliness from Section 4.2 to HyperSTL and STL characterisations. This enables us to complement the model-based testing approach with probabilistic falsification [95, 2] techniques to systematically find violations of robust cleanliness or func-cleanliness.

5.1 Cleanliness of Labelled Transition Systems

In this chapter we will focus on robust cleanliness and func-cleanliness of mixed-IO systems (cf., Section 3.3). Mixed-IO systems are defined by means of sets of traces that describe the behaviour of a system. Hence, it is well suited for black-box settings that we target in this chapter. In the following, we will use labelled transition systems to make the reasoning about sets of traces more convenient. The paths in an LTS originating from the LTS' initial state induce a set of traces. Conversely, for every set of traces an LTS can be constructed that induces this set of traces. Notably, constructing mixed-IO systems from LTS does not necessarily satisfy the input and output enabledness that is required for mixed-IO systems. To ensure that the induced mixed-IO system is input enabled, we will work with input-output transition systems, which constitute the subset of labelled transition systems that are input enabled. If an IOTS L is not output enabled, then there is some state that does not have an outgoing

transition labelled with an output symbol. To overcome this problem, instead of using L , we use its quiescence closure L_δ , i.e., we use the special output symbol δ to represent the absence of an output (cf. Section 2.3). The IOTS L_δ induces a mixed-IO system that is input and output enabled. By default, a distance function for outputs d_{Out} is not prepared to handle output traces that contain the quiescence symbol. The cleanness definitions we will show below use an extended output distance function d_{Out_δ} that is based on the original function d_{Out} . Concretely, $d_{\text{Out}_\delta}(\sigma_1, \sigma_2) := d_{\text{Out}}(\sigma_1 \setminus \delta, \sigma_2 \setminus \delta)$ if $\sigma_1[k] = \delta \Leftrightarrow \sigma_2[k] = \delta$ for all k , and $d_{\text{Out}_\delta}(\sigma_1, \sigma_2) := \infty$ otherwise, where $\sigma \setminus \delta$ is the same as σ with all δ removed. d_{Out_δ} returns the distance infinity for two traces if they have quiescence symbols at different positions. Otherwise, the quiescence symbols are removed and what remains after that removal is passed to the original output distance function.

We could use robust cleanness for mixed-IO systems straightforwardly for cleanness of IOTS. An IOTS L would be robustly clean w.r.t. some contract \mathcal{C} if and only if $\text{traces}_\omega(L_\delta)$ is robustly clean w.r.t. \mathcal{C} . To make our approach better usable in practice, we generalise this notion of cleanness to allow also partial knowledge about the output behaviour of a system for standard inputs. Concretely, instead of defining only inputs as standard, we define inputs and outputs for such inputs as standard. Notably, for an input, one of its outputs can be considered as standard behaviour while a different output can be excluded. This corresponds to a testing context, in which recordings of the system executing standard inputs are the baseline for testing. We adapt the cleanness contracts from Chapter 3 by replacing the set StdIn with an LTS Std that defines the standard behaviour and call the resulting tuple *cleanness context*. Conceptually, cleanness contracts and contexts are very similar: For a mixed-IO robust cleanness contract $\mathcal{C}' = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ and an IOTS L that induces a mixed-IO system $L' = \text{traces}_\omega(L_\delta)$, the tuple $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ is a cleanness context, if Std is an LTS such that $\text{traces}_\omega(\text{Std}_\delta) \subseteq L'(\text{StdIn})$, i.e., the behaviour captured by Std_δ is also a behaviour of L_δ . Ideally, $\text{traces}_\omega(\text{Std}_\delta)$ is equal to $L'(\text{StdIn})$, because missing standard behaviour may lead to false positives of doping detection. Notice that a cleanness context implicitly defines a set StdIn ; it is determined by the input sequences $\text{traces}_\omega(\text{Std}_\delta) \downarrow_i$ occurring in Std_δ .

Robust cleanness for LTS closely resembles robust cleanness for mixed-IO systems, except for the special handling of the standard behaviour and the usage of IOTS instead of mere trace sets.

Definition 5.1. An IOTS L is *l-robustly clean* with respect to cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(L_\delta)$ and for all $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$, $\sigma' \in \text{traces}_\omega(L_\delta)$ and $k > 0$ it holds that whenever

$d_{\text{In}}(\sigma[..j]_{\downarrow i}, \sigma'[..j]_{\downarrow i}) \leq \kappa_i$ for all $j \leq k$ then there exists $\sigma'' \in \text{traces}_{\omega}(\mathbf{L}_{\delta})$ such that $\sigma'_{\downarrow i} = \sigma''_{\downarrow i}$ and $d_{\text{Out}_{\delta}}(\sigma[..k]_{\downarrow o}, \sigma''[..k]_{\downarrow o}) \leq \kappa_o$.

We remark that cleanness definitions for IOTS are *system-specific*, i.e., they are only applicable to the IOTS from which the standard behaviour Std is obtained from (or could have been obtained from). This is in contrast to all cleanness notions from Chapter 3 that are defined w.r.t. contracts, which are applicable to all systems matching the sets In and Out implicitly defined through the distance functions. For an IOTS to be clean according to the above definition, the traces of the (quiescence-closed) standard behaviour must be fully contained in the behaviour of the IOTS.

Definition 5.2. An IOTS \mathbf{L} is *u-robustly clean* with respect to cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if $\text{traces}_{\omega}(\text{Std}_{\delta}) \subseteq \text{traces}_{\omega}(\mathbf{L}_{\delta})$ and for all $\sigma \in \text{traces}_{\omega}(\text{Std}_{\delta})$, $\sigma' \in \text{traces}_{\omega}(\mathbf{L}_{\delta})$ and $k > 0$ it holds that whenever $d_{\text{In}}(\sigma[..j]_{\downarrow i}, \sigma'[..j]_{\downarrow i}) \leq \kappa_i$ for all $j \leq k$ then there exists $\sigma'' \in \text{traces}_{\omega}(\text{Std}_{\delta})$ such that $\sigma'_{\downarrow i} = \sigma''_{\downarrow i}$ and $d_{\text{Out}_{\delta}}(\sigma'[..k]_{\downarrow o}, \sigma''[..k]_{\downarrow o}) \leq \kappa_o$.

A peculiarity of u-robust cleanness of IOTS is in the existential quantification: Instead of quantifying over traces of \mathbf{L}_{δ} , the quantifier requests a trace from the standard LTS Std_{δ} . This is necessary, because, as explained above, the standard LTS may be incomplete. So, to satisfy Definition 5.2 there must be an output for input $\sigma_{\downarrow i}$ that has been actually observed; it does not suffice if the system is in principle capable of showing an output that satisfies the output distance requirement.

Finally, the full robust cleanness definition is given below using the quantifier-based characterisation analogue to its mixed-IO counterpart.

Definition 5.3. An IOTS \mathbf{L} is *robustly clean* with respect to cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if $\text{traces}_{\omega}(\text{Std}_{\delta}) \subseteq \text{traces}_{\omega}(\mathbf{L}_{\delta})$ and for all $\sigma \in \text{traces}_{\omega}(\text{Std}_{\delta})$, $\sigma' \in \text{traces}_{\omega}(\mathbf{L}_{\delta})$ and $k > 0$ it holds that whenever $d_{\text{In}}(\sigma[..j]_{\downarrow i}, \sigma'[..j]_{\downarrow i}) \leq \kappa_i$ for all $j \leq k$ then

1. there exists $\sigma'' \in \text{traces}_{\omega}(\mathbf{L}_{\delta})$, such that $\sigma'_{\downarrow i} = \sigma''_{\downarrow i}$ and

$$d_{\text{Out}_{\delta}}(\sigma'[..k]_{\downarrow o}, \sigma''[..k]_{\downarrow o}) \leq \kappa_o,$$
2. there exists $\sigma'' \in \text{traces}_{\omega}(\text{Std}_{\delta})$, such that $\sigma'_{\downarrow i} = \sigma''_{\downarrow i}$ and

$$d_{\text{Out}_{\delta}}(\sigma'[..k]_{\downarrow o}, \sigma''[..k]_{\downarrow o}) \leq \kappa_o.$$

In the sequel, we will often use the predicate $\forall j \leq k: d_{\text{In}}(\sigma[..j]_{\downarrow i}, \sigma'[..j]_{\downarrow i}) \leq \kappa_i$; we abbreviate this predicate by $\mathcal{V}_{(d_{\text{In}}, \kappa_i)}(k, \sigma, \sigma')$. If d_{In} and κ_i are known from the context, we omit the index.

In a similar way, we obtain LTS variants of func-cleanness.

Definition 5.4. An IOTS L is *l-func-clean* with respect to cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(L_\delta)$ and for all $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$, $\sigma' \in \text{traces}_\omega(L_\delta)$ and $k > 0$, there exists $\sigma'' \in \text{traces}_\omega(L_\delta)$, such that $\sigma' \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i))$.

Definition 5.5. An IOTS L is *u-func-clean* with respect to cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(L_\delta)$ and for all $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$, $\sigma' \in \text{traces}_\omega(L_\delta)$ and $k > 0$, there exists $\sigma'' \in \text{traces}_\omega(\text{Std}_\delta)$, such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}}(\sigma'[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i))$.

Definition 5.6. An IOTS L is *func-clean* with respect to cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(L_\delta)$ and for all $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$, $\sigma' \in \text{traces}_\omega(L_\delta)$ and $k > 0$,

1. there exists $\sigma'' \in \text{traces}_\omega(L_\delta)$, such that $\sigma' \downarrow_i = \sigma'' \downarrow_i$ and
$$d_{\text{Out}}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i)),$$
2. there exists $\sigma'' \in \text{traces}_\omega(\text{Std}_\delta)$, such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and
$$d_{\text{Out}}(\sigma'[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i)).$$

Trace integrity preserving robust cleanness definitions for LTS-defined systems are enumerated below. They deviate from the above definitions as expected (cf. Sections 3.2.4 and 3.3.3).

Definition 5.7. An IOTS L is *trace integral l-robustly clean* w.r.t. cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(L_\delta)$ and for all $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$ and $\sigma' \in \text{traces}_\omega(L_\delta)$, there exists $\sigma'' \in \text{traces}_\omega(L_\delta)$ such that $\sigma' \downarrow_i = \sigma'' \downarrow_i$ and for every $k > 0$ it holds that whenever $d_{\text{In}}(\sigma[..j] \downarrow_i, \sigma'[..j] \downarrow_i) \leq \kappa_i$ for all $j \leq k$, then $d_{\text{Out}_\delta}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq \kappa_o$.

Definition 5.8. An IOTS L is *trace integral u-robustly clean* w.r.t. cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(L_\delta)$ and for all $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$ and $\sigma' \in \text{traces}_\omega(L_\delta)$, there exists $\sigma'' \in \text{traces}_\omega(\text{Std}_\delta)$ such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and for every $k > 0$ it holds that whenever $d_{\text{In}}(\sigma[..j] \downarrow_i, \sigma'[..j] \downarrow_i) \leq \kappa_i$ for all $j \leq k$, then $d_{\text{Out}_\delta}(\sigma'[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq \kappa_o$.

Definition 5.9. An IOTS L is *trace integral robustly clean* w.r.t. cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ if and only if L is trace integral l-robustly clean w.r.t. \mathcal{C} and L is trace integral u-robustly clean w.r.t. \mathcal{C} .

Accordingly, there are trace integral definitions for func-cleanness.

Definition 5.10. An IOTS L is *trace integral l-func-clean* with respect to cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(L_\delta)$ and for all $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$ and $\sigma' \in \text{traces}_\omega(L_\delta)$, there exists $\sigma'' \in \text{traces}_\omega(L_\delta)$, such that $\sigma' \downarrow_i = \sigma'' \downarrow_i$ and for every $k > 0$, it holds that $d_{\text{Out}}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[..k] \downarrow_i, \sigma'[..k] \downarrow_i))$.

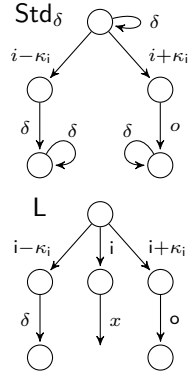
Definition 5.11. An IOTS L is *trace integral u-func-clean* with respect to cleanliness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if $\text{traces}_{\omega}(\text{Std}_{\delta}) \subseteq \text{traces}_{\omega}(L_{\delta})$ and for all $\sigma \in \text{traces}_{\omega}(\text{Std}_{\delta})$ and $\sigma' \in \text{traces}_{\omega}(L_{\delta})$, there exists $\sigma'' \in \text{traces}_{\omega}(\text{Std}_{\delta})$, such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and for every $k > 0$, it holds that $d_{\text{Out}}(\sigma'[\cdot k] \downarrow_o, \sigma''[\cdot k] \downarrow_o) \leq f(d_{\text{In}}(\sigma[\cdot k] \downarrow_i, \sigma''[\cdot k] \downarrow_i))$.

Definition 5.12. An IOTS L is *trace integral func-clean* with respect to cleanliness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ if and only if L is trace integral l-func-clean w.r.t. \mathcal{C} and L is trace integral u-func-clean w.r.t. \mathcal{C} .

5.2 Reference Implementation for Robust Cleaness

As we will show in this section, for many robust cleanliness contexts, there exists an IOTS that is robustly clean w.r.t. this context and that is maximal in the sense that whatever behaviour is deemed robustly clean w.r.t. this context, it is part of this IOTS. The condition for such an IOTS to exist is a cleanliness context that actually allows robustly clean IOTS to exist at all. As the following example shows, this is not always the case.

Example 5.13. The LTS Std_{δ} on the right is a quiescence-closed standard LTS Std_{δ} for the implementation L shown below Std_{δ} . For simplicity some input transitions are omitted. Assume $\text{Out} = \{\circ\}$ and $\text{In} = \{i, i - \kappa_i, i + \kappa_i\}$. Consider the transition labelled x of L . This must be one of either \circ or δ , but we will see that either choice leads to a contradiction w.r.t. the output distances induced. The input projection of the middle path in L is $i - \kappa_i$ and the input distance to $(i - \kappa_i) - \kappa_i$ and $(i + \kappa_i) - \kappa_i$ is exactly κ_i , so both branches $(i + \kappa_i) \circ$ and $(i - \kappa_i) \delta$ of Std_{δ} must be considered to determine x . For $x = \circ$, the output distance of $-\circ x$ to $-\circ \circ$ in the right branch of Std_{δ} is 0, i.e. less than κ_o . However, $d_{\text{Out}_{\delta}}(-\circ \delta, -\circ \circ) = \infty > \kappa_o$. Thus the output distance to the left branch of Std_{δ} is too high if picking \circ . Instead picking $x = \delta$ does not work either for the symmetric reasons – the problem switches sides. Thus, neither picking \circ nor δ for x satisfies robust cleanliness here (more precisely, u-robust cleanliness in this case). Indeed, no implementation satisfying robust cleanliness exists for the given context.



The problem of unsatisfiable contexts is unique to IOTS; for the contract-based cleanliness definitions for other system models, it is always possible to describe a system that satisfies a contract. The crucial difference is that a cleanliness context defines the outputs for the standard inputs. Irrespective of how an IOTS looks like, the outputs this IOTS must exhibit for standard inputs are

fixed by Std_δ . In the above example, we cannot change the implementation L to get better outputs for inputs $i - \kappa_i$ and $i + \kappa_i$ (without changing the cleanness context).

Definition 5.14 (Satisfiable Context). A context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ is *satisfiable*, if and only if, there exists an IOTS L that is robustly clean w.r.t. \mathcal{C} . A contract that is not satisfiable is called *unsatisfiable*.

If a context \mathcal{C} is satisfiable, then this means in particular that there exists an IOTS L that is l-robustly clean w.r.t. \mathcal{C} and that is u-robustly clean w.r.t. \mathcal{C} . Moreover, the following proposition shows that the standard LTS of a satisfiable context is always u-robustly clean on its own. That is, every behaviour that Std shows is justifiable by every trace of Std . The LTS Std may, however, lack behaviour to satisfy l-robust cleanness.

Proposition 5.15. Let $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be a cleanness context. If \mathcal{C} is satisfiable, then Std is u-robustly clean w.r.t. \mathcal{C} .

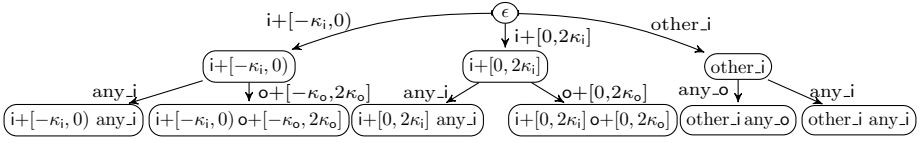
Proof. First, observe that obviously $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(\text{Std}_\delta)$. Further, let $\sigma, \sigma' \in \text{traces}_\omega(\text{Std}_\delta)$ and $k > 0$. We may assume that $\mathcal{V}_{(d_{\text{In}}, \kappa_i)}(k, \sigma, \sigma')$ holds, and have to show that there exists a trace in $\sigma'' \in \text{traces}_\omega(\text{Std}_\delta)$, such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}_\delta}(\sigma'[\dots k] \downarrow_o, \sigma''[\dots k] \downarrow_o) \leq \kappa_o$.

From satisfiability of \mathcal{C} , we get an IOTS L that is robustly clean w.r.t. \mathcal{C} , and, hence, u-robustly clean w.r.t. \mathcal{C} . From Definition 5.2 we get that $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(L_\delta)$; thus, $\sigma' \in \text{traces}_\omega(L_\delta)$. Further, since $\mathcal{V}_{(d_{\text{In}}, \kappa_i)}(k, \sigma, \sigma')$ holds, we get for σ and σ' from u-robust cleanness of L a trace $\sigma'' \in \text{traces}_\omega(\text{Std}_\delta)$ such that $\sigma \downarrow_i = \sigma'' \downarrow_i$ and $d_{\text{Out}_\delta}(\sigma'[\dots k] \downarrow_o, \sigma''[\dots k] \downarrow_o) \leq \kappa_o$. \square

For every satisfiable context, there exists an IOTS that is the largest implementation that is robustly clean w.r.t. this context. “Largest” is defined on the set of traces of an IOTS:

Definition 5.16 (Largest Implementation). Let \mathcal{C} be a context and L an IOTS that is robustly clean w.r.t. \mathcal{C} . L is the *largest implementation for \mathcal{C}* if and only if for every L' that is robustly clean w.r.t. \mathcal{C} , it holds that $\text{traces}_\omega(L'_\delta) \subseteq \text{traces}_\omega(L_\delta)$.

If a cleanness context is satisfiable, there is always a (up to trace equivalence) unique IOTS that is the largest, robustly clean implementation for this context. We will construct this largest implementation and call it *reference implementation* \mathcal{R} . To construct \mathcal{R} , we will in the following assume, for simplicity of the construction, past-forgetful output distance functions. Thus, we simply assume that $d_{\text{Out}} : (\text{Out} \cup \{-o\}) \times (\text{Out} \cup \{-o\}) \rightarrow \overline{\mathbb{R}}_{\geq 0}$. We remark that $d_{\text{Out}_\delta}(\delta, o) = \infty$ for all $o \neq \delta$.

Figure 5.1: The reference implementation \mathcal{R} of Std in Example 5.18.

The reference implementation \mathcal{R} is derived from a concrete cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$. Since \mathcal{R} must be input-enabled, it supports any possible sequence of inputs. For every state in \mathcal{R} that is reachable through an input σ , \mathcal{R} can nondeterministically choose an output that satisfies u-robust cleanness. That is, an edge leaving such a state is labelled with output \mathfrak{o} only if for every trace σ_i of Std_δ with an input distance of at most κ_i , there is a state in Std_δ reachable through σ_i that has an outgoing edge labelled with output \mathfrak{o}' that is at most κ_o away from \mathfrak{o} . Moreover, every state in \mathcal{R} offers every output that is permitted by u-robust cleanness. Consequently, for inputs beyond the κ_i radius of all standard inputs, all outputs in Out_δ are possible in the respective state in \mathcal{R} .

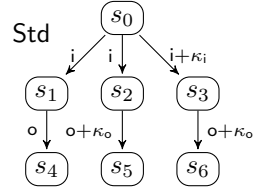
To construct the reference implementation \mathcal{R} we decide to model the quiescence transitions explicitly instead of using the quiescence closure. We preserve the property, that in each state of the LTS it is possible to do an output or a quiescence transition. The construction of \mathcal{R} proceeds by adding all transitions that satisfy u-robust cleanness.

Definition 5.17. Let $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be a context. The *reference implementation* \mathcal{R} for \mathcal{C} is the LTS $\langle (\text{In} \cup \text{Out}_\delta)^*, \text{In}, \text{Out}_\delta, \rightarrow_{\mathcal{R}}, \epsilon \rangle$ where $\rightarrow_{\mathcal{R}}$ is defined by

$$\begin{aligned} & \forall \sigma_i \in \text{traces}_\omega(\text{Std}_\delta) \downarrow_i: \\ & (\forall j \leq |\sigma| + 1: d_{\text{In}}((\sigma \cdot a)[..j] \downarrow_i, \sigma_i[..j]) \leq \kappa_i) \\ & \Rightarrow \exists \sigma_S \in \text{traces}_\omega(\text{Std}_\delta): \sigma_S \downarrow_i = \sigma_i \wedge d_{\text{Out}_\delta}(a \downarrow_o, \sigma_S[|\sigma| + 1] \downarrow_o) \leq \kappa_o \\ \hline & \sigma \xrightarrow{\mathfrak{a}}_{\mathcal{R}} \sigma \cdot a \end{aligned}$$

Notably, \mathcal{R} is deterministic, since only transitions of the form $\sigma \xrightarrow{\mathfrak{a}}_{\mathcal{R}} \sigma \cdot a$ are added. Even further, the construction of \mathcal{R} is such that we are always able to identify for each trace the (unique) state which can be reached by that trace. This is also expressed formally in Lemma 5.19 and Corollary 5.20. Before, we provide a detailed example to explain the construction of a reference implementation.

Example 5.18. Figure 5.1 gives a schematic representation of the reference implementation \mathcal{R} for the LTS Std on the right. Input (output) actions are denoted with letter i (o , respectively), quiescence transitions are omitted. We use the absolute difference of the values, so that $d_{\text{In}}(i, i') := |i - i'|$ and $d_{\text{Out}}(o, o') := |o - o'|$. For this example, the quiescence closure Std_δ looks like Std but with δ -loops in states s_0 , s_4 , s_5 , and s_6 . Label $r+[a, b]$ should be interpreted as any value $r' \in [a + r, b + r]$ and similarly $r+[a, b)$ and $r+(a, b]$, appropriately considering closed and open boundaries; “other $_i$ ” represents any other input not explicitly considered leaving the same state; and “any $_i$ ” and “any $_o$ ” represent any possible input and output (including δ), respectively. In any case $-_i$ and $-_o$ are not considered since they are not part of the alphabet of the LTS. Also, we note that any possible sequence of inputs becomes enabled in the bottom states in \mathcal{R} in Figure 5.1 (omitted in the picture).



The reference implementation \mathcal{R} is obtained according to Definition 5.17. In order to give an idea of its construction we focus on the states σ such that $|\sigma| = 1$ (i.e., $\sigma \in \text{In} \cup \{\delta\}$) – other cases are simpler. First, notice that

$$\text{traces}_\omega(\text{Std}_\delta) = \delta^\omega + \delta^* i o \delta^\omega + \delta^* i (o + \kappa_o) \delta^\omega + \delta^* (i + \kappa_i) (o + \kappa_o) \delta^\omega.$$

Here we use ω -regular notation to describe the set of traces. This means that $\text{traces}_\omega(\text{Std}_\delta)$ contains the trace that remains quiet indefinitely (namely δ^ω), all traces that may stay quiet for a while, receive an input i , produce and output o , and remain quiet indefinitely (i.e., any trace in $\delta^* i o \delta^\omega$), and so on. Hence, the set $\text{traces}_\omega(\text{Std}_\delta) \downarrow_i$ is then

$$\text{traces}_\omega(\text{Std}_\delta) \downarrow_i = -_i^\omega + -_i^* i -_i^\omega + -_i^* (i + \kappa_i) -_i^\omega.$$

Suppose $\sigma \in i+[-\kappa_i, 0)$ and $a \in o + [-\kappa_o, 2\kappa_o]$. Then, $\sigma_i = i -_i^\omega \in \text{traces}_\omega(\text{Std}_\delta) \downarrow_i$ is the only standard trace satisfying $\forall j \leq 2: d_{\text{In}}((\sigma \cdot a) \downarrow_i[..j], \sigma_i[..j]) \leq \kappa_i$. If $a \in o + [-\kappa_o, \kappa_o]$ take $\sigma_S = i o \delta^\omega$, then $\sigma_S \downarrow_i = \sigma_i \wedge d_{\text{Out}_\delta}(a \downarrow_o, \sigma_S[|\sigma| + 1] \downarrow_o) \leq \kappa_o$ holds. If $a \in o + [0, 2\kappa_o]$, then $\sigma_S = i(o + \kappa_o) \delta^\omega$ is the one that does the job. Therefore $\sigma \xrightarrow{a}_{\mathcal{R}} \sigma \cdot a$. This case defines the schematic transition $i+[-\kappa_i, 0) \xrightarrow{o+[-\kappa_o, 2\kappa_o]}_{\mathcal{R}} i+[-\kappa_i, 0) o+[-\kappa_o, 2\kappa_o]$.

If instead $a \in \text{Out}$ but $a \notin o + [-\kappa_o, 2\kappa_o]$, then no a -outgoing transition from $\sigma \in i+[-\kappa_i, 0)$ is possible since no matching σ_S can be found. However, if $a \in \text{In}$, no $\sigma_i \in \text{traces}_\omega(\text{Std}_\delta) \downarrow_i$ satisfies $\forall j \leq 2: d_{\text{In}}((\sigma \cdot a) \downarrow_i[..j], \sigma_i[..j]) \leq \kappa_i$. As the antecedent of the implication is false, any input defines a valid outgoing transition from a state $\sigma \in i+[-\kappa_i, 0)$. This yields the schematic transition $i+[-\kappa_i, 0) \xrightarrow{\text{any}_i}_{\mathcal{R}} i+[-\kappa_i, 0) \text{any}_i$.

Suppose now $\sigma \in i+[0, 2\kappa_i]$ and $a \in o + [0, 2\kappa_o]$. We consider the two subcases $\sigma \in i+[0, \kappa_i]$ and $\sigma \in i+(\kappa_i, 2\kappa_i]$. If $\sigma \in i+(\kappa_i, 2\kappa_i]$ then $\sigma_i = (i+\kappa_i) \dashv_i^\omega \in \text{traces}_\omega(\text{Std}_\delta) \dashv_i$ is the only one satisfying $\forall j \leq 2: d_{\text{In}}((\sigma \cdot a) \dashv_i[.j], \sigma_i[.j]) \leq \kappa_i$ and the construction follows similarly as above. If instead $\sigma \in i+[0, \kappa_i]$, then every $\sigma_i \in \{i \dashv_i^\omega, (i+\kappa_i) \dashv_i^\omega\}$ satisfies $\forall j \leq 2: d_{\text{In}}((\sigma \cdot a) \dashv_i[.j], \sigma_i[.j]) \leq \kappa_i$. If $\sigma_i = i \dashv_i^\omega$, choose $\sigma_S = i(o+\kappa_o) \delta^\omega$, and if $\sigma_i = (i+\kappa_i) \dashv_i^\omega$, choose $\sigma_S = (i+\kappa_i)(o+\kappa_o) \delta^\omega$. In both of these cases, $\sigma_S \dashv_i = \sigma_i \wedge d_{\text{Out}_\delta}(a \dashv_o, \sigma_S[|\sigma|+1] \dashv_o) \leq \kappa_o$ is satisfied. Hence $\sigma \xrightarrow{a} \mathcal{R} \sigma \cdot a$. Putting both subcases together yields the schematic transition $i+[0, 2\kappa_i] \xrightarrow{o+[0, 2\kappa_o]} \mathcal{R} i+[0, 2\kappa_i] o + [0, 2\kappa_o]$.

The case in which $\sigma \in i+[0, 2\kappa_i]$ but $a \notin o + [0, 2\kappa_o]$ follows as before.

If $\sigma \notin i+[-\kappa_i, 2\kappa_i]$ (in other words, “ $\sigma \in \text{other}_i$ ”), there is no trace $\sigma_i \in \text{traces}_\omega(\text{Std}_\delta) \dashv_i$ such that $\forall j \leq 2: d_{\text{In}}((\sigma \cdot a) \dashv_i[.j], \sigma_i[.j]) \leq \kappa_i$, so any transition is possible.

Finally, if $\sigma = \delta$ (omitted in Figure 5.1) the construction would follow just like for the initial state ϵ .

Properties of the Reference Implementation \mathcal{R} . To show that \mathcal{R} is defined in a reasonable way, we will establish some important properties of \mathcal{R} . We start with a fundamental property, which exploits the way \mathcal{R} is constructed to serve as a basis for many of the following proofs. Essentially, every state in \mathcal{R} is “labelled” with the unique trace by which the state is reachable, if it is reachable at all.

Lemma 5.19. Let \mathcal{C} be a context and \mathcal{R} the reference implementation for \mathcal{C} . Then, for all finite paths $\rho \in \text{paths}_*(\mathcal{R})$ it holds that $\text{last}(\rho) = \text{trace}(\rho)$.

Proof. We proceed by induction on the number of states in ρ . If ρ has only one state then $\rho = \epsilon = \text{last}(\rho) = \text{trace}(\rho)$, since ϵ is the initial state in \mathcal{R} .

Suppose now, that $\rho = (\rho' a s) \in \text{paths}_*(\mathcal{R})$. By induction, $\text{last}(\rho') = \text{trace}(\rho')$. By Definition 5.17, $\text{last}(\rho') \xrightarrow{a} \mathcal{R} s$ only if $s = \text{last}(\rho') \cdot a$. But $\text{last}(\rho) = s = \text{last}(\rho') \cdot a = \text{trace}(\rho') \cdot a = \text{trace}(\rho)$, which proves the lemma. \square

As a consequence, for every trace of \mathcal{R} we can reconstruct the unique path with this trace.

Corollary 5.20. Let \mathcal{C} be a context, \mathcal{R} the reference implementation for \mathcal{C} , $\rho \in \text{paths}_*(\mathcal{R})$ a finite path of \mathcal{R} and $\sigma = \text{trace}(\rho)$ its trace. Then ρ is exactly the path $\epsilon \sigma[1] (\sigma[.1]) \sigma[2] (\sigma[.2]) \cdots (\sigma[.|\sigma|-1]) \sigma[|\sigma|] (\sigma[.|\sigma|])$.

One of the most desired properties of \mathcal{R} that we will show is that it is the largest implementation for the context it is constructed from. The following lemma shows a similar property. It assumes implementations to be LTS (rather than IOTS) and it considers only u-robust cleanness. This Lemma will be central to many of the following proofs.

Lemma 5.21. Let \mathcal{C} be a context and \mathcal{R} the reference implementation for \mathcal{C} . Then, for every LTS L that is u-robustly clean, it holds that $\text{traces}_\omega(\mathsf{L}_\delta) \subseteq \text{traces}_\omega(\mathcal{R})$.

Proof. For a proof by contradiction, suppose that there is some L that is u-robustly clean, but which has some trace $\sigma \in \text{traces}_\omega(\mathsf{L}_\delta)$ that is not a trace of \mathcal{R} , i.e. $\sigma \notin \text{traces}_\omega(\mathcal{R})$. Since $\sigma \notin \text{traces}_\omega(\mathcal{R})$, there must be some $k > 0$ for which $\sigma[..(k-1)] \in \text{traces}_*(\mathcal{R})$, but $\sigma[..k] \notin \text{traces}_*(\mathcal{R})$. Hence, there is no transition $\sigma[..(k-1)] \xrightarrow{\sigma[k]}_{\mathcal{R}} \sigma[..k]$ in \mathcal{R} . This can only be, because the premise of Definition 5.17 is not satisfied, i.e., there is some $\sigma_i \in \text{traces}_\omega(\text{Std}_\delta) \downarrow_i$, such that (1) $\mathcal{V}(k, \sigma, \sigma_i)$ and (2) for all standard traces $\sigma_S \in \text{traces}_\omega(\text{Std}_\delta)$ with $\sigma_S \downarrow_i = \sigma_i$ it holds that $d_{\text{Out}_\delta}(\sigma[k] \downarrow_o, \sigma_S[k] \downarrow_o) > \kappa_o$.

Let $\sigma_{io} \in \text{traces}_\omega(\text{Std}_\delta)$ such that $\sigma_{io} \downarrow_i = \sigma_i$. From Definition 5.2 we get for $\mathsf{L}, \sigma_{io}, \sigma$ and k with (1) a trace $\sigma'' \in \text{traces}_\omega(\text{Std}_\delta)$ with $\sigma'' \downarrow_i = \sigma_{io} \downarrow_i = \sigma_i$ and $d_{\text{Out}_\delta}(\sigma[..k] \downarrow_o, \sigma''[..k] \downarrow_o) \leq \kappa_o$. From the assumption that d_{Out_δ} is past-forgetful, we get that $d_{\text{Out}_\delta}(\sigma[k] \downarrow_o, \sigma''[k] \downarrow_o) \leq \kappa_o$, which is a contradiction to (2). \square

Definition 5.17 models an LTS that is deterministic and quiescence is added explicitly instead of relying on the quiescence closure. As a consequence, outputs and quiescence may coexist as options in a state, i.e., they are not mutually exclusive. Lemma 5.22 shows that this is done in the spirit of model-based testing theory and **io**co, that is, \mathcal{R}_δ is identical to \mathcal{R} .

Lemma 5.22. Let \mathcal{C} be a satisfiable context and \mathcal{R} the reference implementation for \mathcal{C} . Then, the quiescence closure \mathcal{R}_δ of \mathcal{R} is exactly \mathcal{R} .

Proof. We have to show that for every state $\sigma \in (\text{In} \cup \text{Out}_\delta)^*$, there is a transition $\sigma \xrightarrow{o}_{\mathcal{R}} \sigma \cdot o$ in \mathcal{R} with $o \in \text{Out}_\delta$. Let $\sigma_i = \sigma \downarrow_i \cdot (-)_i^\omega$ an infinite input trace. We proceed by case-distinction on whether there is a trace $\sigma_S \in \text{traces}_\omega(\text{Std}_\delta)$ such that $\mathcal{V}(|\sigma| + 1, \sigma_i, \sigma_S)$ holds. If this is not the case, the premise of Definition 5.17 does not hold and hence we get that for all $o \in \text{Out}_\delta$ a transition $\sigma \xrightarrow{o}_{\mathcal{R}} \sigma \cdot o$ in \mathcal{R} .

