Doping Tests for Cyber-Physical Systems

SEBASTIAN BIEWER, Saarland University, Saarland Informatics Campus, Germany
PEDRO R. D'ARGENIO, Universidad Nacional de Córdoba, FAMAF, Argentina, CONICET, Argentina, and Saarland University, Saarland Informatics Campus, Germany
HOLGER HERMANNS, Saarland University, Saarland Informatics Campus, Germany and Institute of Intelligent Software, China

The software running in embedded or cyber-physical systems (CPS) is typically of proprietary nature, so users do not know precisely what the systems they own are (in)capable of doing. Most malfunctionings of such systems are not intended by the manufacturer, but some are, which means these cannot be classified as bugs or security loopholes. The most prominent examples have become public in the diesel emissions scandal, where millions of cars were found to be equipped with software violating the law, altogether polluting the environment and putting human health at risk. The behaviour of the software embedded in these cars was intended by the manufacturer, but it was not in the interest of society, a phenomenon that has been called *software doping*. Due to the unavailability of a specification, the analysis of doped software is significantly different from that for buggy or insecure software and hence classical verification and testing techniques have to be adapted.

The work presented in this paper builds on existing definitions of software doping and lays the theoretical foundations for conducting software doping tests, so as to enable uncovering unethical manufacturers. The complex nature of software doping makes it very hard to effectuate doping tests in practice. We explain the main challenges and provide efficient solutions to realise doping tests despite this complexity.

ACM Reference Format:
Sebastian Biewer, Pedro R. D'Argenio, and Holger Hermanns. 2021. Doping Tests for Cyber-Physical Systems. *ACM Trans. Model. Comput. Simul.* 31, 3, Article 16 (August 2021), 28 pages. https://doi.org/10.1145/3449354

1 INTRODUCTION

Embedded and cyber-physical systems are becoming more widespread as part of our daily life. Printers, mobile phones, smart watches, smart home equipment, virtual assistants, drones and battery-equipped gadgets are just a few examples. Modern cars are composed of a multitude of such systems. These systems can have a significant impact on our lives, especially if they do not work as expected. As a result, numerous approaches exist to assure quality of a system. The classical and most common type of malfunctioning is what is widely called "bug". Usually, a bug is a small mistake in the software or hardware that causes a behaviour that is not intended or expected. Other types of malfunctioning are caused by incorrect or wrongly interpreted sensor data, physical deficiencies of a component, or might for instance be radiation-induced.

Authors' addresses: Sebastian Biewer, biewer@depend.uni-saarland.de, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany; Pedro R. D'Argenio, pedro.dargenio@unc.edu.ar, Universidad Nacional de Córdoba, FAMAF, Córdoba, Argentina, CONICET, Córdoba, Argentina, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany; Holger Hermanns, hermanns@cs.uni-saarland.de, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, Institute of Intelligent Software, Guangzhou, China.

XXXX-XXXX/2021/8-ART16 $15.00
https://doi.org/10.1145/3449354

Another interesting kind of malfunction (also from an ethical perspective [4]) arises if the expectation of how the system should behave is different for two (or more) parties. Examples for such scenarios are widespread in the context of personal data privacy, where product manufacturers and data protection agencies have notoriously different opinions about how a software is supposed to handle personal data. Another example with a considerable history is related to the usage of third-party cartridges in printers. Manufacturers and users do not agree on whether their printer should work with third-party cartridges (the user's opinion) or only with those sold by the manufacturer (the manufacturer's opinion). Lastly, an example context that received very high media attention is that of emission cleaning systems in diesel cars. These systems are meant to reduce the amount of dangerous particles and gases like CO_2 and NO_2 released through the exhaust pipe, and there are regulations in place defining how much of these substances are allowed to be emitted during car operation. Part of these regulations are emissions tests, precisely defined test cycles that a car has to undergo on a chassis dynamometer [34]. For every car model the manufacturers need to obey to these regulations in order to get admission to the market. The central weakness of these regulations is that the relevant behaviour of the car is only a very small fraction of the possible behaviour on the road. Indeed, several manufacturers equip their cars with defeat devices that recognise if the car is undergoing an official emissions test. During the test, the car obeys the regulation, but outside test conditions, the emissions emitted are often significantly higher than allowed. Generally speaking, the phenomena described above are considered as incorrect software behaviour by one party, but as intended software behaviour by the other party (usually the manufacturer). In the literature, such phenomena are called software doping [3, 12].

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 only be discovered after production, by the other party facing the final product.

This problem is worsened by the fact that embedded software is typically proprietary, so (unless one finds a way to breach the intellectual property [11]) it is only possible to detect software doping by observation of the behaviour of the product, i.e., by black-box testing.

This paper develops the foundations for black-box testing approaches geared towards discovering doped software in concrete cases. We will start off from an established formal notion of robust cleanness (which is the negation of software doping) [12]. Essentially, the idea of robust cleanness is based on a succinct specification (called a "contract") that all involved parties are assumed to have agreed on upfront and which captures the intended behaviour of a system with respect to all inputs to the system. Inputs are considered to be user inputs or environmental inputs given by sensors. The contract is defined by input and output distances on standard system trajectories supplemented by input and output thresholds. Intuitively, the input distance and threshold induce a tube around the standard inputs, and similar for outputs. For any input in the tube around some standard input the system must be able to react with an output that is in the tube around the output possible according to the standard. In many cases, the radii of the tubes are identical to the respective thresholds. This is similar to the tube illustration for the robustness degree of temporal logics formulas [18].

Example 1.1. For a diesel car the standard trajectory is the behaviour exhibited during the official emissions test cycle. The input distance measures the deviation in car speed from the standard. The input threshold is a small number, larger than the acceptable error tolerance of the cycle, limiting the inputs considered of interest. The output distance then is the difference between (the total amount of) NO_2 released by the car facing inputs of interest and the NO_2 emitted if on the standard test cycle. For cars with an active defeat device we expect to see a violation of the contract even for relatively large output thresholds.

A cyber-physical system (CPS) is influenced by physical or chemical dynamics. Some of these influences can be observed by the sensors the CPS is equipped with, but some portion might remain unknown, making proper analysis difficult. Nondeterminsm is a powerful way of representing such uncertainty faithfully, and indeed the notion of robust cleanness supports nondeterministic reactive systems [12]. Furthermore, the analysis needs to consider (at least) two trajectories simultaneously, namely the standard trajectory and another that stays within the input tube. 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 [10]. In this respect, expressing robust cleanness as a hyperproperty needs both $\forall$ and $\exists$ trajectory quantifiers. Formulas containing only one type of quantifier can be analysed efficiently, e.g., using model-checking techniques, but checking properties with alternating quantifiers is known to be computationally hard [9, 20]. 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 infinite 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, not observing an output in finitely many trials does not rule out that this output can be generated. As a result, there is no prior work (we are aware of) that targets the automatic generation of test cases for hyperproperties, let alone robust cleanness.

This work is an extension of a recent paper [6]. These extensions shed light on the complete process from developing the theory, several transformations of a state-of-the-art testing algorithm towards an implementation of a testing framework, which we used to do doping tests in practice. The contribution of this paper is three-fold. (1) We observe that standard behaviour, in particular when derived by common standardisation procedures, can be represented by finite models, and we identify under which conditions the resulting contracts are (un)satisfiable. (2) For a given satisfiable contract we construct the largest nondeterministic model that is robustly clean w.r.t. this contract. We integrate this model into a model-based testing theory, which can provide a nondeterministic algorithm to derive sound test suites. (3) We develop a testing algorithm for bounded-length tests and discretised input/output values. We present a concrete implementation of this algorithm and demonstrate its effectiveness using examples for the diesel emissions scandal. Two of these test cases were executed with a real car on a chassis dynamometer.

2 SOFTWARE DOPING ON REACTIVE PROGRAMS

Embedded software is reactive, it reacts to inputs received from sensors by producing outputs that are meant to control the device functionality. We consider a reactive program as a function $P\colon \mathsf{In}^\omega \to 2^{(\mathsf{Out}^\omega)}$ on infinite sequences of inputs so that the program reacts to the k-th input in the input sequence by producing nondeterministically the k-th output in each respective output sequence. Thus, the program can be seen, for instance, as a (nondeterministic) Mealy or Moore machine. Moreover, we consider an equivalence relation $\approx \subseteq \mathsf{In}^\omega \times \mathsf{In}^\omega$ that equates sequences of inputs. To illustrate this, think of the program embedded in a printer. Here $\approx$ would for instance equate input sequences that agree with respect to submitting the same documents regardless of the cartridge brand, the level of the toner (as long as there is sufficient), etc. We furthermore consider the set $\mathsf{StdIn} \subseteq \mathsf{In}^\omega$ of inputs of interest or *standard inputs*. In the previous example, StdIn contains all the input sequences with compatible cartridges and printable documents. The definitions given below are simple adaptations of those given in [12] (but where parameters are instead treated as parts of the inputs).

Definition 2.1. A reactive program P is *clean* if for all inputs $\mathsf{i}, \mathsf{i}' \in \mathsf{StdIn}$ such that $\mathsf{i} \approx \mathsf{i}'$, $P(\mathsf{i}) = P(\mathsf{i}')$. Otherwise it is *doped.*

This definition states that a program is *clean* if its execution exhibits the same visible sequence of output when supplied with two equivalent inputs, provided such inputs comply with the given standard StdIn. Notice that the behaviour outside StdIn is deemed immediately clean since it is of no interest.

In the context of the printer example, a program that would fail to print a document when provided with an ink cartridge from a third-party manufacturer, but would otherwise succeed to print would be considered doped, since this difference in output behaviour is captured by the above definition. For this, the inputs (being pairs of document and printer cartridge) must be considered equivalent (not identical), which comes down to ink cartridges being compatible.

However, the above definition is not very helpful for cases that need to preserve certain intended behaviour *outside* of the standard inputs StdIn. This is clearly the case in the diesel emissions scandal where the standard inputs are given precisely by the emissions test, but the behaviour observed there is assumed to generalise beyond the singularity of this test setup. It is meant to ensure that the amount of NO_2 and NO (abbreviated as NO_x) in the car exhaust gas does not deviate considerably *in general*, and comes with a legal prohibition of defeat mechanisms that simply turn off the cleaning mechanism. This legal framework is obviously a bit short sighted, since it can be circumvented by mechanisms that alter the behaviour gradually in a continuous manner, but in effect drastically. In a nutshell, one expects that if the input values observed by the electronic control unit (ECU) of a diesel vehicle deviate within "reasonable distance" from the *standard* input values provided during the lab emission test, the amount of NO_x found in the exhaust gas is still within the regulated threshold, or at least it does not exceed it more than a "reasonable amount".

This motivates the need to introduce the notion of distances on inputs and outputs. More precisely, we consider distances on finite traces: $d_{\mathsf{In}} \colon (\mathsf{In}^* \times \mathsf{In}^*) \longrightarrow \mathbb{R}_{\geq 0}$ and $d_{\mathsf{Out}} \colon (\mathsf{Out}^* \times \mathsf{Out}^*) \longrightarrow \mathbb{R}_{\geq 0}$. Such distances are required to be pseudometrics. (d is a pseudometric if $d(x,x) = 0$, $d(x,y) = d(y,x)$ and $d(x,y) \leq d(x,z) + d(z,y)$ for all x, y, and z.) With this, D'Argenio et. al [12] provide a definition of robust cleanness that considers two parameters: parameter κ_{i} refers to the acceptable distance an input may deviate from the norm to be still considered, and parameter κ_{o} that tells how far apart outputs are allowed to be in case their respective inputs are within κ_{i} distance (Def. 2.2 spells out the Hausdorff distance used in [12]).

Definition 2.2. Let $\sigma[..k]$ denote the k-th prefix of the sequence σ. A reactive program P is *robustly clean* if for all input sequences $\mathsf{i}, \mathsf{i}' \in \mathsf{In}^\omega$ with $\mathsf{i} \in \mathsf{StdIn}$, it holds for arbitrary $k \geq 0$ that whenever $d_{\mathsf{In}}(\mathsf{i}[..j], \mathsf{i}'[..j]) \leq \kappa_{\mathsf{i}}$ for all $j \leq k$, then

(1) for all $\mathsf{o} \in P(\mathsf{i})$ there exists $\mathsf{o}' \in P(\mathsf{i}')$ such that $d_{\mathsf{Out}}(\mathsf{o}[..k], \mathsf{o}'[..k]) \leq \kappa_{\mathsf{o}}$, and
(2) for all $\mathsf{o}' \in P(\mathsf{i}')$ there exists $\mathsf{o} \in P(\mathsf{i})$ such that $d_{\mathsf{Out}}(\mathsf{o}[..k], \mathsf{o}'[..k]) \leq \kappa_{\mathsf{o}}$.

Notice that this is what we actually need for the nondeterministic case: each possible output generated along one of the executions of the program should be matched within "reasonable distance" by some output generated by the other execution of the program. Also notice that i' does not need to satisfy StdIn, but it will be considered as long as it is within κ_{i} distance of any input satisfying StdIn. In such a case, outputs generated by $P(\mathsf{i}')$ will be requested to be within κ_{o} distance of some output generated by the respective execution induced by a standard input.