In the case that the assumption does hold, we get from satisfiability of \mathcal{C} and Definition 5.14 an implementation L that is robustly clean and, hence, in particular l-robustly clean. Using Definition 5.1, we get a trace $\sigma'' \in \text{traces}_\omega(\mathsf{L}_\delta)$ with $\sigma'' \downarrow_i = \sigma_i$ and $d_{\text{Out}_\delta}(\sigma''[|\sigma| + 1] \downarrow_o, \sigma_S[|\sigma| + 1] \downarrow_o) \leq \kappa_o$. From Lemma 5.21 we get that $\sigma'' \in \text{traces}_\omega(\mathcal{R})$. For this trace to exist it is necessary that there is the transition $\sigma''[..|\sigma|] \xrightarrow{\sigma''[|\sigma|+1]}_{\mathcal{R}} \sigma''[..|\sigma| + 1]$ in \mathcal{R} . Hence, we know (from Definition 5.17) that for every trace $\sigma_S \in \text{traces}_\omega(\text{Std}_\delta) \downarrow_i$ for which $\mathcal{V}(|\sigma| + 1, \sigma'', \sigma_S)$ holds, there is some $\hat{\sigma} \in \text{traces}_\omega(\text{Std}_\delta)$ with $\hat{\sigma} \downarrow_i = \sigma_S \downarrow_i$ and $d_{\text{Out}_\delta}(\sigma''[|\sigma| + 1], \hat{\sigma}[|\sigma| + 1] \downarrow_o) \leq \kappa_o$. Since $\sigma'' \downarrow_i = \sigma_i$ and in particular $\sigma \downarrow_i \cdot (-)_i = \sigma''[..|\sigma| + 1] \downarrow_i$, we

have that for every σ_S and $j \leq |\sigma| + 1$, the equivalence $d_{\text{In}}(\sigma''[\cdot \cdot j] \downarrow_i, \sigma_S[\cdot \cdot j] \downarrow_i) \leq \kappa_i \iff d_{\text{In}}((\sigma \cdot \sigma''[|\sigma| + 1])[\cdot \cdot j] \downarrow_i, \sigma_S[\cdot \cdot j] \downarrow_i) \leq \kappa_i$ holds. Hence, for every $\sigma_S \in \text{traces}_\omega(\text{Std}_\delta) \downarrow_i$ with $\mathcal{V}(|\sigma| + 1, (\sigma \cdot \sigma''[|\sigma| + 1]), \sigma_S)$, we can provide a $\hat{\sigma} \in \text{traces}_\omega(\text{Std}_\delta)$ with $\hat{\sigma} \downarrow_i = \sigma_S \downarrow_i$ and $d_{\text{Out}_\delta}(\sigma''[|\sigma| + 1], \hat{\sigma}[|\sigma| + 1] \downarrow_o) \leq \kappa_o$. By Definition 5.17 we know that the transition $\sigma[\cdot \cdot |\sigma|] \xrightarrow{\sigma''[|\sigma| + 1]}_{\mathcal{R}} \sigma[\cdot \cdot |\sigma|] \cdot \sigma''[|\sigma| + 1]$ exists in \mathcal{R} . Since $\sigma'' \downarrow_i = \sigma_i$, we know that $\sigma'' \downarrow_i[|\sigma| + 1] = \text{--}_i$ and hence $\sigma''[|\sigma| + 1] \in \text{Out}_\delta$. \square

The LTS \mathcal{R} is supposed to serve as an implementation. Hence, Lemma 5.23 shows that \mathcal{R} is input-enabled and hence is an IOTS.

Lemma 5.23. Let \mathcal{C} be a satisfiable context with standard Std and let \mathcal{R} be constructed from \mathcal{C} . Then \mathcal{R} is an input-output transition system.

Proof. By construction, \mathcal{R} is a labelled transition system. By Definition 2.1 an LTS is an IOTS, if it is input-enabled. Hence, we have to show that for every state $\sigma \in (\text{In} \cup \text{Out})^*$ it holds for every $i \in \text{In}$ that there is a transition $\sigma \xrightarrow{i}_{\mathcal{R}} \sigma \cdot i$ in \mathcal{R} . To have this transition, the premise of Definition 5.17 must be satisfied. Let $\sigma_i \in \text{traces}_*(\text{Std}_\delta) \downarrow_i$ and accordingly $\sigma_S \in \text{traces}_*(\text{Std}_\delta)$ a trace with $\sigma_S \downarrow_i = \sigma_i$. Assume that $\mathcal{V}(|\sigma| + 1, (\sigma \cdot i), \sigma_i)$ holds (otherwise the lemma holds trivially). We pick σ_S for the existential quantifier. By definition $\sigma_S \downarrow_i = \sigma_i$, so it suffices to show that $d_{\text{Out}_\delta}(i \downarrow_o, \sigma_S[|\sigma| + 1] \downarrow_o) = d_{\text{Out}_\delta}(-_o, \sigma_S[|\sigma| + 1] \downarrow_o) \leq \kappa_o$. We continue by case distinction of whether $\sigma_S[|\sigma| + 1] \in \text{In}$. If this is the case, we are immediately done, because $d_{\text{Out}_\delta}(-_o, -_o) = 0 \leq \kappa_o$.

If instead $\sigma_S[|\sigma| + 1] \in \text{Out}_\delta$, from satisfiability of \mathcal{C} we get an LTS L that is robustly clean. Hence, L is l-robustly clean and u-robustly clean. From l-robust cleaness of L we get, with σ_S for σ , $(\sigma \cdot i)$ for σ' and $k = |\sigma| + 1$, a trace $\hat{\sigma} \in \text{traces}_\omega(\text{L}_\delta)$ with $\hat{\sigma} \downarrow_i = (\sigma \cdot i) \downarrow_i$ and $d_{\text{Out}_\delta}(\hat{\sigma}[|\sigma| + 1] \downarrow_o, \sigma_S[|\sigma| + 1] \downarrow_o) \leq \kappa_o$. From u-robust cleaness of L and Lemma 5.21 we get that $\hat{\sigma} \in \text{traces}_\omega(\mathcal{R})$. We get from $d_{\text{Out}_\delta}(\hat{\sigma}[|\sigma| + 1] \downarrow_o, \sigma_S[|\sigma| + 1] \downarrow_o) \leq \kappa_o$ and $\hat{\sigma} \downarrow_i = (\sigma \cdot i) \downarrow_i$ that $d_{\text{Out}_\delta}(-_o, \sigma_S[|\sigma| + 1] \downarrow_o) \leq \kappa_o$, which concludes the proof. \square

Each context contains some standard behaviour modelled as an LTS Std . The reference implementation for a context should be constructed in a way such that the standard behaviour is contained in \mathcal{R} , i.e., $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(\mathcal{R}_\delta)$. Lemma 5.24 shows that this is the case for \mathcal{R} .

Lemma 5.24. Let $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a satisfiable context and \mathcal{R} the reference implementation for \mathcal{C} . Then $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(\mathcal{R}_\delta)$.

Proof. The lemma follows from Proposition 5.15 and Lemmas 5.21 and 5.22. \square

\mathcal{R} is modelled by adding all transitions satisfying Definition 5.2. Lemma 5.25 confirms that, conversely, \mathcal{R} satisfies u-robust cleanness. Then, Lemma 5.26 shows that \mathcal{R} satisfies also l-robust cleanness.

Lemma 5.25. Let \mathcal{C} be a context and \mathcal{R} the reference implementation for \mathcal{C} . Then, \mathcal{R} is u-robustly clean w.r.t. \mathcal{C} .

Proof. From Lemma 5.23 we know that \mathcal{R} is an IOTS and from Lemma 5.24 that $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(\mathcal{R}_\delta)$. Let $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$, $\sigma' \in \text{traces}_\omega(\mathcal{R}_\delta)$, $k > 0$ and assume that $d_{\text{In}}(\sigma[..j]\downarrow_i, \sigma'[..j]\downarrow_i) \leq \kappa_i$ for all $j \leq k$. Using Lemma 5.22 we know that $\sigma' \in \text{traces}_\omega(\mathcal{R})$. By Corollary 5.20, we get that there must be some path $\rho = \epsilon \sigma'[1](\sigma'[..1]) \dots (\sigma'[..k-1])\sigma'[k](\sigma'[..k]) \in \text{paths}_*(\mathcal{R})$. In particular $\sigma'[..k-1] \xrightarrow{\sigma'[k]} \sigma'[..k]$ is a transition in \mathcal{R} . By Definition 5.17, we know that for all $\sigma_i \in \text{traces}(\text{Std}_\delta)\downarrow_i$ with $\mathcal{V}(k, \sigma'[..k], \sigma_i)$ (which is equivalent to $\mathcal{V}(k, \sigma', \sigma_i)$), there is some $\sigma_S \in \text{traces}_\omega(\text{Std}_\delta)$ with $\sigma_S\downarrow_i = \sigma_i$ and $d_{\text{Out}_\delta}(\sigma[k]\downarrow_o, \sigma_S[k]\downarrow_o) \leq \kappa_o$ (*).

Since $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$ then $\sigma\downarrow_i \in \text{traces}_\omega(\text{Std}_\delta)\downarrow_i$. Suppose $\mathcal{V}(k, \sigma', \sigma)$ holds (otherwise the lemma holds trivially). Then, we get from (*) a trace $\sigma_S \in \text{traces}_\omega(\text{Std}_\delta)$ with $\sigma_S\downarrow_i = \sigma\downarrow_i$ such that $d_{\text{Out}_\delta}(\sigma[k]\downarrow_o, \sigma_S[k]\downarrow_o) \leq \kappa_o$. \square

Lemma 5.26. Let \mathcal{C} be a satisfiable context and \mathcal{R} the reference implementation for \mathcal{C} . Then, \mathcal{R} is l-robustly clean w.r.t. \mathcal{C} .

Proof. From Lemma 5.23 we know that \mathcal{R} is an IOTS and from Lemma 5.24 that $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(\mathcal{R}_\delta)$. Let $\sigma_1 \in \text{traces}_\omega(\text{Std}_\delta)$, $\sigma_2 \in \text{traces}_\omega(\mathcal{R}_\delta)$ and $k > 0$. Using Lemma 5.22 we know that $\sigma_2 \in \text{traces}_\omega(\mathcal{R})$. Suppose $\mathcal{V}(k, \sigma_1, \sigma_2)$ holds (otherwise, the lemma holds trivially). From satisfiability of \mathcal{C} we get an LTS L that is robustly clean w.r.t. \mathcal{C} . Hence, L is l-robustly clean and u-robustly clean. Since L is an IOTS, it must be input-enabled; hence, there is some $\sigma_3 \in \text{traces}_\omega(\text{L}_\delta)$ with $(\sigma_3)\downarrow_i = \sigma_2\downarrow_i$. From Definition 5.1, with σ_1 for σ , σ_3 for σ' , k and $\mathcal{V}(k, \sigma_1, \sigma_3)$, we get a trace $\sigma'' \in \text{traces}_\omega(\text{L})$ with $\sigma''\downarrow_i = \sigma_3\downarrow_i$ (and hence $\sigma''\downarrow_i = \sigma_2\downarrow_i$) and $d_{\text{Out}_\delta}(\sigma''[k]\downarrow_o, \sigma_1[k]\downarrow_o) \leq \kappa_o$. Since L is u-robustly clean, we get from Lemma 5.21 that $\sigma'' \in \text{traces}_\omega(\mathcal{R})$, which concludes the proof. \square

With the properties of \mathcal{R} established in this section it is easy to show that \mathcal{R} is robustly clean w.r.t. the context it is constructed from.

Theorem 5.27. Let $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be a satisfiable context and \mathcal{R} the reference implementation for \mathcal{C} . Then \mathcal{R} is robustly clean w.r.t. \mathcal{C} .

Proof. Definition 5.3 requires that \mathcal{R} is an IOTS, which is shown in Lemma 5.23. Furthermore, from Lemma 5.24 we get that $\text{traces}_\omega(\text{Std}_\delta) \subseteq \text{traces}_\omega(\mathcal{R})$. With Lemmas 5.22, 5.25 and 5.26 we get that \mathcal{R} satisfies Definitions 5.1 and 5.2. Hence, \mathcal{R} satisfies Definition 5.3. \square

Furthermore, it is not difficult to show that \mathcal{R} is indeed the largest implementation that is allowed by the context it was constructed from.

Theorem 5.28. Let $\mathcal{C} = \langle \text{Std}, d_{\text{in}}, d_{\text{out}}, \kappa_i, \kappa_o \rangle$ be a satisfiable context and \mathcal{R} the reference implementation for \mathcal{C} . Then \mathcal{R} is the largest implementation for \mathcal{C} .

Proof. We know from Theorem 5.27 that \mathcal{R} is an IOTS, which is robustly clean w.r.t. \mathcal{C} . It remains to show that for every LTS L' that is robustly clean w.r.t. \mathcal{C} , $\text{traces}_\omega(L_\delta) \subseteq \text{traces}_\omega(\mathcal{R})$. Any such L' satisfies in particular Definition 5.2, so it follows directly from Lemma 5.21 that \mathcal{R} is the largest implementation for \mathcal{C} . \square

5.3 Model-Based Doping Tests

In this section, we develop a testing algorithm for robust cleanness of mixed-IO systems. Following the conceptual ideas behind **io**co, we need to construct a specification that is compatible with our notion of robust cleanness in such a way that a test suite can be derived. Intuitively, such a specification must be able to foresee every behaviour of the system that is allowed by the contract. It turns out that we can take up the model-based testing theory right away with \mathcal{R} as the specification *Spec*. We get an algorithm that can generate doping test suites provided we are able to prove that \mathcal{R} is constructed in such a way that whenever an IUT \mathcal{I} is robustly clean \mathcal{I} **io**co \mathcal{R} holds, i.e.,

$$\forall \sigma \in \text{traces}_*(\mathcal{R}_\delta) : \text{out}(\mathcal{I}_\delta \text{ after } \sigma) \subseteq \text{out}(\mathcal{R}_\delta \text{ after } \sigma). \quad (5.1)$$

Working out this proof requires frequent reasoning about the functions **out** and **after**. However, there is a strong connection between these functions and reasoning about traces, which is established in Lemma 5.29. This enables us to use all the properties considering traces of \mathcal{R} from Section 5.2.

Lemma 5.29. Let L be an LTS, $\sigma \in \text{traces}_*(L_\delta)$ a suspension trace of L and o an output. Then, $o \in \text{out}(L_\delta \text{ after } \sigma)$ if and only if $\sigma \cdot o \in \text{traces}_*(L_\delta)$.

Proof. By definition, $o \in \text{out}(L_\delta \text{ after } \sigma)$ if and only if there is some $q \in (L_\delta \text{ after } \sigma)$ for which there is some q' and a transition $q \xrightarrow{o} q'$. This holds if and only if there is a path $\rho \in \text{paths}_*(L_\delta)$ with $\text{trace}(\rho) = \sigma$, $\text{last}(\rho) = q$ and $q \xrightarrow{o} q'$. Equivalently, there can be a path $\rho' \in \text{paths}_*(L_\delta)$ with $\text{trace}(\rho') = \sigma \cdot o$, which is the case if and only if $\sigma \cdot o \in \text{traces}_*(L_\delta)$. \square

The following theorem shows that \mathcal{R} , indeed, satisfies the conditions to serve as a specification for model-based testing. Its proof translates the requirements enforced by **io**co into trace properties and exploits the properties of \mathcal{R} established in Section 5.2.

Theorem 5.30. Let $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be a satisfiable context, \mathcal{R} the reference implementation for \mathcal{C} and let \mathcal{I} be an IOTS that is robustly clean w.r.t. \mathcal{C} . Then, it holds that \mathcal{I} **ioco** \mathcal{R} .

Proof. We have to show that for all $\sigma \in \text{traces}_*(\mathcal{R}_\delta)$ it holds that $\text{out}(\mathcal{I}_\delta \text{ after } \sigma) \subseteq \text{out}(\mathcal{R}_\delta \text{ after } \sigma)$. From Lemma 5.22 we know that $\sigma \in \text{traces}_*(\mathcal{R})$. If $\text{out}(\mathcal{I}_\delta \text{ after } \sigma) = \emptyset$ the theorem trivially holds. Otherwise, there is some $o \in \text{out}(\mathcal{I}_\delta \text{ after } \sigma) \subseteq \text{Out}_\delta$ and $\sigma \cdot o \in \text{traces}_*(\mathcal{I}_\delta)$ follows with Lemma 5.29. By Definition 2.2, every state in \mathcal{I}_δ has an outgoing output or quiescence transition and hence there is an infinite trace $\sigma' \in \text{traces}_\omega(\mathcal{I}_\delta)$ with $\sigma'[\cdot, |\sigma| + 1] = \sigma \cdot o$. From Definition 5.16, Theorems 5.27 and 5.28 and robust cleanness of \mathcal{I} , we can conclude that $\sigma' \in \text{traces}_\omega(\mathcal{R}_\delta)$. Since $\sigma \cdot o$ is a finite prefix of σ' , we get that $\sigma \cdot o \in \text{traces}_*(\mathcal{R}_\delta)$. Finally, Lemma 5.29 gives us that $o \in \text{out}(\mathcal{R}_\delta \text{ after } \sigma)$. \square

Theorem 5.30 establishes that we can use Algorithm TG to generate doping tests (in the form of LTS) by using \mathcal{R} as the specification model. From a theoretical point of view, the problem of finding doping tests is solved with Corollary 5.31, which follows directly from the completeness of TG [120, 121].

Corollary 5.31. Let $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a satisfiable context and \mathcal{R} the reference implementation for \mathcal{C} . Then \mathcal{I} **ioco** \mathcal{R} if and only if \mathcal{I} **passes** $\text{TG}(\{\epsilon\})$.

However, there are several issues regarding the practicality of TG. To perform a doping test for a given context \mathcal{C} , we first have to construct \mathcal{R} . \mathcal{R} is the largest implementation for \mathcal{C} and is, hence, infinite in size. Constructing \mathcal{R} is necessary, because \mathcal{R} serves as the specification for model-based testing. In general, a specification LTS may not be computable on-the-fly and hence TG assumes the availability of the full specification upon test case generation. The following test generation algorithm DTG echoes Algorithm TG, however, it does not need \mathcal{R} as input but constructs on-the-fly the relevant information that TG obtains from \mathcal{R} .

$\text{DTG}(h) :=$ choose nondeterministically one of the following processes:

1. **pass**
2. $i; p_i$ where $i \in \text{In}$ and $p_i \in \text{DTG}(h \cdot i)$
 $+ \sum \{o; \text{fail} \mid o \in \text{Out} \wedge o \notin \Theta(h)\}$
 $+ \sum \{o_j; p_{o_j} \mid o_j \in \text{Out} \wedge o_j \in \Theta(h)\}$, where for each $o_j, p_{o_j} \in \text{DTG}(h \cdot o_j)$
3. $\sum \{o; \text{fail} \mid o \in \text{Out} \cup \{\delta\} \wedge o \notin \Theta(h)\}$
 $+ \sum \{o_j; p_{o_j} \mid o_j \in \text{Out} \cup \{\delta\} \wedge o_j \in \Theta(h)\}$, where for each $o_j, p_{o_j} \in \text{DTG}(h \cdot o_j)$

There are two main differences between DTG and TG. First, the input h to DTG is a single trace instead of a set of states. That is because the construction

of DTG follows the same ideas as the construction of \mathcal{R} , where a trace represents a state of the LTS. Moreover, \mathcal{R} is deterministic, so when using TG with \mathcal{R} , the set S always contains exactly one state of \mathcal{R} , which is a trace. The second difference is that DTG uses a function Θ instead of out . Essentially, $\Theta(h)$ captures all output transitions leaving state h in \mathcal{R} (i.e., $\text{out}(\{h\})$) without knowing (or constructing) \mathcal{R} . Thus, $\Theta(h)$ is precisely the set of outputs that satisfies the premise in the definition of \mathcal{R} after the trace h , as stipulated in Definition 5.17. The definition of Θ is shown in Equation (5.2).

$$\begin{aligned} \Theta(h) := \{o \in \text{Out}_\delta \mid & \tag{5.2} \\ & \forall \sigma_i \in \text{traces}_\omega(\text{Std}_\delta) \downarrow_i : \\ & (\forall j \leq |h|+1: d_{\text{In}}(\sigma_i[..j] \downarrow_i, (h \cdot o)[..j] \downarrow_i) \leq \kappa_i) \\ & \Rightarrow \exists \sigma \in \text{traces}_\omega(\text{Std}_\delta) : \sigma \downarrow_i = \sigma_i \downarrow_i \wedge d_{\text{Out}_\delta}(o, \sigma[|h|+1] \downarrow_o) \leq \kappa_o \} \end{aligned}$$

The following lemma confirms that Θ can be used to compute out without knowing \mathcal{R} . Instead, the definition of Θ is defined directly for a context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$. We emphasize this difference in Lemma 5.32 by annotating the functions appropriately, i.e., by $\Theta^{(\mathcal{C})}$ and $\text{out}^{(\mathcal{R})}$.

Lemma 5.32. Let $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be a satisfiable context and \mathcal{R} the reference implementation for \mathcal{C} . For all $h \in (\text{In} \cup \text{Out}_\delta)^*$, $\Theta^{(\mathcal{C})}(h) = \text{out}^{(\mathcal{R})}(\{h\})$.

Proof. Let $h \in (\text{In} \cup \text{Out}_\delta)^*$ and $o \in \text{Out}_\delta$. As per Equation (5.2), $o \in \Theta^{(\mathcal{C})}(h)$ if and only if for any $\sigma_i \in \text{traces}_\omega(\text{Std}_\delta) \downarrow_i$ with $\mathcal{V}(|h|+1, \sigma_i, h \cdot o)$ there exists $\sigma \in \text{traces}_\omega(\text{Std}_\delta)$ such that $\sigma \downarrow_i = \sigma_i \downarrow_i$ and $d_{\text{Out}}(o, \sigma[|h|+1] \downarrow_o) \leq \kappa_o$. However, this is equivalent to the premise of the rule from Definition 5.17, hence $o \in \Theta^{(\mathcal{C})}(h)$ if and only if there is a transition $h \xrightarrow{o} \mathcal{R} h \cdot o$ in \mathcal{R} . In turn, such transition exists if and only if $o \in \text{out}^{(\mathcal{R})}(\{h\})$. \square

Although Algorithm DTG does not require \mathcal{R} as an input, \mathcal{R} still is the specification for which DTG is supposed to generate test cases. Hence, we have to show that \mathcal{I} **ioco** \mathcal{R} if and only if \mathcal{I} **passes** DTG(ϵ) (as in Corollary 5.31). For this, it is serviceable to realise that for every history $h \in (\text{In} \cup \text{Out}_\delta)^*$, the set of test cases TG and DTG generate are identical (i.e., the processes defining the LTS are identical). This is expressed by the following Lemma.

Lemma 5.33. Let \mathcal{C} be a satisfiable context and \mathcal{R} the reference implementation for \mathcal{C} . Then, for every process $p \in \mathcal{P}$ and history $h \in (\text{In} \cup \text{Out}_\delta)^*$, it holds that $p \in \text{TG}(\{h\})$ if and only if $p \in \text{DTG}(h)$.

Proof. We prove the claim by structural induction on p . If p is a process name, then $p = \mathbf{fail}$ or $p = \mathbf{pass}$. Neither TG nor DTG produce **fail** for any input, however, both can always produce **pass**.

If $p = \sum_{z \in Z} a_z; p_z$, both TG and DTG can use choices (2) and (3) to generate p . We first show $p \in \text{TG}(\{h\}) \Rightarrow p \in \text{DTG}(h)$ and distinguish between whether p is constructed by choice (2) or (3) of TG.

For case (2), we fix some arbitrary $i \in \text{In}$ and $p_i \in \text{TG}(\{h\} \text{ after } i)$. Notice that $(\{h\} \text{ after } i)$ is always non-empty, because \mathcal{R} is input-enabled (Lemma 5.23). Furthermore, we fix a mapping from accepted outputs to one of the possible recursively computed subprocess $\mathcal{F} := \{(o, p_o) \mid o \in \text{Out} \cap \text{out}(\{h\}) \wedge p_o \in \text{TG}(\{h\} \text{ after } o)\}$. Then, choice (2) of TG produces exactly one test, which is $p = i; p_i + \sum\{o; \text{fail} \mid o \in \text{Out} \wedge o \notin \text{out}(\{h\})\} + \sum\{o; \mathcal{F}(o) \mid o \in \text{Out} \wedge o \in \text{out}(\{h\})\}$. We can rewrite the test case to $p' = i; p_i + \sum\{o; \text{fail} \mid o \in \text{Out} \wedge o \notin \Theta(h)\} + \sum\{o; \mathcal{F}(o) \mid o \in \text{Out} \wedge o \in \Theta(h)\}$ by using that $\text{out}(\{h\}) = \Theta(h)$ from Lemma 5.32. From Definition 5.17 it follows that for every $o \in \text{out}(\{h\})$, $(\{h\} \text{ after } o) = \{h \cdot o\}$. Hence, $\mathcal{F} = \{(o, p_o) \mid o \in \text{Out} \cap \Theta(h) \wedge p_o \in \text{TG}(\{h \cdot o\})\} = \{(o, p_o) \mid o \in \text{Out} \cap \Theta(h) \wedge p_o \in \text{DTG}(h \cdot o)\}$ with the inductive hypothesis. From Lemma 5.23 and Definition 5.17 we know that $p_i \in \text{TG}(\{h \cdot i\})$ and hence by the inductive hypothesis $p_i \in \text{DTG}(h \cdot i)$. Now, for the fixed i, p_i and \mathcal{F} , p' is exactly the test that is generated by choice (2) of DTG(h).

For case (3) of TG, we fix a mapping from accepted outputs to one of the possible recursively computed subprocess $\mathcal{F} := \{(o, p_o) \mid o \in \text{Out}_\delta \cap \text{out}(\{h\}) \wedge p_o \in \text{TG}(\{h\} \text{ after } o)\}$. Then, choice (3) of TG produces exactly one test, which is $\sum\{o; \text{fail} \mid o \in \text{Out}_\delta \wedge o \notin \text{out}(\{h\})\} + \sum\{o; \mathcal{F}(o) \mid o \in \text{Out}_\delta \wedge o \in \text{out}(\{h\})\}$. We can rewrite the test case to $p' = i; p_i + \sum\{o; \text{fail} \mid o \in \text{Out}_\delta \wedge o \notin \Theta(h)\} + \sum\{o; \mathcal{F}(o) \mid o \in \text{Out}_\delta \wedge o \in \Theta(h)\}$ by using $\text{out}(\{h\}) = \Theta(h)$ as per Lemma 5.32. From Definition 5.17 it follows that for every $o \in \text{out}(\{h\})$, $(\{h\} \text{ after } o) = \{h \cdot o\}$. From this, we can conclude $\mathcal{F} = \{(o, p_o) \mid o \in \text{Out}_\delta \cap \Theta(h) \wedge p_o \in \text{TG}(\{h \cdot o\})\}$ and then $\mathcal{F} = \{(o, p_o) \mid o \in \text{Out}_\delta \cap \Theta(h) \wedge p_o \in \text{DTG}(h \cdot o)\}$ with the inductive hypothesis. Now, for the fixed \mathcal{F} , p' is exactly the test that is generated by choice (3) of DTG(h).

The proof for $p \in \text{DTG}(h) \Rightarrow p \in \text{TG}(\{h\})$ is analogue. \square

With Lemma 5.33 and Corollary 5.31 we get soundness and exhaustiveness of DTG. Altogether, DTG serves as an algorithm that can generate sound doping tests. If a test fails for some implementation, we know that it is doped.

Theorem 5.34. Let $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a satisfiable context and \mathcal{I} an implementation. If \mathcal{I} is robustly clean w.r.t. \mathcal{C} , then \mathcal{I} **passes** DTG(ϵ).

Proof. Let \mathcal{R} be the reference implementation for \mathcal{C} . With Lemma 5.33 and Corollary 5.31 we get that \mathcal{I} **passes** DTG(ϵ) if and only if \mathcal{I} **ioco** \mathcal{R} . According to Theorem 5.30 the latter holds if \mathcal{I} is robustly clean w.r.t. \mathcal{C} . \square

It is worth noting that this theorem does not imply that \mathcal{I} is robustly clean if \mathcal{I} always passes DTG. This is due to the intricacies of actual hyperproperties.

By testing, we will never be able to verify l-robust cleanness (even if we consider infinitely large test suites), because this needs a simultaneous view on all possible execution traces of \mathcal{I} . During testing, however, we always can observe only a single trace. Intuitively, this is because l-robust cleanness effectively puts a constraint on the lower bound of the size of the sets of outputs that a system must be able to produce, whereas u-robust cleanness enforces an upper bound. A violation of the upper-bound constraint is irrevocable, i.e., once observed, the system is for sure not robustly clean. However not having observed an output does not exclude the possibility for observing it in the future.

Bounded-Depth Doping Tests We developed and proved correct Algorithm DTG, which enables model-based testing for some context \mathcal{C} w.r.t. **ioco** without the need to explicitly construct \mathcal{R} , which is infinite in size. Nevertheless, practical problems remain. First, it might still be the case that a generated test case is an LTS of infinite size. Second, even for finite test cases a practitioner might consider it a waste of computing resources to construct a test covering all possible answers of the implementation under test, instead of dynamically checking conformance of an output once it was received from the implementation. Third, function Θ , although independent of the availability of \mathcal{R} , can be hard to compute (in terms of finding an algorithm), as it involves infinite traces. So, in light of the nature of testing, namely that every test eventually has to end, it seems reasonable to modify the acceptance predicate Θ so that it considers finite traces for its decision. Such a bounded-depth construction is provided as Equation (5.3).

$$\begin{aligned} \Theta_b(h) := \{o \in \text{Out}_\delta \mid & \hspace{15em} (5.3) \\ & \forall \sigma_i \in \text{traces}_b(\text{Std}_\delta) \downarrow_i: \\ & (\forall j \leq |h|+1: d_{\text{In}}(\sigma_i[..j] \downarrow_i, (h \cdot o)[..j] \downarrow_i) \leq \kappa_i) \\ & \Rightarrow \exists \sigma \in \text{traces}_b(\text{Std}_\delta): \sigma \downarrow_i = \sigma_i \downarrow_i \wedge d_{\text{Out}_\delta}(o, \sigma[|h|+1] \downarrow_o) \leq \kappa_o \} \end{aligned}$$

For test history of length at most b , Θ_b delivers all outputs that are accepted by some context. It is computable provided that the standard behaviour **Std** is modelled by a finite LTS and that **In** and **Out** are bounded and discretised. The only variation w.r.t. Θ in Equation (5.2) lies in the use of the set $\text{traces}_b(\text{Std}_\delta)$, instead of $\text{traces}_\omega(\text{Std}_\delta)$, so as to return all traces of Std_δ whose length is exactly b . Since Std_δ is finite, Θ_b can indeed be implemented.

We get a bounded-depth test generation algorithm DTG_b by replacing every occurrence of Θ in DTG by Θ_b and by forcing case 1 if and only if $|h| = b$. Since Θ_b only considers finite traces, it conservatively includes extra outputs thus making tests more permissive. This is due to the existential quantifier in the last line of Equation (5.3): it may be the case that the b -prefix of some

infinite trace satisfies this expression, but no infinite extension of such prefix in Std_δ does. Therefore, we have the following variation of Lemma 5.32.

Lemma 5.35. Let $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ be a context and \mathcal{R} the reference implementation for \mathcal{C} . For all $b > 0$ and $h \in (\text{In} \cup \text{Out}_\delta)^*$ with $|h| < b$, $\Theta_b^{(\mathcal{C})}(h) \supseteq \text{out}^{(\mathcal{R})}(\{h\})$.

Proof. From Lemma 5.32 we get that $\Theta^{(\mathcal{C})}(h) = \text{out}^{(\mathcal{R})}(\{h\})$, hence it suffices to show that $\Theta^{(\mathcal{C})}(h) \subseteq \Theta_b^{(\mathcal{C})}(h)$. Let $o \in \Theta^{(\mathcal{C})}(h)$. To show that $o \in \Theta_b(h)$ we may assume an arbitrary $\sigma_i \in \text{traces}_b(\text{Std}_\delta) \downarrow_i$ with $\mathcal{V}(|h| + 1, \sigma_i, h \cdot o)$ (notice that $|h| + 1 \leq b$). $\sigma_i \in \text{traces}_b(\text{Std}_\delta) \downarrow_i$ implies that there is some $\sigma_{io} \in \text{traces}_b(\text{Std}_\delta)$ with $\sigma_{io} \downarrow_i = \sigma_i$. By Definition 2.2, there is an infinite trace $\hat{\sigma}_{io} \in \text{traces}_\omega(\text{Std}_\delta)$ with $\hat{\sigma}_{io}[\cdot b] = \sigma_{io}$. $\mathcal{V}(|h| + 1, \hat{\sigma}_{io}, h \cdot o)$ still holds, as $|h| + 1 \leq b = |\sigma_{io}|$ and $\hat{\sigma}_{io}[\cdot |h| + 1] \downarrow_i = \sigma_i$. From $o \in \Theta^{(\mathcal{C})}(h)$ and Equation (5.2) we get for $\hat{\sigma}_{io}$ and $\mathcal{V}(|h| + 1, \hat{\sigma}_{io}, h \cdot o)$ a trace $\hat{\sigma} \in \text{traces}_\omega(\text{Std}_\delta)$ with $\hat{\sigma} \downarrow_i = \hat{\sigma}_{io} \downarrow_i$ and $d_{\text{Out}_\delta}(o, \hat{\sigma}[\cdot |h| + 1] \downarrow_o) \leq \kappa_o$. Let $\sigma = \hat{\sigma}[\cdot |h| + 1]$, then $\sigma \downarrow_i = \sigma_i$ and $d_{\text{Out}_\delta}(o, \sigma[\cdot |h| + 1] \downarrow_o) \leq \kappa_o$. This proves $o \in \Theta_b(h)$. \square

As a consequence of Lemma 5.35, we have that any robustly clean implementation passes the test suite generated by DTG_b , or, expressed inversely, if an implementation fails a test generated by DTG_b , then it is doped. This is stated in the following lemma.

Lemma 5.36. Let Std be a finite LTS, $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a satisfiable context and \mathcal{I} an implementation. If \mathcal{I} is robustly clean w.r.t. \mathcal{C} , then \mathcal{I} **passes** $\text{DTG}_b(\epsilon)$ for every positive integer b .

Proof. Let $b \in \mathbb{N}_+$. We prove the claim by contraposition, i.e., we show that \mathcal{I} is not robustly clean w.r.t. \mathcal{C} , if $\neg(\mathcal{I} \text{ passes } \text{DTG}_b(\epsilon))$. Let $\mathcal{I} = \langle Q, \text{In}, \text{Out}, \rightarrow, q_0 \rangle$. Assume, there is some $t \in \text{DTG}_b(\epsilon)$ and $q' \in Q$, such that **fail** $\parallel q'$ is reachable from $t \parallel q_0$. Let $P = \{p \in \text{paths}_*(t \parallel q_0) \mid \exists q' \in Q. \text{last}(p) = \text{fail} \parallel q'\}$ the set of paths by which such a state can be reached. Let $(t \parallel q_0) a_0 \cdots a_{n-1} (t_n \parallel q_n) a_n$ (**fail** $\parallel q'$) = $p \in P$ be the shortest of these paths, $\sigma = \text{trace}(p)$ be its trace and let $h = \sigma[\cdot |\sigma| - 1]$. Since p is the shortest path in P , evidently $t_n \neq \text{fail}$ and hence $t_n \in \text{DTG}_b(h)$. By definition of DTG_b , a transition from t_n to **fail** is only possible in cases (2) and (3) if $a_n \in \text{Out}_\delta$ (notice that $\text{Out} \subset \text{Out}_\delta$) and $a_n \notin \Theta_b(a_n)$. With Lemma 5.35, we get that $a_n \notin \Theta(a_n)$. Moreover, it is easy to see from the definition of P , that if $\sigma \in \text{traces}_*(t \parallel q_0)$, then also $\sigma \in \text{traces}_*(q_0)$ and hence $\sigma \in \text{traces}_*(\mathcal{I}_\delta)$. As $a_n \notin \Theta(a_n)$ (although $a_n \in \text{Out}_\delta$), according to Equation (5.2), there is some $\sigma_i \in \text{traces}_\omega(\text{Std}_\delta) \downarrow_i$ with $\mathcal{V}(n + 1, \sigma_i, \sigma)$, such that for all $\sigma'' \in \text{traces}_\omega(\text{Std}_\delta)$ with $\sigma'' \downarrow_i = \sigma_i$, it is the case that $d_{\text{Out}_\delta}(a_n, \sigma''[n + 1] \downarrow_o) > \kappa_o$ (*). Let $\sigma_{io} \in \text{traces}_\omega(\text{Std}_\delta)$ be such that

$\sigma_{io} \downarrow_i = \sigma_i$. By Definition 2.2, each state in \mathcal{I}_δ can proceed by some output or quiescence. Hence, there is some infinite suffix $\sigma_+ \in \text{Out}_\delta^\omega$ to σ , such that $(\sigma \cdot \sigma_+) \in \text{traces}_\omega(\mathcal{I}_\delta)$.

Now, assume that \mathcal{I} is robustly clean w.r.t. context \mathcal{C} . Then, \mathcal{I} is in particular u-robustly clean w.r.t. \mathcal{C} and we get from Definition 5.2 for $\sigma_{io}, (\sigma \cdot \sigma_+), (n+1)$ and with $\mathcal{V}(n+1, \sigma_i, \sigma) \iff \mathcal{V}(n+1, \sigma_{io}, \sigma \cdot \sigma_+)$, that there is some trace $\sigma'' \in \text{traces}_\omega(\text{Std}_\delta)$ with $\sigma'' \downarrow_i = \sigma_{io} \downarrow_i = \sigma_i$ and $d_{\text{Out}_\delta}((\sigma \cdot \sigma_+)[n+1] \downarrow_o, \sigma''[n+1] \downarrow_o) = d_{\text{Out}_\delta}(a_n, \sigma''[n+1] \downarrow_o) \leq \kappa_o$. However, this is a contradiction to (*), which concludes the proof. \square

Since \mathcal{I} passes $\text{DTG}_b(\epsilon)$ implies \mathcal{I} passes $\text{DTG}_a(\epsilon)$ for any $a \leq b$, we have in summary arrived at a computable algorithm DTG_b that for sufficiently large b (corresponding to the length of the test) will be able to generate a doping test that will be a convicting witness for any IUT \mathcal{I} that is not robustly clean w.r.t. a given context \mathcal{C} . The transformation of the model-based testing algorithm gets its finishing touch with Algorithm 5.1 presented below, which, similar to the transformation from TG to DTG, circumvents the need to construct the entire test LTS upfront by instead actively reacting to the implementation under test. In this, DT_b constructs on-the-fly only those parts of the test LTS that are necessary at the given point of execution.

The algorithm shares several characteristics with DTG_b . Each call receives the current history of the test as a finite trace of inputs and outputs. DTG_b eventually reaches the **fail** or **pass** state, whereas DT_b explicitly returns either of two values **fail** or **pass**. It chooses one of three cases, where the first case exactly imitates the first case of DTG_b – the test terminates by indicating success. Cases 2 and 3 are similar, however not identical since DT_b explicitly resolves nondeterminism when the IUT offers some output. Case 2 of DTG_b allows to decide nondeterministically to either process this output or to pass some input to the IUT. DT_b instead gives priority to processing the output. Hence, in this case DT_b enforces to use the third case of the algorithm. Notice that we consider DT_b as an on-the-fly algorithm simulating the parallel composition of a test case LTS with \mathcal{I} . Consequently, we assume that one call of the algorithm executes atomically, i.e., if \mathcal{I} does not offer an output in line 4, it also does not offer outputs in line 5. Case 3 handles reception of outputs or detects quiescence. Quiescence can be recognized by using a timeout mechanism that returns δ if no output has been received in a given amount of time, and DT_b verifies whether the output (or its absence) is valid by consulting Θ_b . In case the output is among those foreseen by Θ_b , the test continues recursively. Otherwise, the algorithm terminates with a **fail** verdict. If instead the IUT is not offering an output, it is legitimate (but not necessary) to choose Case 2 so as to pick some input, pass it to the IUT and continue recursively to simulate a transition in the test LTS.

Algorithm 5.1 Bounded-Length Doping Test (DT_b)

Input: history $h \in (\text{In} \cup \text{Out} \cup \{\delta\})^*$
Output: pass or fail

```

1:  $c \leftarrow \Omega_{\text{case}}(h)$  /* Pick from one of three cases */
2: if  $c = 1$  or  $|h| = b$  then
3:   return pass /* Finish test generation */
4: else if  $c = 2$  and no output from  $\mathcal{I}$  is available then
5:    $i \leftarrow \Omega_{\text{in}}(h)$  /* Pick next input */
6:    $i \rightarrow \mathcal{I}$  /* Forward input to IUT */
7:   return  $DT_b(h \cdot i)$  /* Continue with next step */
8: else if  $c = 3$  or output from  $\mathcal{I}$  is available then
9:    $o \leftarrow \mathcal{I}$  /* Receive output from IUT */
10:  if  $o \in \Theta_b(h)$  then
11:    /* If  $o$  is foreseen by oracle continue with next step */
12:    return  $DT_b(h \cdot o)$ 
13:  else
14:    return fail /* Otherwise, report test failure */
15:  end if
16: end if

```

DTG_b chooses the case to apply and the input to provide next nondeterministically. DT_b is parameterized by Ω_{case} and Ω_{in} which can be instantiated by either nondeterminism or some optimized test-case selection.

With Algorithm DT_b , we finish a journey of transformations. The bounded-depth algorithm effectively circumvents the fact that, except for Std and Std_δ , all other objects we need to deal with are countably or uncountably infinite and that the property we need to check is a hyperproperty. By furthermore relegating the construction of the test LTS and its parallel composition (with the implementation under test) into an on-the-fly-algorithm, akin to [43], a practically usable and elegant algorithm for real-world doping tests results.

5.4 HyperSTL

With the model-based testing theory, we have a rigorous foundation to do doping tests in practice. The theory provides an algorithm that, given some input, specifies how the output from the system under test must be interpreted. However, the theory leaves out *how* test inputs can be obtained. In the following we will use probabilistic falsification methods to strategically obtain test inputs. We first introduce HyperSTL, an extension of STL to express hyperproperties over

real-valued or mixed-IO systems. Then, based on the HyperLTL characterisations in Section 4.2, we develop HyperSTL characterisations for robust cleanness and func-cleanness and explain how these HyperSTL formulas can be used with a probabilistic falsification method to obtain test inputs.

Before we introduce HyperSTL, we remark that there exists previous work by Nguyen et al. [96] that discusses an extension of STL to HyperSTL though using a non-standard semantic underpinning. In this context, they present a falsification approach restricted to the fragment “t-HyperSTL” where, according to the authors, “a nesting structure of temporal logic formulas involving different traces is not allowed”. Therefore, none of our cleanness definitions belongs to this fragment. Finally, Nguyen et al. assume that the system under falsification can be simulated. This is a reasonable assumption, because probabilistic falsification is traditionally used for Simulink models. Nevertheless, STL-based falsification techniques are applicable also to black-box systems, as long as the outputs of the system can be passed into the falsification engine.

For HyperSTL this is different; the problem is 1) the potentially continuous time domain, 2) that the evaluation of the HyperSTL formula is based on *sampling*, and 3) that two or more traces must be compared. What Nguyen et al. propose is an algorithm that constructs a new (Simulink) model by cloning the system under falsification and letting all clones run in parallel. This self-composition technique is common to realise evaluation of hyperproperties [12, 34]; we explicitly use self-composition in Section 4.1. In Nguyen’s parallel composition approach, snapshots of the composed system are effectively snapshots of the individual copies of the model at exactly the same instant of time – despite the continuous time domain. This approach is not available when interacting with (black-box) real-world cyber-physical systems (CPS). Here, two samples of different executions are taken at different times with probability 1. Consequently, traces are sampled with different timing functions. To overcome this problem, there are several options. One option is offered by the logic HyperSTL* [20] (an extension of STL* [27]), which enables the comparison of values at different times in different traces. HyperSTL* characterisations have been proposed for various cleanness notions and is beyond the scope of this thesis [20]. The second option is to ensure by means of preprocessing the input and output traces to and from the system that all traces use the same timing function (as we implicitly did for the model-based testing theory and for the HyperLTL characterisations). In the following, we will investigate the second option.

To this end, we propose a HyperSTL syntax similar to that of Nguyen et al. [96]:

$$\begin{aligned} \psi &::= \exists\pi. \psi \mid \forall\pi. \psi \mid \phi \\ \phi &::= \top \mid f > 0 \mid \neg\phi \mid \phi \wedge \phi \mid \phi \mathcal{U} \phi . \end{aligned}$$

The meaning of the universal and existential quantifier is as for HyperLTL. A crucial difference to the other logics presented above is the proposition $f > 0$. In contrast to HyperLTL and to the existing definition of HyperSTL, we consider it insufficient to allow propositions to refer to only a single trace. In HyperLTL that does not cause harm, because atomic propositions of individual traces can be compared by means of the Boolean connectives. To formulate thresholds for real values, however, we feel the need to allow real values from multiple traces to be combined in the function f , and thus to appear as arguments of f . Hence, in our semantics of HyperSTL, $f > 0$ holds if and only if the result of f , applied to all traces quantified over, is greater than 0. For this to work formally, the arity of function f is the product of the trace width n and the number m of traces quantified over at the occurrence of $f > 0$ in the formula, so $f : (\mathbb{R}^n)^m \rightarrow \mathbb{R}$.

A trace assignment [34] $\Pi : \mathcal{V} \rightarrow \mathbb{M}$ is a partial function assigning traces of \mathbb{M} to variables. Let $\Pi[\pi := w]$ denote the same function as Π , except that π is mapped to trace w . The Boolean semantics of HyperSTL is defined below.

Definition 5.37. Let ψ be a HyperSTL formula, $t \in \mathcal{T}$ a time point, $\mathbb{M} \subseteq (\mathcal{T} \rightarrow \mathbb{R}^n)$ a real-valued system, and Π a trace assignment. Then, the Boolean semantics for $\mathbb{M}, \Pi, t \models \psi$ is defined inductively:

$$\begin{aligned}
\mathbb{M}, \Pi, t \models \exists \pi. \psi &\Leftrightarrow \exists w \in \mathbb{M}. \mathbb{M}, \Pi[\pi := w], t \models \psi \\
\mathbb{M}, \Pi, t \models \forall \pi. \psi &\Leftrightarrow \forall w \in \mathbb{M}. \mathbb{M}, \Pi[\pi := w], t \models \psi \\
\mathbb{M}, \Pi, t \models \top & \\
\mathbb{M}, \Pi, t \models f > 0 &\Leftrightarrow f(\Pi(\pi_1)(t), \dots, \Pi(\pi_m)(t)) > 0 \text{ for } \text{dom}(\Pi) = \{\pi_1, \dots, \pi_m\}^1 \\
\mathbb{M}, \Pi, t \models \neg \phi &\Leftrightarrow \mathbb{M}, \Pi, t \not\models \phi \\
\mathbb{M}, \Pi, t \models \phi_1 \wedge \phi_2 &\Leftrightarrow \mathbb{M}, \Pi, t \models \phi_1 \text{ and } \mathbb{M}, \Pi, t \models \phi_2 \\
\mathbb{M}, \Pi, t \models \phi_1 \mathcal{U} \phi_2 &\Leftrightarrow \exists t' \geq t. \mathbb{M}, \Pi, t' \models \phi_2 \text{ and } \forall t'' \in [t, t'). \mathbb{M}, \Pi, t'' \models \phi_1
\end{aligned}$$

A system \mathbb{M} satisfies a formula ψ if and only if $\mathbb{M}, \emptyset, 0 \models \psi$. Analogue to the quantitative semantics of STL, the quantitative semantics for HyperSTL is defined below:

Definition 5.38. Let ψ be a HyperSTL formula, $t \in \mathcal{T}$ a time point, $\mathbb{M} \subseteq (\mathcal{T} \rightarrow \mathbb{R}^n)$ a real-valued system, and Π a trace assignment. Then, the quantitative semantics for $\rho(\psi, \mathbb{M}, \Pi, t)$ is defined inductively:

$$\begin{aligned}
\rho(\exists \pi. \psi, \mathbb{M}, \Pi, t) &= \sup_{w \in \mathbb{M}} \rho(\psi, \mathbb{M}, \Pi[\pi := w], t) \\
\rho(\forall \pi. \psi, \mathbb{M}, \Pi, t) &= \inf_{w \in \mathbb{M}} \rho(\psi, \mathbb{M}, \Pi[\pi := w], t) \\
\rho(\top, \mathbb{M}, \Pi, t) &= \infty
\end{aligned}$$

$$\begin{aligned}
\rho(f > 0, \mathbf{M}, \Pi, t) &= f(\Pi(\pi_1)(t), \dots, \Pi(\pi_m)(t)) \text{ for } \text{dom}(\Pi) = \{\pi_1, \dots, \pi_m\}^1 \\
\rho(\neg\phi, \mathbf{M}, \Pi, t) &= -\rho(\phi, \mathbf{M}, \Pi, t) \\
\rho(\phi_1 \wedge \phi_2, \mathbf{M}, \Pi, t) &= \min(\rho(\phi_1, \mathbf{M}, \Pi, t), \rho(\phi_2, \mathbf{M}, \Pi, t)) \\
\rho(\phi_1 \mathcal{U} \phi_2, \mathbf{M}, \Pi, t) &= \sup_{t' \geq t} \min\{\rho(\phi_2, \mathbf{M}, \Pi, t'), \inf_{t'' \in [t, t']} \rho(\phi_1, \mathbf{M}, \Pi, t'')\}
\end{aligned}$$

It is an easy exercise to show that for continuous-time signals this quantitative semantics of HyperSTL is a conservative extension of the quantitative semantics of STL discussed above. For discrete-time signals it is important to understand that discrete time points often represent points in continuous time. It is widely accepted, that this can be cast into a (strictly monotonic) timing function $\tau : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ [7, 54]. The HyperSTL semantics given above is meaningful in a discrete-time setting if all traces share the same timing function.

HyperSTL characterisation The HyperLTL characterisations in Section 4.2 assume the system to be a subset of $(2^{\text{AP}})^\omega$ and works with distances between traces by means of a Boolean encoding into atomic propositions. By using HyperSTL, we can characterise cleanness for systems that are representable as subsets of $(\mathcal{T} \rightarrow \mathbb{R}^n)$ for some width $n \in \mathbb{N}$. That is, input sets In and output sets Out represent real-valued signals of width m , respectively width l , and a system is encoded by a set $\mathbf{M} \subseteq (\mathcal{T} \rightarrow \mathbb{R}^{m+l})$ that captures the full system behaviour.

We can take the HyperLTL formulas from Propositions 4.12 to 4.15 and transform them into HyperSTL formulas by applying simple syntactic changes. We consider here cleanness for non-parametrised systems. Therefore, those parts of the HyperLTL formulas reasoning about parameters do not appear in the following HyperSTL formulas. We get for l-robust cleanness the formula

$$\begin{aligned}
\psi_{\text{l-rob}} &:= \forall \pi_1. \forall \pi_2. \exists \pi'_2. \text{Std}_{\pi_1} > 0 \\
&\rightarrow \left(\square(\text{eq}(\pi_2 \downarrow_i, \pi'_2 \downarrow_i) \leq 0) \wedge \right. \\
&\quad \left. ((d_{\text{Out}}(\pi_1 \downarrow_o, \pi'_2 \downarrow_o) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(\pi_1 \downarrow_i, \pi'_2 \downarrow_i) - \kappa_i > 0)) \right),
\end{aligned} \tag{5.4}$$

u-robust cleanness is characterised by

$$\begin{aligned}
\psi_{\text{u-rob}} &:= \forall \pi_1. \forall \pi_2. \exists \pi'_1. \text{Std}_{\pi_1} > 0 \\
&\rightarrow \left(\text{Std}_{\pi'_1} > 0 \wedge \square(\text{eq}(\pi_1 \downarrow_i, \pi'_1 \downarrow_i) \leq 0) \wedge \right. \\
&\quad \left. ((d_{\text{Out}}(\pi'_1 \downarrow_o, \pi_2 \downarrow_o) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(\pi'_1 \downarrow_i, \pi_2 \downarrow_i) - \kappa_i > 0)) \right),
\end{aligned} \tag{5.5}$$

¹We admit some sloppiness; the set $\text{dom}(\Pi)$ should have a fixed order.

for l-func-cleanness we get the formula

$$\begin{aligned} \psi_{\text{l-fun}} &:= \forall \pi_1. \forall \pi_2. \exists \pi'_2. \text{Std}_{\pi_1} > 0 \\ &\rightarrow \left(\square(\text{eq}(\pi_2 \downarrow_i, \pi'_2 \downarrow_i) \leq 0) \wedge \left(\square(d_{\text{Out}}(\pi_1 \downarrow_o, \pi'_2 \downarrow_o) - f(d_{\text{In}}(\pi_1 \downarrow_i, \pi'_2 \downarrow_i)) \leq 0) \right) \right), \end{aligned} \quad (5.6)$$

and, finally, u-func-cleanness is encoded by

$$\begin{aligned} \psi_{\text{u-fun}} &:= \forall \pi_1. \forall \pi_2. \exists \pi'_1. \text{Std}_{\pi_1} > 0 \\ &\rightarrow \left(\text{Std}_{\pi'_1} > 0 \wedge \square(\text{eq}(\pi_1 \downarrow_i, \pi'_1 \downarrow_i) \leq 0) \wedge \right. \\ &\quad \left. \left(\square(d_{\text{Out}}(\pi'_1 \downarrow_o, \pi_2 \downarrow_o) - f(d_{\text{In}}(\pi'_1 \downarrow_i, \pi_2 \downarrow_i)) \leq 0) \right) \right). \end{aligned} \quad (5.7)$$

The quantifiers remain unchanged relative to the formulas in Propositions 4.12 to 4.15. The formulas use generic projection functions \downarrow_i and \downarrow_o to extract the input values, respectively output values from a trace. To apply the formulas, these functions must be instantiated with functions for the concrete value domain of the traces to be analysed. For example, for $\text{In} = \mathbb{R}^m$, $\text{Out} = \mathbb{R}^l$ and $\text{M} \subseteq (\mathcal{T} \rightarrow \mathbb{R}^{m+l})$, the projections could be defined for every $w = (s_1, \dots, s_m, s_{m+1}, \dots, s_{m+l})$ as $w \downarrow_i = (s_1, \dots, s_m)$ and $w \downarrow_o = (s_{m+1}, \dots, s_{m+l})$. The input equality requirement for two traces π and π' is ensured by globally enforcing $\text{eq}(\pi \downarrow_i, \pi' \downarrow_i) \leq 0$. eq is a generic function that must return zero if and only if its arguments are identical and a positive value otherwise. It has to be instantiated for concrete value domains. For example, for $\text{In} = \mathbb{R}^m$, $\text{eq}((s_1, \dots, s_m), (s'_1, \dots, s'_m))$ could be defined as the sum of the component-wise distances $\sum_{1 \leq i \leq m} |s_i - s'_i|$. Finally, in the above formulas we replace the AP-encoded versions of d_{In} and d_{Out} by the original distance functions d_{In} and d_{Out} and perform simple arithmetic operations to match the syntactic requirements of HyperSTL.

Formulas (5.5) and (5.7) are prepared to express u-robust cleanness, respectively u-func-cleanness w.r.t. both cleanness *contracts* or cleanness *contexts*. That is, we assume the existence of a function Std_π that returns a positive value if and only if the trace assigned to π encodes a standard input (when considering cleanness *contracts*) or encodes an input and output that constitute a standard behaviour (when considering cleanness *contexts*). In the latter case, we must additionally check that the existentially quantified trace π'_1 represents standard behaviour. For cleanness *contracts* this additional check is not necessary (but also not harmful), because the formula enforces that if π_1 represents a standard input, then π'_1 represents the same standard input, too. Explicitly requiring that π'_1 represents a standard behaviour echoes the setup in Definitions 5.8 and 5.11, which define trace integral u-robust cleanness and u-func-cleanness w.r.t. cleanness *contexts*.

We remark that for encoding Std_π , due to the absence of the Next-operator in HyperSTL, it might be necessary to add a clock signal $s(t) = t$ to traces in a preprocessing step. This is not considered here for the sake of avoiding cluttered notation.

Correctness under Mixed-IO Interpretation For a model M , the structure of the real-valued traces requires inputs and outputs to form pairs synchronised in time. A more realistic scenario is that of inputs and outputs occurring independently of each other. In particular, when testing a real-world CPS, the testing interface can either pass an input to the system under test or receive an output, but not both at the same time. Furthermore, certain tests require to pass a series of inputs before receiving an output at all (examples are provided in Chapter 6). The mixed-IO model supports such real-world testing scenarios.

Mixed-IO signals are defined in the discrete time domain \mathbb{N} . A mixed-IO signal $s \in (\text{In} \cup \text{Out})^\omega$ (or, equivalently, $s : \mathbb{N} \rightarrow \text{In} \cup \text{Out}$) is similar to a real-valued discrete-time signal, but the value domain \mathbb{R} is replaced by the domain $\text{In} \cup \text{Out}$. A mixed-IO trace always consists of a single mixed-IO signal, e.g., $w = (s)$. Accordingly, predicates of the form $f > 0$ must use functions f that produce real values for mixed-IO signals.

The formulas above are applicable to mixed-IO system models (and, hence, to LTS-defined systems), too. The abstract functions \downarrow_i , \downarrow_o , eq and Std can be instantiated for mixed-IO models. \downarrow_i and \downarrow_o can be defined equally to the syntactically identical projection functions for mixed-IO models defined in Section 3.3. The function $\text{eq}(i_1, i_2) \leq 0$ can be defined using the distance function d_{In} and some arbitrary small $\varepsilon > 0$:

$$\text{eq}(i_1, i_2) := \begin{cases} 0, & \text{if } i_1 = i_2 \\ d_{\text{In}}(i_1, i_2) + \varepsilon, & \text{if } i_1 \neq i_2 \wedge i_1, i_2 \in \text{In} \\ \infty, & \text{otherwise.} \end{cases} \quad (5.8)$$

In the second clause of the above definition we add some positive value ε to the result of d_{In} , because $d_{\text{In}}(i_1, i_2)$ could be 0 even if $i_1 \neq i_2$. For the correctness of the above HyperSTL formulas, however, it is crucial that $\text{eq}(i_1, i_2) = 0$ if and only if $i_1 = i_2$. For a good performance of the falsification algorithm, we will nevertheless want to make use of d_{In} if $i_1 \neq i_2$.

Propositions 5.39 and 5.40 show that HyperSTL formulas (5.4) and (5.5) under the mixed-IO interpretation outlined above indeed characterise trace integral l-robust cleanness and u-robust cleanness.

Proposition 5.39. Let $L \subseteq \mathbb{N} \rightarrow (\text{In} \cup \text{Out})$ be a mixed-IO system and $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract or context for robust cleanness with $\text{Std} \subseteq L$. Further,

let Std_π be a quantifier-free HyperSTL subformula, such that $\mathbb{L}, \{\pi := w\}, 0 \models \text{Std}_\pi$ if and only if $w \in \text{Std}$. Then, \mathbb{L} is trace integral l-robustly clean w.r.t. \mathcal{C} if and only if $\mathbb{L}, \emptyset, 0 \models \psi_{\text{l-rob}}$.

Proposition 5.40. Let $\mathbb{L} \subseteq \mathbb{N} \rightarrow (\text{In} \cup \text{Out})$ be a mixed-IO system and $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a contract or context for robust cleanliness with $\text{Std} \subseteq \mathbb{L}$. Further, let Std_π be a quantifier-free HyperSTL subformula, such that $\mathbb{L}, \{\pi := w\}, 0 \models \text{Std}_\pi$ if and only if $w \in \text{Std}$. Then, \mathbb{L} is trace integral u-robustly clean w.r.t. \mathcal{C} if and only if $\mathbb{L}, \emptyset, 0 \models \psi_{\text{u-rob}}$.

The proof for these propositions are almost identical. For u-robust cleanliness it is slightly more difficult, because of the additional requirement that π'_1 in formula (5.5), respectively σ'' in Definition 5.8, represent standard behaviour from Std . Thus, we only present the proof for Proposition 5.40. Before the actual proof, we first establish two lemmas. Lemma 5.41 destructs the globally (\square) and weak until (\mathcal{W}) operators such that the timing constraints encoded by these operators become explicit.

Lemma 5.41. Let $\sigma : \mathcal{T} \rightarrow X$ be a trace with $\mathcal{T} = \mathbb{N}$ or $\mathcal{T} = \mathbb{R}_{\geq 0}$ and let ϕ and ψ be STL formulas. Then the following equivalences hold.

1. $\sigma, 0 \models \square\phi$ if and only if $\forall t \geq 0. \sigma, t \models \phi$,
2. if $\mathcal{T} = \mathbb{N}$, then $\sigma, 0 \models \phi \mathcal{W} \psi$ if and only if $\forall t \geq 0. (\forall t' \leq t. \sigma, t' \models \neg\psi) \Rightarrow \sigma, t \models \phi$.

Proof. We prove the two statements separately.

1. Using the definition of the derived operators \square and \diamond , we get that $\sigma, 0 \models \square\phi$ holds if and only if $\sigma, 0 \models \neg(\top \mathcal{U} \neg\phi)$ holds. Using the (Boolean) semantics of STL (from Section 2.6.2), we get that this is equivalent to $\neg(\exists t \geq 0. \sigma, t \models \neg\phi \wedge \forall t' < t. \sigma, t' \models \top)$. After simple logical operations, we get that this is equivalent to $\forall t \geq 0. \sigma, t \models \phi$ as required.
2. Using 1, the definition of \mathcal{W} , the (Boolean) semantics of STL, and considering that $\mathcal{T} = \mathbb{N}$, we get that $\sigma, 0 \models \phi \mathcal{W} \psi$ if and only if $\exists t \in \mathbb{N}. \sigma, t \models \psi \wedge \forall t' < t. \sigma, t' \models \phi$ or $\forall t \in \mathbb{N}. \sigma, t \models \phi$. We denote this proposition as V . It is easy to see that the right operand of the equivalence to prove can be rewritten to $\forall t \in \mathbb{N}. (\exists t' \leq t. \sigma, t' \models \psi) \vee \sigma, t \models \phi$. We denote this proposition as W and must show that $V \Rightarrow W$ and $W \Rightarrow V$. To prove that V implies W , we distinguish two cases.
 - For the first case, assume that the left operand of the disjunction in V holds, i.e., there is some $t \in \mathbb{N}$, such that $\sigma, t \models \psi \wedge \forall t' < t. \sigma, t' \models \phi$. To show W , let $t_0 \in \mathbb{N}$ be arbitrary. If $t \leq t_0$, then there exists $t' \leq t_0$

(namely $t' = t$) such that $\sigma, t' \models \psi$; hence W holds. If $t > t_0$, then we know from $\forall t' < t. \sigma, t' \models \phi$ that $\sigma, t_0 \models \phi$ is true; hence, W holds.

- For the second case, assume that the right operand of the disjunction in V holds, i.e., $\forall t \in \mathbb{N}. \sigma, t \models \phi$. Then, obviously W holds.

To prove that W implies V , let $PV = \{t \in \mathbb{N} \mid \sigma, t \models \psi\}$ be the set of all time points at which ψ holds. If PV is the empty set, it follows immediately from W that $\forall t \in \mathbb{N}. \sigma, t \models \phi$ and that, hence, V holds. If PV is not empty, let $t = \min PV$ be the smallest time in PV (the minimum always exists, because $\mathcal{T} = \mathbb{N}$). Then, obviously, $\exists t \in \mathbb{N}. \sigma, t \models \psi$. To show that V holds, it suffices to show that $\forall t' < t. \sigma, t' \models \phi$. This follows from W , because t is the smallest time at which $\sigma, t \models \psi$ holds and, therefore, for every $t' < t$ it does not hold that $\sigma, t' \models \psi$.

□

Lemma 5.42 converts HyperSTL formula (5.5) into a first-order logic formula. Notably, it does so only for traces with an underlying discrete time domain. If the time domain is continuous, it is, for example, possible to have a trace such that, for some $t > 0$, in the time interval $[0, t]$ the input distances and the output distances are within the κ_i and κ_o thresholds, and in the open time interval (t, ∞) both input and output distances are beyond their thresholds. Due to the open interval, we are unable to determine a time t' at which the input distance grows beyond the κ_i threshold; consequently, we can also not determine the time at which the obligations for the output distance ends. The robust cleanness definition can handle this instance and would label the above behaviour as clean. The characterisation in formula (5.5), however, would be violated. The semantics of the weak until operator is unable to capture a behaviour like the one above, because in this example it needs a concrete time t' at which the input distance grows beyond the κ_i threshold. This is also the reason, why Lemma 5.41.2 is restricted to $\mathcal{T} = \mathbb{N}$.

To get a HyperSTL characterisation for robust cleanness and a continuous time domain, we believe that it is possible to add past-time operators to HyperSTL. One of these operators is the *historically* operator [13]. $\Box\phi$, which, similar to $\square\phi$ for the future, requires that ϕ must always have been true in the past up until now. With this operator, robust cleanness could be expressed by replacing in formula (5.5) the weak-until construction $(d_{\text{Out}}(\pi'_1 \downarrow_o, \pi_2 \downarrow_o) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(\pi'_1 \downarrow_i, \pi_2 \downarrow_i) - \kappa_i > 0)$ with the historic construction $\Box((\Box d_{\text{In}}(\pi'_1 \downarrow_i, \pi_2 \downarrow_i) - \kappa_i \leq 0) \rightarrow d_{\text{Out}}(\pi'_1 \downarrow_o, \pi_2 \downarrow_o) - \kappa_o \leq 0)$. A HyperSTL* characterisation of robust cleanness is defined in this manner; we refer to [20, Proposition 6.9] for the details.

Lemma 5.42. Let M be a discrete-time real-valued or mixed-IO system, i.e., $M \subseteq (\mathbb{N} \rightarrow \mathbb{R}^n)$ or $M \subseteq (\mathbb{N} \rightarrow (\text{In} \cup \text{Out}))$, and let $\text{Std} \subseteq M$ be a set of standard traces. Also, let Std_π be a quantifier-free HyperSTL subformula, such that $M, \{\pi := w\}, 0 \models \text{Std}_\pi$ if and only if $w \in \text{Std}$. Then, $M, \emptyset, 0 \models \psi_{\text{u-rob}}$ if and only if

$$\begin{aligned} \forall w \in \text{Std}. \forall w' \in M. \exists w'' \in \text{Std}. (\forall t \geq 0. \text{eq}(w \downarrow_i[t], w'' \downarrow_i[t]) \leq 0) \wedge \\ \forall t \geq 0. (\forall t' \leq t. d_{\text{In}}(w'' \downarrow_i[t'], w' \downarrow_i[t']) - \kappa_i \leq 0) \Rightarrow \\ d_{\text{Out}}(w'' \downarrow_o[t], w' \downarrow_o[t]) - \kappa_o \leq 0. \end{aligned}$$

Proof. Using Lemma 5.41.1, Lemma 5.41.2 and Definition 5.37, we get that

$$\begin{aligned} M, \emptyset, 0 \models \forall \pi. \forall \pi'. \exists \pi''. \text{Std}_\pi \\ \rightarrow \left(\text{Std}_{\pi''} \wedge \Box(\text{eq}(\pi \downarrow_i, \pi'' \downarrow_i) \leq 0) \wedge \right. \\ \left. ((d_{\text{Out}}(\pi'' \downarrow_o, \pi' \downarrow_o) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(\pi'' \downarrow_i, \pi' \downarrow_i) - \kappa_i > 0)) \right) \end{aligned}$$

holds if and only if

$$\begin{aligned} \forall w \in M. \forall w' \in M. \exists w'' \in M. (M, \Pi, 0 \models \text{Std}_\pi) \\ \rightarrow \left((M, \Pi, 0 \models \text{Std}_{\pi''}) \wedge (\forall t \geq 0. (M, \Pi, t \models \text{eq}(\pi \downarrow_i, \pi'' \downarrow_i) \leq 0)) \wedge \right. \\ \left. (\forall t \geq 0. (\forall t' \leq t. (M, \Pi, t' \models \neg d_{\text{In}}(\pi'' \downarrow_i, \pi' \downarrow_i) - \kappa_i > 0)) \Rightarrow \right. \\ \left. (M, \Pi, t \models d_{\text{Out}}(\pi'' \downarrow_o, \pi' \downarrow_o) - \kappa_o \leq 0)) \right) \end{aligned}$$

holds for $\Pi = \{\pi := w, \pi' := w', \pi'' := w''\}$. Using that $M, \{\pi := w\}, 0 \models \text{Std}_\pi$ if and only if $w \in \text{Std}$, and by further applying Definition 5.37 and basic logical operations, we get that the above proposition is equivalent to

$$\begin{aligned} \forall w \in M. \forall w' \in M. \exists w'' \in M. w \in \text{Std} \\ \rightarrow \left(w'' \in \text{Std} \wedge (\forall t \geq 0. \text{eq}(w \downarrow_i[t], w'' \downarrow_i[t]) \leq 0) \wedge \right. \\ \left. (\forall t \geq 0. (\forall t' \leq t. d_{\text{In}}(w'' \downarrow_i[t'], w' \downarrow_i[t']) - \kappa_i \leq 0) \Rightarrow d_{\text{Out}}(w'' \downarrow_o[t], w' \downarrow_o[t]) - \kappa_o \leq 0) \right). \end{aligned}$$

Finally, after carefully reordering premises, we get that the above holds if and only if

$$\begin{aligned} \forall w \in \text{Std}. \forall w' \in M. \exists w'' \in \text{Std}. (\forall t \geq 0. \text{eq}(w \downarrow_i[t], w'' \downarrow_i[t]) \leq 0) \wedge \\ \forall t \geq 0. (\forall t' \leq t. d_{\text{In}}(w'' \downarrow_i[t'], w' \downarrow_i[t']) - \kappa_i \leq 0) \Rightarrow d_{\text{Out}}(w'' \downarrow_o[t], w' \downarrow_o[t]) - \kappa_o \leq 0. \end{aligned}$$

□

To prove Proposition 5.40, we must further transform the first-order characterisation obtained from Lemma 5.42 to see that it indeed matches the definition of trace integral u-robust cleanliness.