We remark that Def. 2.2 entails the existence of a *contract* which defines the set of standard inputs StdIn, the tolerance parameters κ_i and κ_o as well as the distances d_In and d_Out. In the context of diesel engines, one might imagine that the values to be considered, especially the tolerance parameters κ_i and κ_o for a particular car model are made publicly available (or are even advertised by the car manufacturer), so as to enable potential customers to discriminate between different car models according to the robustness they reach in being clean. It is also imaginable that the tolerances and distances are fixed by the legal authorities as part of environmental regulations.

3 ROBUSTLY CLEAN LABELLED TRANSITION SYSTEMS

This section develops the framework needed for an effective theory of black-box doping tests based on the above concepts. In this, the standard behaviour (e.g. as defined by the emission tests) and the robust cleanness definitions together will induce a set of reference behaviours that then serve as a model in a model-based conformance testing approach. To set the stage for this, we recall the definitions of labelled transition systems (LTS) and input-output transitions systems (IOTS) together with Tretmans' notion on model-based conformance testing [31]. We then recast the characterisation of robust cleanness (Def. 2.2) in terms of LTS.

Definition 3.1. A *labelled transition system (LTS)* with inputs and outputs $\mathcal{L} = \langle Q, \mathsf{In}, \mathsf{Out}, \rightarrow, q_0 \rangle$ is a five-tuple where (i) Q is a (possibly uncountable) non-empty set of states; (ii) $L = \mathsf{In} \uplus \mathsf{Out}$ is a (possibly uncountable) set of labels; (iii) $\rightarrow \subseteq Q \times L \times Q$ is the transition relation; (iv) $q_0 \in Q$ is the initial state. 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 $a \in \mathsf{In}$ there is some $q' \in Q$ such that $q \xrightarrow{a} q'$.

For ease of presentation, we do not consider internal transitions. The following definitions will be used throughout the paper. A *finite path* p in an LTS $\mathcal{L} = \langle Q, \mathsf{In}, \mathsf{Out}, \rightarrow, q_0 \rangle$ 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 $\mathsf{last}(p)$ as the last state occurring in p, i.e., $\mathsf{last}(p) = q_n$. An *infinite path* p in $\mathcal{L}$ is a sequence $q_1 a_1 q_2 a_2 \ldots$ with $q_i \xrightarrow{a_i} q_{i+1}$ for all $i \in \mathbb{N}_{>0}$. Let $\mathsf{paths}_*(q)$ and $\mathsf{paths}_\omega(q)$ be the sets of all finite and infinite paths of $\mathcal{L}$ beginning in state q, respectively. If $q = q_0$, we also write $\mathsf{paths}_*(\mathcal{L})$ ($\mathsf{paths}_\omega(\mathcal{L})$) instead of $\mathsf{paths}_*(q_0)$ ($\mathsf{paths}_\omega(q_0)$). The sequence $a_1 a_2 \cdots a_n$ is a *finite trace* σ of $\mathcal{L}$ if there is a finite path $q_1 a_1 q_2 a_2 \ldots a_n q_{n+1} \in \mathsf{paths}_*(q_1)$. We denote $\mathsf{last}(\sigma)$ as the last action label occurring in σ, i.e., $\mathsf{last}(\sigma) = a_n$. $a_1 a_2 \cdots$ is an *infinite trace* if there is an infinite path $q_1 a_1 q_2 a_2 \ldots \in \mathsf{paths}_\omega(q_1)$. If p is a path, we let $\mathsf{trace}(p)$ denote the trace defined by p. For states $q \in Q$, let $\mathsf{traces}_*(q)$ and $\mathsf{traces}_\omega(q)$ be the set of all finite and infinite traces beginning in q, respectively, and let $\mathsf{traces}_*(\mathcal{L}) = \mathsf{traces}_*(q_0)$ and $\mathsf{traces}_\omega(\mathcal{L}) = \mathsf{traces}_\omega(q_0)$. We will use $\mathcal{L}_1 \subseteq \mathcal{L}_2$ to denote that $\mathsf{traces}_\omega(\mathcal{L}_1) \subseteq \mathsf{traces}_\omega(\mathcal{L}_2)$.

Model-Based Conformance Tests. In the following we recall the basic notions of *input-output conformance* (**ioco**) testing [31–33], and refer to the mentioned literature for more details. 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*. The idea of whether the IUT $\mathcal{I}$ *conforms to* the specification *Spec* is formalized by means of the **ioco** relation.

In the remainder of this article it is necessary to identify *quiescent* (or *suspended*) states. A state is quiescent whenever it cannot proceed autonomously, i.e., it cannot produce an output. A quiescent state may be explicitly highlighted as such by having an outgoing quiescence transition with the distinct label δ. An implementation can observe quiescence with a timeout mechanism. 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 3.2. Let $\mathcal{L} = \langle Q, \mathsf{In}, \mathsf{Out}, \rightarrow, q_0 \rangle$ be an LTS. The *quiescence closure* (or *δ-closure*) of $\mathcal{L}$ is defined as the LTS $\mathcal{L}_\delta := \langle Q, \mathsf{In}, \mathsf{Out}_\delta, \rightarrow_\delta, q_0 \rangle$ with $\mathsf{Out}_\delta := \mathsf{Out} \cup \{\delta\}$ and $\rightarrow_\delta := \rightarrow \cup \{s \xrightarrow{\delta}_\delta s \mid \forall o \in \mathsf{Out}, t \in Q\colon s \overset{o}{\not\rightarrow} t\}$. Using this the *suspension traces* of $\mathcal{L}$ are defined by $\mathrm{traces}_*(\mathcal{L}_\delta)$.

Let $\mathcal{L} = \langle Q, \mathsf{In}, \mathsf{Out}, \rightarrow, q_0 \rangle$ be an LTS with $\sigma = a_1\, a_2 \dots a_n \in \mathrm{traces}_*(\mathcal{L})$ and let $Q' \subseteq Q$. The set $\mathcal{L}$ after σ is defined as $\{q_n \mid q_0 a_1 q_1 a_2 \dots a_n q_n \in \mathrm{paths}_*(\mathcal{L})\}$ and Q' after a as $\{q' \mid \exists q \in Q'\colon q \xrightarrow{a} q'\}$. For a state q, let $\mathrm{out}(q) = \{o \in \mathsf{Out} \cup \{\delta\} \mid \exists q'\colon q \xrightarrow{o} q'\}$ and for a set of states Q', let $\mathrm{out}(Q') = \bigcup_{q \in Q'} \mathrm{out}(q)$.

The idea behind the **ioco** relation is that any output produced by the IUT must have been foreseen by its specification, and moreover, any input in the IUT not foreseen in the specification may introduce new functionality. As a result, $\mathcal{I}$ **ioco** *Spec* is defined to hold whenever $\mathrm{out}(\mathcal{I}_\delta$ after $\sigma) \subseteq \mathrm{out}(\mathit{Spec}_\delta$ after $\sigma)$ for all $\sigma \in \mathrm{traces}_*(\mathit{Spec}_\delta)$.

The base principle of *conformance testing* now is to assess by means of testing whether the IUT conforms to its specification w.r.t. **ioco**. Tretmans defines test cases as LTS. These LTS are described by means of a basic process algebra [33]. A *process* is a term defined in the language $\mathcal{P}$ given by

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

where Z is an index set, each a_z is a label, and 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 [33], 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.

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

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** $:= \sum\{a;$**pass** $\mid a \in \mathsf{Out}_\delta\}$ and **fail** $:= \sum\{a;$**fail** $\mid a \in \mathsf{Out}_\delta\}$, (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 [33]. This can be captured by parallel composition according to the following transition rule:

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

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 $t \in \mathsf{TG}(S)$ to denote that t is one of the processes that can be generated by an execution of $\mathsf{TG}(S)$.
$\mathsf{TG}(S)$:= choose nondeterministically one of the following processes:

(1) **pass**

(2) $i; t_i$ where $i \in \mathsf{In}$, S after $i \neq \emptyset$ and $t_i \in \mathsf{TG}(S \text{ after } i)$
$+ \sum\{o; \mathbf{fail} \mid o \in \mathsf{Out} \wedge o \notin \mathsf{out}(S)\}$
$+ \sum\{o_j; t_{o_j} \mid o_j \in \mathsf{Out} \wedge o_j \in \mathsf{out}(S)\}$, where for each o_j, $t_{o_j} \in \mathsf{TG}(S \text{ after } o_j)$
(3) $\sum\{o; \mathbf{fail} \mid o \in \mathsf{Out} \cup \{\delta\} \wedge o \notin \mathsf{out}(S)\}$
$+ \sum\{o_j; t_{o_j} \mid o_j \in \mathsf{Out} \cup \{\delta\} \wedge o_j \in \mathsf{out}(S)\}$, where for each o_j, $t_{o_j} \in \mathsf{TG}(S \text{ after } o_j)$

Given a specification *Spec* with initial state s_0, $\mathsf{TG}(\{s_0\})$ 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 it continues with the testing process. The third option is similar to the previous one only that it considers the possibility of quiescence instead of inputs. When the absence of an output (label δ) is observed, the test fails if quiescence is not accepted and otherwise continues with the selected execution. TG can produce a (possibly infinitely large) test suite T, for which a system $\mathcal{I}$ **passes** T if $\mathcal{I}$ **ioco** *Spec* and, conversely, $\mathcal{I}$ **ioco** *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 by Tretmans [32, 33] for more details and intuitions about **ioco**, $\mathcal{P}$ and TG.

It is important to note that in the setting of robust cleanness the specification *Spec* is missing. Instead, we need to construct the specification from the standard inputs and the respective observed outputs, together with the distance functions and the thresholds given by the contract. Furthermore, this needs to respect the $\forall - \exists$ interaction required by the cleanness property (Def. 2.2).

Software Doping on LTS. To capture the notion of software doping in the context of LTS, we provide two projections of a trace, projecting to a sequence of the appearing inputs, respectively outputs. To do this, we extend the set of labels by adding the input $-_i$, that indicates that in the respective step some output (or quiescence) was produced (but masking the precise output), and the output $-_o$ that indicates that in this step some (masked) input was given. *Projection on inputs* $\downarrow_i$: $(\mathsf{In} \cup \mathsf{Out}_\delta)^\omega \to (\mathsf{In} \cup \{-_i\})^\omega$ and *projection on outputs* $\downarrow_o$: $(\mathsf{In} \cup \mathsf{Out}_\delta)^\omega \to (\mathsf{Out}_\delta \cup \{-_o\})^\omega$ are defined for all traces $\sigma \in (\mathsf{In} \cup \mathsf{Out}_\delta)^\omega$ and $k \in \mathbb{N}$ as follows: $\sigma{\downarrow_i}[k] := \textbf{if } \sigma[k] \in \mathsf{In} \textbf{ then } \sigma[k] \textbf{ else } -_i$ and similarly $\sigma{\downarrow_o}[k] := \textbf{if } \sigma[k] \in \mathsf{Out}_\delta \textbf{ then } \sigma[k] \textbf{ else } -_o$. They are lifted to sets of traces in the usual elementwise way.

Definition 3.3. An LTS $\mathcal{S}$ is a *standard* for an LTS $\mathcal{L}$, if $\mathsf{traces}_\omega(\mathcal{S}_\delta) \subseteq \mathsf{traces}_\omega(\mathcal{L}_\delta)$.

The above definition provides an interpretation of the notion of StdIn for a given program P modelled in terms of LTS $\mathcal{L}$. This interpretation relaxes the original definition of StdIn, because it requires to fix only a subset of the behaviour that $\mathcal{L}$ exhibits when executed with standard inputs. This corresponds to a testing context, in which recordings of the system executing standard inputs are the baseline for testing. StdIn can then be considered as implicitly determined as the input sequences $\mathsf{traces}_\omega(\mathcal{S}){\downarrow_i}$ occurring in $\mathcal{S}$. If instead $\mathcal{L}$ and $\mathsf{StdIn} \subseteq (\mathsf{In} \cup -_i)^\omega$ are given, we denote by $\mathcal{S}^{(\mathcal{L},\mathsf{StdIn})}$ a standard LTS which is maximal w.r.t. StdIn and $\mathcal{L}$, i.e., for all $\sigma \in (\mathsf{In} \cup \mathsf{Out}_\delta)^\omega$, $\sigma \in \mathsf{traces}_\omega({\mathcal{S}_\delta}^{(\mathcal{L},\mathsf{StdIn})})$ if and only if $\sigma{\downarrow_i} \in \mathsf{StdIn}$ and $\sigma \in \mathsf{traces}_\omega(\mathcal{L}_\delta)$.

In this new setting, we assume that the distance functions d_{In} and d_{Out} apply on traces containing labels $-_i$ and $-_o$, i.e. they are pseudometrics in $(\mathsf{In} \cup \{-_i\})^* \times (\mathsf{In} \cup \{-_i\})^* \to \mathbb{R}_{\geq 0}$ and, respectively, $(\mathsf{Out} \cup \{-_o\})^* \times (\mathsf{Out} \cup \{-_o\})^* \to \mathbb{R}_{\geq 0}$. In the context of this article, we maintain the notion of a contract for robust cleanness and denote it explicitly by a 5-tuple $\mathcal{C} = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$. It contains an LTS $\mathcal{S}$ representing some standard behaviour, the distance functions and thresholds (the domains In and Out are captured implicitly as the domains of d_{In}, respectively d_{Out}). With this, we state robust cleanness for LTS as follows.