Proof of Proposition 5.40. Using Lemma 5.42 we get that

$$\begin{aligned} \mathbf{L}, \emptyset, 0 \models & \forall \pi_1. \forall \pi_2. \exists \pi'_1. \mathbf{Std}_{\pi_1} \\ & \rightarrow \left(\mathbf{Std}_{\pi'_1} \wedge \square(\text{eq}(\pi_1 \downarrow_i, \pi'_1 \downarrow_i) \leq 0) \wedge \right. \\ & \quad \left. ((d_{\text{Out}}(\pi'_1 \downarrow_o, \pi_2 \downarrow_o) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(\pi'_1 \downarrow_i, \pi_2 \downarrow_i) - \kappa_i > 0)) \right) \end{aligned}$$

holds if and only if

$$\begin{aligned} \forall w_1 \in \mathbf{Std}. \forall w_2 \in \mathbf{L}. \exists w'_1 \in \mathbf{Std}. (\forall t \geq 0. \text{eq}(w_1 \downarrow_i[t], w'_1 \downarrow_i[t]) \leq 0) \wedge \\ \forall t \geq 0. (\forall t' \leq t. d_{\text{In}}(w'_1 \downarrow_i[t'], w_2 \downarrow_i[t']) - \kappa_i \leq 0) \Rightarrow d_{\text{Out}}(w'_1 \downarrow_o[t], w_2 \downarrow_o[t]) - \kappa_o \leq 0. \end{aligned}$$

After applying simple logical operations and using that $\text{eq}(i_1, i_2) = 0$ if and only if $i_1 = i_2$, we get that this is equivalent to

$$\begin{aligned} \forall w_1 \in \mathbf{Std}. \forall w_2 \in \mathbf{L}. \exists w'_1 \in \mathbf{Std} \text{ with } w_1 \downarrow_i = w'_1 \downarrow_i. \\ \forall t \geq 0. (\forall t' \leq t. d_{\text{In}}(w'_1 \downarrow_i[t'], w_2 \downarrow_i[t']) \leq \kappa_i) \Rightarrow d_{\text{Out}}(w'_1 \downarrow_o[t], w_2 \downarrow_o[t]) \leq \kappa_o, \end{aligned}$$

which, since we assumed $\mathbf{Std} \subseteq \mathbf{L}$, is equivalent to the definition of trace integral u-robust cleanness for mixed-IO systems. \square

We can state propositions similar to Propositions 5.39 and 5.40 for l-func-cleanness and u-func-cleanness.

Proposition 5.43. Let $\mathbf{L} \subseteq \mathbb{N} \rightarrow (\text{In} \cup \text{Out})$ be a mixed-IO system and $\mathcal{C} = \langle \mathbf{Std}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract or context for func-cleanness with $\mathbf{Std} \subseteq \mathbf{L}$. Further, let \mathbf{Std}_{π} be a quantifier-free HyperSTL subformula, such that $\mathbf{L}, \{\pi := w\}, 0 \models \mathbf{Std}_{\pi}$ if and only if $w \in \mathbf{Std}$. Then, \mathbf{L} is trace integral l-func-clean w.r.t. \mathcal{C} if and only if $\mathbf{L}, \emptyset, 0 \models \psi_{\text{l-fun}}$.

Proposition 5.44. Let $\mathbf{L} \subseteq \mathbb{N} \rightarrow (\text{In} \cup \text{Out})$ be a mixed-IO system and $\mathcal{C} = \langle \mathbf{Std}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a contract or context for func-cleanness with $\mathbf{Std} \subseteq \mathbf{L}$. Further, let \mathbf{Std}_{π} be a quantifier-free HyperSTL subformula, such that $\mathbf{L}, \{\pi := w\}, 0 \models \mathbf{Std}_{\pi}$ if and only if $w \in \mathbf{Std}$. Then, \mathbf{L} is trace integral u-func-clean w.r.t. \mathcal{C} if and only if $\mathbf{L}, \emptyset, 0 \models \psi_{\text{u-fun}}$.

The proofs for Propositions 5.43 and 5.44 are conceptually similar to the one for Proposition 5.40. The only difference is that instead of the reasoning about the \mathcal{W} construct the globally enforced relation between output distances and the result of f must be proven equivalent in the HyperSTL formulas and the l- and u-func-cleanness. We omit the proofs here.

STL Characterisation for Finite Standard Behaviour In many practical settings – when the different standard behaviours are spelled out upfront explicitly, as in NEDC and WLTC – it can be assumed that the number of distinct standard behaviours Std is finite (while there are infinitely many possible behaviours in \mathbf{M} or \mathbf{L}). Finiteness of Std makes it possible to remove by enumeration the quantifiers from the u-robust cleanness and u-func-cleanness HyperSTL formulas. This opens the way – after proper adjustments – to work with the STL fragment of HyperSTL. In the following, we assume that the set $\text{Std} = \{w_1, \dots, w_c\}$ is an arbitrary standard set with c unique standard traces. We further assume a fixed system from which we get traces of width n . In particular, we assume that $w_1 = (s_{11}, \dots, s_{1n}), \dots$, and $w_c = (s_{c1}, \dots, s_{cn})$ are the individual standard traces in Std . We denote by $w = (s_1, \dots, s_n)$ the trace under analysis.

To encode the HyperSTL formulas (5.5) and (5.7) in STL, we use the concept of *self-composition*, which has proven useful for the analysis of hyperproperties [57, 12]. We concatenate all signals of w and the standard traces w_1 to w_c to the composed trace $w_+ = (s_1, \dots, s_n, s_{11}, \dots, s_{1n}, \dots, s_{c1}, \dots, s_{cn})$ of width $n + nc$. Given a trace width $n \in \mathbb{N}_+$, a system $\mathbf{M} \subseteq (\mathcal{T} \rightarrow X^n)$ and a set $\text{Std} = \{w_1, \dots, w_c\}$ with $w_1 = (s_{11}, \dots, s_{1n}), \dots$, and $w_c = (s_{c1}, \dots, s_{cn})$, we denote by $\mathbf{M} \circ \text{Std} := \{(s_1, \dots, s_n, s_{11}, \dots, s_{1n}, \dots, s_{c1}, \dots, s_{cn}) \mid (s_1, \dots, s_n) \in \mathbf{M}\}$ the system in which every trace in \mathbf{M} is composed with the standard traces in Std .

For every $w_+ \in \mathbf{M} \circ \text{Std}$, we will in the following STL formula write w to mean the projection on w_+ to the signals (s_1, \dots, s_n) , and we write w_k , for $1 \leq k \leq c$, to mean the projection on w_+ to the signals (s_{k1}, \dots, s_{kn}) of the k th standard trace. More formally, w_k in an STL formula is a function such that $w_k(w_+)$ returns the trace $w_k \in \text{Std}$ and w is a function such that $w(w_+)$ returns the trace $w \in \mathbf{M}$ embedded in w_+ .

Proposition 5.45. Let \mathbf{M} be a discrete-time real-valued or mixed-IO system, i.e., $\mathbf{M} \subseteq (\mathbb{N} \rightarrow \mathbb{R}^n)$ or $\mathbf{M} \subseteq (\mathbb{N} \rightarrow (\text{In} \cup \text{Out}))$, and let $\text{Std} = \{w_1, \dots, w_c\} \subseteq \mathbf{M}$ be a finite set of standard traces. Also, let Std_π be a quantifier-free HyperSTL subformula, such that $\mathbf{M}, \{\pi := w\}, 0 \models \text{Std}_\pi$ if and only if $w \in \text{Std}$. Then, $\mathbf{M}, \emptyset, 0 \models \psi_{\text{u-rob}}$ if and only if $(\mathbf{M} \circ \text{Std}) \models \varphi_{\text{u-rob}}$, where

$$\varphi_{\text{u-rob}} := \bigwedge_{1 \leq a \leq c} \bigvee_{1 \leq b \leq c} \left(\square(\text{eq}(w_a \downarrow_i, w_b \downarrow_i) \leq 0) \wedge \right. \\ \left. ((d_{\text{Out}}(w_b \downarrow_o, w \downarrow_o) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(w_b \downarrow_i, w \downarrow_i) - \kappa_i > 0)) \right).$$

Proof. Using Lemma 5.42 we get that

$$\begin{aligned} \mathbf{M}, \emptyset, 0 \models \forall \pi'. \forall \pi''. \exists \pi'''. \mathbf{Std}_{\pi'} \\ \rightarrow \left(\mathbf{Std}_{\pi'''} \wedge \Box(\mathbf{eq}(\pi' \downarrow_i, \pi'' \downarrow_i) \leq 0) \wedge \right. \\ \left. ((d_{\text{Out}}(\pi''' \downarrow_o, \pi'' \downarrow_o) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(\pi''' \downarrow_i, \pi'' \downarrow_i) - \kappa_i > 0)) \right) \end{aligned}$$

holds if and only if

$$\begin{aligned} \forall w' \in \mathbf{Std}. \forall w'' \in \mathbf{M}. \exists w''' \in \mathbf{Std}. (\forall t \geq 0. \mathbf{eq}(w' \downarrow_i[t], w''' \downarrow_i[t]) \leq 0) \wedge \\ \forall t \geq 0. (\forall t' \leq t. d_{\text{In}}(w''' \downarrow_i[t'], w'' \downarrow_i[t']) - \kappa_i \leq 0) \Rightarrow d_{\text{Out}}(w''' \downarrow_o[t], w'' \downarrow_o[t]) - \kappa_o \leq 0. \end{aligned}$$

Since $\mathbf{Std} = \{w_1, \dots, w_c\}$, we can replace the universal and existential quantifiers over \mathbf{Std} by a conjunction, respectively disjunction, over the standard traces [103]. We get

$$\begin{aligned} \forall w \in \mathbf{M}. \bigwedge_{1 \leq a \leq c} \bigvee_{1 \leq b \leq c} (\forall t \geq 0. \mathbf{eq}(w_a \downarrow_i[t], w_b \downarrow_i[t]) \leq 0) \wedge \\ \forall t \geq 0. (\forall t' \leq t. d_{\text{In}}(w_b \downarrow_i[t'], w \downarrow_i[t']) - \kappa_i \leq 0) \Rightarrow d_{\text{Out}}(w_b \downarrow_o[t], w \downarrow_o[t]) - \kappa_o \leq 0. \end{aligned}$$

For the next step, we replace the universal quantification of w over \mathbf{M} by a universal quantification of the composed traces w_+ over $\mathbf{M} \circ \mathbf{Std}$. Then, from the Boolean semantics of STL and by replacing all traces w , respectively w_k , by the corresponding w_+ -projections, we get the equivalent proposition

$$\begin{aligned} \forall w_+ \in (\mathbf{M} \circ \mathbf{Std}). \bigwedge_{1 \leq a \leq c} \bigvee_{1 \leq b \leq c} (\forall t \geq 0. (w_+, t \models \mathbf{eq}(w_a \downarrow_i, w_b \downarrow_i) \leq 0)) \wedge \\ \forall t \geq 0. (\forall t' \leq t. (w_+, t' \models \neg d_{\text{In}}(w_b \downarrow_i, w \downarrow_i) - \kappa_i > 0)) \Rightarrow \\ (w_+, t \models d_{\text{Out}}(w_b \downarrow_o, w \downarrow_o) - \kappa_o \leq 0). \end{aligned}$$

With the Boolean semantics of STL and Lemmas 5.41.1 and 5.41.2 we get the equivalent statement that

$$\begin{aligned} \forall w_+ \in (\mathbf{M} \circ \mathbf{Std}). w_+, 0 \models \bigwedge_{1 \leq a \leq c} \bigvee_{1 \leq b \leq c} (\Box(\mathbf{eq}(w_a \downarrow_i, w_b \downarrow_i) \leq 0)) \wedge \\ ((d_{\text{Out}}(w_b \downarrow_o, w \downarrow_o) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(w_b \downarrow_i, w \downarrow_i) - \kappa_i > 0)), \end{aligned}$$

which concludes the proof. \square

Notice that by definition, the behaviour of mixed-IO systems is modelled as a set of traces that entail a single signal $\mathbb{N} \rightarrow (\text{In} \cup \text{Out})$. The composed trace w_+ , however, consists of $c+1$ signals, i.e., $w_+ \in \mathbb{N} \rightarrow (\text{In} \cup \text{Out})^{c+1}$. Nevertheless, the usage of \mathbf{eq} , \downarrow_i , \downarrow_o , d_{In} and d_{Out} is not affected by this, because these functions are applied after using the projections w_k , respectively w , that recover the single-signal mixed-IO traces from which w_+ is constructed from.

By combining Propositions 5.40 and 5.45 we get that $\varphi_{\text{u-rob}}$ characterises u-robust cleanness w.r.t. contracts with finite standard behaviour:

Corollary 5.46. Let $L \subseteq \mathbb{N} \rightarrow (\text{In} \cup \text{Out})$ be a mixed-IO system and $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ a context for robust cleanness with finite standard behaviour $\text{Std} = \{w_1, \dots, w_c\} \subseteq L$. Then, L is trace integral u-robustly clean w.r.t. \mathcal{C} if and only if $(L \circ \text{Std}) \models \varphi_{\text{u-rob}}$.

The proposition below for u-func-cleanness is analogue to Proposition 5.45 – except for the absence of the condition that M must be defined over a discrete time domain. Its proof is, up to the different reasoning for $\square(d_{\text{Out}}(w_b \downarrow_o, w \downarrow_o) - f(d_{\text{In}}(w_b \downarrow_i, w \downarrow_i)) \leq 0)$ instead of $(d_{\text{Out}}(w_b \downarrow_o, w \downarrow_o) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(w_b \downarrow_i, w \downarrow_i) - \kappa_i > 0)$, identical to that of Proposition 5.45. Hence, we omit it.

Proposition 5.47. Let M be a real-valued or mixed-IO system, i.e., $M \subseteq (\mathcal{T} \rightarrow \mathbb{R}^n)$ or $M \subseteq (\mathbb{N} \rightarrow (\text{In} \cup \text{Out}))$ for $\mathcal{T} = \mathbb{N}$ or $\mathcal{T} = \mathbb{R}_{\geq 0}$, and let $\text{Std} = \{w_1, \dots, w_c\} \subseteq M$ be a finite set of standard traces. Also, let Std_π be a quantifier-free HyperSTL subformula, such that $M, \{\pi := w\}, 0 \models \text{Std}_\pi$ if and only if $w \in \text{Std}$. Then, $M, \emptyset, 0 \models \psi_{\text{u-fun}}$ if and only if $(M \circ \text{Std}) \models \varphi_{\text{u-fun}}$, where

$$\varphi_{\text{u-fun}} := \bigwedge_{1 \leq a \leq c} \bigvee_{1 \leq b \leq c} \left(\square(\text{eq}(w_a \downarrow_i, w_b \downarrow_i)) \leq 0 \right) \wedge \left(\square(d_{\text{Out}}(w_b \downarrow_o, w \downarrow_o) - f(d_{\text{In}}(w_b \downarrow_i, w \downarrow_i)) \leq 0) \right).$$

Again, by combining Propositions 5.44 and 5.47 we get that $\varphi_{\text{u-fun}}$ characterises u-func-cleanness w.r.t. contracts with finite standard behaviour:

Corollary 5.48. Let $L \subseteq \mathbb{N} \rightarrow (\text{In} \cup \text{Out})$ be a mixed-IO system and $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, f \rangle$ a context for func-cleanness with finite standard behaviour $\text{Std} = \{w_1, \dots, w_c\} \subseteq L$. Then, L is trace integral u-func-clean w.r.t. \mathcal{C} if and only if $(L \circ \text{Std}) \models \varphi_{\text{u-fun}}$.

Example 5.49. We consider the robust cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ where $\text{Std} = \{w_1, w_2\}$ contains the two standard traces $w_1 = 1; 2; 3; 7_o; 0; \delta^\omega$ and $w_2 = 0; 1; 2; 3; 6_o; \delta^\omega$. We here decorate inputs with index i and outputs with index o , i.e., w_1 describes a system receiving the three inputs 1, 2, and 3, then producing the output 7, and finally receiving input 0 before entering quiescence. We take

$$d_{\text{In}}(i_1, i_2) = \begin{cases} |i_1 - i_2|, & \text{if } i_1, i_2 \in \text{In} \\ 0, & \text{if } i_1 = i_2 = _i \\ \infty, & \text{otherwise,} \end{cases}$$

and

$$d_{\text{Out}}(\mathfrak{o}_1, \mathfrak{o}_2) = \begin{cases} |\mathfrak{o}_1 - \mathfrak{o}_2|, & \text{if } \mathfrak{o}_1, \mathfrak{o}_2 \in \text{Out} \setminus \{\delta\} \\ 0, & \text{if } \mathfrak{o}_1 = \mathfrak{o}_2 = -_{\circ} \text{ or } \mathfrak{o}_1 = \mathfrak{o}_2 = \delta \\ \infty, & \text{otherwise.} \end{cases}$$

The contractual value thresholds are assumed to be $\kappa_i = 1$ and $\kappa_o = 6$.

Assume we are observing the trace $w = 0; 1; 2; 6_{\circ} 0; \delta^{\omega}$ to be monitored with STL formula $\varphi_{\text{u-rob}}$ (from Proposition 5.45). First notice, that for combinations of a and b in $\varphi_{\text{u-rob}}$, where $a \neq b$, the subformula $\Box(\text{eq}(w_a \downarrow_i, w_b \downarrow_i) \leq 0)$ is always false, because w_1 and w_2 have different (input) values at time point 0. Hence, it remains to show that

$$\begin{aligned} & (d_{\text{Out}}(w_1 \downarrow_{\circ}, w \downarrow_{\circ}) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(w_1 \downarrow_i, w \downarrow_i) - \kappa_i > 0) \wedge \\ & (d_{\text{Out}}(w_2 \downarrow_{\circ}, w \downarrow_{\circ}) - \kappa_o \leq 0) \mathcal{W}(d_{\text{In}}(w_2 \downarrow_i, w \downarrow_i) - \kappa_i > 0). \end{aligned}$$

For the first part, the input distance between inputs in w and w_1 is always 1 at positions 1 to 3, it is 0 at position 4 (because $-_i$ is compared to $-_i$) and in position 5 and beyond. Thus, $d_{\text{In}}(w_1 \downarrow_i, w \downarrow_i) - \kappa_i$ is always at most 0, and the right hand-side of the \mathcal{W} operator is always false. Consequently, by definition of \mathcal{W} , the left operand of \mathcal{W} must always hold, i.e., $d_{\text{Out}}(w_1 \downarrow_{\circ}, w \downarrow_{\circ})$ must always be less or equal to 6. This is the case for w_1 and w : at all positions except for 4, $-_{\circ}$ is compared to $-_{\circ}$ (or δ to δ), so the difference is 0, and at position 4, the distance of 6 and 7 is 1.

For the second \mathcal{W} -formula, w is compared to w_2 . These two traces are comparable only to a limited extent: the order of input and output is altered at the last two positions of the signals before quiescence. Hence, the right operand of \mathcal{W} is true at position 4, and the formula holds for the remaining trace. For positions 1 to 3, the input distances are 0, because the input values are identical. At these positions, the left operand must hold. The values are input values, so $-_{\circ}$ is compared to $-_{\circ}$ at each position. This distance is defined to be 0, so it holds that $-6 \leq 0$, and the formula is satisfied. Since both formulas hold, the conjunction of both holds, too, and trace w is qualified as trace integral robustly clean. There could however be other system traces not considered in this example, that overall could violate trace integral robust cleanness of the system.

Restriction of input space Robust cleanness puts semantic requirements on fragments of a system's input space, outside of which the system's behaviour remains unspecified. Typically, the fragment of the input space covered is rather small. To falsify the STL formula $\varphi_{\text{u-rob}}$ from Proposition 5.45, the falsifier has two challenging tasks. First, it has to find a way to stay in the relevant input space,

i.e., select inputs with a distance of at most κ_i from the standard behaviour. Only if this is assured it can search for an output large enough to violate the κ_o requirement. In this, a large robustness estimate provided by the quantitative semantics of STL cannot serve as an indicator for deciding whether an input is too far off or whether an output stays too close to the standard behaviour.

The general strength of the falsification technique is its proven ability to discover outputs of a black-box system violating a property. That is why the technique is considered suitable for real-world robust cleanness tests. We can improve its efficiency significantly by narrowing upfront the input space the falsifier uses.

As mentioned in Section 5.3, real-life test execution traces are always finite, i.e., the trace lengths can be bounded by some constant $B \in \mathbb{N}$ and systems can be represented as sets of finite traces $M \subseteq (\text{In} \cup \text{Out})^B$ (which for formality reasons each can be considered suffixed with δ^ω). In this bounded horizon, we can provide a predicate discriminating between relevant and irrelevant input sequences. Formally, the restriction to the relevant input space fragment of a system $M \subseteq (\text{In} \cup \text{Out})^B$ is given by the set $\text{In}_{\text{Std}, \kappa_i} = \{w \in M \mid \exists w' \in \text{Std}. \bigwedge_{k=0}^{B-1} (d_{\text{In}}(w[k] \downarrow_i, w'[k] \downarrow_i) \leq \kappa_i)\}$. Since Std and B are finite, membership is computable.

There are rare cases in which this optimisation may prevent the falsifier from finding a counterexample. This is only the case if there is an input prefix leading to a violation of the formula for which there is no suffix such that the whole trace satisfies the κ_i constraint. Below is a pathological example in which this could make a difference.

Example 5.50. Apart from NO_x emissions, NEDC (and WLTC) tests are used to measure fuel consumption. Consider a cleanness context where inputs represent the speed of a car and where outputs represent the amount of fuel injected into the engine. Assuming a “normal” fuel rate behaviour during the standard test, there might be a test within a reasonable κ_i distance, where the fuel is wasted insanely. Then, the fuel tank might run empty before the intended end of the test, which therefore could not be finished within the κ_i distance, because speed would be constantly 0 at the end. The actually driven test is not in set $\text{In}_{\text{Std}, \kappa_i}$, but there is a prefix within κ_i distance that violates the robust cleanness property.

5.5 An Integrated Testing Approach

From Section 5.3 we get a model-based testing algorithm that provably provides correct verdicts for u-robust cleanness. Section 5.4 develops a quantitative semantics for u-robust cleanness, which can be plugged into existing probabilistic

falsification algorithms, such as the one shown in Algorithm 2.1. The probabilistic falsification approach determines inputs that falsify u-robust cleanness. Under the assumption that inputs can be passed automatically to the system under test, and that this system produces an output in adequate time, the two approaches constitute alternative ways to serve as a black-box verification technique for u-robust cleanness.

For cyber-physical systems these assumptions may be questioned. To conduct a test with a car, for example, the input to the system is a test cycle that is passed to the vehicle by driving it. This imposes several challenges (Chapter 6 discusses this in detail), among which are the violations of the the above assumptions: the test cycle must be driven *manually* and the execution time of the system is the duration of the test cycle and, hence, *inadequately long*. These concerns are shared by many CPS. Notably, we consider here the scenario that the CPS is tested by an entity that is not the manufacturer. While the latter might have tools to overcome these technical challenges, the former typically does not have access to them.

We propose the following solution for effective doping tests of cyber-physical systems. The big picture is provided in Figure 5.2. In a first step, the CPS is used under real-world conditions without enforcing any specific constraints on the inputs to the system. For all executions, the inputs and outputs are recorded. So, essentially, the system can be used as it is needed by the user, but all interactions with it are recorded. From these recordings, a *model* can be learned, which for arbitrary inputs (whether they were covered in the recorded data or not) predicts outputs of the system. For the learned model, the probabilistic falsification algorithm computes a test input that falsifies it – inputs to this model can be passed automatically and an output is produced almost instantly. The result of the falsification procedure is an input sequence that can serve in a second step as an input that the model-based testing algorithm can use for the real-world system. If the prediction was correct, also the real system is falsified. If it was incorrect, the learned model can be refined and the process starts again.

5.6 Related Work & Contributions

This chapter uses an existing model-based testing approach and instantiates it with u-robust cleanness. We take robust cleanness and identify the conditions under which it can be used for model-based testing. We adapt the notion of robust cleanness for mixed-IO systems from Section 3.3 so that it meets these conditions. We continue by constructing a largest LTS that is u-robustly clean and prove essential properties of it. This LTS can serve as a specification for

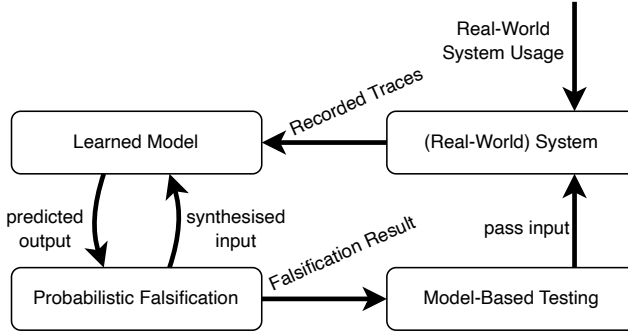


Figure 5.2: Integrated testing approach

model-based testing and we develop a provably correct on-the-fly testing algorithm for u-robust cleanness. Our algorithm is similar to existing test frameworks like TGV [75] or TorX [43], but they cannot test for u-robust cleanness out of the box and, consequently, they cannot provide (proved) soundness guarantees for the detection of software doping.

The work in Section 5.4 is related to an existing, for our cleanness definitions unsuitable, quantitative semantics of HyperSTL [96] and to existing work about probabilistic falsification [48, 2, 54, 8, 13]. Inspired by the HyperLTL characterisations in Section 4.2, we propose HyperSTL characterisation for robust cleanness and func-cleanness. Under the assumptions made in Section 5.3 we transform the HyperSTL characterisation of u-robust cleanness and u-func-cleanness to STL formulas that allow the automatic falsification of u-robust cleanness and, respectively, u-func-cleanness.

Finally, we combine the model-based testing technique and the falsification technique for CPS to propose an integrated testing approach. Using a learned model in addition to the real system has similarities to the idea of digital twins [64, 78].

The contributions made in Sections 5.1 to 5.3 mostly come from [17, 18]. Many of the contributions in Sections 5.4 and 5.5 were first published in [25]. All contributions were primarily developed by myself. New contributions in this thesis are the HyperSTL characterisation for l-robust cleanness and func-cleanness, the STL characterisation of u-func-cleanness under finite standard behaviour, all proofs in Section 5.4, and Section 5.5.

6 Hands-On: Diesel Doping Tests

Using the concepts and techniques defined in the previous chapters, we now show how it can be applied to analyse diesel cars. With the Diesel Emissions Scandal, it became obvious that software doping exists in products that are used by millions of people. Despite that, the cheating software in diesel cars was not discovered immediately. One reason for this is, that a car falls into the class of *cyber-physical systems*, i.e., digital systems that interact with the physical world. In particular, testing cyber-physical systems turns out to be often very difficult when the test inputs must be passed “through the physical world” instead of a purely digital interface. For example, to make a car follow a test cycle, some human must drive the car accordingly.

This chapter identifies several challenges of automotive doping tests – challenges that these tests have in common with tests of many other cyber-physical systems. We develop techniques to overcome these challenges, or at least significantly reduce the effort needed to conduct these tests. Our testing approaches are based on the model-based testing technique developed in Chapter 5. We first give a demonstration of how this technique is able to identify software doping as it was found in a multitude of Volkswagen cars. Then, we take a real car and conduct a variety of tests on a chassis dynamometer, which are all based on cleanness notions that we introduced in Chapter 3. Finally, we present the *Car Data Platform*, which is supposed to be a long-term research project to perform mass-monitoring of cars and to provide a rich toolbox for automotive software doping analysis.

6.1 Model-Based Testing in Practice

Algorithm 5.1 serves as a basis for real-world doping tests. It is the core of a testing framework we have implemented in Python. This implementation along with implementations specific to the use case described here. Further accompanying documentation is archived and publicly available at DOI 10.5281/zenodo.4709389 [19]. The framework defines the minimal requirements for implementations of distance functions, value domains and the communication interface to the implementation under test as abstract classes. We call the instantiation of the pair $(\Omega_{\text{case}}, \Omega_{\text{In}})$ a *test case selection*, which can be implemented as desired,

as long as it complies to the interface defined by the framework. We want to remark that our framework implements Case 2 of DT_b different from what is explained in Section 5.3. In practice, we cannot assume atomicity of one iteration of the test execution. This is a well-known practical impediment of model-based testing [63]. The common approach to circumvent this issue proceeds by delegating the decision of Algorithm 5.1 which case to pick to the driver component [43] (connecting to the IUT), which is configured to be able to look one output (or quiescence) ahead. We have adapted this approach, giving preference to Case 3 if the driver holds some output. Except for the structural constraints explained above, there are no limitations for the specification of concrete contracts or IUTs.

Software doping tests are typically executed physically rather than in simulation. When testing passenger cars, the driver component is a human driver. Having a human in the loop has severe consequences. In many cases, they will fail to make the car under test behave exactly as specified by the designated test inputs. To overcome this problem of human imprecisions, we will use a technique related to testing, which is *monitoring*. A monitor can read the inputs and outputs of a system in order to detect incorrect behaviour of the system. In contrast to testing, the inputs are not provided by the test, but instead the system is monitored during normal operation. Monitors can be either online (evaluation is done while inputs are still received) or offline (observed behaviour is evaluated after the observation). A monitor can easily be extended to a test by controlling the environment providing the inputs to the system. In contrast to classical testing, however, the monitor has the flexibility to handle human imprecisions. We made offline monitoring explicitly part of our testing framework. To this end, we use its flexibility to specify a virtual implementation under test with an associated test case selection that can run a recorded trace with the testing algorithm being in the loop. We present two examples showing two different approaches of how our framework can be used. Both examples consider the Diesel Emissions Scandal.

6.1.1 The Volkswagen Case

Among the first car manufacturers convicted of cheating with emission cleaning systems is Volkswagen. As explained in Section 2.7, their cars used pairs of piecewise linear functions to distinguish between emission tests and normal driving behaviour. Before we report about doping tests with real cars, we use the Volkswagen case to showcase how robust cleanness can identify this instance of doping. The first step is to construct a suitable cleanness context.

Inputs & Outputs The input dimension ln is spanned by (a subset of) the sensors the car model is equipped with (among them e.g. temperature of the exhaust,

outside temperature, vertical and lateral acceleration, throttle position, time after engine start, engine rpm, possibly height above ground level etc.). Most substances leaving the exhaust pipe are gases or small particles that are a result of the chemical reactions in the engine. The processes inside the engine depends to a very large extend on the amount of injected fuel, which is controlled by the position of the throttle. The typical way of defining how the throttle is supposed to be used is by means of a speed trajectory. The vehicle speed is the decisive quantity specified to vary along the test cycle NEDC (cf. Figure 2.2), hence, we take $\text{In} = \mathbb{R}$. Nevertheless, it is possible to add further dimensions of inputs; ambient air, for example, is also part of the reactions in the engine, but has much less influence on the results than the amount of fuel. Gear changes, on the other hand, naturally produce extreme variations on the output. Therefore, following a pattern of gear changes different from what is prescribed by the NEDC should be considered as an extreme variation of the input value (i.e., causing exceedance of κ_i). Thus, in our experiments, we carefully follow the gear change instructions of the NEDC. Gear information is omitted from our input domain. There are similar practical reasons why other physical characteristics are neglected, but which our theory can handle easily. For every input dimension added, there needs to be a technical counterpart that is able to identify the appropriate values and that is synchronised with the speed and emissions sensors. To avoid this technical overhead and for ease of presentation we do not consider additional input dimensions.

The outputs Out depend on the actual objective of the test. Most tests related to the diesel scandal involve the measurement of the amount of NO_x per kilometre emitted since engine start, but it could also be the amount of CO_2 , any other gases, or fuel consumption. Sometimes, the outputs of interest are not accessible directly. For example, when using only the OBD interface of the car (cf. Section 2.7) the values reported by the on-board NO_x sensors are expressed in parts-per-million (ppm). In this case, other sensor values (e.g. mass air flow, fuel rate and others) [82] can be used to compute the amount of NO_x emitted in mg/km. All sensor values necessary for this computation would then be considered being part of Out ; the distance function d_{Out} would have to perform the necessary conversions as part of the distance computation. In the following examples we use an external emissions measurement system, that internally performs the computation of the amount of NO_x in mg/km. Hence, this is the decisive output quantity and thus $\text{Out} = \mathbb{R}$.