Definition 3.4. An IOTS $\mathcal{L}$ is *robustly clean* w.r.t. some contract $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$ if and only if $\mathcal{S}$ is a standard for $\mathcal{L}$ and for all $\sigma \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$, $\sigma' \in \mathrm{traces}_\omega(\mathcal{L}_\delta)$ and $k \geq 0$ it holds that whenever $d_{\mathsf{In}}(\sigma[..j]{\downarrow_i}, \sigma'[..j]{\downarrow_i}) \leq \kappa_{\mathsf{i}}$ for all $j \leq k$ then

(1) there exists $\sigma'' \in \mathrm{traces}_\omega(\mathcal{L}_\delta)$ such that $\sigma'{\downarrow_i} = \sigma''{\downarrow_i}$ and $d_{\mathsf{Out}_\delta}(\sigma[..k]{\downarrow_o}, \sigma''[..k]{\downarrow_o}) \leq \kappa_{\mathsf{o}}$,
(2) there exists $\sigma'' \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$ such that $\sigma{\downarrow_i} = \sigma''{\downarrow_i}$ and $d_{\mathsf{Out}_\delta}(\sigma'[..k]{\downarrow_o}, \sigma''[..k]{\downarrow_o}) \leq \kappa_{\mathsf{o}}$.

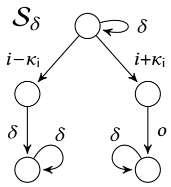
The definition is in the fashion of the HyperLTL interpretation of Proposition 19 of [12] (restricted to programs with no parameters). There, contracts define the set StdIn instead of the LTS $\mathcal{S}$ and hence the standard behaviour is fully defined. The relaxed interpretation of standard behaviour $\mathcal{S}$ is reflected in the last line of Def. 3.4, which requires σ'' to be a trace of $\mathcal{S}_\delta$ instead of $\mathcal{L}_\delta$. For the maximal standard LTS $\mathcal{S}^{(\mathcal{L},\mathsf{StdIn})}$, Def. 3.4 echoes the HyperLTL semantics. Thus, the proof showing that Def. 3.4 is the correct interpretation of Def. 2.2 in terms of LTS, can be obtained in a way similar to that of Prop. 19 in [12]. In the spirit of model-based testing with **ioco**, Def. 3.4 takes specific care of quiescence in a system. In order to properly integrate quiescence into the context of robust cleanness it must be considered as a unique output. As a consequence, in the presence of a contract $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$, we use—instead of $\mathcal{S}$, Out and d_{Out}—the quiescence closure $\mathcal{S}_\delta$ of $\mathcal{S}$, $\mathsf{Out}_\delta = \mathsf{Out} \cup \{\delta\}$ and an extended output distance defined as $d_{\mathsf{Out}_\delta}(\sigma_1, \sigma_2) := d_{\mathsf{Out}}(\sigma_{1\setminus\delta}, \sigma_{2\setminus\delta})$ if $\sigma_1[i] = \delta \Leftrightarrow \sigma_2[i] = \delta$ for all i, and $d_{\mathsf{Out}_\delta}(\sigma_1, \sigma_2) := \infty$ otherwise, where $\sigma_{\setminus\delta}$ is the same as σ with all δ removed.

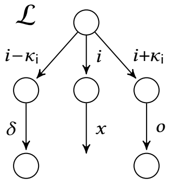
In the sequel, we will at some places need to refer to Def. 3.4 only considering the second condition (but not the first one). We denote this as Def. 3.4.2. We will also use the predicate $\forall j \leq k \colon d_{\mathsf{In}}(\sigma[..j]{\downarrow_i}, \sigma'[..j]{\downarrow_i}) \leq \kappa_{\mathsf{i}}$, which we abbreviate by a predicate $\mathcal{V}_{(d_{\mathsf{In}},\kappa_{\mathsf{i}})}(k, \sigma, \sigma')$. If d_{In} and κ_{i} are known from the context, we omit the index.

4 REFERENCE IMPLEMENTATION FOR CONTRACTS

As mentioned before, doping tests need to be based on a contract C, which we assume given. C specifies the domains In, Out, a standard LTS $\mathcal{S}$, the distance functions d_{In} and d_{Out} and the bounds κ_{i} and κ_{o}. We intuitively expect the contract to be satisfiable in the sense that it never enforces a single input sequence of the implementation to keep outputs close enough to two different executions of the specification while their outputs stretch too far apart. We show such a problematic case in the following example.

Example 2. On the right a quiescence-closed standard LTS $\mathcal{S}_\delta$ for an implementation $\mathcal{L}$ (shown below) is depicted. For simplicity some input transitions are omitted. Assume $\mathsf{Out} = \{o\}$ and $\mathsf{In} = \{i, i{-}\kappa_{\mathsf{i}}, i{+}\kappa_{\mathsf{i}}\}$. Consider the transition labelled x of $\mathcal{L}$. This must be one of either o 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 $\mathcal{L}$ is $i\ -_i$ and the input distance to $(i{-}\kappa_{\mathsf{i}})\ -_i$ and $(i + \kappa_{\mathsf{i}})\ -_i$ is exactly κ_{i}, so both branches $(i + \kappa_{\mathsf{i}})\ o$ and $(i - \kappa_{\mathsf{i}})\ \delta$ of $\mathcal{S}_\delta$ must be considered to determine x. For $x = o$, the output distance of $-_o\ x$ to $-_o\ o$ in the right branch of $\mathcal{S}_\delta$ is 0, i.e. less than κ_{o}. However, $d_{\mathsf{Out}_\delta}(-_o\ \delta, -_o\ o) = \infty > \kappa_{\mathsf{o}}$. Thus the output distance to the left branch of $\mathcal{S}_\delta$ is too high if picking o. Instead picking $x = \delta$ does not work either, for the symmetric reasons, the problem switches sides. Thus, neither picking o nor δ for x satisfies robust cleanness here. Indeed, no implementation satisfying robust cleanness exists for the given contract.





We would expect that a correct implementation fully entails the standard behaviour. So, to satisfy a contract, the standard behaviour itself must be robustly clean. This and the need for satisfiability of particular inputs lead to Def. 4.1.

Definition 4.1 (Satisfiable Contract). Let $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_\mathsf{i}, \kappa_\mathsf{o} \rangle$ be a contract and let $\sigma_i \in (\mathsf{In} \cup \{-_i\})^\omega$ be the input projection of some trace. σ_i is *satisfiable* for C if and only if for every standard trace $\sigma_S \in \mathsf{traces}_\omega(\mathcal{S}_\delta)$ and $k > 0$ such that for all $j \le k$ $d_{\mathsf{In}}(\sigma_i[..j], \sigma_S[..j]\downarrow_i) \le \kappa_\mathsf{i}$ there is some implementation $\mathcal{L}$ that satisfies Def. 3.4.2 w.r.t. C and has some trace $\sigma \in \mathsf{traces}_\omega(\mathcal{L}_\delta)$ with $\sigma\downarrow_i = \sigma_i$ and $d_{\mathsf{Out}_\delta}(\sigma[..k]\downarrow_o, \sigma_S[..k]\downarrow_o) \le \kappa_\mathsf{o}$.
C is *satisfiable* if and only if all inputs $\sigma_i \in (\mathsf{In} \cup \{-_i\})^\omega$ are satisfiable for C and $\mathcal{S}$ is robustly clean w.r.t. C. A contract that is not satisfiable is called *unsatisfiable*.

Given a satisfiable contract it is always possible to construct an implementation that is robustly clean w.r.t. to this contract. Furthermore, for every contract there is exactly one implementation (modulo trace equivalence) that contains all possible outputs that satisfy robust cleanness. Such an implementation is called the *largest implementation*.

Definition 4.2 (Largest Implementation). Let C be a contract and $\mathcal{L}$ an implementation that is robustly clean w.r.t. C. $\mathcal{L}$ is the *largest implementation within* C if and only if for every $\mathcal{L}'$ that is robustly clean w.r.t. C it holds that $\mathsf{traces}_\omega(\mathcal{L}'_\delta) \subseteq \mathsf{traces}_\omega(\mathcal{L}_\delta)$.

In the following, we will focus on the fragment of satisfiable contracts with standard behaviour defined by finite LTS. For unsatisfiable contracts, testing is not necessary, because every implementation is not robustly clean w.r.t. to C. Finiteness of $\mathcal{S}$ will be necessary to make testing feasible in practice. For simplicity we will further assume *past-forgetful* output distance functions. That is, $d_{\mathsf{Out}}(\sigma_1, \sigma_2) = d_{\mathsf{Out}}(\sigma'_1, \sigma'_2)$ whenever $\mathsf{last}(\sigma_1) = \mathsf{last}(\sigma'_1)$ and $\mathsf{last}(\sigma_2) = \mathsf{last}(\sigma'_2)$. Thus, we simply assume that $d_{\mathsf{Out}} \colon (\mathsf{Out} \cup \{-_o\} \times \mathsf{Out} \cup \{-_o\}) \to \mathbb{R}_{\ge 0}$, i.e., the output distances are determined by the last output only. We remark that $d_{\mathsf{Out}_\delta}(\delta, o) = \infty$ for all $o \neq \delta$.

Reference implementation. We will now show how to construct the largest implementation for any contract (of the fragment we consider), which we name *reference implementation* $\mathcal{R}$. It is derived from $\mathcal{S}_\delta$ by adding inputs and outputs in such a way that whenever the input sequence leading to a particular state is within κ_i distance of an input sequence σ_i of $\mathcal{S}_\delta$, then the outputs possible in such a state should be at most κ_o distant from those outputs possible in the unique state on $\mathcal{S}_\delta$ reached through σ_i. This ensures that $\mathcal{R}$ will satisfy condition 2) in Def. 3.4. 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 Def. 3.4.2.

Definition 4.3. Let $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_\mathsf{i}, \kappa_\mathsf{o} \rangle$ be a contract. The *reference implementation* $\mathcal{R}$ for contract C is the LTS $\langle (\mathsf{In} \cup \mathsf{Out}_\delta)^*, \mathsf{In}, \mathsf{Out}_\delta, \to_{\mathcal{R}}, \epsilon \rangle$ where $\to_{\mathcal{R}}$ is defined by

$$\frac{\begin{array}{l}\forall \sigma_i \in \mathsf{traces}_\omega(\mathcal{S}_\delta)\downarrow_i :\\ \quad (\forall j \le |\sigma| + 1 \colon d_{\mathsf{In}}((\sigma \cdot a)\downarrow_i[..j], \sigma_i[..j]) \le \kappa_\mathsf{i})\\ \qquad \Rightarrow \exists \sigma_S \in \mathsf{traces}_\omega(\mathcal{S}_\delta) \colon \sigma_S\downarrow_i = \sigma_i \wedge d_{\mathsf{Out}_\delta}(a\downarrow_o, \sigma_S[|\sigma| + 1]\downarrow_o) \le \kappa_\mathsf{o}\end{array}}{\sigma \xrightarrow{a}_{\mathcal{R}} \sigma \cdot a}$$

Notably, $\mathcal{R}$ is deterministic, since only transitions of the form $\sigma \xrightarrow{a}_{\mathcal{R}} \sigma \cdot a$ 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 4.4 and Corollary 4.5.

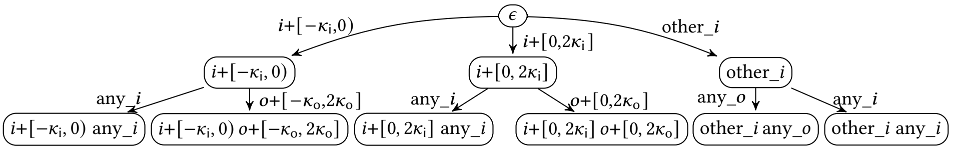
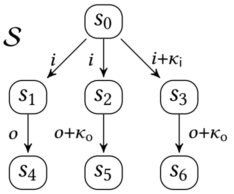

ϵ
i+$[-\kappa_{\mathrm{i}},0)$
i+$[0,2\kappa_{\mathrm{i}}]$
other_i
i+$[-\kappa_{\mathrm{i}},0)$
i+$[0,2\kappa_{\mathrm{i}}]$
other_i
any_i
o+$[-\kappa_{\mathrm{o}},2\kappa_{\mathrm{o}}]$
any_i
o+$[0,2\kappa_{\mathrm{o}}]$
any_o
any_i
i+$[-\kappa_{\mathrm{i}},0)$ any_i
i+$[-\kappa_{\mathrm{i}},0)$ o+$[-\kappa_{\mathrm{o}},2\kappa_{\mathrm{o}}]$
i+$[0,2\kappa_{\mathrm{i}}]$ any_i
i+$[0,2\kappa_{\mathrm{i}}]$ o+$[0,2\kappa_{\mathrm{o}}]$
other_i any_o
other_i any_i

Fig. 1. The reference implementation $\mathcal{R}$ of $\mathcal{S}$ in Example 3.


$\mathcal{S}$
s_0
i
i
i+κ_{i}
s_1
s_2
s_3
o
o+κ_{o}
o+κ_{o}
s_4
s_5
s_6

Example 3. Fig. 1 gives a schematic representation of the reference implementation $\mathcal{R}$ for the LTS $\mathcal{S}$ 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 $d_{\mathsf{In}}(i,i') := |i-i'|$ and $d_{\mathrm{Out}}(o,o') := |o-o'|$. For this example, the quiescence closure $\mathcal{S}_\delta$ looks like $\mathcal{S}$ 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 Fig. 1 (omitted in the picture).

The reference implementation $\mathcal{R}$ is obtained according to Def. 4.3. In order to give an idea of its construction we focus on the states σ such that $|\sigma| = 1$ (i.e., $\sigma \in \mathsf{In} \cup \{\delta\}$)—other cases are simpler. First, notice that

$$\mathrm{traces}_\omega(\mathcal{S}_\delta) \;=\; \delta^\omega \;+\; \delta^*\, i\, o\, \delta^\omega \;+\; \delta^*\, i\, (o{+}\kappa_{\mathrm{o}})\, \delta^\omega \;+\; \delta^*\, (i{+}\kappa_{\mathrm{i}})\, (o{+}\kappa_{\mathrm{o}})\, \delta^\omega.$$

Here we use ω-regular notation to describe the set of traces. This means that $\mathrm{traces}_\omega(\mathcal{S}_\delta)$ contains the trace that remains quiet indefininitely (namely δ^ω), all traces that may stay quiet for a while, receive an input i, produce and output o, and remain quiet indefininitely (i.e., any trace in $\delta^*\, i\, o\, \delta^\omega$), and so on. Hence, the set $\mathrm{traces}_\omega(\mathcal{S}_\delta){\downarrow_i}$ is then

$$\mathrm{traces}_\omega(\mathcal{S}_\delta){\downarrow_i} \;=\; -_i^\omega \;+\; -_i^*\, i\, -_i^\omega \;+\; -_i^*\, (i{+}\kappa_{\mathrm{i}})\, -_i^\omega.$$

Suppose $\sigma \in i$+$[-\kappa_{\mathrm{i}},0)$ and $a \in o + [-\kappa_{\mathrm{o}}, 2\kappa_{\mathrm{o}}]$. In this case, $\sigma_i = i\, -_i^\omega \in \mathrm{traces}_\omega(\mathcal{S}_\delta){\downarrow_i}$ is the only standard trace satisfying $\forall j \leq 2\colon d_{\mathsf{In}}((\sigma \cdot a){\downarrow_i}[..j], \sigma_i[..j]) \leq \kappa_{\mathrm{i}}$. If $a \in o + [-\kappa_{\mathrm{o}}, \kappa_{\mathrm{o}}]$ take $\sigma_S = i\, o\, \delta^\omega$ and then $\sigma_S{\downarrow_i} = \sigma_i \wedge d_{\mathrm{Out}_\delta}(a{\downarrow_o}, \sigma_S[|\sigma|+1]{\downarrow_o}) \leq \kappa_{\mathrm{o}}$ holds. If $a \in o + [0, 2\kappa_{\mathrm{o}}]$, then $\sigma_S = i\, (o{+}\kappa_{\mathrm{o}})\, \delta^\omega$ is the one that does the job. Therefore $\sigma \xrightarrow{a}_{\mathcal{R}} \sigma \cdot a$. This case defines the schematic transition i+$[-\kappa_{\mathrm{i}},0) \xrightarrow{o+[-\kappa_{\mathrm{o}},2\kappa_{\mathrm{o}}]}_{\mathcal{R}} i$+$[-\kappa_{\mathrm{i}},0)\, o$+$[-\kappa_{\mathrm{o}}, 2\kappa_{\mathrm{o}}]$.

If instead $a \in \mathrm{Out}$ but $a \notin o + [-\kappa_{\mathrm{o}}, 2\kappa_{\mathrm{o}}]$, then no a-outgoing transition from $\sigma \in i$+$[-\kappa_{\mathrm{i}},0)$ is possible since no matching σ_S can be found. However, if $a \in \mathsf{In}$, no $\sigma_i \in \mathrm{traces}_\omega(\mathcal{S}_\delta){\downarrow_i}$ satisfies $\forall j \leq 2\colon d_{\mathsf{In}}((\sigma \cdot a){\downarrow_i}[..j], \sigma_i[..j]) \leq \kappa_{\mathrm{i}}$. As the antecedent of the implication is false, any input defines a valid outgoing transition from a state $\sigma \in i$+$[-\kappa_{\mathrm{i}},0)$. This yields the schematic transition i+$[-\kappa_{\mathrm{i}},0) \xrightarrow{\text{any\_}i}_{\mathcal{R}} i$+$[-\kappa_{\mathrm{i}},0)$ any_i.

Suppose now $\sigma \in i$+$[0,2\kappa_{\mathrm{i}}]$ and $a \in o + [0, 2\kappa_{\mathrm{o}}]$. We consider the two subcases $\sigma \in i$+$[0,\kappa_{\mathrm{i}}]$ and $\sigma \in i$+$(\kappa_{\mathrm{i}}, 2\kappa_{\mathrm{i}}]$. If $\sigma \in i$+$(\kappa_{\mathrm{i}}, 2\kappa_{\mathrm{i}}]$ then $\sigma_i = (i{+}\kappa_{\mathrm{i}})\, -_i^\omega \in \mathrm{traces}_\omega(\mathcal{S}_\delta){\downarrow_i}$ is the only one satisfying $\forall j \leq 2\colon d_{\mathsf{In}}((\sigma \cdot a){\downarrow_i}[..j], \sigma_i[..j]) \leq \kappa_{\mathrm{i}}$ and the construction follows similarly as above. If instead $\sigma \in i$+$[0,\kappa_{\mathrm{i}}]$, then every $\sigma_i \in \{i\, -_i^\omega, (i{+}\kappa_{\mathrm{i}})\, -_i^\omega\}$ satisfies $\forall j \leq 2\colon d_{\mathsf{In}}((\sigma \cdot a){\downarrow_i}[..j], \sigma_i[..j]) \leq \kappa_{\mathrm{i}}$. If $\sigma_i = i\, -_i^\omega$, choose $\sigma_S = i\, (o{+}\kappa_{\mathrm{o}})\, \delta^\omega$, and if $\sigma_i = (i{+}\kappa_{\mathrm{i}})\, -_i^\omega$, choose $\sigma_S = (i{+}\kappa_{\mathrm{i}})\, (o{+}\kappa_{\mathrm{o}})\, \delta^\omega$. In both of these cases, $\sigma_S{\downarrow_i} = \sigma_i \wedge d_{\mathsf{Out}_\delta}(a{\downarrow_o}, \sigma_S[|\sigma|+1]{\downarrow_o}) \leq \kappa_\mathrm{o}$ is satisfied. Hence $\sigma \xrightarrow{a}_{\mathcal{R}} \sigma \cdot a$. Putting both subcases together yields the schematic transition $i{+}[0, 2\kappa_\mathrm{i}] \xrightarrow{o+[0,2\kappa_\mathrm{o}]}_{\mathcal{R}} i{+}[0, 2\kappa_\mathrm{i}]\ o + [0, 2\kappa_\mathrm{o}]$.

The case in which $\sigma \in i{+}[0, 2\kappa_\mathrm{i}]$ but $a \notin o + [0, 2\kappa_\mathrm{o}]$ follows as before.

If $\sigma \notin i{+}[-\kappa_\mathrm{i}, 2\kappa_\mathrm{i}]$ (in other words, "$\sigma \in \mathrm{other\_}i$"), there is no $\sigma_i \in \mathsf{traces}_\omega(\mathcal{S}_\delta){\downarrow_i}$ such that $\forall j \leq 2\colon d_{\mathsf{In}}((\sigma \cdot a){\downarrow_i}[..j], \sigma_i[..j]) \leq \kappa_\mathrm{i}$, so any transition is possible.

Finally, if $\sigma = \delta$ (omitted in Fig. 1) the construction would follow just like for the initial state ϵ.

Properties of the Reference Implementation $\mathcal{R}$. In order 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 in order 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 4.4. *Let C be a contract and $\mathcal{R}$ the reference implementation for C. Then, for all finite paths $p \in \mathsf{paths}_*(\mathcal{R})$ it holds that* $\mathsf{last}(p) = \mathsf{trace}(p)$.

PROOF. We proceed by induction on the number of states in p. If p has only one state then $p = \epsilon = \mathsf{last}(p) = \mathsf{trace}(p)$, since ϵ is the initial state in $\mathcal{R}$.
Suppose now, that $p = (p'\,a\,s) \in \mathsf{paths}_*(\mathcal{R})$. By induction, $\mathsf{last}(p') = \mathsf{trace}(p')$. By Def. 4.3, $\mathsf{last}(p') \xrightarrow{a}_{\mathcal{R}} s$ only if $s = \mathsf{last}(p') \cdot a$. But $\mathsf{last}(p) = s = \mathsf{last}(p') \cdot a = \mathsf{trace}(p') \cdot a = \mathsf{trace}(p)$, which proves the lemma. □

As a consequence, for every trace of $\mathcal{R}$ we can reconstruct the unique path with this trace.

COROLLARY 4.5. *Let C be a contract, $\mathcal{R}$ the reference implementation for C, $p \in \mathsf{paths}_*(\mathcal{R})$ a finite path of $\mathcal{R}$ and $\sigma = \mathsf{trace}(p)$ its trace. Then p 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 within the contract it is constructed from. The following lemma shows a similar property. It is stronger in not assuming implementations to be IOTS (thus LTS are sufficient) and it considers only the second condition of robust cleanness. It is also weaker in not concluding $\mathcal{R}$ to be robustly clean. However, this Lemma will be central to many of the following proofs.

LEMMA 4.6. *Let C be a contract and $\mathcal{R}$ the reference implementation for C. Then, for every LTS $\mathcal{L}$ satisfying Def. 3.4.2, it holds that* $\mathsf{traces}_\omega(\mathcal{L}_\delta) \subseteq \mathsf{traces}_\omega(\mathcal{R})$.

PROOF. For a proof by contradiction, suppose that there is some $\mathcal{L}$ satisfying Def. 3.4.2, but which has some trace $\sigma \in \mathsf{traces}_\omega(\mathcal{L}_\delta)$ that is not a trace of $\mathcal{R}$, i.e. $\sigma \notin \mathsf{traces}_\omega(\mathcal{R})$. Since $\sigma \notin \mathsf{traces}_\omega(\mathcal{R})$, there must be some $k > 0$ for which $\sigma[..k-1] \in \mathsf{traces}_*(\mathcal{R})$, but $\sigma[..k] \notin \mathsf{traces}_*(\mathcal{R})$. 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 Def. 4.3 is not satisfied, i.e., there is some $\sigma_i \in \mathsf{traces}_\omega(\mathcal{S}_\delta){\downarrow_i}$, such that (1) $\mathcal{V}(k, \sigma, \sigma_i)$ and (2) for all standard traces $\sigma_S \in \mathsf{traces}_\omega(\mathcal{S}_\delta)$ with $\sigma_s{\downarrow_i} = \sigma_i$ it holds that $d_{\mathsf{Out}_\delta}(\sigma[k]{\downarrow_o}, \sigma_S[k]{\downarrow_o}) > \kappa_\mathrm{o}$.
Let $\sigma_{io} \in \mathsf{traces}_\omega(\mathcal{S}_\delta)$ such that $\sigma_{io}{\downarrow_i} = \sigma_i$. From Def. 3.4.2 we get for $\mathcal{L}, \sigma_{io}, \sigma$ and k with (1) a trace $\sigma'' \in \mathsf{traces}_\omega(\mathcal{S}_\delta)$ with $\sigma''{\downarrow_i} = \sigma_{io}{\downarrow_i} = \sigma_i$ and $d_{\mathsf{Out}_\delta}(\sigma[..k]{\downarrow_o}, \sigma''[..k]{\downarrow_o}) \leq \kappa_\mathrm{o}$. From the assumption that d_{Out_δ} is past-forgetful, we get that $d_{\mathsf{Out}_\delta}(\sigma[k]{\downarrow_o}, \sigma''[k]{\downarrow_o}) \leq \kappa_\mathrm{o}$, which is a contradiction to (2). □

Def. 4.3 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 4.7 shows that this is done in the spirit of model-based testing theory and **ioco**, that is, $\mathcal{R}_\delta$ is identical to $\mathcal{R}$.