Cleanness Context For the experiments we will consider robust cleanness to analyse whether a car exhibits software doping. Thus, before conducting the experiments, we must define a cleanness context.

A standard LTS Std can be constructed from the results of driving the NEDC

cycle several times on a chassis dynamometer, and logging both input and output values. The specific setting we consider is that of a trace σ_S recorded with an emissions measurement system which is attached to the exhaust pipe and reports the accumulated amount of NO_x gases during the entire test procedure upon its termination. Each such experiment constitutes a trace with an infinite suffix of δ s (because the experiment is finite), say $\sigma_S := i_1 \cdots i_{1180} o_S \delta \delta \delta \cdots$. The inputs i_1, \dots, i_{1180} are given by the NEDC over its 20 minutes (1180 seconds) duration, possibly deviating by up to 2 km/h due to human driving imprecision (as per the official NEDC regulations), and are followed by a single output o_S reporting the NO_x amount.

Suitable distance functions are past-forgetful and compute the absolute difference of the speed of the car for d_{In} and the discrepancy of the amount of gases (in mg/km) for d_{Out} . Formally, we define $d_{\text{In}}(a, b) = |a - b|$ if $a, b \in \text{In}$, $d_{\text{In}}(-i, -i) = 0$ and $d_{\text{In}}(a, b) = \infty$ otherwise. Similarly, $d_{\text{Out}}(a, b) = |a - b|$ if $a, b \in \text{Out}$, $d_{\text{Out}}(-o, -o) = d_{\text{Out}}(\delta, \delta) = 0$ and $d_{\text{Out}}(a, b) = \infty$ otherwise.

For the distance thresholds, we pick $\kappa_i = 15$ km/h and $\kappa_o = 180$ mg/km. The input bound allows more variation than foreseen within the NEDC itself (2 km/h). Notably, the output bound is very generous. It is more than the double of the currently allowed legal limit (80 mg/km) of how much NO_x a car is allowed to emit at all. Ultimately, this induces a concrete cleanness context $\mathcal{C} = \langle \text{Std}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o \rangle$ that we are going to use in the sequel. The contract is strictly speaking hypothetical (since no car manufacturer agreed on it), but from a common-sense perspective it appears generous enough to serve as a valid discriminator to accuse any party violating it of software doping.

Test Case Selection We implemented a toy version of an emission cleaning system [19] that encodes the pair of piecewise linear functions found in the early Volkswagen emission cheating cases. As standard behaviour, we encoded the NEDC with 50 mg/km of emitted NO_x . Moreover, we implemented the above mentioned input distance function and output distance function. For the test case selection, we instantiated Ω_{case} and Ω_{In} of Algorithm 5.1 with

$$\Omega_{\text{case}}(h) = \begin{cases} 2 & \text{if } |h| \leq 1180 \\ 3 & \text{if } |h| = 1181 \end{cases} \quad \text{and}$$

$$\Omega_{\text{In}}(h) = \text{rand}_{\text{unif}} [\max(0, \text{last}(h) - \kappa_i), \text{last}(h) + \kappa_i].$$

That is, the test stops after 1181 steps and otherwise meanders randomly through the speed variations possible ($\text{rand}_{\text{unif}}$ implements uniform randomness). Running Algorithm DT_b (with $b = 1182$) with these parameters is extremely likely to lead to a **fail**, i.e., to indicate software doping. During these tests the chance is high to leave the white area in Figure 2.3 defined by the pairs of piecewise

linear functions encoded in VW's control units. In our experiments, we had to take $\kappa_i \leq 4$ in order to see tests passing with some perceivable chance.

6.1.2 The Nissan Case

The Volkswagen example above shows how our testing framework works in theory. In practice, if we test cyber-physical systems like cars, it is usually not possible (or at least very difficult) to effectuate the interface between DT_b and the IUT. Testing a car, for example, requires a human driver who can drive the car as specified by DT_b . However, the driver needs to be made aware of the upcoming input values a few seconds in advance in order to be able to prepare for changes. This is not in the spirit of our algorithm (and neither that of model-based testing), because there is no support for look-ahead. Furthermore, human imprecisions must be taken into account. Even well trained drivers will likely not be able to reach the prescribed speed values accurately at precisely the right time points. Thus, for these kinds of experiments, we propose the following three-step approach.

1. Use the test case selection in order to generate a sequence of inputs that serve as a test case instruction for a human driver. Considering a tolerance of η for human imprecisions, the input sequences should be generated for a contract where the input threshold is $\kappa'_i = \kappa_i - \eta$, i.e., assuming the driver controls the car with an imprecision of at most tolerance η , the actually driven input sequence will still be considered acceptable as per Definition 5.2.
2. Utilise that test case to guide a human driver effectuating the test on the chassis dynamometer, record the entire experiment, and store it as a trace.
3. Use the monitoring capabilities of our framework to simulate the experiment with Algorithm DT_b analysing it. To this end, we provide an implementation to parse traces and to generate a virtual IUT and a test case selection, which, when used with DT_b , simulate the recorded experiment. Algorithm DT_b will return either **pass** or **fail** (i.e., there are no inconclusive tests).

It is worth mentioning that whatever happens during the execution of a test, the observable input sequence is handled correctly by DT_b . In particular, if the input deviates too much from a standard input, the test is trivially passed. In this case our framework will additionally flag that the test is passed due to inputs not covered by robust cleanness. In practice, we try to eliminate such unproductive experiments by adequately configuring the human imprecision estimate η upfront.

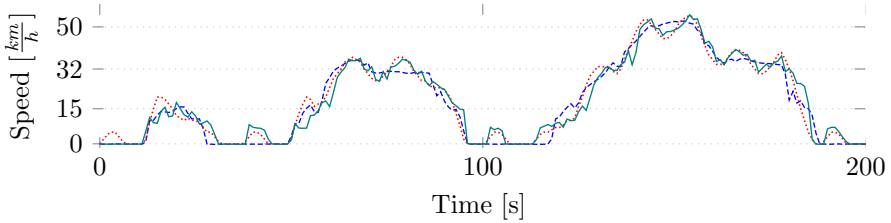


Figure 6.1: Initial 200s of a SineNEDC (red, dotted), its test drive (green) and the NEDC driven (blue, dashed).

For the purpose of practically demonstrating this three-step approach, we picked a Renault 1.5 dci (110hp) (Diesel) engine. This engine runs, among others, inside a Nissan NV200 Evalia which is classified as a EURO 6B car. The test cycle used in the original type approval of the car was NEDC. Emissions are cleaned using *exhaust gas recirculation* (EGR). The technical core of EGR is a valve between the exhaust and intake pipe, controlled by software. EGR is known to possibly cause performance losses, especially at higher speed. Car manufacturers might be tempted to optimise EGR usage for engine performance unless facing a known test cycle such as the NEDC.

We report here on two of the tests we executed apart from the NEDC reference test. PowerNEDC is a variation of the NEDC, where acceleration is increased from 0.94 m/s^2 to 1.5 m/s^2 in phase 6 of the NEDC elementary urban cycle (i.e. after 56 s, 251 s, 446 s and 641 s). It can be described by the same Ω_{case} as for the Volkswagen example. Ω_{in} is easy to write, but we omit it here as it is rather space consuming. The second test, called SineNEDC, defines the speed at time t to be the speed of the NEDC at time t plus $5 \cdot \sin(0.5t)$ (but capped at 0). Again, Ω_{case} matches the Volkswagen one. The input selection is given by

$$\Omega_{\text{in}}(h) = \max \left\{ \begin{array}{l} 0, \\ \text{NEDC}(|h|) + 5 \cdot \sin(0.5|h|) \end{array} \right\}.$$

Figure 6.1 shows the initial 200s of SineNEDC (red, dotted). The car was fixed on a *Maha LPS 2000* dynamometer and attached to an *AVL M.O.V.E iS* portable emissions measurement system (PEMS, see Figure 6.2) with speed data sampling at a rate of 20 Hz, averaged to match the 1 Hz rate of the NEDC. The human driver effectuated the NEDC with a deviation of at most 9 km/h relative to the reference (notably, the results obtained for NEDC are not consistent with the car data sheet, likely caused by lacking calibration and absence of any further manufacturer-side optimisations). The NEDC drive is depicted in Figure 6.1 as



Figure 6.2: Nissan NV200 Evalia on a dynamometer

	NEDC	Power	Sine
<i>Distance [m]</i>	11,029	11,081	11,171
<i>Avg. Speed [km/h]</i>	33	29	34
<i>CO₂ [g/km]</i>	189	186	182
<i>NO_x [mg/km]</i>	180	204	584

Table 6.1: Dynamometer measurements

a blue solid line.

The PowerNEDC test drive as well as the SineNEDC test drive both deviated by less than 15 km/h from the NEDC test drive, and hence less than κ_i , as per the contract described at the beginning of this section. The green solid line in Figure 6.1 shows the SineNEDC as driven. The test outcomes are summarised in Table 6.1. They show that the amount of CO₂ for the two tests is lower than for the one for NEDC driven. The NO_x emissions of PowerNEDC deviate by around 24 mg/km, which is clearly below κ_o . But the SineNEDC produces about 3.24 times the amount of NO_x, namely 404 mg/km more than what we measured for the NEDC, which is a violation of the cleanness context. Thus, this experiment reveals that the car under test is doped, since $\kappa_o = 180$ mg/km.

6.2 Conformance-Based Testing in Practice

In a second set of experiments we used the Nissan car that appeared in the previous section to practically evaluate the cleanness notions for hybrid systems

(cf. Section 3.4). In particular, we are interested in two types of experiments. First, we evaluate if hybrid cleanness is suitable to better overcome the human (timing) imprecisions when driving a car – in our case on a chassis dynamometer. This problem has already been discussed in Section 6.1.2, where we proposed a solution that was (and could) only be based on adjusting the threshold for value errors. With hybrid conformance, we are able to explicitly distinguish between tolerances in the value domain as well as the time domain. For the second type of experiments in this section we will experiment with the ability of conf-cleanness to control time and value deviations to drive the NEDC segments in a different order and to lengthen tests beyond the NEDC’s duration of 1180 s.

The experiments in this section are based on the conf-cleanness contract $\mathcal{C} = \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, \kappa_o, \text{Ret}_{\text{id}}, \text{Ret}_{\text{id}}, 0, 0 \rangle$ that encodes robust cleanness (cf. Proposition 3.111). We use the same $d_{\text{In}}, d_{\text{Out}}, \kappa_i$ and κ_o as in Section 6.1. The set $\text{StdIn} = \{\text{NEDC}\}$ is the singleton set that contains only the NEDC¹. The cleanness parameters $\text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_o$ and for the output retiming Ret_{id} and $\tau_i = 0$ are identical in all contracts in the remainder of this section. We explain for every experiment how we adjust \mathcal{C} ’s cleanness parameters for inputs. The experiments will demonstrate, how software doping analysis benefits from input retimings different than Ret_{id} .

NEDC Permutations First, we propose a new test cycle PermNEDC in which NEDC segments are permuted on the time axis. Figure 6.3 shows the test cycle. In each of the four UDC segments the three non-zero speed-phases are permuted. The transformation from NEDC to PermNEDC can be described by a retiming function r_p . An explicit definition of r_p is space consuming, hence we omit it. Along with the new cycle, we propose two suitable variants of contract \mathcal{C} with different input conformances. Neither input conformance is constrained by a time threshold; in other words, $\tau_i = \infty$.

- We define contract \mathcal{C}_a that is as \mathcal{C} , but entails the family of retimings $\text{Ret}_a = \{(r, r^{-1}) \mid r \in \mathcal{T} \rightarrow \mathcal{T} \text{ and } r \text{ is total and bijective}\}$ for inputs. Ret_a allows any reordering of the NEDC inputs. Notably, no inputs can be added or removed. This contract enforces the input conformance $\text{Conf}_{d_{\text{In}}, \infty, \kappa_i}^{\text{Ret}_a}$.
- Contract \mathcal{C}_p adjusts \mathcal{C} by enforcing for inputs the family of retimings $\text{Ret}_p = \{(r_p, r_p^{-1})\}$ that only allows the particular retiming r_p used to design the test cycle as discussed above. The input conformance induced by this contract is $\text{Conf}_{d_{\text{In}}, \infty, \kappa_i}^{\text{Ret}_p}$. This input conformance is

¹Strictly seen, we take the singleton set that contains the speed trajectory of our car when the driver *tried* to drive the NEDC. This speed trajectory slightly differs from the NEDC.

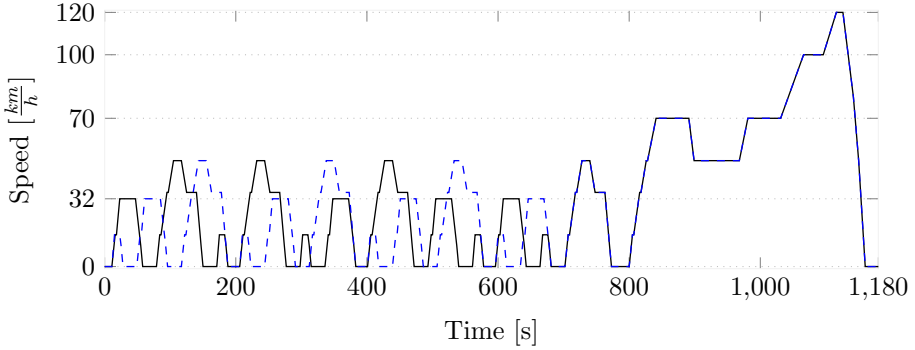


Figure 6.3: PermNEDC (solid, black line) compared to NEDC (dashed, blue line)

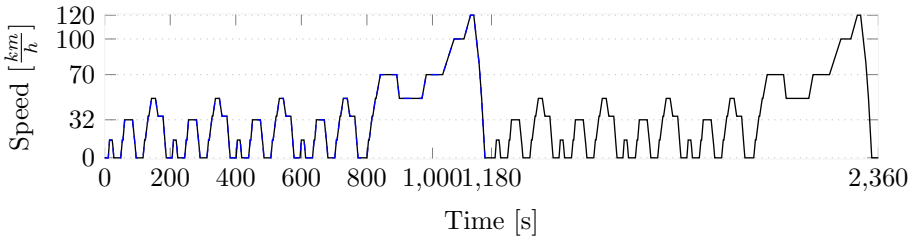


Figure 6.4: DoubleNEDC (solid, black line) compared to NEDC (dashed, blue line)

stricter than $\text{Conf}_{d_{in}, \infty, \kappa_i}^{\text{Ret}_a}$ above; it enforces that PermNEDC is not permuted any further by the driver.

NEDC Lengthening Conf-cleanness tests can run longer than the NEDC; this is not possible with robust cleanness. We propose the test cycle DoubleNEDC, which consists of two consecutive NEDCs. In contrast to all other test cycles in this paper, DoubleNEDC produces two outputs: the first after 1180s, and the second after 2360s. The first half of this cycle is a classical “cold” NEDC (i.e., the engine cooled down before the test execution). The second half is a “hot” NEDC, since the cool-down phase was implicitly skipped. Also, the PreCon phase is skipped implicitly; there is only a single EUDC (instead of three) prior to the second NEDC. The

inputs of both NEDCs can be compared by using the retiming functions id and $r_d = \lambda t. t \bmod 1180$. r_d maps time points of the global “test case clock” to the local time point in the NEDC time domain. DoubleNEDC requires to adapt contract \mathcal{C} to \mathcal{C}_d by replacing the input retiming family with $\text{Ret}_d = \{(\text{id}, r_d)\}$. Furthermore, to be able to check the second output against the (standard) NEDC output we need sync-conf-cleanness. Hence, we add the synchronisation retiming function $\text{Sync}_d(r_1, r_2) = (\text{id}, r_d)$ to \mathcal{C}_d , which enforces that both DoubleNEDC outputs are compared to the single NEDC output (independent of the input retimings r_1 and r_2).

Human Time Imprecision Tolerance Diesel doping tests are executed by humans driving a car. Humans tend to make mistakes when driving. Mistakes can be the over- or undershooting of the targeted speed (the error is on the value axis), or accelerations or decelerations happening too early or too late (the error is a shift on the time axis), or superpositions thereof. To compensate for both value and time errors, we use hybrid conformance. As a formal cleanness contract, this would be expressed by a variant of \mathcal{C} in which the input conformance is replaced by $\text{HybridConf}_{d, \tau_i, \kappa_i}$ for some $\tau_i > 0$. For the purpose of demonstration, we will later analyse several such variants of \mathcal{C} , each variant with a unique value for τ_i and κ_i , i.e., we consider the contract $\mathcal{C}(\tau_i, \kappa_i)$ parametrised in τ_i and κ_i . Concrete values for τ_i and κ_i must be specified when using the contract.

A test cycle that reflects drivings rich of acceleration and deceleration phases – and is hence particularly prone to human driving errors – is SineNEDC from Section 6.1.2. We will evaluate SineNEDC under several variants of $\mathcal{C}(\tau_i, \kappa_i)$.

Human time imprecision is as yet not considered in test cycles PermNEDC and DoubleNEDC; the cleanness contracts for both cycles require cycle-specific families of retimings. However, tolerance for human imprecision can be added to these predicates by means of conformance and retiming composition. Let $\text{Ret}^{(1)}$ and $\text{Ret}^{(2)}$ be two families of retimings. Then,

$$\text{Ret}^{(2)} \circ \text{Ret}^{(1)} := \{(r_1^{(2)} \circ r_1^{(1)}, r_2^{(2)} \circ r_2^{(1)}) \mid (r_1^{(2)}, r_2^{(2)}) \in \text{Ret}^{(2)} \text{ and } (r_1^{(1)}, r_2^{(1)}) \in \text{Ret}^{(1)}\}$$

is the component-wise function composition. The definition for conformance composition is $\text{Conf}_{d, \tau_2, \epsilon}^{\text{Ret}^{(2)}} \circ \text{Conf}_{d, \tau_1, \epsilon}^{\text{Ret}^{(1)}} := \text{Conf}_{d, \infty, \epsilon}^{\text{Ret}^{(3)}}$, where $\text{Ret}^{(3)} = \text{Ret}_{\tau_2}^{(2)} \circ \text{Ret}_{\tau_1}^{(1)}$ composes the individual retimings. Notably, the τ_1 - and τ_2 -constraints on $\text{Ret}^{(1)}$ and $\text{Ret}^{(2)}$ are applied before the composition. It is not necessary to apply further timing constraints to the resulting retiming, hence we allow infinite τ .

\mathcal{C}	$= \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, 15 \text{ km/h}, 180 \text{ mg/km}, \text{Ret}_{\text{id}}, \text{Ret}_{\text{id}}, 0, 0 \rangle$
\mathcal{C}_a	$= \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, 15 \text{ km/h}, 180 \text{ mg/km}, \text{Ret}_a, \text{Ret}_{\text{id}}, \infty, 0 \rangle$
\mathcal{C}_p	$= \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, 15 \text{ km/h}, 180 \text{ mg/km}, \text{Ret}_p, \text{Ret}_{\text{id}}, \infty, 0 \rangle$
\mathcal{C}_d	$= \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, 15 \text{ km/h}, 180 \text{ mg/km}, \text{Ret}_d, \text{Ret}_{\text{id}}, \infty, 0, \text{Sync}_d \rangle$
$\mathcal{C}(\tau_i, \kappa_i)$	$= \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, 180 \text{ mg/km}, \text{Ret}_{\text{hy}}, \text{Ret}_{\text{id}}, \tau_i, 0 \rangle$
$\mathcal{C}_p(\tau_i, \kappa_i)$	$= \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, 180 \text{ mg/km}, \text{Ret}_{\text{hy}+p}, \text{Ret}_{\text{id}}, \infty, 0 \rangle$
$\mathcal{C}_d(\tau_i, \kappa_i)$	$= \langle \text{StdIn}, d_{\text{In}}, d_{\text{Out}}, \kappa_i, 180 \text{ mg/km}, \text{Ret}_{\text{hy}+d}, \text{Ret}_{\text{id}}, \infty, 0, \text{Sync}_d \rangle$

Table 6.2: Overview of the evaluated contracts. $d_{\text{In}}(i_1, i_2) = |i_1 - i_2|$ and $d_{\text{Out}}(o_1, o_2) = |o_1 - o_2|$. **StdIn**, the families of retimings and **Sync_d** are as defined in the main text.

To overcome the human imprecisions for PermNEDC and DoubleNEDC, we use the parametrised contracts $\mathcal{C}_p(\tau_i, \kappa_i)$ and $\mathcal{C}_d(\tau_i, \kappa_i)$, adaptations of \mathcal{C}_p and \mathcal{C}_d , with input retiming families $\text{Ret}_{\text{hy}+p} := (\text{Ret}_{\text{hy}})_{\tau_i} \circ \text{Ret}_p$ and $\text{Ret}_{\text{hy}+d} := (\text{Ret}_{\text{hy}})_{\tau_i} \circ \text{Ret}_d$, respectively, and both with value threshold κ_i . Thus, contract $\mathcal{C}_p(\tau_i, \kappa_i)$ enforces conformance $\text{HybridPermConf}_{d, \infty, \kappa_i} = \text{HybridConf}_{d_{\text{In}}, \tau_i, \kappa_i} \circ \text{Conf}_{d_{\text{In}}, \infty, \kappa_i}^{\text{Ret}_p}$ and contract $\mathcal{C}_d(\tau_i, \kappa_i)$ enforces conformance $\text{HybridDoubleConf}_{d_{\text{In}}, \infty, \kappa_i} = \text{HybridConf}_{d_{\text{In}}, \tau_i, \kappa_i} \circ \text{Conf}_{d_{\text{In}}, \infty, \kappa_i}^{\text{Ret}_d}$ on inputs.

As for hybrid conformance in $\mathcal{C}(\tau_i, \kappa_i)$, we will specify concrete τ_i and κ_i upon usage of the contract. Notably, for DoubleNEDC, this does not have effects on the output conformance, because **Sync_d** does not consider the input retiming. This is important, because outputs are available only at time points 1180 and 2360 and must not be moved to time points different than that. We do not compose Ret_a and $(\text{Ret}_{\text{hy}})_{\tau_i}$, because Ret_a allows any possible NEDC permutation, which naturally reduces the effect of timing imprecisions.

Table 6.2 summarises the contracts presented above.

Computing Parameters of Hybrid Conformance In some experiments we compute, for a fixed time threshold τ_i and two test cycles, the minimal value error κ_i such that hybrid conformance holds for the input. The implementation of this computation is inspired by the the *HyperSTL** formula $\varphi_{d, \tau, \epsilon}^{\text{HybridConf}}$ (cf. Equation (2.2)), i.e., it computes $\min_{\kappa_i} \varphi_{d_{\text{In}}, \tau_i, \kappa_i}^{\text{HybridConf}}$ for two traces π_1 and π_2 . The

algorithm is sketched below.

$$\begin{aligned} \text{localMin}(t_1, i_1, i_2, d, \tau) &= \min \{d(i_2[t_2], i_1[t_1]) \mid t_2 \in [t_1 - \tau; t_1 + \tau] \cap \text{dom}(i_2)\} \\ \text{globalMin}(i_1, i_2, d, \tau) &= \max \{\text{localMin}(t_1, i_1, i_2, d, \tau) \mid t_1 \in \text{dom}(i_1)\} \\ \epsilon_{\min}(i_1, i_2, d, \tau) &= \max \{\text{globalMin}(i_1, i_2, d, \tau), \text{globalMin}(i_2, i_1, d, \tau)\} \end{aligned}$$

Here, $\text{localMin}(t_1, i_1, i_2, d, \tau)$ computes the minimal ϵ for subformula

$$\diamond_{[0, \tau]} d(i_{\pi_2}, i_{\pi_1}^*) \leq \epsilon \vee \diamond_{[0, \tau]} d(i_{\pi_2}, i_{\pi_1}^*) \leq \epsilon,$$

where the value of $i_{\pi_1}^*$ is frozen at time t_1 . $\text{globalMin}(i_1, i_2, d, \tau)$ reflects the *Globally* and *Freeze* operator: it finds the maximum by quantifying over all $t_1 \in \text{dom}(i_1)$ and by calling localMin with the frozen time value t_1 . To reflect the complete formula, $\epsilon_{\min}(i_1, i_2, d, \tau)$ returns the maximum of the conjuncts, which are the results of globalMin for both combinations of i_1 and i_2 . Thus, given d_{In} and τ_i , the minimal κ_i for two input traces i_1 and i_2 is computed by $\epsilon_{\min}(i_1, i_2, d_{\text{In}}, \tau_i)$.

The computations for `HybridPermConf` and `HybridDoubleConf` proceed in two steps. Both Ret_p and Ret_d are singleton sets; it is known which retiming must be applied first. For two input traces i_1 and i_2 and retiming (r_1, r_2) , there are shifted traces $i'_1 = i_1 \circ r_2$ and $i'_2 = i_2 \circ r_1$. The minimal κ_i for hybrid conformance is given by $\max \{\text{globalMin}(i_1, i'_2, d_{\text{In}}, \tau_i), \text{globalMin}(i_2, i'_1, d_{\text{In}}, \tau_i)\}$.

Test Results & Verdicts We executed each of NEDC, PermNEDC, DoubleNEDC and SineNEDC two times. We identify a concrete test execution by a suffix -1 or -2 to test cycle identifier (e.g., NEDC-1 is the first and NEDC-2 the second execution of NEDC). Raw data and the implementation of the analysis is available online [23]. For NEDC, we combined the result of both executions to an average value of 182 mg/km of NO_x . Notably, the Euro 6b regulation (to which our car is supposed to conform to) allows at most 80 mg/km, and the car under test is certified with 60.8 mg/km according to its documentation. The car is 3 years old.

For doping detection, a test verdict is only meaningful if its input trace is conformant to that of the average NEDC execution; otherwise, the test is trivially passed. We will first evaluate PermNEDC w.r.t. \mathcal{C}_a and \mathcal{C}_p , DoubleNEDC w.r.t. \mathcal{C}_d , and SineNEDC w.r.t. \mathcal{C} . To demonstrate the effects of hybrid conformance, we then analyse the experiments w.r.t. the parametrised variants of the contracts \mathcal{C} , \mathcal{C}_p and \mathcal{C}_d , respectively. By definition of the test cycles, the nominal value difference for PermNEDC and DoubleNEDC after retiming is zero, and for SineNEDC it is 5 km/h. Though, due to human imprecisions, the actual differences are significantly higher.

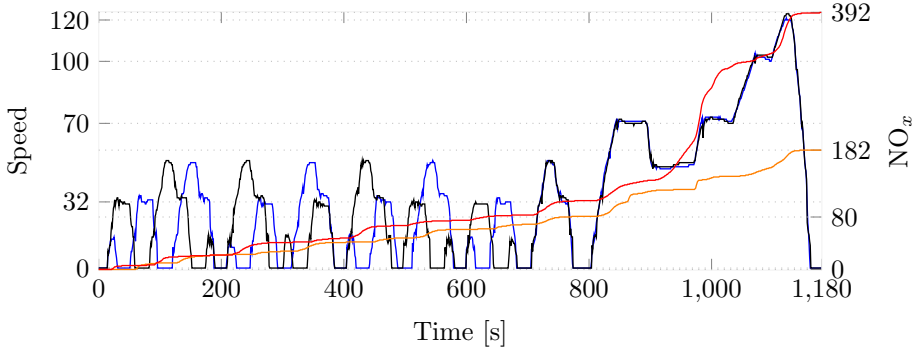


Figure 6.5: PermNEDC-1 speed (black) and NEDC speed (blue) in km/h, and accumulated NO_x for PermNEDC-1 (red) and NEDC (orange) in mg/km.

- The executions of PermNEDC are shown in Figures 6.5 and 6.6. The amount of emitted NO_x were 392 mg/km for PermNEDC-1 and 316 mg/km for PermNEDC-2. $\text{Conf}_{d_{\text{in}}, \infty, \kappa_i}^{\text{Ret}_a}$ does hold for $\kappa_i \geq 3$ km/h for both executions; with contract \mathcal{C}_a , which defines $\kappa_i = 15$ km/h, drastic deviations of NO_x witness doping. Hence, doping is detected for PermNEDC-1, i.e., the cleanness test fails, as the difference of NO_x (compared to NEDC) is 210 mg/km and hence greater than $\kappa_o = 180$ mg/km defined by \mathcal{C}_a . Test PermNEDC-2 passes with an NO_x difference of 134 mg/km, which is within the contract.

With contract \mathcal{C}_p and input conformance $\text{Conf}_{d_{\text{in}}, \infty, \kappa_i}^{\text{Ret}_p}$, the test verdict for PermNEDC-1 is different. $\text{Conf}_{d_{\text{in}}, \infty, \kappa_i}^{\text{Ret}_p}$ would only hold for $\kappa_i \geq 16$ km/h, which is above the contract defined threshold of 15 km/h. Hence, the speed trajectory of PermNEDC-1 is not adduced and the test trivially passed.

- DoubleNEDC-1 and 2, shown in Figures 6.7 and 6.8, lead to an average emission of 305 mg/km, respectively 308 mg/km of NO_x . Executions of DoubleNEDC are twice as long as regular NEDC tests and produce two outputs. The measurements for DoubleNEDC-1 report (229, 382) mg/km, for DoubleNEDC-2 (207, 408) mg/km. To determine the verdicts for contract \mathcal{C}_d , we first check if $\text{Conf}_{d_{\text{in}}, \infty, \kappa_i}^{\text{Ret}_d}$ holds. This turns out not to hold for DoubleNEDC-2, because we observed value deviations of up to 25 km/h. This test is therefore trivially passed. For DoubleNEDC-1 all value deviations remain below the 15 km/h threshold; this test run is thus to be

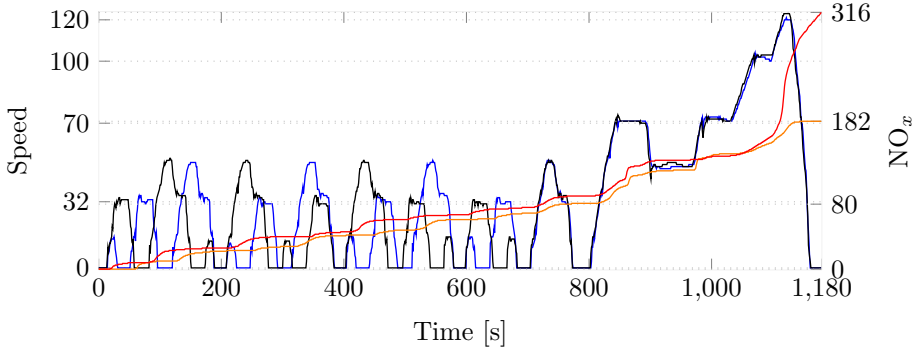


Figure 6.6: PermNEDC-2 speed (black) and NEDC speed (blue) in km/h, and accumulated NO_x for PermNEDC-2 (red) and NEDC (orange) in mg/km.

considered relevant for output comparison. According to the retiming synchronisation in \mathcal{C}_d , each of the outputs 229 and 382 must be compared to the NEDC output 182. The output conformance is violated for the second output with a difference of 200 mg/km exceeding the allowed $\kappa_o = 180$ mg/km threshold. Hence, DoubleNEDC-1 fails and doping is detected.

- During the test executions of SineNEDC, we measured 483 mg/km and 632 mg/km. The test progression is shown in Figures 6.9 and 6.10. In SineNEDC-1, speed values deviate by up to 18 km/h, which exceeds the κ_i threshold in \mathcal{C} , so this test run is trivially passed. SineNEDC-2 respects the κ_i threshold because inputs never deviate by more than 13 km/h. Consequently, SineNEDC-2 convicts our test car of doping, as the output difference of 450 mg/km is 2.5 times the allowed threshold κ_o .
- As discussed, we use hybrid conformance to compensate for human driving imprecisions. In this context, Table 6.3 details the effect of a choice of τ_i on the maximal value error. We fix a maximum value that we allow for the time offset τ_i . For this τ_i we analyse our dataset to find the minimal κ_i such that for the combination of τ_i and κ_i the input traces under consideration satisfy the cycle-specific hybrid conformance (defined by the corresponding cleanness contract). For $\tau_i = 0$ we get exactly the κ_i for which the two traces satisfy $\text{Conf}_{d_{in}, \infty, \kappa_i}^{\text{Ret}_p}$ (for PermNEDC), $\text{Conf}_{d_{in}, \infty, \kappa_i}^{\text{Ret}_d}$ (for DoubleNEDC), and $\text{TraceConf}_{d_{in}, 0, \kappa_i}$ (for SineNEDC). Table 6.3 shows

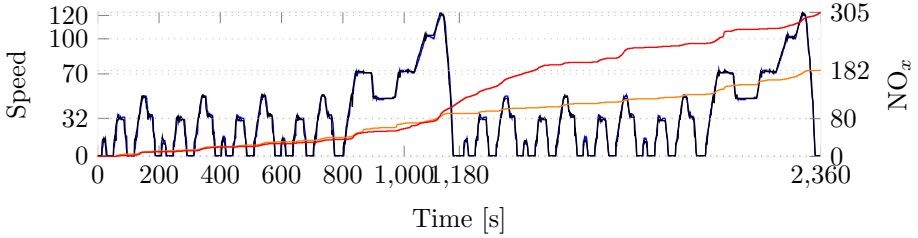


Figure 6.7: DoubleNEDC-1 speed (black) and NEDC speed (blue) in km/h, and accumulated NO_x for DoubleNEDC-1 (red) and NEDC (orange) in mg/km.