Lemma 4.7. *Let C be a satisfiable contract and $\mathcal{R}$ the reference implementation for C. Then, the quiescence closure $\mathcal{R}_\delta$ of $\mathcal{R}$ is exactly $\mathcal{R}$.*

Proof. We have to show that for every state $\sigma \in (\mathsf{In} \cup \mathsf{Out}_\delta)^*$, there is a transition $\sigma \xrightarrow{o}_{\mathcal{R}} \sigma \cdot o$ in $\mathcal{R}$ with $o \in \mathsf{Out}_\delta$. Let $\sigma_i = \sigma{\downarrow_i} \cdot (-_i)^\omega$ an infinite input trace. We proceed by case-distinction on whether there is a trace $\sigma_S \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$ such that $\mathcal{V}(|\sigma| + 1, \sigma_i, \sigma_S)$ holds. If this is not the case, the premise of Def. 4.3 does not hold and hence we get that for all $o \in \mathsf{Out}_\delta$ 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 C and Def. 4.1 an implementation $\mathcal{L}$ and a trace $\sigma'' \in \mathrm{traces}_\omega(\mathcal{L}_\delta)$ with $\sigma''{\downarrow_i} = \sigma_i$ and $d_{\mathsf{Out}_\delta}(\sigma''[|\sigma|+1]{\downarrow_o}, \sigma_S[|\sigma|+1]{\downarrow_o}) \leq \kappa_{\mathrm{o}}$. From Lemma 4.6 we get that $\sigma'' \in \mathrm{traces}_\omega(\mathcal{R})$. 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 Def. 4.3) that for every trace $\sigma_S \in \mathrm{traces}_\omega(\mathcal{S}_\delta){\downarrow_i}$ for which $\mathcal{V}(|\sigma| + 1, \sigma'', \sigma_S)$ holds, there is some $\hat{\sigma} \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$ with $\hat{\sigma}{\downarrow_i} = \sigma_S{\downarrow_i}$ and $d_{\mathsf{Out}_\delta}(\sigma''[|\sigma| + 1], \hat{\sigma}[|\sigma| + 1]{\downarrow_o}) \leq \kappa_{\mathrm{o}}$. 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 $d_{\mathsf{In}}(\sigma''[..j]{\downarrow_i}, \sigma_S[..j]{\downarrow_i}) \leq \kappa_{\mathrm{i}} \iff d_{\mathsf{In}}((\sigma \cdot \sigma''[|\sigma| + 1])[..j]{\downarrow_i}, \sigma_S[..j]{\downarrow_i}) \leq \kappa_{\mathrm{i}}$ holds. Hence, for every $\sigma_S \in \mathrm{traces}_\omega(\mathcal{S}_\delta){\downarrow_i}$ with $\mathcal{V}(|\sigma| + 1, (\sigma \cdot \sigma''[|\sigma| + 1]), \sigma_S)$, we can provide a $\hat{\sigma} \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$ with $\hat{\sigma}{\downarrow_i} = \sigma_S{\downarrow_i}$ and $d_{\mathsf{Out}_\delta}(\sigma''[|\sigma| + 1], \hat{\sigma}[|\sigma| + 1]{\downarrow_o}) \leq \kappa_{\mathrm{o}}$. By Def. 4.3 we know that the transition $\sigma[..|\sigma|] \xrightarrow{\sigma''[|\sigma|+1]}_{\mathcal{R}} \sigma[..|\sigma|] \cdot \sigma''[|\sigma|+1]$ exists in $\mathcal{R}$. Since $\sigma''{\downarrow_i} = \sigma_i$, we know that $\sigma''{\downarrow_i}[|\sigma|+1] = -_i$ and hence $\sigma''[|\sigma| + 1] \in \mathsf{Out}_\delta$. □

The LTS $\mathcal{R}$ is supposed to serve as an implementation. To this end, Lemma 4.8 shows that $\mathcal{R}$ is input-enabled and hence is an IOTS.

Lemma 4.8. *Let C be a satisfiable contract with standard $\mathcal{S}$ and let $\mathcal{R}$ be constructed from C. Then $\mathcal{R}$ is an input-output transition system.*

Proof. By construction, $\mathcal{R}$ is a labelled transition system. By Def. 3.1 an LTS is an IOTS, if it is input-enabled. Hence, we have to show that for every state $\sigma \in (\mathsf{In} \cup \mathsf{Out})^*$ it holds for every $i \in \mathsf{In}$ that there is a transition $\sigma \xrightarrow{i}_{\mathcal{R}} \sigma \cdot i$ in $\mathcal{R}$. To have this transition, the premise of Def. 4.3 must be satisfied. Let $\sigma_i \in \mathit{traces}_*(\mathcal{S}_\delta){\downarrow_i}$ and accordingly $\sigma_S \in \mathrm{traces}_*(\mathcal{S}_\delta)$ 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 $d_{\mathsf{Out}_\delta}(i{\downarrow_o}, \sigma_S[|\sigma|+1]{\downarrow_o}) = d_{\mathsf{Out}_\delta}(-_o, \sigma_S[|\sigma|+1]{\downarrow_o}) \leq \kappa_{\mathrm{o}}$. We continue by case distinction of whether $\sigma_S[|\sigma| + 1] \in \mathsf{In}$. If this is the case, we are immediately done, because $d_{\mathsf{Out}_\delta}(-_o, -_o) = 0 \leq \kappa_{\mathrm{o}}$.
If instead $\sigma_S[|\sigma|+1] \in \mathsf{Out}_\delta$, satisfiability of C with $(\sigma \cdot i){\downarrow_i}$ for σ_i, σ_S for σ_S and $k = |\sigma|+1$ provides some implementation $\mathcal{L}$ satisfying Def. 3.4.2 and a trace $\hat{\sigma} \in \mathrm{traces}_\omega(\mathcal{L}_\delta)$ with $\hat{\sigma}{\downarrow_i} = (\sigma \cdot i){\downarrow_i}$ and $d_{\mathsf{Out}_\delta}(\hat{\sigma}[|\sigma| + 1]{\downarrow_o}, \sigma_S[|\sigma| + 1]{\downarrow_o}) \leq \kappa_{\mathrm{o}}$. From Lemma 4.6 we get that $\hat{\sigma} \in \mathrm{traces}_\omega(\mathcal{R})$. We get from $d_{\mathsf{Out}_\delta}(\hat{\sigma}[|\sigma| + 1]{\downarrow_o}, \sigma_S[|\sigma| + 1]{\downarrow_o}) \leq \kappa_{\mathrm{o}}$ and $\hat{\sigma}{\downarrow_i} = (\sigma \cdot i){\downarrow_i}$ that $d_{\mathsf{Out}_\delta}(-_o, \sigma_S[|\sigma| + 1]{\downarrow_o}) \leq \kappa_{\mathrm{o}}$, which concludes the proof. □

$\mathcal{R}$ is modelled by adding all transitions satisfying Def. 3.4.2. Lemma 4.9 confirms that, conversely, $\mathcal{R}$ satisfies Def. 3.4.2. Then, Lemma 4.10 shows that $\mathcal{R}$ satisfies also the first condition of Def. 3.4.

Lemma 4.9. *Let C be a contract and $\mathcal{R}$ the reference implementation for C. Then, for all $\sigma \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$ and $\sigma' \in \mathrm{traces}_\omega(\mathcal{R})$, it holds that for all $k \geq 0$ such that $d_{\mathsf{In}}(\sigma[..j]{\downarrow_i}, \sigma'[..j]{\downarrow_i}) \leq \kappa_{\mathrm{i}}$ for all $j \leq k$, there exists $\sigma'' \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$ such that $\sigma{\downarrow_i} = \sigma''{\downarrow_i}$ and $d_{\mathsf{Out}_\delta}(\sigma'[k]{\downarrow_o}, \sigma''[k]{\downarrow_o}) \leq \kappa_{\mathrm{o}}$.*

Proof. Let $\sigma \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$ and let $\sigma' \in \mathrm{traces}_\omega(\mathcal{R})$. By Corollary 4.5, we get that there must be some path $p = \epsilon\, \sigma'[1](\sigma'[..1]) \dots (\sigma'[..k-1])\sigma'[k](\sigma'[..k]) \in \mathrm{paths}_*(\mathcal{R})$. In particular

$\sigma'[..k-1] \xrightarrow{\sigma'[k]} \sigma'[..k]$ is a transition in $\mathcal{R}$. By Def. 4.3, we know that for all $\sigma_i \in \text{traces}(\mathcal{S}_\delta)\downarrow_i$ with $\mathcal{V}(k, \sigma'[..k], \sigma_i)$ (which is equivalent to $\mathcal{V}(k, \sigma', \sigma_i)$), there is some $\sigma_S \in \text{traces}_\omega(\mathcal{S}_\delta)$ with $\sigma_S\downarrow_i = \sigma_i$ and $d_{\text{Out}_\delta}(\sigma[k]\downarrow_o, \sigma_S[k]\downarrow_o) \leq \kappa_\text{o}$ (*).
Since $\sigma \in \text{traces}_\omega(\mathcal{S}_\delta)$ then $\sigma\downarrow_i \in \text{traces}_\omega(\mathcal{S}_\delta)\downarrow_i$. Suppose $\mathcal{V}(k, \sigma', \sigma)$ holds (otherwise the lemma holds trivially). Then, by (*), there exists $\sigma_S \in \text{traces}_\omega(\mathcal{S}_\delta)$ with $\sigma_S\downarrow_i = \sigma\downarrow_i$ such that $d_{\text{Out}_\delta}(\sigma[k]\downarrow_o, \sigma_S[k]\downarrow_o) \leq \kappa_\text{o}$. □

Lemma 4.10. *Let C be a satisfiable contract and $\mathcal{R}$ the reference implementation for C. Then, for all $\sigma, \sigma' \in \text{traces}_\omega(\mathcal{R})$, if $\sigma \in \text{traces}_\omega(\mathcal{S}_\delta)$, it holds that for all $k \geq 0$ such that $d_\text{In}(\sigma[..j]\downarrow_i, \sigma'[..j]\downarrow_i) \leq \kappa_\text{i}$ for all $j \leq k$, there exists $\sigma'' \in \text{traces}_\omega(\mathcal{R})$ such that $\sigma'\downarrow_i = \sigma''\downarrow_i$ and $d_{\text{Out}_\delta}(\sigma[k]\downarrow_o, \sigma''[k]\downarrow_o) \leq \kappa_\text{o}$.*

Proof. Let $\sigma \in \text{traces}_\omega(\mathcal{S}_\delta)$, $\sigma' \in \text{traces}_\omega(\mathcal{R})$ and $k \geq 0$. Suppose $\mathcal{V}(k, \sigma, \sigma')$ holds (otherwise, the lemma holds trivially). From satisfiability of C we know that input $\sigma'\downarrow_i$ is satisfiable and, hence, we get from Def. 4.1 for σ and k an implementation $\mathcal{L}$ satisfying Def. 3.4.2 for which there is $\sigma'' \in \text{traces}_\omega(\mathcal{L}_\delta)$ with $\sigma''\downarrow_i = \sigma'\downarrow_i$ and $d_{\text{Out}_\delta}(\sigma''[k]\downarrow_o, \sigma[k]\downarrow_o) \leq \kappa_\text{o}$. From Lemma 4.6 we get that $\text{traces}_\omega(\mathcal{L}_\delta) \subseteq \text{traces}_\omega(\mathcal{R})$, so we can conclude that $\sigma'' \in \text{traces}_\omega(\mathcal{R})$. Hence, σ'' is the desired trace to conclude the proof. □

Each contract contains some standard behaviour as an LTS $\mathcal{S}$. The reference implementation for a contract should be constructed in a way such that $\mathcal{S}$ is a standard for the implementation (according to Def. 3.3). Lemma 4.11 shows that this is the case for $\mathcal{R}$. In this work, we have a particular interest into practical applicability. Hence, we assume for the proof of this Lemma, that $\mathcal{S}$ is finite. This enables us to make the proof by arguing about finite traces.

Lemma 4.11. *Let $\mathcal{S}$ be a finite LTS, $C = \langle \mathcal{S}, d_\text{In}, d_\text{Out}, \kappa_\text{i}, \kappa_\text{o} \rangle$ a satisfiable contract and $\mathcal{R}$ the reference implementation for C. Then $\mathcal{S}$ is a standard for $\mathcal{R}$.*

Proof. We have to show that $\text{traces}_\omega(\mathcal{S}_\delta) \subseteq \text{traces}_\omega(\mathcal{R}_\delta)$. Since $\text{traces}_\omega(\mathcal{R}) = \text{traces}_\omega(\mathcal{R}_\delta)$ according to Lemma 4.7, it suffices to show that $\text{traces}_\omega(\mathcal{S}_\delta) \subseteq \text{traces}_\omega(\mathcal{R})$. Also, since $\mathcal{S}$ is finite also $\mathcal{S}_\delta$ is finite, so we can construct a deterministic (image-finite) LTS $\mathcal{S}_\delta{}'$ (where states are finite traces of the original) with $\text{traces}_\omega(\mathcal{S}_\delta) = \text{traces}_\omega(\mathcal{S}_\delta{}')$. With $\mathcal{R}$ being deterministic, too, we only need to prove $\text{traces}_*(\mathcal{S}_\delta) \subseteq \text{traces}_*(\mathcal{R})$ (see [35] for a proof).
Let $\sigma \in \text{traces}_*(\mathcal{S}_\delta)$. We proceed by induction on $k = |\sigma|$. If $k = 0$ then $\sigma = \epsilon$ and hence $\sigma \in \text{traces}_*(\mathcal{R})$. If $k > 0$, we know that $\sigma[..k-1] \in \text{traces}_*(\mathcal{S}_\delta)$ and from the inductive hypothesis that $\sigma[..k-1] \in \text{traces}_*(\mathcal{R})$. There is a path $p \in \text{paths}_*(\mathcal{R})$ with $\text{trace}(p) = \sigma[..k-1]$ and it follows from Lemma 4.4, that $\text{last}(p) = \sigma[..k-1]$. To show that $\sigma \in \text{traces}_*(\mathcal{R})$, we need to show that there is a transition $\sigma[..k-1] \xrightarrow{\sigma[k]}_{\mathcal{R}} \sigma$ in $\mathcal{R}$. By Def. 4.3, this holds if for any $\sigma_i \in \text{traces}_\omega(\mathcal{S}_\delta)\downarrow_i$ with $\forall j \leq k \colon d_\text{In}(\sigma[..j]\downarrow_i, \sigma_i[..j]) \leq \kappa_\text{i}$, $\mathcal{V}(k, \sigma, \sigma_i)$, there is some $\sigma_S \in \text{traces}_\omega(\mathcal{S}_\delta)$ with $\sigma_S\downarrow_i = \sigma_i$ for which $d_{\text{Out}_\delta}(\sigma[k]\downarrow_o, \sigma_S[k]\downarrow_o) \leq \kappa_\text{o}$. The existence of σ_i implies the existence of some $\sigma_{io} \in \text{traces}_\omega(\mathcal{S}_\delta)$ with $\sigma_{io}\downarrow_i = \sigma_i$. Also, notice that σ can be extended to an infinite trace $\sigma' \in \text{traces}_\omega(\mathcal{S}_\delta)$ such that $\sigma'[..k] = \sigma$. From satisfiability of C we know that $\mathcal{S}$ is robustly clean w.r.t. C. Then, by Def. 3.4.2, there is some $\sigma'' \in \text{traces}_\omega(\mathcal{S}_\delta)$ with $\sigma''\downarrow_i = \sigma_{io}\downarrow_i$ $(= \sigma_i)$ and $d_\text{Out}(\sigma'[k]\downarrow_o, \sigma''[k]\downarrow_o) \leq \kappa_\text{o}$. Taking $\sigma_S = \sigma''$ concludes the proof. □

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 contract it is constructed from.

Theorem 4.12. *Let $\mathcal{S}$ be a finite LTS, $C = \langle \mathcal{S}, d_\text{In}, d_\text{Out}, \kappa_\text{i}, \kappa_\text{o} \rangle$ a satisfiable contract and $\mathcal{R}$ the reference implementation for C. Then $\mathcal{R}$ is robustly clean w.r.t. C.*

Proof. Def. 3.4 requires that $\mathcal{R}$ is an IOTS, which is shown in Lemma 4.8. Furthermore, from Lemma 4.11 we get that $\mathcal{S}$ is a standard for $\mathcal{R}$. With Lemmas 4.10, 4.9 and 4.7 we get that $\mathcal{R}$ satisfies both conditions of Def. 3.4. □

Furthermore, it is not difficult to show that $\mathcal{R}$ is indeed the largest implementation that is allowed by the contract it was constructed from.

Theorem 4.13. *Let $\mathcal{S}$ be a finite LTS, $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$ a satisfiable contract and $\mathcal{R}$ the reference implementation for C. Then $\mathcal{R}$ is the largest implementation within C.*

Proof. We know from Theorem 4.12 that $\mathcal{R}$ is an IOTS, which is robustly clean w.r.t. C. It remains to show that for every LTS $\mathcal{L}'$ that is robustly clean w.r.t. C, $\mathsf{traces}_\omega(\mathcal{L}_\delta) \subseteq \mathsf{traces}_\omega(\mathcal{R})$. Any such $\mathcal{L}'$ satisfies in particular Def. 3.4.2, so it follows directly from Lemma 4.6 that $\mathcal{R}$ is the largest implementation within C. □

5 MODEL-BASED DOPING TESTS

Following the conceptual ideas behind **ioco**, 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 $\mathcal{I}$ **ioco** $\mathcal{R}$ holds, i.e.,

$$\forall \sigma \in \mathsf{traces}_*(\mathcal{R}_\delta) \colon \mathsf{out}(\mathcal{I}_\delta \text{ after } \sigma) \subseteq \mathsf{out}(\mathcal{R}_\delta \text{ after } \sigma). \tag{1}$$

To work 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 5.1. This enables us to use all the properties considering traces of $\mathcal{R}$ from Section 4.

Lemma 5.1. *Let $\mathcal{L}$ be an LTS, $\sigma \in \mathsf{traces}_*(\mathcal{L}_\delta)$ a suspension trace of $\mathcal{L}$ and o an output. Then, $o \in \mathsf{out}(\mathcal{L}_\delta \text{ after } \sigma)$ if and only if $\sigma \cdot o \in \mathsf{traces}_*(\mathcal{L}_\delta)$.*

Proof. By definition, $o \in \mathsf{out}(\mathcal{L}_\delta \text{ after } \sigma)$ if and only if there is some $q \in (\mathcal{L}_\delta \text{ after } \sigma)$ for which there is some q' and a transition $q \xrightarrow{o} q'$. This holds if and only if there is a path $p \in \mathsf{paths}_*(\mathcal{L}_\delta)$ with $\mathsf{trace}(p) = \sigma$, $\mathsf{last}(p) = q$ and $q \xrightarrow{o} q'$. Equivalently, there can be a path $p' \in \mathsf{paths}_*(\mathcal{L}_\delta)$ with $\mathsf{trace}(p') = \sigma \cdot o$, which is the case if and only if $\sigma \cdot o \in \mathsf{traces}_*(\mathcal{L}_\delta)$. □

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 **ioco** into trace properties and exploits the properties of $\mathcal{R}$ established in Section 4.

Theorem 5.2. *Let $\mathcal{S}$ be a finite LTS, $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$ a satisfiable contract, $\mathcal{R}$ the reference implementation for C and let $\mathcal{I}$ be an IOTS, which is robustly clean w.r.t. C. Then, it holds that $\mathcal{I}$* **ioco** *$\mathcal{R}$.*

Proof. We have to show that for all $\sigma \in \mathsf{traces}_*(\mathcal{R}_\delta)$ it holds that $\mathsf{out}(\mathcal{I}_\delta \text{ after } \sigma) \subseteq \mathsf{out}(\mathcal{R}_\delta \text{ after } \sigma)$. From Lemma 4.7 we know that $\sigma \in \mathsf{traces}_*(\mathcal{R})$. If $\mathsf{out}(\mathcal{I}_\delta \text{ after } \sigma) = \emptyset$ the theorem trivially holds. Otherwise, there is some $o \in \mathsf{out}(\mathcal{I}_\delta \text{ after } \sigma) \subseteq \mathsf{Out}_\delta$ and $\sigma \cdot o \in \mathsf{traces}_*(\mathcal{I}_\delta)$ follows with Lemma 5.1. By Def. 3.2, every state in $\mathcal{I}_\delta$ has an outgoing output or quiescence transition and hence there is an infinite trace $\sigma' \in \mathsf{traces}_\omega(\mathcal{I}_\delta)$ with $\sigma'[..|\sigma| + 1] = \sigma \cdot o$. By Def. 4.2, Theorem 4.12, Theorem. 4.13 and robust cleanness of $\mathcal{I}$, we can conclude that $\sigma' \in \mathsf{traces}_\omega(\mathcal{R}_\delta)$. Since $\sigma \cdot o$ is a finite prefix of σ', we get that $\sigma \cdot o \in \mathsf{traces}_*(\mathcal{R}_\delta)$. Finally, Lemma 5.1 gives us that $o \in \mathsf{out}(\mathcal{R}_\delta \text{ after } \sigma)$. □

Theorem 5.2 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 5.3, which follows directly from the completeness of TG [32, 33].

COROLLARY 5.3. *Let $\mathcal{S}$ be a finite LTS, $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$ a satisfiable contract and $\mathcal{R}$ the reference implementation for C. Then $\mathcal{I}$* **ioco** *$\mathcal{R}$ if and only if $\mathcal{I}$* **passes** $\mathsf{TG}(\{\epsilon\})$.

However, there are several issues regarding the practicality of TG. Among them, to perform a doping test for a given contract C, we first have to construct $\mathcal{R}$. $\mathcal{R}$ is the largest implementation within C and as such is 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 only what is needed.

```
DTG($h$) := choose nondeterministically one of the following processes:
  (1) pass
  (2)   $i; t_i$   where $i \in \mathsf{In}$ and $t_i \in \mathsf{DTG}(h \cdot i)$
      + $\sum\{o; \mathbf{fail} \mid o \in \mathsf{Out} \wedge o \notin \mathrm{acc}(h)\}$
      + $\sum\{o_j; t_{o_j} \mid o_j \in \mathsf{Out} \wedge o_j \in \mathrm{acc}(h)\}$, where for each $o_j$, $t_{o_j} \in \mathsf{DTG}(h \cdot o_j)$
  (3)   $\sum\{o; \mathbf{fail} \mid o \in \mathsf{Out} \cup \{\delta\} \wedge o \notin \mathrm{acc}(h)\}$
      + $\sum\{o_j; t_{o_j} \mid o_j \in \mathsf{Out} \cup \{\delta\} \wedge o_j \in \mathrm{acc}(h)\}$, where for each $o_j$, $t_{o_j} \in \mathsf{DTG}(h \cdot o_j)$
```

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 acc instead of out. Essentially, $\mathrm{acc}(h)$ captures all output transitions leaving state h in $\mathcal{R}$ (i.e., $\mathrm{out}(\{h\})$) without knowing (or constructing) $\mathcal{R}$. Thus, $\mathrm{acc}(h)$ is precisely the set of outputs that satisfies the premise in the definition of $\mathcal{R}$ after the trace h, as stipulated in Def. 4.3. The definition of acc is shown in eq. (2).

$$
\begin{aligned}
\mathrm{acc}(h) := \{o \in & \mathsf{Out}_\delta \mid \\
& \forall \sigma_i \in \mathsf{traces}_\omega(\mathcal{S}_\delta){\downarrow_i} : \\
& \quad (\forall j \leq |h|{+}1 : d_{\mathsf{In}}(\sigma_i[..j]{\downarrow_i}, (h \cdot o)[..j]{\downarrow_i}) \leq \kappa_{\mathsf{i}}) \\
& \qquad \Rightarrow \exists \sigma \in \mathsf{traces}_\omega(\mathcal{S}_\delta) : \sigma{\downarrow_i} = \sigma_i{\downarrow_i} \wedge d_{\mathsf{Out}_\delta}(o, \sigma[|h|+1]{\downarrow_o}) \leq \kappa_{\mathsf{o}}\}
\end{aligned}
\tag{2}
$$

The following lemma confirms that acc can be used to compute out without knowing $\mathcal{R}$. Instead, the definition of acc is defined directly for a contract $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$. We emphasize this difference in Lemma 5.4 by annotating the functions appropriately, i.e., by $\mathrm{acc}^{(C)}$ and $\mathrm{out}^{(\mathcal{R})}$.

LEMMA 5.4. *Let $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$ be a satisfiable contract and $\mathcal{R}$ the reference implementation for C. For all $h \in (\mathsf{In} \cup \mathsf{Out}_\delta)^*$, $\mathrm{acc}^{(C)}(h) = \mathrm{out}^{(\mathcal{R})}(\{h\})$.*

PROOF. Let $h \in (\mathsf{In} \cup \mathsf{Out}_\delta)^*$ and $o \in \mathsf{Out}_\delta$. As per eq. (2), $o \in \mathrm{acc}^{(C)}(h)$ if and only if for any $\sigma_i \in \mathsf{traces}_\omega(\mathcal{S}_\delta){\downarrow_i}$ with $\mathcal{V}(|h|+1, \sigma_i, h \cdot o)$ there exists $\sigma \in \mathsf{traces}_\omega(\mathcal{S}_\delta)$ such that $\sigma{\downarrow_i} = \sigma_i{\downarrow_i}$ and $d_{\mathsf{Out}}(o, \sigma[|h|+1]{\downarrow_o}) \leq \kappa_{\mathsf{o}}$. However, this is equivalent to the premise of the rule from Def. 4.3, hence $o \in \mathrm{acc}^{(C)}(h)$ 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 $o \in \mathrm{out}^{(\mathcal{R})}(\{h\})$. □

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** $\mathsf{DTG}(\epsilon)$ (as in Corollary 5.3). For this, it is serviceable to realise that for every history $h \in (\mathsf{In} \cup \mathsf{Out}_\delta)^*$, 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 5.5. *Let C be a satisfiable contract and $\mathcal{R}$ the reference implementation for C. Then, for every process $p \in \mathcal{P}$ and history $h \in (\mathsf{In} \cup \mathsf{Out}_\delta)^*$, it holds that $p \in \mathsf{TG}(\{h\})$ if and only if $p \in \mathsf{DTG}(h)$.*

Proof. We prove the claim by structural induction on p. If p is a process name, then $p = \textbf{fail}$ or $p = \textbf{pass}$. 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 $p \in \mathsf{TG}(\{h\}) \Rightarrow p \in \mathsf{DTG}(h)$ and distinguish between whether p is constructed by choice (2) or (3) of TG.