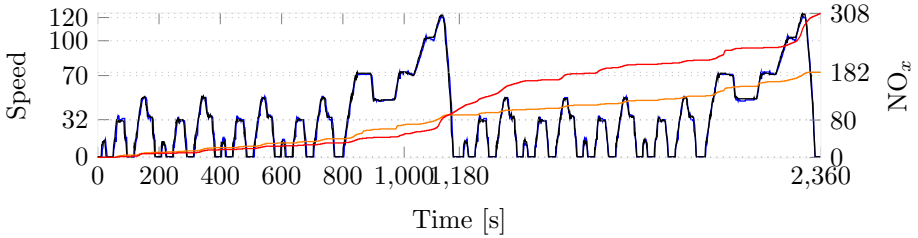


Figure 6.8: DoubleNEDC-2 speed (black) and NEDC speed (blue) in km/h, and accumulated NO_x for DoubleNEDC-2 (red) and NEDC (orange) in mg/km.

the computed κ_i values for $\tau_i = 0, 1, 2, 5, 10, 15$ and 20 seconds. As expected, an increasing τ_i induces the minimal κ_i to decrease. At $\tau_i = 5$ the decrease in the value error reduces notably. This happens because the error is only partially caused by the incorrect timing of the driver. From the values reported in Table 6.3 we see that if we allow deviation for the input $\tau_i = 2$, and keep $\kappa_i = 15$, then we have that the hybrid conformances $\text{HybridDoubleConf}_{d_{ln}, \tau_i, \kappa_i}(\text{NEDC}, \text{DoubleNEDC-2})$ and $\text{HybridConf}_{d_{ln}, \tau_i, \kappa_i}(\text{NEDC}, \text{SineNEDC-1})$ hold. For time threshold $\tau_i = 3$ seconds $\text{HybridPermConf}_{d_{ln}, \tau_i, \kappa_i}(\text{NEDC}, \text{PermNEDC-1})$ also holds. Thus, under hybrid conformance these pairs of traces will be considered in the cleanness test for contracts $\mathcal{C}_d(2, 15)$, $\mathcal{C}(2, 15)$ and $\mathcal{C}_p(3, 15)$, respectively, while under their original contract and input conformance they are to be dismissed.

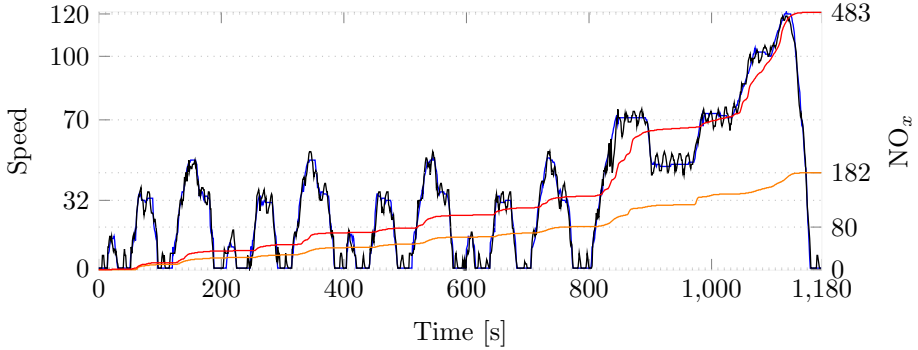


Figure 6.9: SineNEDC-1 speed (black) and NEDC speed (blue) in km/h, and accumulated NO_x for SineNEDC-1 (red) and NEDC (orange) in mg/km.

Test Name	Contract	$\tau_i = 0$	$\tau_i = 1$	$\tau_i = 2$	$\tau_i = 3$	$\tau_i = 5$	$\tau_i = 10$	$\tau_i = 15$	$\tau_i = 20$
PermNEDC-1	$\mathcal{C}_p(\tau_i, \kappa_i)$	$\kappa_i = 16$	$\kappa_i = 16$	$\kappa_i = 16$	$\kappa_i = 11$	$\kappa_i = 8$	$\kappa_i = 8$	$\kappa_i = 8$	$\kappa_i = 8$
PermNEDC-2	$\mathcal{C}_p(\tau_i, \kappa_i)$	$\kappa_i = 11$	$\kappa_i = 10$	$\kappa_i = 7$	$\kappa_i = 7$	$\kappa_i = 7$	$\kappa_i = 7$	$\kappa_i = 7$	$\kappa_i = 7$
DoubleNEDC-1	$\mathcal{C}_d(\tau_i, \kappa_i)$	$\kappa_i = 15$	$\kappa_i = 12$	$\kappa_i = 11$	$\kappa_i = 9$	$\kappa_i = 6$	$\kappa_i = 6$	$\kappa_i = 6$	$\kappa_i = 6$
DoubleNEDC-2	$\mathcal{C}_d(\tau_i, \kappa_i)$	$\kappa_i = 25$	$\kappa_i = 18$	$\kappa_i = 10$	$\kappa_i = 8$	$\kappa_i = 8$	$\kappa_i = 8$	$\kappa_i = 8$	$\kappa_i = 8$
SineNEDC-1	$\mathcal{C}(\tau_i, \kappa_i)$	$\kappa_i = 18$	$\kappa_i = 16$	$\kappa_i = 15$	$\kappa_i = 12$	$\kappa_i = 9$	$\kappa_i = 7$	$\kappa_i = 6$	$\kappa_i = 6$
SineNEDC-2	$\mathcal{C}(\tau_i, \kappa_i)$	$\kappa_i = 13$	$\kappa_i = 11$	$\kappa_i = 9$	$\kappa_i = 9$	$\kappa_i = 7$	$\kappa_i = 7$	$\kappa_i = 7$	$\kappa_i = 7$

Table 6.3: Comparison of minimal value thresholds κ_i for fixed τ_i . Values are given as km/h and time in seconds.

Evaluation and Discussion The amounts of emitted NO_x observed during our experiments provide clear indications of software doping regarding the car’s emission cleaning system. The conformance-based contracts provide the formal basis for this verdict, as discussed above. We here complement this fact with a more intuitive explanation of the behaviour observed.

- PermNEDC slightly reorders NEDC segments in the UDC part of the test cycle. During this part, the measured NO_x does not significantly differ from the NEDC reference. However, during the (unmodified) EUDC part, the amount of emissions grows significantly. It is very unlikely to find a physical explanation for the NO_x increase; and very likely, that the cleaning system is optimised specifically for the NEDC.
- The DoubleNEDC executions appear to reveal that the emission cleaning system optimisation can also rely on engine temperature or execution time

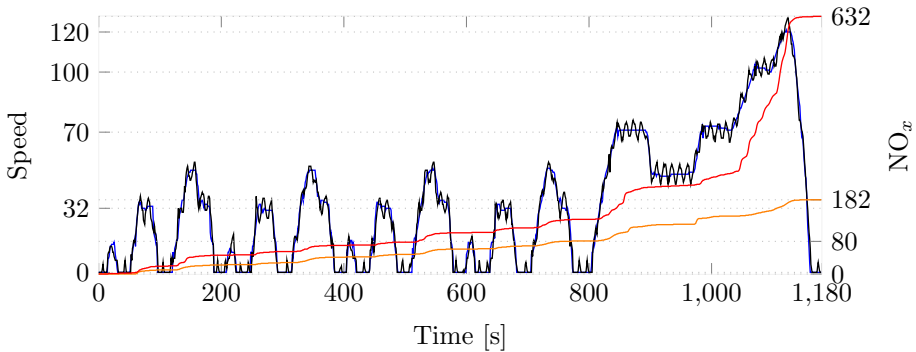


Figure 6.10: SineNEDC-2 speed (black) and NEDC speed (blue) in km/h, and accumulated NO_x for SineNEDC-2 (red) and NEDC (orange) in mg/km.

instead of speed data. Physically, many of the common emission cleaning techniques require a hot engine to work properly (and none of them requires a cold engine). Therefore, a lower NO_x value can be expected if the NEDC is run with a hot engine. In our experiments, however, the NO_x emissions in the hot half are almost two times higher than in the initial cold part. In other words, the emission cleaning performance is reduced after the first NEDC execution. There is no physical explanation for this behaviour. Inside the software, detecting the end of an NEDC trip can be implemented very easily, for instance with a timer counting from 1180 – the length of NEDC – to zero.

- With SineNEDC, we test the cleaning system during driving behaviour which is rich in accelerations and decelerations. An increased amount of NO_x can possibly be explained by physical phenomena. However, we measured an increase of factors 2.7 and 3.5; these numbers can be safely considered as too high for a trustworthy emission cleaning system.

Software doping theory provides the basis for detecting software behaviour violating a formal contract. In this, physical aspects of the emission cleaning system should be considered during the construction of test cases, and test cycles for which drastically higher emissions can be explained physically, should not be considered. The test cycles we used for our experiments were picked with automotive expertise to avoid physically stressful cycles. If test cases are generated automatically from a contract, the physical constraints could be captured

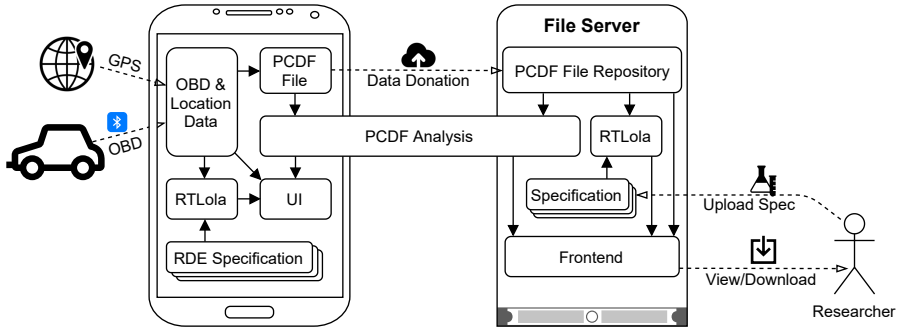


Figure 6.11: Components of the Car Data Platform

by the contract.

The contracts we use for our experiments can be interpreted as very generous in favour of the manufacturers. Input thresholds such as 15 km/h and 2 seconds appear as reasonable values, keeping all tests close enough to the original NEDC. For the output threshold, we use a very large deviation value of 180 mg/km, which allows NO_x emissions to almost double compared to the original NEDC value. Despite the generosity of the contracts, our experiments have been able to reveal doping for all experiments except PermNEDC-2 (and PermNEDC-1).

The analysis of the data shows that it is indeed necessary to not only consider a deviation of value, but to also allow for timing deviations. Considering value and timing deviations offers a rich set of potential test cycles for doping tests and allows to realistically verify conformance of a test cycle and a reference cycle; especially when the quality of the studied driving tests suffers from the human-caused input distortions. In this regard, cleanness notions entailing hybrid conformance are more adequate than conformance notions demanding punctual test executions, such as robust cleanness. Without hybrid conformance, more of the doping cases we have detected would slip through.

Finally, while hybrid conformance is central to the case study considered here, our generic theory of conformance-based cleanness allows for using other conformance notions as appropriate for the CPS under test.

6.3 Car Data Platform and LolaDrives

When we do software doping tests with cars, we must assume that the system under investigation is a black-box system. “Observing the system” is the only possibility to gain insights into the software controlling the system. For cars,

this raises three concrete questions: 1) how can we observe what the car is doing, 2) how can we accumulate sufficiently many observations to draw conclusions, and 3) how can we address the above two questions in a cost efficient manner? Our answer to all three questions is the *Car Data Platform* (CDP). It combines several car-related tools and services.

Figure 6.11 summarises the main components that belong to the CDP. At the core is the *Portable Car Data Format* (PCDF) to encode car-related diagnostics data in a well-defined way [107]. The central place to collect PCDF files is a data server, which provides interfaces to submit new files, to analyse existing files, and to view analysis results. New files are typically submitted by instances of the mobile app LolaDrives, which run in-the-field connected to the diagnostics interface OBD of a car. LolaDrives enables and encourages users to donate their PCDF files. CDP offers a repository of analyses for PCDF files [26] that is used by LolaDrives instances as well as the server. A server frontend provides researchers with convenient access options regarding the analysis results computed for donated data.

6.3.1 LolaDrives

LolaDrives is an Android application publicly available in Google’s Play Store [1]. It is compatible with many Bluetooth OBD adapters to access diagnostic data from cars. The app supports two main diagnosis modes: real-time diagnostics monitoring and RDE test guiding.

Diagnostic Monitoring. In diagnostics monitoring mode, the user selects a set of diagnostic parameters (e.g., vehicle speed, ambient air temperature, etc.), for which real-time values are shown on the screen (see Figure 6.12). Monitoring is supported for all cars with combustion engine built since 2005².

RDE Testing. In RDE test mode, the app constantly analyses the driving behaviour of the user to check whether it satisfies the RDE constraints. The constraints [118] require the driver to equally partition the test into an urban, rural and motorway mode, and to adhere to realistic acceleration and deceleration behaviour. LolaDrives displays the most critical RDE parameters (that the driver can influence) by visualizing the evaluations of the RTLOLA streams presented in Section 2.8.2. This allows the test personnel to easily detect and understand constraint violations. Figure 6.13 shows the RDE feedback view of LolaDrives. From top to bottom, it shows the total time, which must be between 90 and 120 min

²Some electric vehicles are supported, too. However, it is legally not enforced that electric vehicles expose the OBD protocol at an OBD interface, but they may.

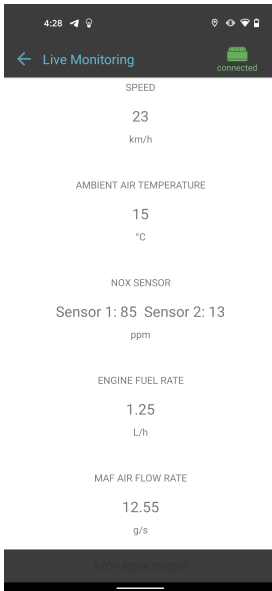


Figure 6.12: Live monitoring view

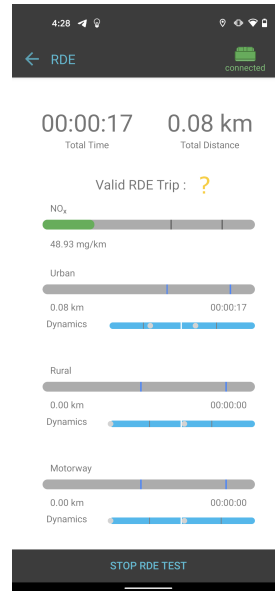


Figure 6.13: RDE test guide view

to finish the test, and the total distance travelled (corresponding to the RTLOLA stream d in Section 2.8.2). The next line indicates the current state of the conditions for a valid RDE test drive disregarding emission data. In the screenshot, the drive is still in progress and inconclusive, indicated by the question mark. Instead, the UI can also indicate success or failure. The latter verdict can occur far before the time limit is reached, caused by an irrecoverable situation such as transgression of the 160 km/h speed limit. We remark that currently LolaDrives does not always detect if for a test the RDE constraints are irrecoverably violated. For example, if a test has run for 119 minutes, but there are still at least 3 km remaining to drive in the motorway mode to cover the 23% share of the total trip distance, then this would require the driver to drive faster than 160 km/h for the remaining minute. Since this is forbidden by the regulation, the RDE constraints are in this moment irrecoverably violated. In the converse case in which the indicator reports a successful drive, this concerns the trip up until this moment. Together with the regulatory constraints, this implies that the current verdict can alternate between success and inconclusive from minute 90 to 120 and may also jump to failure. As there is no specific point in time when the test ends, the app continues to compute statistics until the tester manually stops it or the 120 min mark is reached. Beneath the status indicator is the green NO_x



Figure 6.14: Log of OBD events

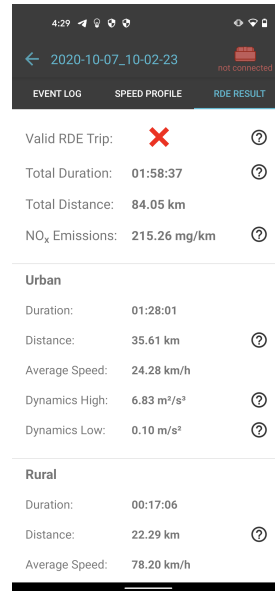


Figure 6.15: RDE result summary

bar displaying the total NO_x emissions (RTLOLA `nox_per_kilometer` stream in Section 2.8.2). The two markings denote the permitted thresholds of 168 mg/km for cars admitted before 2021, and 120 mg/km for cars admitted in 2021 or later.

The next three UI groups represent the progress in each of the distinct modes: urban, rural, and motorway. Each group consists of two horizontal bars. The gray progress bar displays the distance covered in the respective mode (e.g., RTLOLA `r_d` stream for the rural mode in Section 2.8.2). The vertical blue indicators denote lower and upper bounds as per official regulation, for an expected trip length configured by the user. We remark that the configured trip length is solely used to determine the initial position of the distance indicators. In particular, when the user drives the car so that the distance would cross the upper bound of a mode, then LolaDrives instead increases the upper bound as necessary to avoid an overstepping and updates the distance bound indicators for the other two modes accordingly. The blue bar below the gray one illustrates two different metrics for the driving dynamics (e.g., RTLOLA `r_rpa` and `r_pct1_dyn` streams for the rural mode in Section 2.8.2). Both dots need to eventually remain in the middle of the bar below/above their thresholds. A more aggressive acceleration behaviour shifts the dots to the right and a passive driving style to the left.

The RDE test guide is available for cars with compatible diagnosis character-

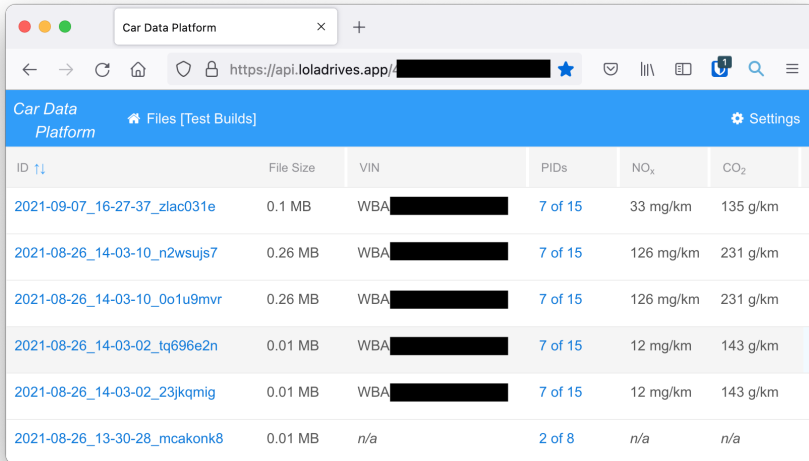
istics. Necessary diagnostic parameters to check the RDE constraints are vehicle speed, fuel type and the ambient air temperature. Furthermore, *LolaDrives* needs access to the location services of the phone to check RDE conditions that are concerned with the altitude of the car. To compute the amount of emitted NO_x , the exhaust mass flow and relative amount of NO_x as measured in the exhaust pipe are necessary. The exhaust mass flow can be approximated from the mass air flow into the engine and the fuel rate [82]. Instead of the fuel rate, the air-fuel ratio can also be used. In case, neither is available, an expected air-fuel ratio based on the fuel type can be used.

Dynamic Specification. While the specification of the RDE regulation itself is fixed, the app may be connected to a variety of cars, not all of which provide the relevant data. For instance, we need the exhaust mass flow (EMF) which is usually measured directly by the PEMS. In case the car does not come equipped with an EMF sensor and we do not have a PEMS at our disposal, we may still be able to calculate the EMF from other data. This has already been demonstrated in previous work and has been the nucleus of the lightweight and low-cost variant of the RDE test procedure [82] *LolaDrives* relies on. Clearly, having the layperson end user change or modify the RTLOLA specification to account for different car configurations is infeasible. Hence, *LolaDrives* automatically adapts the specification to a specific car. To this end, it queries the car for the supported sensors and then automatically pieces together the best specification for the car. This specification then includes the necessary formulae to compute values such as the EMF from data the car actually provides.

Drive History. In both monitoring and RDE testing mode, the data received from the car is stored in a PCDF file. All recorded files can be inspected in the “History” section of *LolaDrives*. It is possible to inspect the raw data received from the car (Figure 6.14), but also results of the analyses once it is available (e.g., RDE results in Figure 6.15). Every analysis has individual requirements about the set of diagnostics parameters that must be available in the record.

6.3.2 Technical Setup

To collect PCDF files at a central place, CDP provides a file server with an easy to use API to submit new files and to get analysis results. The server provides two distinct repositories, one for internal usage by the developers, and one for official usage of the PlayStore version protected by a strong privacy policy. Figure 6.16 shows the web frontend for the internal repository. Internal files can be downloaded, and several analysis results can be inspected. Accessing and submitting data is protected by authorisation tokens. The implementation of



ID ↑	File Size	VIN	PIDs	NO _x	CO ₂
2021-09-07_16-27-37_zlac031e	0.1 MB	WBA [REDACTED]	7 of 15	33 mg/km	135 g/km
2021-08-26_14-03-10_n2wsujs7	0.26 MB	WBA [REDACTED]	7 of 15	126 mg/km	231 g/km
2021-08-26_14-03-10_0o1u9mvr	0.26 MB	WBA [REDACTED]	7 of 15	126 mg/km	231 g/km
2021-08-26_14-03-02_tq696e2n	0.01 MB	WBA [REDACTED]	7 of 15	12 mg/km	143 g/km
2021-08-26_14-03-02_23jkqmig	0.01 MB	WBA [REDACTED]	7 of 15	12 mg/km	143 g/km
2021-08-26_13-30-28_mcakonk8	0.01 MB	n/a	2 of 8	n/a	n/a

Figure 6.16: Server Frontend: List of internal test files

the server is an interplay of individual components, deliberately held flexible so that components can be replaced, removed or extended by other components in the future.

The Car Data Platform is a collection of tools and services for car-related research. It is designed in a modular way to allow for a flexible development in the future. Modules are encapsulated in software packages, some of them are publicly available online. The implementation of the analyses shared by LolaDrives and the file server, and the PCDF core for easy handling of PCDF files is written in Kotlin and published online [107, 26]. Maven artifacts are available for integration in other projects. An analysis worker (written in Kotlin) is running on the server to regularly check if new files have been uploaded and to run the analyses on new files in the background. The file server backend is an npm package written in TypeScript and Express. The frontend is also an npm package providing a react UI to show the contents on the server.

The entire RTLOLA toolkit is written in Rust and available online³. The Rust compiler uses LLVM as a backend, which enables compilation for Android devices. Moreover, the implementation of the interpreter⁴ contains both a stan-

³<https://rtlola.org>

⁴<https://crates.io/crates/rtlola-interpreter>

dalone interpreter and a library. The library exposes a C-compatible interface, which can in turn interface with Java Virtual Machine (JVM)-based languages such as Kotlin. This enables linking of `RTLOLA` and `LolaDrives`. The latter is written in Kotlin and also freely available online [108].

Car Simulator. By the nature of the app’s functionality, the testing of new features of `LolaDrives` requires it running on a phone, connected via Bluetooth to an OBD adapter, which is plugged into a car being driven by a human. This overhead makes testing quite inefficient. We therefore simplified the test procedure by instead constructing a physical car simulator, to which the OBD adapter can be connected. The simulator consists of two parts: 1) a regular PC software to parse and prepare `PCDF` files for simulation, and 2) an Arduino board attached to a CAN bus shield. The Arduino board serves as a “diagnosis storage device” to which the PC repeatedly writes diagnostic data. The CAN bus shield is connected to the OBD adapter; it reads the diagnostics data from the diagnosis storage via the OBD protocol. The PC transmits each event in the source `PCDF` file to the board in the same order and with the same delay as it was recorded. PC and Arduino communicate via a protocol based on *Consistent Overhead Byte Stuffing* (COBS) [31]. The PC software is written in Kotlin and the code on the board is written in the typical C++ based Arduino language. Figure 6.17 shows the simulator in action.

Privacy. An important feature of `LolaDrives` and `CDP` is the support for data donation; users can opt-in to upload the files recorded by `LolaDrives` during monitoring or RDE test mode. But `PCDF` files may contain personal data, for example, the vehicle identification number or GPS coordinates. Collecting personal data is regulated by data protection laws; in our case, the General Data Protection Regulation (GDPR). The GDPR concedes every EU citizen the rights to receive a copy of their data (in a machine readable format), to have it corrected, or deleted. All privacy policies in the EU must educate these rights to the users. `LolaDrives` does so in full. Moreover, our privacy policy [24] explains that data uploads are automatically deleted after at most 15 years (counting 5 years for doing research with it and 10 years data retention time after publication recommended by the German Research Council DFG [44]). Data donations are voluntary. Refusing or withdrawing consent does not restrict the available features of `LolaDrives` in any way.

6.3.3 Demonstration

This section discusses the user perspective on `LolaDrives`. We report on the use of `LolaDrives` for conducting RDE test drives with two rented vehicles (the precise



Figure 6.17: OBD Simulator

car model being unknown upfront).

Overview. The preparation of the test requires the user to plug the OBD-adapter into the OBD-port of the car. After starting car and app, LolaDrives receives data packets and determines the sensor profile of the car, assuming phone and adapter are paired via Bluetooth. As the provided diagnostics data suffices to evaluate the RDE constraints, the app selects the appropriate RTLOLA specification and initializes the RTLOLA monitor. LolaDrives then starts filtering and visualising the data output and trigger notifications provided by the monitor, as explained in Section 6.3.1.

Test Drive. The technical framework and visual feedback of the app were tested in two experiments. The first experiments involved two RDE test drives that were both conducted with an Audi A6 Avant 45-TDI hybrid diesel, which was admitted in 2020 under the EURO 6D-TEMP(-EVAP-ISC) regulation with an NO_x threshold of 80 mg/km under lab conditions and 168 mg/km for RDE conditions. We denote this car as *A20* and the RDE tests as *A20.1* and *A20.2*. The second experiment involves four RDE tests that were conducted with the successor of the above car – an Audi A6 50-TDI hybrid diesel admitted in 2021 under the

	Drive A20.1			Drive A20.2		
	Distance [km]	NO _x [mg/km]	CO ₂ [g/km]	Distance [km]	NO _x [mg/km]	CO ₂ [g/km]
Urban	35.61	138	221	37.42	111	250
Rural	22.29	303	155	27.46	82	170
Motorway	26.15	245	153	25.33	103	176
Total	84.05	215	183	90.24	100	205

	Drive A21.1			Drive A21.2		
	Distance [km]	NO _x [mg/km]	CO ₂ [g/km]	Distance [km]	NO _x [mg/km]	CO ₂ [g/km]
Urban	43.54	31	221	28.75	42	230
Rural	29.17	11	121	30.88	19	164
Motorway	28.73	25	166	15.66	17	137
Total	101.44	23	183	75.29	27	184

	Drive A21.3			Drive A21.4		
	Distance [km]	NO _x [mg/km]	CO ₂ [g/km]	Distance [km]	NO _x [mg/km]	CO ₂ [g/km]
Urban	37.96	76	227	36.46	36	219
Rural	29.49	222	199	25.29	21	142
Motorway	43.01	41	145	34.42	29	150
Total	110.46	101	188	96.17	30	174

Table 6.4: Aggregation of the emission data based on the CDP.

(moderately stricter) EURO 6D(-ISC-FCM) regulation enforcing the same NO_x threshold of 80 mg/km under lab conditions and a smaller 120 mg/km threshold for RDE conditions.⁵ We denote this car as *A21* and the RDE tests done with this car as *A21.1* to *A21.4*. Among the diagnosis parameters available within these cars are vehicle and engine speed, ambient temperature, engine fuel rate and mass air flow. The A20 car has two NO_x-sensors – one in front and one behind the emission cleaning system in the exhaust pipe. The A21 car has three NO_x-sensors – presumably one in front, one between components of the emission cleaning system and one behind it. With this set of sensors, the car is compatible

⁵We determined the precise car model and the variant of the Euro 6d norm using the registration certificate of the car and the German Wikipedia [132, 133]

for RDE tests with LolaDrives. We configured LolaDrives to assume an expected trip length of 83 km for the visual guidance.

Test drives A20.1 and A21.4 meet all conditions to be considered as a valid RDE test. Test drives A20.2 and A21.1 did not experience sufficiently much accelerations in the urban mode to be a valid RDE test; in A21.2 there was a malfunctioning of the OBD adapter and the test was forced to end before reaching the minimal 23 % share of the total trip length in the motorway mode; and test drive A21.3 is invalid, because we failed to comply to the maximum altitude difference of 100 m between start and end point. In all cases, LolaDrives correctly confirmed the satisfaction and violation of the RDE criteria. For the valid tests A20.1 and A21.4 we measured 215 mg/km and, respectively, 30 mg/km of NO_x emissions. Hence, test A20.1 reveals a violation of the RDE regulation while the emissions measured during test A21.4 are conforming. A comprehensive overview for all measured emissions is shown in Table 6.4. In this table, the distances and the total amounts of emitted NO_x are computed directly by LolaDrives; all other values have been computed using a custom RTLOLA specification that was applied to the trip recordings after they were uploaded to the CDP by LolaDrives's data donation feature. Notice that the NO_x value for A21.3 is significantly higher than for the other A21 drives. Additional diagnostics data recorded made it evident that the car had been cleaning its diesel particulate filter and NO_x adsorber during the test. We are, however, not certain if this is the reason for the higher emissions. Anyway, if A21.3 were a valid RDE test, the overall NO_x emissions are still below the threshold of 120 mg/km defined in the regulation.

Figure 6.18 shows the route of test drive A20.2. The first half of the time constituted the urban segment (green). The next 30-40% of the test mainly consisted of the rural segment (purple) followed by the motorway segment (red). The map shows that the rural and motorway segments are regularly interrupted by other segments when the driver had to slow down for traffic reasons; the three phases are solely defined by the vehicle speed. As a result, depending on external circumstances, the driver cannot freely choose their environment, potentially exceeding the distance thresholds for a different segment by accident. It is, therefore, advisable to start with the urban environment and progress to the next environment as early as possible.

6.4 CDP-Based Test Input Selection

In Section 5.4 we presented a probabilistic falsification technique that strategically searches for a test cycle that minimises the robustness estimate for u-robust cleanliness; optimally, the robustness estimate becomes negative in which case u-

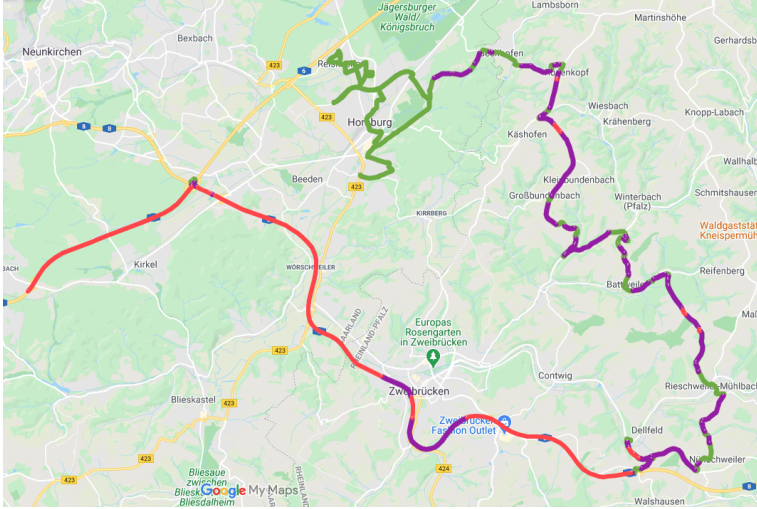


Figure 6.18: Map with the A20.2 test route highlighted.

robust cleanness (and hence robust cleanness) is falsified. Again, this technique is not straightforwardly applicable to real cars. The falsification process requires hundreds of iterations or more until it finds a suitable test cycle. That is, it requires hundreds of experiments (or more) on a chassis dynamometer, which is practically not feasible. Additionally, the human imprecision when driving a test cycle remains a problem, and it is unclear how this influences the effectiveness of the falsification technique.

This section discusses how to tailor the generic probabilistic falsification approach for STL based on Algorithm 2.1 to the particular case of diesel emissions and how to overcome the problems explained above using the integrated testing approach proposed in Section 5.5.

Robustness Taking up the experiments in Section 6.1, assume a cleanness context $\mathcal{C} = \langle \{\text{NEDC} \cdot \circ\}, \kappa_i, \kappa_o, d_{\text{In}}, d_{\text{Out}} \rangle$, where the standard behaviour Std is described by a single trace that is the result of driving the NEDC and getting the average amount of NO_x measured during this test. That is, NEDC here represents the sequence of 1180 inputs with the k th input defining the speed of the car after k seconds from the beginning of the test. The (average) amount of emitted NO_x follows the inputs. By restricting the input space to $\text{In}_{\{\text{NEDC} \cdot \circ\}, \kappa_i}$ as explained in Section 5.4, STL formula $\varphi_{\text{U-rob}}$ from Proposition 5.45 can be

simplified to

$$\Box(d_{\text{Out}}((\text{NEDC} \cdot \circ)\downarrow_{\circ}, s\downarrow_{\circ}) - \kappa_{\circ} \leq 0). \quad (6.1)$$

This is because the conjunction and disjunction over standard traces becomes obsolete for only a single standard trace. For the same reason, the requirement $\Box(\text{eq}(s_a\downarrow_i, s_b\downarrow_i) \leq 0)$ becomes obsolete, as the compared traces are always identical. In the \mathcal{W} subformula, the right proposition is always false, because of the restricted input space: the proposition collapses to $d_{\text{In}}((\text{NEDC} \cdot \circ)\downarrow_i, s\downarrow_i) - \kappa_i > 0$ and the input domain $\text{In}_{\{\text{NEDC} \cdot \circ\}, \kappa_i}$ is $\{w \in \mathbf{M} \mid \forall k \in [0, 1180]. d_{\text{In}}(w[k]\downarrow_i, (\text{NEDC} \cdot \circ)[k]\downarrow_i) \leq \kappa_i\}$. Thus, by the definition of \mathcal{W} and \mathcal{U} , the \mathcal{W} subformula is equivalent to formula (6.1). We implemented Algorithm 2.1 for the robustness computation according to formula (6.1).

Emissions Approximation In practice, running tests like NEDC with real cars is a time consuming and expensive endeavour. Furthermore, tests on chassis dynamometers are usually prohibited to be carried out with rented cars by rental companies. On the other hand, car emission models for simulation are not available to the public – and, anyway, models provided by the manufacturer cannot be considered trustworthy. To carry out our experiments, we instead follow the integrated testing approach from Section 5.5 and use an approximation technique that estimates the amount of NO_x emissions of a car along a certain trajectory based on data recorded during previous trips with the same car, sampled at a frequency of 1 Hz (one sample per second). Notably, these trips do not need to have much in common with the trajectory to be approximated. A trip is represented as a finite sequence $\mathbf{t} \in (\mathbb{R} \times \mathbb{R} \times \mathbb{R})^*$ of triples, where each such triple (v, a, n) represents the speed, the acceleration and the absolute amount of NO_x emitted at a particular time instant in the sample. Speed and acceleration can be considered as the main parameters influencing the instant emission of NO_x . This is, for instance, reflected in the RDE regulation [82, 118] where the decisive quantities to validate the test route and driving behaviour during RDE tests are speed and acceleration.

A data recording \mathcal{D} is the union of finitely many trips \mathbf{t} . We can turn such a recording into a predictor function \mathcal{P} that predicts NO_x values for pairs of speed and acceleration:

$$\mathcal{P}(v, a) = \text{average}[n \mid (\exists v', a'. (|v - v'| \leq 2 \wedge |a - a'| \leq 2 \wedge (v', a', n) \in \mathcal{D}))].$$

The amount of NO_x assigned to a pair (v, a) here is the average of all NO_x values seen in the recording \mathcal{D} for $v \pm \ell$ and $a \pm \ell$, with $0 \leq \ell \leq 2$. To overcome measurement inaccuracies and to increase the robustness of the approximated emissions, the speed and acceleration may deviate up to 2 km/h, and 2 m/s²,

respectively. This tolerance is adopted from the official NEDC regulation [124], which allows up to 2 km/h of deviations while driving the NEDC.

Experiment setup To demonstrate the practical applicability of our implementation of Algorithm 2.1 and our NO_x approximation, we report here on two experiments. The first experiment evaluates the recordings A20.1 and A20.2 from Section 6.3.3. The predictor defined above estimates that the NO_x emission for car A20 when driving the NEDC is 86 mg/km. The second experiment involves the RDE tests A21.1, A21.2 and A21.4 (leaving out A21.3, because it produced significantly higher emissions than the other three trips). Car A21 seems to have a significantly better emission cleaning system: the estimated amount of NO_x emitted during the NEDC is 9 mg/km. Recall that car A20 was falsified w.r.t. the RDE specification in Section 6.3.3. Neither A20 nor A21 has been falsified w.r.t. a cleanness notion defined in Chapter 3.

Cleanness Context Before turning to falsification of robust cleanness, we need to spell out meaningful cleanness contexts. We assume that the behaviour of the car can be modelled as an IOTS and take as input domain the set $\text{In} \subseteq \mathbb{R}_{\geq 0}$ of speed values, and as output domain the set $\text{Out} \subseteq \mathbb{R}_{\geq 0}$ representing the average amount of NO_x emitted during the test. The standard behaviour is the singleton set containing the NEDC drive followed by a single output that is the (predicted) average amount of NO_x emitted for NEDC. That is, for A20 we get $\text{Std} = \{\text{NEDC} \cdot 86_{\circ}\}$ and for A21 $\text{Std} = \{\text{NEDC} \cdot 9_{\circ}\}$. As distance functions, we use the past-forgetful functions from Section 6.1.1 (defined on page 166). For κ_i , the cleanness context uses the same value $\kappa_i = 15$ km/h that has been proven to be a reasonable choice in the sections above. The threshold for NO_x emissions under lab conditions is 80 mg/km. The emission limits for RDE tests depend on the admission date of the car. Cars admitted in 2020 or earlier, must emit 168 mg/km at most, and cars admitted later must adhere to the limit of 120 mg/km. For our experiments, we use $\kappa_{\circ} = 88$ mg/km for A20 and $\kappa_{\circ} = 40$ mg/km for A21 to have the same (absolute) tolerances as for RDE tests. Effectively, the upper threshold for A20 is $84 + 88 = 172$ mg/km, and for A21 the limit is $9 + 40 = 49$ mg/km. Notice that for software doping analysis, the output observed for a certain standard behaviour and the constant κ_{\circ} define the effective threshold; this threshold is typically different from the threshold defined by the regulation.

Evaluation We modified Algorithm 2.1 by adding a timeout condition, i.e., if the algorithm is not able to find a falsifying counterexample within 3,000 iterations, it terminates and returns both the trace for which the smallest robustness

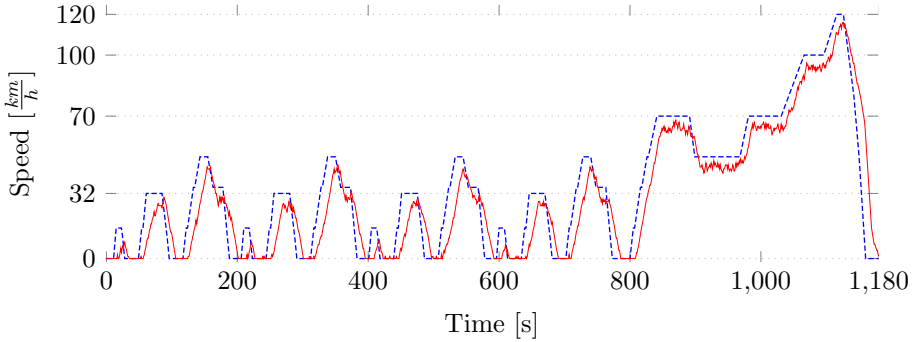


Figure 6.19: NEDC speed profile (blue, dashed) and input falsifying \mathcal{C} for $\kappa_o = 88 \text{ mg/km}$ (red) with 182 mg/km of emitted NO_x .

has been observed and its corresponding robustness value. Hence, if falsification of robust cleanness for a system is not possible, the algorithm outputs an approximation on how robust the system satisfies robust cleanness.

For the concrete case of the diesel emissions, the robustness value during the first 1180 inputs (sampled from the restricted input space $\text{In}_{\text{Std}, \kappa_i}$) is always κ_o . When the NEDC output \mathbf{o}_{NEDC} and the non-standard output \mathbf{o} are compared, the robustness value is $\kappa_o - |\mathbf{o}_{\text{NEDC}} - \mathbf{o}|$ (cf., formula (6.1), the quantitative semantics of STL, and definition of d_{Out}). Hence, for test cycles with small robustness values, we get NO_x emissions \mathbf{o} that are either very small or very large compared to \mathbf{o}_{NEDC} . We ran the modified Algorithm 2.1 on the predictor functions for A20 and A21 using the contexts defined above. For A20, it found a robustness value of -8 , i.e., it was able to falsify robust cleanness relative to the assumed contract and found a test cycle for which NO_x emissions of 182 mg/km are predicted. The test cycle is shown in Figure 6.19. For A21, the smallest robustness estimate found – even after 100 independent executions of the algorithm – was 38 , i.e., A21 is predicted to satisfy robust cleanness with a very high robustness upper bound. The corresponding test cycle is shown in Figure 6.20.

6.5 Related Work & Contributions

This chapter applies the theory developed in this thesis (in particular that in Chapter 5) to the Diesel Emissions Scandal. In particular, Section 6.4 instanti-

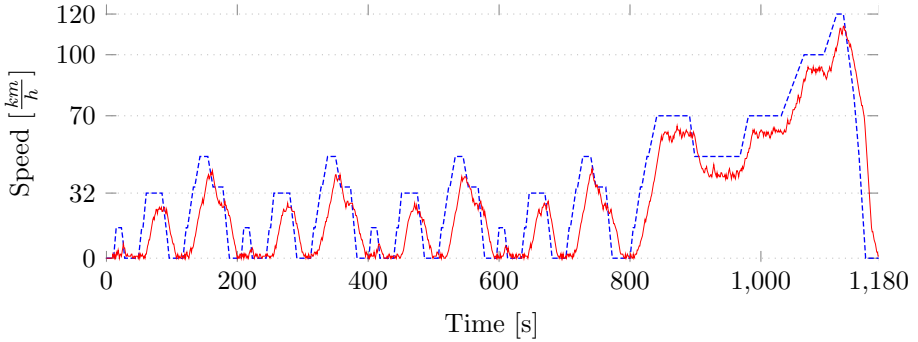


Figure 6.20: NEDC speed profile (blue, dashed) and input maximising NO_x emissions to 11 mg/km (red).

ates the integrated testing approach from Section 5.5. To this end, we proposed an easy to understand emissions prediction technique that can be seen as a simple instance of machine learning. Learning or approximating the behaviour of a system under test has been studied intensively. Meinke and Sindhu [90] were among the first to present a testing approach incrementally learning a Kripke structure representing a reactive system. Volpato and Tretmans [127] propose a learning approach which gradually refines an under- and over-approximation of an input-output transition system representing the system under test. The correctness of this approach needs several assumptions, e.g., an oracle indicating when, for some trace, all outputs, which extend the trace to a valid system trace, have been observed.

Sections 6.1, 6.2 and 6.4 present contributions that are primarily developed by myself. Section 6.1 has its origin in [17, 18]. The use cases for conf-cleanliness and sync-conf-cleanliness tests in Section 6.2 were originally published in [46, 20]. Most of the tests conducted on the chassis dynamometer were driven by a professional test lab driver. The contents in Section 6.3 are based on [21, 22]. All co-authors worked together to realise this project. My main contribution in Section 6.3 was the development of the CDP server components in tight cooperation with the development of `LolaDrives` and I conducted five of the six RDE test drives. The falsification powered test cycle selection in Section 6.4 is taken from [25].

7 Conclusion & Future Work

The introduction takes up a sociological definition of trust that is explicitly detached from “the ability to monitor or control” the software manufacturer. This ability, in case of software doping a very technical one, is in the focus of this thesis. It proposes a collection of techniques to give society and users the ability to detect software doping by means of a post-production inspection of software.

7.1 Summary

From a foundational perspective, Chapter 3 proposes cleanness definitions for sequential programs, and reactive, hybrid and mixed-IO systems. The concrete cleanness notions vary in their expressiveness. Strict cleanness is the most strict notion; it enforces that equivalent inputs must lead to identical outputs. Robust cleanness is a generalisation thereof. It assumes distance functions for inputs and outputs and stipulates that similar inputs must lead to similar outputs. Similarity is expressed by threshold parameters that define up to which distance inputs, respectively outputs, are similar. Func-cleanness is a further generalisation that replaces the notion of “similarity” with a function f establishing a direct relation between input distances and output distances. For hybrid systems, the timing of events is explicitly considered. This allows the notion of conf-cleanness, which enforces that conformant inputs must lead to conformant outputs. Hybrid conformance and Skorokhod conformance are two prominent examples from literature that work well with conf-cleanness. A further generalisation is sync-conf-cleanness that requires that inputs conformant w.r.t. a certain retiming lead to outputs that, after applying the input retiming to the outputs, are conformant.

Chapters 4 to 6 use cleanness definitions from Chapter 3 to propose software doping analysis techniques. Chapter 4 proposes two white-box techniques, i.e., techniques that use knowledge about the implementation of a system, for an exhaustive cleanness verification. It depicts how Dijkstra’s weakest conservative precondition approach can be combined with a self-composition technique to prove cleanness of sequential programs. For reactive systems, it gives HyperLTL formulas to prove cleanness by means of model checking.

Black-box analysis techniques are covered in Chapter 5 with a focus on mixed-IO systems. In particular, we target a testing technique for robust cleanness. To capture these tests formally, we build on a model-based testing framework. We discuss challenges of applying it to robust cleanness and, in particular, the impossibility of having test cases that can check for l-robust cleanness. Finally, we get a provably correct u-robust cleanness test algorithm. The testing approach is complemented by HyperSTL and STL characterisations for (u-)robust and func-cleanness, which can be used with probabilistic falsification approaches to find instances of software doping. We outline how the testing and the falsification approach can be combined to an integrated testing approach to make real-world doping tests more effective.

Chapter 6 addresses the Diesel Emissions Scandal. It demonstrates how the model-based testing technique and the falsification technique can be applied to detect tampered emission cleaning system. In a very simple example we show how robust cleanness could have convicted the Volkswagen doping from 2015. With the model-based testing technique we developed the SineNEDC test cycle that convicted a Nissan car of not being robustly clean. Similarly, with an implementation based on a HyperSTL* formalisation of hybrid conformance, we demonstrate that the same Nissan car shows severely increased NO_x emissions under retimed NEDC test cycles.

The chapter also presents the mobile application *LolaDrives* and the Car Data Platform. *LolaDrives* is, when connected to a car via a low-cost OBD-Bluetooth adapter, able to show real-time diagnostics data and to assist Real-Driving Emissions tests; this works without expensive emissions measurement equipment. The app is integrated into the CDP; users can upload their data records as a donation for further research. Our vision is that the CDP becomes a long-term research project and an umbrella term for a multitude of car-related tools and services. One component of the CDP is a car-specific implementation of an integrated CPS testing approach that combines probabilistic falsification and model-based testing.

The white-box approaches in Chapter 4 are intended for software manufacturers that want to verify the absence of software doping during the design phase of their development. Still, it can also be used by third parties (such as researchers, NGOs or activists) if the implementation details are publicly available. Analyses meant explicitly to be carried out by third parties are the black-box approaches in Chapter 5. These approaches reason over the observable behaviour of the system; thus, no cooperation of the manufacturer is necessary. Finally, with *LolaDrives* this thesis also provides a tool for laypersons. *LolaDrives* uses high quality verification techniques (namely RTLOLA), but it hides these techniques from the user and configures the runtime monitor such that the analysis works out of the box.

7.2 Future Work

The contents in this thesis can be developed further in many different directions. The cleanness definitions presented in Chapter 3 cover sequential programs, reactive systems, mixed-IO systems and hybrid systems. Yet, there are many more modelling formalisms, for example, a whole army of automata-based formalisms to model systems that exhibit behaviour that can be captured by probability distributions [9, 69]. There are prominent modelling formalisms that in addition allow to model the timing behaviour of a system [41, 66]. With cleanness definitions for such systems, it may be interesting to strive for a more precise model of the emission cleaning system of cars. The chemical and physical reactions in such systems can presumably be encoded by probability distributions. This would be a significant step towards a white-box model-checking approach for emission cleaning systems. Recall that the examples from Chapter 4 consider toy examples that deliberately ignore the complexity of the chemical processes of the emission cleaning system.

Another step further to a viable white-box verification approach is a precise model of the interplay of the continuous dynamics induced by the chemical and physical reactions, and the discrete nature of the software controlling these reactions by means of injecting the diesel exhaust fluid or actuating valves that control the amount of oxygen that is available for the combustion in the engine. Such hybrid system models [6] (that contain more information about the inner workings of a system than our notion of hybrid systems in Section 3.4) would be also interesting to consider w.r.t. *conf-cleanness* and *sync-conf-cleanness* in Section 3.4. However, hybrid systems are known to be very hard to analyse [105, 68]; further investigations are necessary to evaluate in how far the cleanness notions in Chapter 3 (which are all hyperproperties) can be model-checked on automata-defined hybrid systems.

With new cleanness definitions for additional computation models, it may become necessary to also enhance the testing techniques in Chapter 5 – be it for simulation-based analyses of models or for real-world systems. Also, there is no end-to-end testing approach for *conf-cleanness* and *sync-conf-cleanness*. We demonstrated the effectiveness of these cleanness notions only manually by computing the satisfaction of conformance between input sequences. A promising starting point for this are the HyperSTL* characterisation of hybrid-conformance cleanness and Skorokhod-conformance cleanness that were proposed recently [20].

The Car Data Platform also offers potential for extensions. On the server-side, a fully automatic analysis pipeline could process the data donations provided by LolaDrives users in a privacy preserving way to draw conclusions regarding different car models, driving styles, et cetera. We already started to add on-

device analyses to the LolaDrives app, so that car owners can easily catch key indicators for their cars, like average fuel consumption and emissions. We also started to look into support for electrical vehicles, for which key indicators like power consumption are interesting, and an analysis of how the driving behaviour influences the power consumption to give vehicle specific recommendations about which driving style can positively influence the battery range of the car. Electric vehicles, however, are not forced to support the OBD protocol. Hence, we would need to obtain access to proprietary manufacturer protocols.

The main purpose of the cleaning definitions in Chapter 3 is to formally define a common set of expectations that software users have on software manufacturers. As it turns out, such expectations may also apply to domains other than software. For example, if a company pays higher salaries to men than for women, although there are no differences in work performance between the genders, then robust cleanness is able to catch this deficiency. For an input distance function on employees that takes into account only the work performance, but not the gender, similar inputs (i.e., employees) would yield vastly different outputs (i.e., salaries). I imagine that there are many interdisciplinary use cases where our cleanness definitions can be applied. We are currently working on a first interdisciplinary project that uses func-cleanness to overcome unfair treatment of individuals by decision making algorithms. The next section provides a preview to this ongoing research project.

7.3 Effective Human Oversight with Func-Cleanness

As part of work in progress, software doping has been positioned in an interdisciplinary context to tackle relevant problems regarding unfair treatment of human individuals by partially or fully automated decision making AI systems. Particularly interesting are so-called *high-risk* application areas (where “risk” must be understood with regard to fundamental human rights). Fields of application include credit approval [58], decisions on visa applications [113], university admissions [130, 85], screening of individuals in predictive policing [136], selection in HR [29, 98, 99], judicial decisions (as with COMPAS [33, 50, 80, 76]), tenant screening [53], and more. In many of these areas, there are legitimate interests and valid reasons for using such models, although the risks associated with their use are manifold.

It is widely recognized that discrimination by unfair models is one particularly important risk. As a result, a colorful zoo of different operationalizations of unfairness has emerged [128, 100], which should be seen less as a set of competing approaches and more as mutually complementary [60]. At the same time, a

consensus is emerging that human oversight is an important piece of the puzzle for mitigating and minimizing societal risks of AI [71, 91, 125]. Accordingly, that idea made it into recent drafts of legislation like the EU proposal for an AI Act [39] or certain US state laws [129].

In the following, I will give a preview of current work in progress on a joint project with Kevin Baum, Sarah Sterz, Sven Hetmank, Markus Langer, Anne Lauber-Rönsberg, Franz Lehr and Holger Hermanns. This project aims at promoting effective human oversight with runtime fairness monitoring and is highly interdisciplinary with contributions from philosophy, psychology, law and computer science. I will focus on the computer science aspects, as this is my contribution to this project.

Our claim is that effective human oversight can and should help with regard to the risk of discriminatory systems. We want to show how an established operationalisation of individual unfairness [52] can be suitably generalised by func-cleanness. Furthermore, the probabilistic falsification approach from Section 2.6.2 and the STL characterisation of u-func-cleanness from Corollary 5.46 (which for deterministic systems characterises func-cleanness) can yield a local, model-agnostic Explainable AI (XAI) method that enables human overseers to meet their responsibilities. In other words, we outline a runtime fairness monitor that promotes effective human oversight.

We illustrate the challenge that our work helps to overcome by the following example of a hypothetical university admission system (inspired by [130, 85]).

Example 7.1. A large university assigns scores to those who apply to their computer science PhD program using an automated, model-based procedure P based on three data points: the position of the applicant’s last graduate institution in an official, subject-specific ranking, the applicant’s last grade point average (GPA), and their score in a subject-specific standardised test taken as part of the application procedure. The system then scores candidates based on how successful it expects them to be as students. A dedicated university employee, Unica, supervises P as a human in the loop and is supposed to detect when the output of P is flawed. The university pays especial attention to fairness in the procedure, so she has to watch out to any signs of potential unfairness. Unica is supposed to desk-reject candidates who’s scores are below a certain, predefined threshold – unless she finds problems with P ’s scoring.

Without any additional tools, Unica, as a human in the loop, must manually check all system outputs for signs of unfairness. This can be a tedious, complicated and error-prone task and runs counter to the scalability of the overall process. Therefore, she at least needs a tool that helps her to detect when something is off about the scoring of individual applicants. For example, a participant who is scored poorly with a GPA of 2.5, but receives a much higher score

when they are assumed to have a GPA of 2.4 should raise red flags. This has to happen at runtime, since Unica needs to make a timely decision on whether to include the applicant in further considerations. Only then she can exercise effective human oversight. Our approach describes technical measures that help in mitigating this challenge by providing her with information from an individual fairness analysis in a suitable, purposeful, expedient way. To this end, we propose a formal definition for individual fairness based on func-cleanness and develop a runtime monitor that analyses every output of P immediately after P 's decision. It strategically searches for unfair treatment of a particular individual by comparing them to relevant hypothetical alternative individuals so as to provide a fairness assessment in a timely manner.

AI systems – in the broadest sense of the word – more and more often support human decision makers. Undoubtedly, such systems should be compliant with applicable law (such as the future European AI Act [39] or the Washington State facial recognition law [129]) and ought to minimize any risks to health, safety or fundamental rights. Sometimes, we cannot mitigate all these risks in advance by technical measures and also some risk-mitigation requires trade-off decisions involving features that are either impossible or difficult to operationalise and formalise. This is why it is essential that a human effectively oversees the system (which is also emphasised by several institutions such as UNESCO [125] and the European High Level Expert Group [71]). *Effective* human oversight, however, is only possible with the appropriate technical measures that allow human overseers to better understand the system at runtime [83]. From a technical point of view, this raises the pressing question of what such technical measures can and ought to look like to actually enable humans to live up to these responsibilities. Our contribution is intended as bridging the gap between the normative expectations of law and society and the current reality of technological design.

7.3.1 Individual Fairness

Unica from Example 7.1 should be able to detect individual unfairness. An operationalisation thereof by Dwork et al. [52] is based on the Lipschitz condition to enforce that similar individuals are treated similarly. To measure similarity, they assume the existence of an input distance function d_{in} and an output distance function d_{out} . This assumption is very similar to the one that we implicitly made in the previous sections for robust cleanness and func-cleanness. However, in the case of the fair treatment of humans finding reasonable distance functions is more challenging than it was for the examples in the previous chapters. Dwork et al. assume that both distance functions perfectly measure distances between individuals and between outputs of the system, respectively, but admit that in practice these distance functions are only approximations of a ground truth at

best. They suggest that distance measures might be learned, but there is no one-size-fits-all approach to selecting distance measures. Indeed, obtaining such distance metrics is a topic of active research [138, 94, 74]. Lastly, the Lipschitz condition assumes a Lipschitz constant L to establish a linear constraint between input and output distances.

Definition 7.2. A deterministic sequential program $P : \text{In} \rightarrow \text{Out}$ is *Lipschitz-fair* w.r.t. $d_{\text{In}} : \text{In} \times \text{In} \rightarrow \mathbb{R}$, $d_{\text{Out}} : \text{Out} \times \text{Out} \rightarrow \mathbb{R}$ and a Lipschitz constant L , if and only if for all $i_1, i_2 \in \text{In}$, $d_{\text{Out}}(P(i_1), P(i_2)) \leq L \cdot d_{\text{In}}(i_1, i_2)$.

Lipschitz-fairness comes with some restrictions that limit its suitability for practical application:

$d_{\text{In}}-d_{\text{Out}}$ -relation: High-risk systems are typically complex systems and ask for more complex fairness constraints than the linearly bounded output distances provided by the Lipschitz condition. For example, using the Lipschitz condition prevents us from allowing small local jumps in the output and at the same time forbidding jumps of the same rate of increase over larger ranges of the input space.

Input relevance: The condition quantifies over the entire input domain of a program, without scrutinising whether each input in such domain could plausibly represent a real-world individual. But whether a system is unfair for a purely hypothetical input compared to another possibly purely hypothetical input is largely irrelevant in practice. What is practically important is the ability to determine whether actual applicants are disadvantaged.

Monitorability: In a monitoring scenario with the Lipschitz condition in place, a fixed input i_1 must be compared to potentially all other inputs i_2 . Since the input domain of the system can be arbitrarily large, the Lipschitz condition is not yet suitable for monitoring in practice (for a related point see John et al. [77]).

We propose a notion of individual fairness that is based on func-cleanness. Instead of cleanness contracts we consider here *fairness contracts*, which are tuples $\mathcal{F} = \langle d_{\text{In}}, d_{\text{Out}}, f \rangle$ containing input and output distance functions and the function f relating input distances and output distances. Notably, the set of standard inputs StdIn known from cleanness contracts is not part of a fairness contract; it is unknown what qualifies an input to be ‘standard’ in the context of fairness analyses. Still, our fairness definition evaluates fairness for a set of individuals \mathcal{I} , which has conceptual similarities to the set StdIn . A fairness

contract is an encoding of what fairness means for a concrete context or situation. Such a fairness encoding must apply equally to every individual, thus, the set \mathcal{I} must not be part of it.

Definition 7.3. A deterministic sequential program $P : \text{In} \rightarrow \text{Out}$ is *func-fair* for a set $\mathcal{I} \subseteq \text{In}$ of individuals w.r.t. a fairness contract $\mathcal{F} = \langle d_{\text{In}}, d_{\text{Out}}, f \rangle$, if and only if for every $i \in \mathcal{I}$ and $i' \in \text{In}$, $d_{\text{Out}}(P(i), P(i')) \leq f(d_{\text{In}}(i, i'))$.

The idea behind func-fairness is that every individual in set \mathcal{I} is compared to potential other inputs in the domain of P . These other inputs do not necessarily need to be in \mathcal{I} , nor do these inputs need to have “physical counterparts” in the real world. Driven by the insights of the **Input relevance** restriction of Lipschitz-fairness, we explicitly distinguish inputs in the following and will call inputs that are given to P by a user *actual inputs*, denoted i_a , and call inputs to which such i_a are compared to *synthetic inputs*, denoted i_s . Actual inputs are typically inputs that have a real-world counterpart in most use cases, while this might or might not be true for synthetic inputs. An alternative to using synthetic inputs is to use only actual inputs, i.e., to compare every actual input with every other actual input in \mathcal{I} . For example, for a university admission, all applicants could be compared to every other applicant. However, this would heavily rely on contingencies: the detection of unfair treatment of an applicant depends on whether they were lucky enough that, coincidentally, another candidate has also applied who aids in unveiling the system’s unfairness towards them. Func-fairness prefers to over-approximate the set of plausible inputs that actual inputs are compared to rather than under-approximating it by comparing only to other inputs in \mathcal{I} .

Notice that func-fairness is a conservative extension of the Lipschitz condition. With $\mathcal{I} = \text{In}$ and $f(x) = L \cdot x$, func-fairness mimics Lipschitz-fairness. Wachter et al. [128] classify the fairness-by-awareness approach of Dwork et al. [52] as bias-transforming. As we generalise this and introduce no element that has to be regarded as bias-preserving, our approach arguably is bias-transforming, too.

Func-fairness, with its function f , provides a powerful tool to model complex fairness constraints. How such an f is defined has profound impact on the quality of the fairness analysis. A full discussion about which types of functions make a good f go beyond the scope of this preview. A suitable choice for f and the distance functions d_{In} and d_{Out} heavily depends on the context in which fairness is analysed – there is no one-fits-it-all solution. Func-fairness makes this explicit with the formal fairness contract $\mathcal{F} = \langle d_{\text{In}}, d_{\text{Out}}, f \rangle$.

Algorithm 7.1 FairnessMonitor,with ξ -min $S = (\xi, i_1, i_2)$ only if $(\xi, i_1, i_2) \in S$ and for all $(\xi', i'_1, i'_2) \in S, \xi' \geq \xi$ **Falsification Parameters:** PS: Proposal scheme, β : Temperature parameter**Input:** System $P : \text{In} \rightarrow \text{Out}$, Fairness contract $\mathcal{F} = \langle d_{\text{In}}, d_{\text{Out}}, f \rangle$ and set of actual inputs \mathcal{I} **Output:** A minimal fairness score triple from $\mathbb{R} \times \mathcal{I} \times \text{In}$.

```

1:  $i_s \leftarrow$  any input  $i_a \in \mathcal{I}$ 
2:  $(\xi, i_{\min}, i_s) \leftarrow \xi\text{-min}\{F(i_a, i_s), i_a, i_s \mid i_a \in \mathcal{I}\}$ 
3:  $(\xi_{\min}, i_1, i_2) \leftarrow (\xi, i_{\min}, i_s)$ 
4: while not timeout do
5:    $i'_s \leftarrow \text{PS}(i_s, P(i_s))$ 
6:    $(\xi', i'_{\min}, i'_s) \leftarrow \xi\text{-min}\{F(i_a, i'_s), i_a, i'_s \mid i_a \in \mathcal{I}\}$ 
7:    $(\xi_{\min}, i_1, i_2) \leftarrow \xi\text{-min}\{(\xi_{\min}, i_1, i_2), (\xi', i'_{\min}, i'_s)\}$ 
8:    $\alpha \leftarrow \exp(-\beta(\xi' - \xi))$ 
9:    $r \leftarrow \text{UniformRandomReal}(0, 1)$ 
10:  if  $r \leq \alpha$  then
11:     $i_s \leftarrow i'_s$ 
12:     $\xi \leftarrow \xi'$ 
13:  end if
14: end while
15: return  $(\xi_{\min}, i_1, i_2)$ 

```

7.3.2 Fairness Monitoring

We develop a probabilistic-falsification-based fairness monitor that, given a set of actual inputs, searches for a synthetic counterexample to falsify a system P w.r.t. a fairness contract \mathcal{F} . To this end, it is necessary to provide a quantitative description of func-fairness that satisfies the characteristics of a robustness estimate. We call this description *fairness score*. For an actual input i_a and a synthetic input i_s we define the fairness score as $F(i_a, i_s) := f(d_{\text{In}}(i_a, i_s)) - d_{\text{Out}}(P(i_a), P(i_s))$. F is indeed a robustness estimate function: if $F(i_a, i_s)$ is non-negative, then $d_{\text{Out}}(P(i_a), P(i_s)) \leq f(d_{\text{In}}(i_a, i_s))$, and if it is negative, then $d_{\text{Out}}(P(i_a), P(i_s)) \not\leq f(d_{\text{In}}(i_a, i_s))$. For a set of actual inputs \mathcal{I} , the definition generalises to $F(\mathcal{I}, i_s) = \min\{F(i_a, i_s) \mid i_a \in \mathcal{I}\}$, i.e., the overall fairness score is the minimum of the concrete fairness scores of the inputs in \mathcal{I} . Notice that $\mathcal{R}_{\mathcal{I}}(i_s) := F(\mathcal{I}, i_s)$ is essentially the quantitative interpretation of $\varphi_{\text{u-func}}$ (from Proposition 5.47) after simplifications attributed to the fact that P is a sequential and deterministic program.

Algorithm 7.1 shows FairnessMonitor, which builds on Algorithm 2.1 to search for the minimal fairness score in a system P for fairness contract \mathcal{F} . The algo-

rithm stores fairness scores in triples that also contain the two inputs for which the fairness score was computed. The minimum in a set of such triples is defined by the function ξ -min that returns the triple with the smallest fairness score of all triples in the set. The first line of `FairnessMonitor` initialises the variable i_s with an arbitrary actual input from \mathcal{I} . For this value of i_s , the algorithm checks the corresponding fairness scores for all actual inputs $i_a \in \mathcal{I}$ and stores the smallest one. In line 3, the globally smallest fairness score triple is initialised. In line 5 it uses the proposal scheme to get the next synthetic input i'_s . Line 6 is similar to line 2: for the newly proposed i'_s it finds the smallest fairness scores, stores it, and updates the global minimum if it found a smaller fairness score (line 7). Lines 8-13 come from Algorithm 2.1. The only difference is that in addition to i_s we also store the fairness score ξ . Line 4 of Algorithm 7.1 differs from Algorithm 2.1 by terminating the falsification process after a timeout occurs. Hence, the algorithm does not (exclusively) aim to falsify the fairness property, but aims at minimising the fairness score; even if the fair treatment of the inputs in \mathcal{I} cannot be falsified in a reasonable amount of time, we still learn how robustly they are treated fairly, i.e., how far the least fairly treated individual in \mathcal{I} is away from being treated unfairly. After the timeout occurs, the algorithm returns the triple with the overall smallest seen fairness score ξ_{\min} , together with the actual input i_1 and the synthetic input i_2 for which ξ_{\min} was found. In case ξ_{\min} is negative, i_2 is a counterexample for P being func-fair.

`FairnessMonitor` implements a sound \mathcal{F} -unfairness detection as stated in Proposition 7.4. However, it is not complete, i.e., it is not generally the case that P is func-fair for \mathcal{I} if ξ is positive. It may happen that there is a counterexample, but `FairnessMonitor` did not succeed in finding it before the timeout. This is analogue to the results of the model-agnostic robust cleanness analysis in Chapter 5.

Proposition 7.4. Let $P : \text{In} \rightarrow \text{Out}$ be a sequential program, $\mathcal{F} = \langle d_{\text{In}}, d_{\text{Out}}, f \rangle$ a fairness contract and \mathcal{I} a set of actual inputs. Further, let (ξ_{\min}, i_1, i_2) be the result of `FairnessMonitor`($P, \mathcal{F}, \mathcal{I}$). If ξ_{\min} is negative, then P is not func-fair for \mathcal{I} w.r.t. \mathcal{F} .

Moreover, `FairnessMonitor` circumvents major restrictions of the Lipschitz-fairness:

$d_{\text{In}}-d_{\text{Out}}$ -relation: Func-fairness defines constraints between input and output distances by means of a function f , which allows to express also complex fairness constraints.

Input relevance: Func-fairness explicitly distinguishes between actual and synthetic inputs. This way, func-fairness acknowledges a possible obstacle of the fairness theory when it comes to a real-world usage of the analysis.

Algorithm 7.2 FairnessAwareSystem**Parameters:** System $P : \text{In} \rightarrow \text{Out}$, Fairness contract $\mathcal{F} = \langle d_{\text{In}}, d_{\text{Out}}, f \rangle$ **Input:** Input $i_a \in \text{In}$ **Output:** Tuple of the system output, normalized fairness score, and synthetic values witnessing the fairness score

- 1: $(\xi_{\min}, i_a, i_s) \leftarrow \text{FairnessMonitor}(P, \mathcal{F}, \{i_a\})$
- 2: **return** $(P(i_a), \xi_{\min} \div f(d_{\text{In}}(i_a, i_s)), (i_s, P(i_s)))$

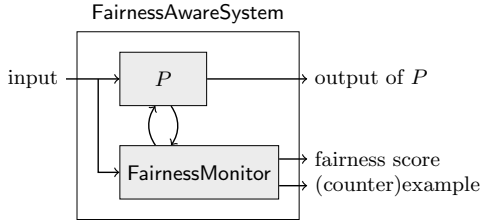


Figure 7.1: Schematic visualisation of FairnessAwareSystem

Monitorability: FairnessMonitor demonstrates that func-fairness is monitorable.

It resolves the quantification over In using concepts from probabilistic falsification using the robustness estimate function F as defined above.