For case (2), we fix some arbitrary $i \in \mathsf{In}$ and $t_i \in \mathsf{TG}(\{h\} \text{ after } i)$. Notice that $(\{h\} \text{ after } i)$ is always non-empty, because $\mathcal{R}$ is input-enabled (Lemma 4.8). Furthermore, we fix a mapping from accepted outputs to one of the possible recursively computed subprocess $\mathcal{F} := \{(o, t_o) \mid o \in \mathsf{Out} \cap \mathsf{out}(\{h\}) \wedge t_o \in \mathsf{TG}(\{h\} \text{ after } o)\}$. Then, choice (2) of TG produces exactly one test, which is $p = i; t_i + \sum\{o; \textbf{fail} \mid o \in \mathsf{Out} \wedge o \notin \mathsf{out}(\{h\})\} + \sum\{o; \mathcal{F}(o) \mid o \in \mathsf{Out} \wedge o \in \mathsf{out}(\{h\})\}$. We can rewrite the test case to $p' = i; t_i + \sum\{o; \textbf{fail} \mid o \in \mathsf{Out} \wedge o \notin \mathsf{acc}(h)\} + \sum\{o; \mathcal{F}(o) \mid o \in \mathsf{Out} \wedge o \in \mathsf{acc}(h)\}$ by using that $\mathsf{out}(\{h\}) = \mathsf{acc}(h)$ from Lemma 5.4. From Def. 4.3 it follows that for every $o \in \mathsf{out}(\{h\})$, $(\{h\} \text{ after } o) = \{h \cdot o\}$. Hence, $\mathcal{F} = \{(o, t_o) \mid o \in \mathsf{Out} \cap \mathsf{acc}(h) \wedge t_o \in \mathsf{TG}(\{h \cdot o\})\} = \{(o, t_o) \mid o \in \mathsf{Out} \cap \mathsf{acc}(h) \wedge t_o \in \mathsf{DTG}(h \cdot o)\}$ with the inductive hypothesis. From Lemma 4.8 and Def. 4.3 we know that $t_i \in \mathsf{TG}(\{h \cdot i\})$ and hence by the inductive hypothesis $t_i \in \mathsf{DTG}(h \cdot i)$. Now, for the fixed i, t_i and $\mathcal{F}$, p' is exactly the test that is generated by choice (2) of $\mathsf{DTG}(h)$.

For case (3) of TG, we fix a mapping from accepted outputs to one of the possible recursively computed subprocess $\mathcal{F} := \{(o, t_o) \mid o \in \mathsf{Out}_\delta \cap \mathsf{out}(\{h\}) \wedge t_o \in \mathsf{TG}(\{h\} \text{ after } o)\}$. Then, choice (3) of TG produces exactly one test, which is $\sum\{o; \textbf{fail} \mid o \in \mathsf{Out}_\delta \wedge o \notin \mathsf{out}(\{h\})\} + \sum\{o; \mathcal{F}(o) \mid o \in \mathsf{Out}_\delta \wedge o \in \mathsf{out}(\{h\})\}$. We can rewrite the test case to $p' = i; t_i + \sum\{o; \textbf{fail} \mid o \in \mathsf{Out}_\delta \wedge o \notin \mathsf{acc}(h)\} + \sum\{o; \mathcal{F}(o) \mid o \in \mathsf{Out}_\delta \wedge o \in \mathsf{acc}(h)\}$ by using that $\mathsf{out}(\{h\}) = \mathsf{acc}(h)$ from Lemma 5.4. From Def. 4.3 it follows that for every $o \in \mathsf{out}(\{h\})$, $(\{h\} \text{ after } o) = \{h \cdot o\}$. From this, we can conclude $\mathcal{F} = \{(o, t_o) \mid o \in \mathsf{Out}_\delta \cap \mathsf{acc}(h) \wedge t_o \in \mathsf{TG}(\{h \cdot o\})\}$ and then $\mathcal{F} = \{(o, t_o) \mid o \in \mathsf{Out}_\delta \cap \mathsf{acc}(h) \wedge t_o \in \mathsf{DTG}(h \cdot o)\}$ with the inductive hypothesis. Now, for the fixed $\mathcal{F}$, p' is exactly the test that is generated by choice (3) of $\mathsf{DTG}(h)$.

The proof for $p \in \mathsf{DTG}(h) \Rightarrow p \in \mathsf{TG}(\{h\})$ is analogue. □

With Lemma 5.5 and Corollary 5.3 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 5.6. *Let $\mathcal{S}$ be a finite LTS, $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$ a satisfiable contract and $\mathcal{I}$ an implementation. If $\mathcal{I}$ is robustly clean w.r.t. C, then $\mathcal{I}$* **passes** $\mathsf{DTG}(\epsilon)$.

Proof. Let $\mathcal{R}$ be the reference implementation for C. With Lemma 5.5 and Corollary 5.3 we get that $\mathcal{I}$ **passes** $\mathsf{DTG}(\epsilon)$ if and only if $\mathcal{I}$ **ioco** $\mathcal{R}$. According to Theorem 5.2 the latter holds if $\mathcal{I}$ is robustly clean w.r.t. C. □

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 the first condition of Def. 3.4 (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.

Bounded-Depth Doping Tests. We developed and proved correct Algorithm DTG, which enables model-based testing for some contract 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 if needing to construct a test covering all possible answers of the implementation under test. Third, function acc, 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 acc so that it considers finite traces for its decision. Such a bounded-depth construction is provided as eq. (3).

$$\begin{aligned}\mathrm{acc}_b(h) \coloneqq \{o \in & \mathrm{Out}_\delta \mid \\ & \forall \sigma_i \in \mathrm{traces}_b(\mathcal{S}_\delta)\!\downarrow_i : \\ & \quad (\forall j \leq |h|{+}1 \colon d_{\mathsf{In}}(\sigma_i[..j]\!\downarrow_i, (h \cdot o)[..j]\!\downarrow_i) \leq \kappa_{\mathrm{i}}) \\ & \quad\quad \Rightarrow \exists \sigma \in \mathrm{traces}_b(\mathcal{S}_\delta) \colon \sigma\!\downarrow_i = \sigma_i\!\downarrow_i \wedge d_{\mathrm{Out}_\delta}(o, \sigma[|h|{+}1]\!\downarrow_o) \leq \kappa_{\mathrm{o}}\}\end{aligned} \tag{3}$$

For test history of length at most b, acc_b delivers all outputs that are accepted by some contract. It is computable provided that In and Out are bounded and discretised. The only variation w.r.t. acc in eq. (2) lies in the use of the set $\mathrm{traces}_b(\mathcal{S}_\delta)$, instead of $\mathrm{traces}_\omega(\mathcal{S}_\delta)$, so as to return all traces of $\mathcal{S}_\delta$ whose length is exactly b. Since $\mathcal{S}_\delta$ is finite, acc_b can indeed be implemented.

We get a bounded-depth test generation algorithm DTG_b by replacing every occurrence of acc in DTG by acc_b and by forcing case 1 when and only when $|h| = b$. Since acc_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 (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 $\mathcal{S}_\delta$ does. Therefore, we have the following variation of Lemma 5.4.

LEMMA 5.7. *Let $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathrm{i}}, \kappa_{\mathrm{o}} \rangle$ be a contract and $\mathcal{R}$ the reference implementation for C. For all $b > 0$ and $h \in (\mathsf{In} \cup \mathsf{Out}_\delta)^*$ with $|h| < b$, $\mathrm{acc}_b^{(C)}(h) \supseteq \mathrm{out}^{(\mathcal{R})}(\{h\})$.*

PROOF. From Lemma 5.4 we get that $\mathrm{acc}^{(C)}(h) = \mathrm{out}^{(\mathcal{R})}(\{h\})$, hence it suffices to show that $\mathrm{acc}^{(C)}(h) \subseteq \mathrm{acc}_b^{(C)}(h)$. Let $o \in \mathrm{acc}^{(C)}(h)$. To show that $o \in \mathrm{acc}_b(h)$ we may assume an arbitrary $\sigma_i \in \mathrm{traces}_b(\mathcal{S}_\delta)\!\downarrow_i$ with $\mathcal{V}(|h+1|, \sigma_i, h \cdot o)$ (notice that $|h| + 1 \leq b$). $\sigma_i \in \mathrm{traces}_b(\mathcal{S}_\delta)\!\downarrow_i$ implies that there is some $\sigma_{io} \in \mathrm{traces}_b(\mathcal{S}_\delta)$ with $\sigma_{io}\!\downarrow_i = \sigma_i$. By Def. 3.2, there is an infinite trace $\hat{\sigma}_{io} \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$ with $\hat{\sigma}_{io}[..b] = \sigma_{io}$. $\mathcal{V}(|h| + 1, \hat{\sigma}_{io}, h \cdot o)$ still holds, as $|h| + 1 \leq b = |\sigma_{io}|$ and $\hat{\sigma}_{io}[..|h| + 1]\!\downarrow_i = \sigma_i$. From $o \in \mathrm{acc}^{(C)}(h)$ and eq. (2) we get for $\hat{\sigma}_{io}$ and $\mathcal{V}(|h+1|, \hat{\sigma}_{io}, h \cdot o)$ a trace $\hat{\sigma} \in \mathrm{traces}_\omega(\mathcal{S}_\delta)$ with $\hat{\sigma}\!\downarrow_i = \hat{\sigma}_{io}\!\downarrow_i$ and $d_{\mathrm{Out}_\delta}(o, \hat{\sigma}[|h| + 1]\!\downarrow_o) \leq \kappa_{\mathrm{o}}$. Let $\sigma = \hat{\sigma}[..|h| + 1]$, then $\sigma\!\downarrow_i = \sigma_i$ and $d_{\mathrm{Out}_\delta}(o, \sigma[|h| + 1]\!\downarrow_o) \leq \kappa_{\mathrm{o}}$. This proves $o \in \mathrm{acc}_b(h)$. □

As a consequence of Lemma 5.7, 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 5.8. *Let $\mathcal{S}$ be a finite LTS, $C = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathrm{Out}}, \kappa_{\mathrm{i}}, \kappa_{\mathrm{o}} \rangle$ a satisfiable contract and $\mathcal{I}$ an implementation. Then, if $\mathcal{I}$ is robustly clean w.r.t. C, $\mathcal{I}$* **passes** *$\mathrm{DTG}_b(\epsilon)$ 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. C, if $\neg(\mathcal{I}$ **passes** $\mathrm{DTG}_b(\epsilon))$. Let $\mathcal{I} = \langle Q, \mathsf{In}, \mathsf{Out}, \longrightarrow, q_0 \rangle$. Assume, there is some $t \in \mathrm{DTG}_b(\epsilon)$

and $q' \in Q$, such that **fail** $\parallel q'$ is reachable from $t \parallel q_0$. Let $P = \{p \in \mathsf{paths}_*(t \parallel q_0) \mid$ there is some $q' \in Q$, such that $\mathsf{last}(p) = \textbf{fail} \parallel q'\}$ the set of paths by which such a state can be reached. Let $(t \parallel q_0)\ a_0\ \cdots\ a_{n-1}\ (t_n \parallel q_n)\ a_n\ (\textbf{fail} \parallel q') = p \in P$ be the shortest of these paths, $\sigma = \mathsf{trace}(p)$ be its trace and let $h = \sigma[..|\sigma|-1]$. Since p is the shortest path in P, evidently $t_n \neq \textbf{fail}$ and hence $t_n \in \mathsf{DTG}_b(h)$. By definition of DTG_b, a transition from t_n to **fail** is only possible in cases (2) and (3) if $a_n \in \mathsf{Out}_\delta$ (notice that $\mathsf{Out} \subset \mathsf{Out}_\delta$) and $a_n \notin \mathsf{acc}_b(a_n)$. With Lemma 5.7, we get that $a_n \notin \mathsf{acc}(a_n)$. Moreover, it is easy to see from the definition of P, that if $\sigma \in \mathsf{traces}_*(t \parallel q_0)$, then also $\sigma \in \mathsf{traces}_*(q_0)$ and hence $\sigma \in \mathsf{traces}_*(\mathcal{I}_\delta)$. As $a_n \notin \mathsf{acc}(a_n)$ (although $a_n \in \mathsf{Out}_\delta$), according to eq. (2), there is some $\sigma_i \in \mathsf{traces}_\omega(\mathcal{S}_\delta)\!\downarrow_i$ with $\mathcal{V}(n+1, \sigma_i, \sigma)$, such that for all $\sigma'' \in \mathsf{traces}_\omega(\mathcal{S}_\delta)$ with $\sigma''\!\downarrow_i = \sigma_i$, it is the case that $d_{\mathsf{Out}_\delta}(a_n, \sigma''[n+1]\!\downarrow_o) > \kappa_{\mathsf{o}}$ (*). Let $\sigma_{io} \in \mathsf{traces}_\omega(\mathcal{S}_\delta)$ be such that $\sigma_{io}\!\downarrow_i = \sigma_i$. By Def. 3.2, each state in $\mathcal{I}_\delta$ can proceed by some output or quiescence. Hence, there is some infinite suffix $\sigma_+ \in {\mathsf{Out}_\delta}^\omega$ to σ, such that $(\sigma \cdot \sigma_+) \in \mathsf{traces}_\omega(\mathcal{I}_\delta)$.
Now, assume that $\mathcal{I}$ is robustly clean w.r.t. contract $\mathcal{C}$. Then we get from Def. 3.4.2 for σ_{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 $\sigma'' \in \mathsf{traces}_\omega(\mathcal{S}_\delta)$ with $\sigma''\!\downarrow_i = \sigma_{io}\!\downarrow_i = \sigma_i$ and $d_{\mathsf{Out}_\delta}((\sigma \cdot \sigma_+)[n+1]\!\downarrow_o, \sigma''[n+1]\!\downarrow_o) = d_{\mathsf{Out}_\delta}(a_n, \sigma''[n+1]\!\downarrow_o) \leq \kappa_{\mathsf{o}}$. However, this is a contradiction to (*), which concludes the proof. □

Since $\mathcal{I}$ **passes** $\mathsf{DTG}_b(\epsilon)$ implies $\mathcal{I}$ **passes** $\mathsf{DTG}_a(\epsilon)$ 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 contract $\mathcal{C}$. The transformation of the model-based testing algorithm gets its finishing touch with Algorithm 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.