Towards func-fairness in the loop If a high-risk system is in operation, a human in the loop must oversee the correct and fair functioning of the outputs of the system. To do this, the human needs real-time fairness information. Figure 7.1 shows how this can be achieved by coupling the system P and the FairnessMonitor in Algorithm 7.1 in a new system called FairnessAwareSystem. This system is sketched in Algorithm 7.2. Intuitively, FairnessAwareSystem is a higher-order program that is parameterised with the original program P and the fairness contract \mathcal{F} . When instantiated with these parameters, the program takes arbitrary (actual) inputs i_a from In . In the first step, it does a fairness analysis using FairnessMonitor with arguments P , \mathcal{F} and $\{i_a\}$. To make fairness scores comparable, FairnessAwareSystem normalises the fairness score ξ received from FairnessMonitor by dividing¹ it by the output distance limit $f(d_{\text{In}}(i_a, i_s))$. For fair outputs, the score will be between 0 (almost unfair) and 1 (as fair as

¹For f that can return 0, there may be a $0 \div 0$ division. The result of this division should be defined depending on the concrete context; reasonable values range from the extreme scores 0 (to indicate that the score is on the edge to becoming ‘unfair’) to 1 (to indicate that more fairness is impossible).

possible). Outputs that are not func-fair are accompanied by a negative score representing how much the limit $f(d_{\ln}(i_a, i_s))$ is exceeded. A fairness score of $-n$ means that the output distance of $P(i_a)$ and $P(i_s)$ is $n + 1$ times as high as that limit. Finally, `FairnessAwareSystem` returns the triple with P 's output for i_a , the normalised fairness score, and the synthetic input with its output witnessing the fairness score.

Interpretation of monitoring results Especially when `FairnessAwareSystem` finds a violation of func-fairness, the suitable interpretation and appropriate response to the normalised fairness score proves to be a non-trivial matter that requires expertise.

Example 7.5. Instead of using P from Example 7.1 on its own, Unica now uses `FairnessAwareSystem` and thereby receive a fairness score along with P 's verdict on each applicant. If the fairness score is negative, she can also take into account the information on the synthetic counterpart returned by `FairnessAwareSystem`. Among the 4096 applicants for the PhD program, the monitoring assigns a negative fairness score to three candidates: Alexa, who received a low score, Eugene, who was scored very highly, and John, who got an average score. According to their scoring, Alexa would be desk-rejected, while Eugene and John would be considered further. Alexa's synthetic counterpart, let's call him Syntbad, is ranked much higher than Alexa. In fact, he is ranked so high that Syntbad would not be desk-rejected. Unica compares Alexa and Syntbad and finds that they only differ in one respect: Syntbad's graduate university is the one in the official ranking that is immediately *below* the one that Alexa attended. Unica does some research and finds that Alexa's institution is predominantly attended by People of Color, while this is not the case for Syntbad's institution. Therefore, `FairnessAwareSystem` helped Unica not only to find an unfair treatment of Alexa, but also to uncover a case of potential racial discrimination. John's counterpart, Synclair, is ranked much lower than him. Unica manually inspects John's previous institution (an infamous online university), his GPA of 1.8, and his test result with only 13%. She finds that this very much suggests that John will not be a successful PhD candidate and desk-rejects him. Therefore, Unica has successfully used `FairnessAwareSystem` to detect a fault in scoring system P whereby John would have been treated unfairly in a way that would have been to his advantage. Eugene received a top score, but his synthetic counterpart, Syna, received only an average one. Unica suspects that Eugene was ranked too highly given his graduate institution, GPA, and test score. However, as he would not have been desk-rejected either way, nothing changes for Eugene, and the unfairness he was subject to, is not of effect to him. The cases of John and Eugene share similarities with the configuration in (b) in Figure 7.2, the one of

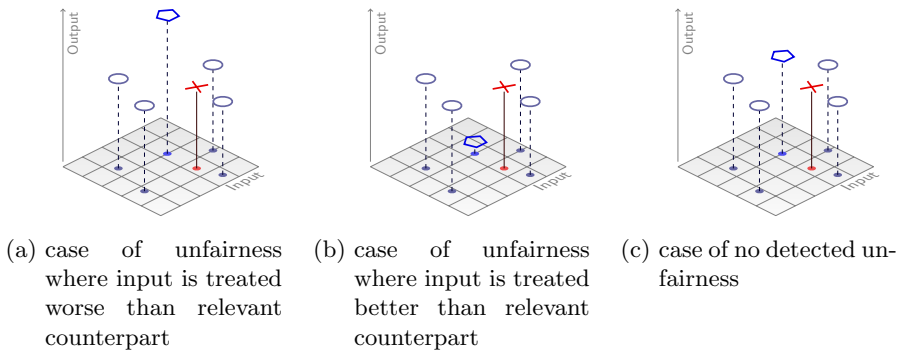


Figure 7.2: Exemplary illustration of configurations of an input (red cross) and its synthetic counterparts (grey circles) and the synthetic counterpart with the minimal fairness score (blue polygon); with a two-dimensional input space (grid) and a one-dimensional output.

Alexa with (a), and the ones of all other 4093 candidates with (c).

If our monitor finds only a few problematic cases in a (sufficiently large and diverse) set of inputs, our monitoring helps Unica from our running example by drawing her attention to cases that require special attention. Thereby, individuals who are judged by the system have a better chance of being treated fairly, since even rare instances of unfair treatment are detected. If, on the other hand, the number of problematic cases found is large, or Unica finds especially concerning cases or patterns, this can point to larger issues within the system. In these cases, Unica should take appropriate steps and make sure that the system is no longer used until clarity is established why so many violations are found. If the system is found to be systematically unfair, it should arguably be removed from the decision process. A possible conclusion could also be that the system is unsuitable for certain use cases, e.g., for the use on individuals from a particular group. Accordingly, it might not have to be removed altogether but only needs to be restricted such that problematic use cases are avoided. In any case, significant findings should also be fed back to developers or operators of the potentially problematic system. A fairness monitoring such as in *FairnessAwareSystem* or a fairness analysis as in *FairnessMonitor* could also be useful to developers, regulating authorities, watchdog organisations, or forensic analysts as it helps them to check the individual fairness of a system in a controlled environment.

As mentioned in the beginning, the contents in this section are work in progress. Still, it demonstrates how our cleanness definitions also serve for pur-

poses other than the detection of software doping (in the original sense presuming some intentionality by the software manufacturer). AI systems, in particular deep neural networks, are trustworthy only to a very limited extent. Verification of such systems is an active research topic, but their complexity makes a comprehensive verification difficult. I advertised the cleanness definitions in this thesis as an ability to control and to monitor systems that we do not trust. AI systems in high-risk application areas are systems that we must not trust, and this section shows that func-cleanness is ready to exercise control as promised.

Bibliography

- [1] LolaDrives web page. <https://loladrives.app>.
- [2] Houssam Abbas, Georgios E. Fainekos, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Trans. Embed. Comput. Syst.*, 12(2s):95:1–95:30, 2013. doi:10.1145/2465787.2465797.
- [3] Kate Abnett and Victoria Waldersee. VW defeat devices were illegal in certain conditions, EU’s top court says, 2022. Online; accessed: 2022-09-21. URL: <https://www.reuters.com/business/sustainable-business/eus-top-court-says-vw-car-emissions-defeat-devices-were-illegal-2022-07-14/>.
- [4] A. Aerts, M. Reniers, and M.R. Mousavi. Chapter 19 – Model-based testing of cyber-physical systems. In Houbing Song, Danda B. Rawat, Sabina Jeschke, and Christian Brecher, editors, *Cyber-Physical Systems, Intelligent Data-Centric Systems*, pages 287 – 304. Academic Press, Boston, 2017. doi:10.1016/B978-0-12-803801-7.00019-5.
- [5] Riham Alkousaa. German city of stuttgart bans older diesel vehicles from April 1, 2019. Online; accessed: 2022-09-21. URL: <https://www.reuters.com/article/us-germany-emissions-stuttgart-idUSKCN1R91P8>.
- [6] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995. doi:10.1016/0304-3975(94)00202-T.
- [7] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 390–401. IEEE Computer Society, 1990. doi:10.1109/LICS.1990.113764.

- [8] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*, pages 254–257. Springer, 2011. doi:10.1007/978-3-642-19835-9_21.
- [9] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.
- [10] Gilles Barthe, Juan Manuel Crespo, and Cesar Kunz. Relational verification using product programs. In Michael J. Butler and Wolfram Schulte, editors, *FM 2011*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011. doi:10.1007/978-3-642-21437-0_17.
- [11] Gilles Barthe, Pedro R. D’Argenio, Bernd Finkbeiner, and Holger Hermanns. Facets of software doping. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, volume 9953 of *Lecture Notes in Computer Science*, pages 601–608, 2016. doi:10.1007/978-3-319-47169-3_46.
- [12] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011. doi:10.1017/S0960129511000193.
- [13] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 135–175. Springer, 2018. doi:10.1007/978-3-319-75632-5_5.
- [14] Kevin Baum. What the hack is wrong with software doping? In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, vol-

- ume 9953 of *Lecture Notes in Computer Science*, pages 633–647, 2016. doi:10.1007/978-3-319-47169-3_49.
- [15] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. RTLOLA cleared for take-off: Monitoring autonomous aircraft. In *CAV 2020*, volume 12225 of *LNCS*, pages 28–39. Springer, 2020. doi:10.1007/978-3-030-53291-8_3.
- [16] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *POPL’04*, pages 14–25. ACM Press, 2004. doi:10.1145/964001.964003.
- [17] Sebastian Biewer, Pedro R. D’Argenio, and Holger Hermanns. Doping tests for cyber-physical systems. In David Parker and Verena Wolf, editors, *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings*, volume 11785 of *Lecture Notes in Computer Science*, pages 313–331. Springer, 2019. doi:10.1007/978-3-030-30281-8_18.
- [18] Sebastian Biewer, Pedro R. D’Argenio, and Holger Hermanns. Doping tests for cyber-physical systems. *ACM Trans. Model. Comput. Simul.*, 31(3):16:1–16:27, 2021. doi:10.1145/3449354.
- [19] Sebastian Biewer, Pedro R. D’Argenio, and Holger Hermanns. Doping tests for cyber-physical systems – Tool, April 2021. doi:10.5281/zenodo.4709389.
- [20] Sebastian Biewer, Rayna Dimitrova, Michael Fries, Maciej Gazda, Thomas Heinze, Holger Hermanns, and Mohammad Reza Mousavi. Conformance relations and hyperproperties for doping detection in time and space. *Log. Methods Comput. Sci.*, 18(1), 2022. doi:10.46298/lmcs-18(1:14)2022.
- [21] Sebastian Biewer, Bernd Finkbeiner, Holger Hermanns, Maximilian A. Köhl, Yannik Schnitzer, and Maximilian Schwenger. RTLOLA on board: Testing real driving emissions on your phone. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 365–372. Springer, 2021. doi:10.1007/978-3-030-72013-1_20.

- [22] Sebastian Biewer, Bernd Finkbeiner, Holger Hermanns, Maximilian A. Köhl, Yannik Schnitzer, and Maximilian Schwenger. On the road with RTLOLA: Testing real driving emissions on your phone. *Int. J. Softw. Tools Technol. Transf.*, 2023. doi:10.1007/s10009-022-00689-5.
- [23] Sebastian Biewer, Michael Fries, and Thomas Heinze. Conformance relations and hyperproperties for doping detection in time and space (supplementary material). <https://www.powver.org/publications/conformance-based-doping-detection>, 2020.
- [24] Sebastian Biewer and Holger Hermanns. LolaDrives (App) Privacy Policy. URL: <https://www.loladrives.app/app-privacy-statement/>.
- [25] Sebastian Biewer and Holger Hermanns. On the detection of doped software by falsification. In Einar Broch Johnsen and Manuel Wimmer, editors, *Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13241 of *Lecture Notes in Computer Science*, pages 71–91. Springer, 2022. doi:10.1007/978-3-030-99429-7_4.
- [26] Sebastian Biewer and Yannik Schnitzer. PCDF analyser, September 2021. URL: <https://github.com/udsdepend/pcdf-analyser>.
- [27] Lubos Brim, Petr Dluhos, David Safranek, and Tomas Vejpustek. STL*: Extending signal temporal logic with signal-value freezing operator. *Inf. Comput.*, 236:52–67, 2014. doi:10.1016/j.ic.2014.01.012.
- [28] Cadence System Analysis. The importance of battery cooling systems in electric vehicles, 2022. Online; accessed: 2022-09-21. URL: <https://resources.system-analysis.cadence.com/blog/msa2022-the-importance-of-battery-cooling-systems-in-electric-vehicles>.
- [29] Cathy O’Neil. How algorithms rule our working lives, 2016. Online; accessed: 2023-02-03. URL: <https://www.theguardian.com/science/2016/sep/01/how-algorithms-rule-our-working-lives>.
- [30] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 57–70. ACM, 2010. doi:10.1145/1706299.1706308.

- [31] Stuart Cheshire and Mary Baker. Consistent overhead byte stuffing. In Christophe Diot, Christian Huitema, Scott Shenker, and Martha Steenstrup, editors, *Proceedings of the ACM SIGCOMM 1997 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, September 14-18, 1997, Cannes, France*, pages 209–220. ACM, 1997. doi:10.1145/263105.263168.
- [32] Siddhartha Chib and Edward Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995. doi:10.1080/00031305.1995.10476177.
- [33] Alexandra Chouldechova. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big Data*, 5(2):153–163, 2017. doi:10.1089/big.2016.0047.
- [34] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.
- [35] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *CSF’08*, pages 51–65, 2008. doi:10.1109/CSF.2008.7.
- [36] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013. doi:10.1109/TSE.2012.86.
- [37] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. The hierarchy of hyperlogics. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785713.
- [38] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 121–139. Springer, 2019. doi:10.1007/978-3-030-25540-4_7.

- [39] European Commission. Laying down harmonised rules on artificial intelligence (artificial intelligence act) and amending certain union legislative acts (proposal for a regulation) no 0106/2021, 2020. <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52021PC0206>.
- [40] Moritz Contag, Guo Li, Andre Pawlowski, Felix Domke, Kirill Levchenko, Thorsten Holz, and Stefan Savage. How they did it: An analysis of emission defeat devices in modern automobiles. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 231–250. IEEE Computer Society, 2017. doi:10.1109/SP.2017.66.
- [41] Pedro R. D’Argenio. *Algebras and Automata for Timed and Stochastic Systems*. PhD thesis, Netherlands, 1999.
- [42] Pedro R. D’Argenio, Gilles Barthe, Sebastian Biewer, Bernd Finkbeiner, and Holger Hermanns. Is your software on dope? – Formal analysis of surreptitiously “enhanced” programs. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 83–110. Springer, 2017. doi:10.1007/978-3-662-54434-1_4.
- [43] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using SPIN. *STTT*, 2(4):382–393, 2000. doi:10.1007/s100090050044.
- [44] Deutsche Forschungsgemeinschaft / German Research Foundation. Guidelines for safeguarding good research practice – code of conduct. URL: https://www.dfg.de/download/pdf/foerderung/rechtliche_rahmenbedingungen/gute_wissenschaftliche_praxis/kodex_gwp_en.pdf.
- [45] Edsger W. Dijkstra. *A discipline of programming*. Prentice Hall series in automatic computation. Prentice Hall, Englewood Cliffs, NJ, 1976.
- [46] Rayna Dimitrova, Maciej Gazda, Mohammad Reza Mousavi, Sebastian Biewer, and Holger Hermanns. Conformance-based doping detection for cyber-physical systems. In Alexey Gotsman and Ana Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12136 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 2020. doi:10.1007/978-3-030-50086-3_4.

- [47] Felix Domke and Daniel Lange. The exhaust emissions scandal (“Dieselgate”). In *30th Chaos Communication Congress*, 2015. Online; accessed: 2022-09-20. URL: <https://events.ccc.de/congress/2015/Fahrplan/events/7331.html>.
- [48] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for STL. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 264–279. Springer, 2013. doi:10.1007/978-3-642-39799-8_19.
- [49] Laurent Doyen, Thomas A. Henzinger, Axel Legay, and Dejan Nickovic. Robustness of sequential circuits. In *10th International Conference on Application of Concurrency to System Design, ACSD 2010, Braga, Portugal, 21-25 June 2010*, pages 77–84. IEEE Computer Society, 2010. doi:10.1109/ACSD.2010.26.
- [50] Julia Dressel and Hany Farid. The accuracy, fairness, and limits of predicting recidivism. *Science Advances*, 4(1), 2018. doi:10.1126/sciadv.aao5580.
- [51] Joe Dunn. Diesel emissions scandal: Fiat under investigation. The Telegraph, <http://www.telegraph.co.uk/cars/news/diesel-emissions-scandal-fiat-under-investigation/>, 2017. Online; accessed: 2023-02-03.
- [52] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*, pages 214–226, 2012. doi:10.1145/2090236.2090255.
- [53] Erin Smith, and Heather Vogell. How Your Shadow Credit Score Could Decide Whether You Get an Apartment , 2022. Online; accessed: 2023-02-06. URL: <https://www.propublica.org/article/how-your-shadow-credit-score-could-decide-whether-you-get-an-apartment>.
- [54] Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theor. Comput. Sci.*, 410(42):4262–4291, 2009. doi:10.1016/j.tcs.2009.06.021.
- [55] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. Stream-LAB: Stream-based Monitoring of Cyber-Physical Systems. In *CAV 2019*,

- volume 11561 of *LNCS*, pages 421–431. Springer, 2019. doi:10.1007/978-3-030-25540-4_24.
- [56] Bernd Finkbeiner and Christopher Hahn. Deciding Hyperproperties. In Josée Desharnais and Radha Jagadeesan, editors, *CONCUR 2016*, volume 59 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.CONCUR.2016.13.
- [57] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2015. doi:10.1007/978-3-319-21690-4_3.
- [58] Organisation for Economic Co-operation and Development (OECD). Artificial intelligence, machine learning and big data in finance: Opportunities, challenges and implications for policy makers, 2021. Online; accessed: 2023-02-03. URL: <https://www.oecd.org/finance/financial-markets/Artificial-intelligence-machine-learning-big-data-in-finance.pdf>.
- [59] Fortune Business Insights. Smartwatch market size, share & covid-19 impact analysis, 2022. Online; accessed: 2022-09-21. URL: <https://www.fortunebusinessinsights.com/smartwatch-market-106625>.
- [60] Sorelle A. Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. The (im)possibility of fairness: Different value systems require different mechanisms for fair decision making. *Commun. ACM*, 64(4):136–143, mar 2021. doi:10.1145/3433949.
- [61] Maciej Gazda and Mohammad Reza Mousavi. Logical characterisation of hybrid conformance. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 130:1–130:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.130.
- [62] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*

- 2004, Venice, Italy, January 14-16, 2004, pages 186–197. ACM, 2004. doi:10.1145/964001.964017.
- [63] Alexander Graf-Brill and Holger Hermanns. Model-based testing for asynchronous systems. In Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti, editors, *Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18-20, 2017, Proceedings*, volume 10471 of *Lecture Notes in Computer Science*, pages 66–82. Springer, 2017. doi:10.1007/978-3-319-67113-0_5.
- [64] Michael Grieves and John Vickers. Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. *Transdisciplinary perspectives on complex systems: New findings and approaches*, pages 85–113, 2017. doi:10.1007/978-3-319-38756-7_4.
- [65] Dick Hamlet. Continuity in software systems. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSA 2002, Roma, Italy, July 22-24, 2002*, pages 196–200. ACM, 2002. doi:10.1145/566172.566203.
- [66] Arnd Hartmanns. *On the analysis of stochastic timed systems*. PhD thesis, Saarland University, 2015. URL: <http://scidok.sulb.uni-saarland.de/volltexte/2015/6054/>.
- [67] Les Hatton and Michiel van Genuchten. When software crosses a line. *IEEE Software*, 33(1):29–31, 2016. doi:10.1109/MS.2016.6.
- [68] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 278–292. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561342.
- [69] Holger Hermanns. *Interactive Markov Chains*. PhD thesis, Universität Erlangen-Nürnberg, 1998.
- [70] Holger Hermanns, Sebastian Biewer, Pedro R. D’Argenio, and Maximilian A. Köhl. Verification, testing, and runtime monitoring of automotive exhaust emissions. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 1–17. EasyChair, 2018. doi:10.29007/6zxt.

- [71] High-Level Expert Group on Artificial Intelligence. Ethics Guidelines for Trustworthy AI, 2019. URL: <https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai>.
- [72] Sebastian Holler, Sebastian Biewer, and Clara Schneidewind. HoRStify: Sound security analysis of smart contracts. In *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 9-13, 2023*, 2023. to appear; preprint available at <https://arxiv.org/abs/2301.13769>.
- [73] William Hurst, Madjid Merabti, and Paul Fergus. A survey of critical infrastructure security. In Jonathan Butts and Sujeet Sheno, editors, *Critical Infrastructure Protection VIII - 8th IFIP WG 11.10 International Conference, ICCIP 2014, Arlington, VA, USA, March 17-19, 2014, Revised Selected Papers*, volume 441 of *IFIP Advances in Information and Communication Technology*, pages 127–138. Springer, 2014. doi:10.1007/978-3-662-45355-1_9.
- [74] Christina Ilvento. Metric learning for individual fairness. 2019. doi:10.48550/ARXIV.1906.00250.
- [75] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005. doi:10.1007/s10009-004-0153-x.
- [76] Jeff Larson, Surya Mattu, Lauren Kirchner and Julia Angwin. How We Analyzed the COMPAS Recidivism Algorithm, 2016. Online; accessed: 2023-02-06. URL: <https://www.propublica.org/article/how-we-analyzed-the-compass-recidivism-algorithm>.
- [77] Philips George John, Deepak Vijaykeerthy, and Diptikalyan Saha. Verifying individual fairness in machine learning models. In Ryan P. Adams and Vibhav Gogate, editors, *Proceedings of the Thirty-Sixth Conference on Uncertainty in Artificial Intelligence, UAI 2020, virtual online, August 3-6, 2020*, volume 124 of *Proceedings of Machine Learning Research*, pages 749–758. AUAI Press, 2020. URL: <http://proceedings.mlr.press/v124/george-john20a.html>.
- [78] David Jones, Chris Snider, Aydin Nassehi, Jason Yon, and Ben Hicks. Characterising the digital twin: A systematic literature review. *CIRP Journal of Manufacturing Science and Technology*, 29:36–52, 2020. doi:10.1016/j.cirpj.2020.02.002.

- [79] Timothy Jones. Boeing reveals further software problem in 737 max airplane, 2019. Online; accessed: 2022-09-21. URL: <https://www.dw.com/en/boeing-reveals-further-software-problem-in-737-max-airplane/a-48214065>.
- [80] Julia Angwin, Jeff Larson, Surya Mattu and Lauren Kirchner. Machine Bias, 2016. Online; accessed: 2023-02-06. URL: <https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing>.
- [81] Narges Khakpour and Mohammad Reza Mousavi. Notions of conformance testing for cyber-physical systems: Overview and roadmap (invited paper). In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1-4, 2015*, volume 42 of *LIPICs*, pages 18–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPICs.CONCUR.2015.18.
- [82] Maximilian A. Köhl, Holger Hermanns, and Sebastian Biewer. Efficient monitoring of real driving emissions. In Christian Colombo and Martin Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2018. doi:10.1007/978-3-030-03769-7_17.
- [83] Markus Langer, Daniel Oster, Timo Speith, Holger Hermanns, Lena Kästner, Eva Schmidt, Andreas Sesing, and Kevin Baum. What do we want from explainable artificial intelligence (XAI)? - A stakeholder perspective on XAI and a conceptual model guiding interdisciplinary XAI research. *Artif. Intell.*, 296:103473, 2021. doi:10.1016/j.artint.2021.103473.
- [84] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005. doi:10.1145/1058150.1058158.
- [85] Lilah Burke. The Death and Life of an Admissions Algorithm, 2020. Online; accessed: 2023-02-06. URL: <https://www.insidehighered.com/admissions/article/2020/12/14/u-texas-will-stop-using-controversial-algorithm-evaluate-phd>.
- [86] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009*,

- Washington, DC, USA, 1-4 December 2009, pages 355–363. IEEE Computer Society, 2009. doi:10.1109/RTSS.2009.17.
- [87] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2004. doi:10.1007/978-3-540-30206-3_12.
- [88] Tiziana Margaria and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Part II*, volume 9953 of *LNCS*, 2016. doi:10.1007/978-3-319-47169-3.
- [89] Roger C Mayer, James H Davis, and F David Schoorman. An integrative model of organizational trust. *Academy of management review*, 20(3):709–734, 1995.
- [90] Karl Meinke and Muddassar A. Sindhu. Incremental learning-based testing for reactive systems. In Martin Gogolla and Burkhart Wolff, editors, *Tests and Proofs - 5th International Conference, TAP@TOOLS 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, volume 6706 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2011. doi:10.1007/978-3-642-21768-5_11.
- [91] Leila Methnani, Andrea Aler Tubella, Virginia Dignum, and Andreas Theodorou. Let me take over: Variable autonomy for meaningful human control. *Frontiers in Artificial Intelligence*, 4, 2021. doi:10.3389/frai.2021.737072.
- [92] Dimiter Milushev and Dave Clarke. Incremental hyperproperty model checking via games. In Hanne Riis Nielson and Dieter Gollmann, editors, *Secure IT Systems - 18th Nordic Conference, NordSec 2013*, volume 8208 of *LNCS*, pages 247–262. Springer, 2013. doi:10.1007/978-3-642-41488-6_17.
- [93] Jad Mouawad. Volkswagen to recall 8.5 million vehicles in europe, 2015. Online; accessed: 2022-09-21. URL: <https://www.nytimes.com/2015/10/16/business/international/volkswagen-germany-recall.html>.

- [94] Debarghya Mukherjee, Mikhail Yurochkin, Moulinath Banerjee, and Yuekai Sun. Two simple ways to learn individual fairness metrics from data. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7097–7107. PMLR, 13–18 Jul 2020. URL: <https://proceedings.mlr.press/v119/mukherjee20a.html>.
- [95] Truong Nghiem, Sriram Sankaranarayanan, Georgios E. Fainekos, Franjo Ivancic, Aarti Gupta, and George J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In Karl Henrik Johansson and Wang Yi, editors, *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, pages 211–220. ACM, 2010. doi:10.1145/1755952.1755983.
- [96] Luan Viet Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V. Deshmukh, and Taylor T. Johnson. Hyperproperties of real-valued signals. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*, pages 104–113. ACM, 2017. doi:10.1145/3127041.3127058.
- [97] Bart Nooteboom. *Trust: Forms, foundations, functions, failures and figures*. Edward Elgar Publishing, 2002.
- [98] Cathy O’Neil. *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. Crown Publishing Group, USA, 2016.
- [99] Oracle. AI in Human Resources: The Time is Now, 2019. Online; accessed: 2023-02-06. URL: <https://www.oracle.com/a/ocom/docs/applications/hcm/oracle-ai-in-hr-wp.pdf>.
- [100] Dana Pessach and Erez Shmueli. A review on fairness in machine learning. *ACM Comput. Surv.*, 55(3), feb 2022. doi:10.1145/3494672.
- [101] S. Pettersson and B. Lennartson. Stability and robustness for hybrid systems. In *Proceedings of 35th IEEE Conference on Decision and Control*, volume 2, pages 1202–1207 vol.2, Dec 1996. doi:10.1109/CDC.1996.572653.
- [102] R Tyrrell Rockafellar and Roger J-B Wets. *Variational analysis*, volume 317. Springer Science & Business Media, 2009.

- [103] Kenneth H Rosen and Kamala Krithivasan. *Discrete mathematics and its applications: with combinatorics and graph theory*. Tata McGraw-Hill Education, 2012.
- [104] Walter Rudin et al. *Principles of mathematical analysis*, volume 3. McGraw-Hill New York, 1976.
- [105] Gerardo Schneider. *Algorithmic Analysis of Polygonal Hybrid Systems*. PhD thesis, VERIMAG – UJF, Grenoble, France, July 2002.
- [106] Bruce Schneier, Matthew Fredrikson, Tadayoshi Kohno, and Thomas Ristenpart. Surreptitiously weakening cryptographic systems. *IACR Cryptology ePrint Archive*, 2015:97, 2015. URL: <http://eprint.iacr.org/2015/097>.
- [107] Yannik Schnitzer. PCDF core, September 2021. URL: <https://github.com/udsdepend/pcdf-core>.
- [108] Yannik Schnitzer and Sebastian Biewer. LolaDrives Android, September 2021. URL: <https://github.com/udsdepend/loladrives-android>.
- [109] Maximilian Schwenger. Statically Analyzed Stream Monitoring for Cyber-Physical Systems. Dissertation, Saarland University, 2022.
- [110] Marcelo Sousa and Isil Dillig. Cartesian Hoare logic for verifying k-safety properties. In Chandra Krintz and Emery Berger, editors, *PLDI 2016*, pages 57–69. ACM, 2016. doi:10.1145/2908080.2908092.
- [111] Statista. Installed base of smart speakers in the united states from 2018 to 2022, 2022. Online; accessed: 2022-09-21. URL: <https://www.statista.com/statistics/967402/united-states-smart-speakers-in-households/>.
- [112] Statista. Number of smartphone subscriptions worldwide from 2016 to 2021, with forecasts from 2022 to 2027, 2022. Online; accessed: 2022-09-21. URL: <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>.
- [113] Steven Meurrens. The Increasing Role of AI in Visa Processing, 2021. Online; accessed: 2023-02-06. URL: <https://canadianimmigrant.ca/immigrate/immigration-law/the-increasing-role-of-ai-in-visa-processing>.

- [114] Arthur Sullivan. German air traffic software glitch one of several problems afflicting sector, 2019. Online; accessed: 2022-09-21. URL: <https://www.dw.com/en/german-air-traffic-software-glitch-one-of-several-problems-afflicting-sector/a-48053507>.
- [115] Paulo Tabuada, Ayca Balkan, Sina Y. Caliskan, Yasser Shoukry, and Rupa Majumdar. Input-output robustness for discrete systems. In Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr, editors, *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 217–226. ACM, 2012. doi:10.1145/2380356.2380396.
- [116] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *SAS 2005*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005. doi:10.1007/11547662_24.
- [117] The European Parliament and the Council of the European Union. Directive 98/69/ec of the european parliament and of the council, 1998. URL: <http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31998L0069:EN:HTML>.
- [118] The European Parliament and the Council of the European Union. Commission Regulation (EU) 2017/1151, June 2017. URL: <http://data.europa.eu/eli/reg/2017/1151/oj>.
- [119] Jan Tretmans. *A formal approach to conformance testing*. PhD thesis, University of Twente, Enschede, Netherlands, 1992. URL: <http://purl.utwente.nl/publications/58114>.
- [120] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996. doi:10.1016/S0169-7552(96)00017-7.
- [121] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. doi:10.1007/978-3-540-78917-8_1.
- [122] Umweltbundesamt (Germany). Nitrogen dioxide has serious impact on health, 2018. Online; accessed: 2022-09-21. URL: <https://www.umweltbundesamt.de/en/press/pressinformation/nitrogen-dioxide-has-serious-impact-on-health>.

- [123] European Union. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation), 2016. <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32016R0679>.
- [124] United Nations. UN Vehicle Regulations - 1958 Agreement, Revision 2, Addendum 100, Regulation No. 101, Revision 3 — E/ECE/324/Rev.2/Add.100/Rev.3, 2013. URL: <http://www.unece.org/trans/main/wp29/wp29regs101-120.html>.
- [125] United Nations Educational, Scientific and Cultural Organization (UNESCO). Recommendation on the ethics of artificial intelligence, 2021. URL: <https://unesdoc.unesco.org/ark:/48223/pf0000380455>.
- [126] John Voelcker. VW diesel owners have lost \$1,500 in value on their cars: price analysis, 2016. Online; accessed: 2022-09-21. URL: https://www.greencarreports.com/news/1104531_vw-diesel-owners-have-lost-1500-in-value-on-their-cars-price-analysis.
- [127] Michele Volpato and Jan Tretmans. Approximate active learning of nondeterministic input output transition systems. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 72, 2015. doi:10.14279/tuj.eceasst.72.1008.
- [128] Sandra Wachter, Brent Mittelstadt, and Chris Russell. Bias preservation in machine learning: the legality of fairness metrics under EU non-discrimination law. *W. Va. L. Rev.*, 123:735, 2020. doi:10.2139/ssrn.3792772.
- [129] Washington State. Certification of Enrollment: Engrossed Substitute Senate Bill 6280 ('Washington State Facial Recognition Law'), 2020. Online; accessed: 2023-02-06. URL: <https://app.leg.wa.gov/billssummary?BillNumber=6280&Year=2019&Initiative=false#documentSection>.
- [130] Austin Waters and Risto Miikkulainen. Grade: Machine learning support for graduate admissions. *AI Magazine*, 35(1):64, Mar. 2014. URL: <https://ojs.aaai.org/index.php/aimagazine/article/view/2504>, doi:10.1609/aimag.v35i1.2504.
- [131] Wikipedia. Volkswagen emissions scandal. Wikipedia, The Free Encyclopedia, 2016. Online; accessed: 2022-09-21. URL: https://en.wikipedia.org/wiki/Volkswagen_emissions_scandal.

- [132] Wikipedia. Abgasnorm — wikipedia, die freie enzyklopädie, 2022. Online; accessed: 2022-05-14. URL: <https://de.wikipedia.org/w/index.php?title=Abgasnorm&oldid=223638095>.
- [133] Wikipedia. Audi A6 C8 — wikipedia, die freie enzyklopädie, 2022. Online; accessed: 2022-05-14. URL: https://de.wikipedia.org/w/index.php?title=Audi_A6_C8&oldid=221632578.
- [134] Wikipedia contributors. Boeing 737 max groundings — Wikipedia, the free encyclopedia, 2022. Online; accessed: 2022-09-21. URL: https://en.wikipedia.org/w/index.php?title=Boeing-737_MAX_groundings&oldid=1111353889.
- [135] Wikipedia contributors. Internet of things — Wikipedia, the free encyclopedia, 2022. Online; accessed: 2022-09-21. URL: https://en.wikipedia.org/w/index.php?title=Internet_of_things&oldid=1110984638.
- [136] Will Douglas Heaven. Predictive policing algorithms are racist. They need to be dismantled., 2020. Online; accessed: 2023-02-06. URL: <https://www.technologyreview.com/2020/07/17/1005396/predictive-policing-algorithms-racist-dismantled-machine-learning-bias-criminal-justice/>.
- [137] Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007. URL: <http://dx.doi.org/10.1016/j.tcs.2006.12.036>, doi:10.1016/j.tcs.2006.12.036.
- [138] Richard S. Zemel, Yu Wu, Kevin Swersky, Toniann Pitassi, and Cynthia Dwork. Learning fair representations. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 325–333. JMLR.org, 2013. URL: <http://proceedings.mlr.press/v28/zemel13.html>.