Algorithm 1 Bounded-Length Doping Test (DT_b)

```
Input: history $h \in (\mathsf{In} \cup \mathsf{Out} \cup \{\delta\})^*$
Output: pass or fail
c $\leftarrow \Omega_{\mathsf{case}}(h)$   /* Pick from one of three cases */
if c = 1 or $|h| = b$ then
    return pass   /* Finish test generation */
else if c = 2 and no output from $\mathcal{I}$ is available then
    $i \leftarrow \Omega_{\mathsf{In}}(h)$   /* Pick next input */
    $i \twoheadrightarrow \mathcal{I}$   /* Forward input to IUT */
    return $\mathsf{DT}_b(h \cdot i)$   /* Continue with next step */
else if c = 3 or output from $\mathcal{I}$ is available then
    $o \twoheadleftarrow \mathcal{I}$   /* Receive output from IUT */
    if $o \in \mathsf{acc}_b(h)$ then
        return $\mathsf{DT}_b(h \cdot o)$ /* If o is foreseen by oracle continue with next step */
    else
        return fail   /* Otherwise, report test failure */
    end if
end if
```

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,

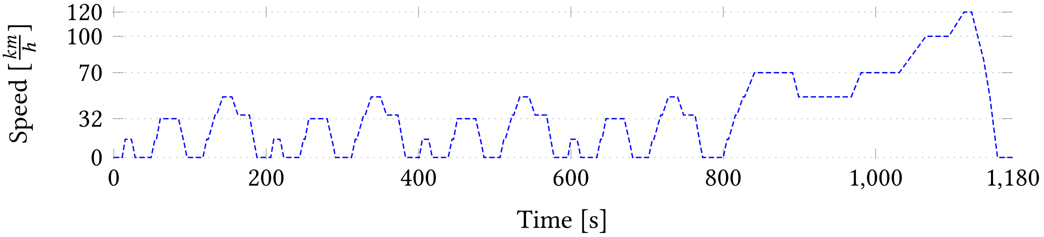


Fig. 2. NEDC speed profile.

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 acc_b. In case the output is among those foreseen by acc_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. 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 complete a journey of transformations. The bounded-depth algorithm effectively circumvents the fact that, except for $\mathcal{S}$ and $\mathcal{S}_\delta$, 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 [14], a practically usable and elegant algorithm for real-world doping tests results.

6 DIESEL DOPING TESTS

The typical problems concerning the Diesel Emissions Scandal involve legally binding frameworks for the admission of passenger cars. For Europe, this framework includes the normed emission test NEDC (New European Driving Cycle) (see Fig. 2) [34]—at the time the scandal surfaced. It is to be carried out on a chassis dynamometer and all relevant parameters are fixed by the norm, including for instance the outside temperature at which it is run. We will explain in this section how our theory and algorithm can be used in practice to detect tampered emission cleaning systems.

Inputs, Outputs, and Standard LTS. The input dimension In 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 NEDC (cf. Fig. 2), hence, we take $\mathsf{In} = \mathbb{R}$. 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 that 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 measure 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 scandal involve the measurement of the amount of NO_x per kilometre that has been 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 on-board diagnostics interface of the car (which is standardised as *OBD-II* [30]) 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) [24] can be used to compute the amount of NO_x emitted in *mg/km*. All sensor values necessary for this computation are considered being part of Out, and d_{Out} is assumed to perform the needed 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 $\text{Out} = \mathbb{R}$.

A standard LTS $\mathcal{S}$ 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 δs (because the experiment is finite), say $\sigma_S := i_1 \cdots i_{1180}\, o_S\, \delta\, \delta\, \delta \cdots$. The inputs $i_1, \cdots i_{1180}$ are given by the NEDC over its 20 minutes (1180 seconds), possibly deviating by up to $2km/h$ 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. Thus, natural 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 $d_{\mathsf{In}}(a,b) = |a-b|$ if $a, b \in \mathsf{In}$, $d_{\mathsf{In}}(-_i, -_i) = 0$ and $d_{\mathsf{In}}(a,b) = \infty$ otherwise. Similarly, $d_{\mathsf{Out}}(a,b) = |a-b|$ if $a, b \in \text{Out}$, $d_{\mathsf{Out}}(-_o, -_o) = d_{\mathsf{Out}}(\delta, \delta) = 0$ and $d_{\mathsf{Out}}(a,b) = \infty$ otherwise.

Contract. As discussed, absence or presence of software doping is understood relative to a contract that is assumed to exist between all involved parties. From the above considerations regarding input/output dimensions and distances we can piece up the blueprint of a contract, except that one needs to fix the input and output distance thresholds. In our experimental evaluation, we instantiate these with $\kappa_{\mathsf{i}} = 15km/h$, respectively $\kappa_{\mathsf{o}} = 180mg/km$. The input bound allows more variation than foreseen within the NEDC itself ($2km/h$). Notably, the output bound is very generous. It is more than the double of the currently allowed legal limit ($80mg/km$) of how much NO_x a car is allowed to emit at all. Ultimately, this induces a concrete contract $\mathcal{C} = \langle \mathcal{S}, d_{\mathsf{In}}, d_{\mathsf{Out}}, \kappa_{\mathsf{i}}, \kappa_{\mathsf{o}} \rangle$ 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.

Testing and Monitoring Framework. Algorithm 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, the use case decribed here, and further accompanying documentation is archived and publicly available at DOI 10.5281/zenodo.4709389 [7]. 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 $(\Omega_{\text{case}}, \Omega_{\text{In}})$ *test case selection*, which can also 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 differently than explained in Section 5. In practice, we cannot assume atomicity of one iteration of the test execution. This is a well-known practical impediment of model-based testing [21]. The common approach to circumvent this issue proceeds by delegating the decision of Algorithm 1 which case to pick to the driver component [14] (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 pass designated test inputs accurately to the car under test. 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.

Volkswagen Case. The first example is based on a toy implementation of an emission cleaning system that was found in several Volkswagen diesel cars [11]. To defeat the regulations, these systems contain (as part of obfuscated code) pairs of piecewise linear functions that delineate certain regions in the time-distance domain. Fig. 3 displays 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 would have travelled 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 [11], and stays like that until engine restart. We implemented a toy version of this emission cleaning approach [7], together with an implementation

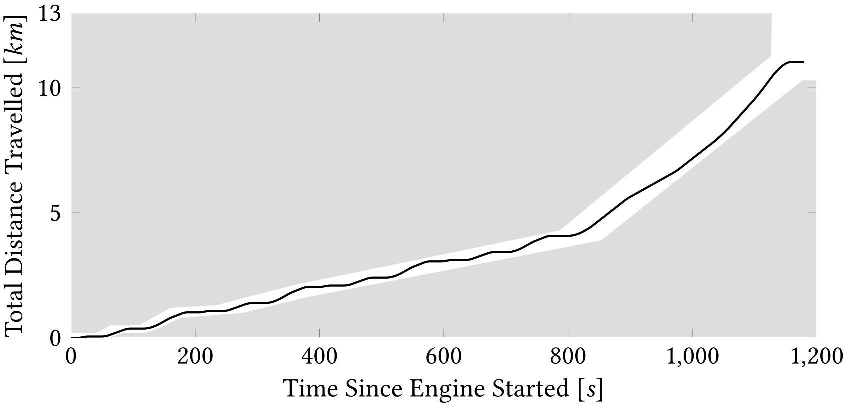


Fig. 3. White region is encoded by a pair of piecewise linear functions found in several Volkswagen ECUs. Black line represents the (imaginary) distance travelled when following the NEDC speed profile on a chassis dynamometer.

of the standard NEDC, the contract as described above, and a test case selection according to

$$\Omega_{\text{case}}(h) = \begin{cases} 2 & \text{if } |h| \leq 1180 \\ 3 & \text{if } |h| = 1181 \end{cases} \qquad \Omega_{\mathsf{In}}(h) = rand_{\mathbf{unif}}\ [\max(0, \mathsf{last}(h) - \kappa_{\mathsf{i}})\ ,\ \mathsf{last}(h) + \kappa_{\mathsf{i}})]$$

which basically stops the test after 1181 steps and otherwise meanders randomly through the speed variations possible ($rand_{\mathbf{unif}}$ implements uniform randomness). Running Algorithm DT_b with these parameters is extremely likely to lead to a **fail**, because the chance of entering a grey area early during these test is very high. In our experiments, we had to take $\kappa_{\mathsf{i}} \leq 4$ in order to see tests passing with some perceivable chance.

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 realise 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 $\kappa'_{\mathsf{i}} = \kappa_{\mathsf{i}} - \eta$, 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 Def. 3.4.
(2) Utilize 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

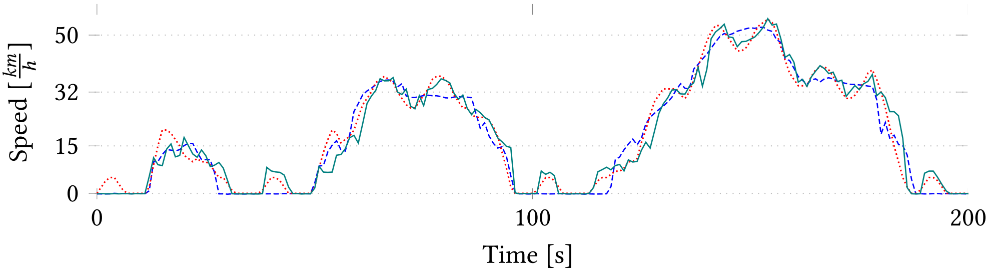


Fig. 4. Initial 200s of a SineNEDC (red, dotted), its test drive (green) and the NEDC driven (blue, dashed).

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.

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 6 car. The test cycle used in the original type approval of the car was NEDC (which corresponds to Euro 6b). 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 optimize 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. 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 $56s$, $251s$, $446s$ and $641s$). 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 below.

$$\Omega_{\mathsf{In}}(h) = \max \left\{ \begin{array}{l} 0, \\ \mathrm{NEDC}(|h|) + 5 \cdot \sin(0.5|h|)) \end{array} \right\}$$

Fig. 4 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 Fig. 5) 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).

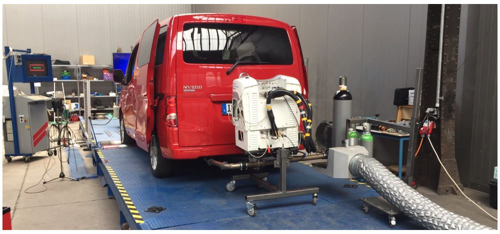


Fig. 5. Nissan NV200 Evalia on a dynamometer

	NEDC	Power	Sine
Distance [m]	11,029	11,081	11,171
Avg. Speed [km/h]	33	29	34
CO_2 [g/km]	189	186	182
NO_x [mg/km]	180	204	**584**

Table 1. Dynamometer measurements

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 line in Fig. 4 shows SineNEDC driven. The test outcomes are summarised in Table 1. They show that the amount of CO_2 for the two tests is lower than for the 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, that is 404 *mg/km* more than what we measured for the NEDC, which is a violation of the contract.

7 DISCUSSION

Related Work. The present work complements white-box approaches to software doping, like model-checking [12] or static code analysis [11] by a black-box testing approach, for which the specification is given implicitly by a contract, and usable for on-the-fly testing. Existing test frameworks like TGV [23] or TorX [14] provide support for the last step, however they fall short on scenarios where (i) the specification is not at hand and, among others, (ii) the test input is distorted in the testing process, e.g., by a human driving a car under test.

Our work is based on the definition of robust cleanness [12] which has conceptual similarities to continuity properties [8, 22] of programs. However, continuity itself does not provide an adequate guarantee of cleanness. This is because physical outputs (e.g. the amount of NO_x gas in the exhaust) 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 NO_x gas concentrations. Established notions of stability and robustness [16, 25, 27, 29] differ from robust cleanness in that the former assure the outputs (of a white-box system model) to stabilize despite transient input disturbances. Robust cleanness does not consider perturbations but (intentionally) different inputs, and needs a hyperproperty formulation.

Concluding Remarks. This work lays the theoretical foundations for black-box testing approaches geared towards uncovering doped software. As in the diesel emissions scandal—where manufacturers were forced to pay heavy fines [28] and where executives face lawsuits and possible prison sentences [5, 17]—doped behaviour is typically related to illegal behaviour.

As we have discussed, software doping analysis comes with several challenges. It can be performed (i) only after production time on the final embedded or cyber-physical product, (ii) notoriously without support by the manufacturer, and (iii) the property belongs to the class of hyperproperties with alternating quantifiers. (iv) Nondeterminism and imprecision caused by a human in-the-loop complicate doping analysis of CPS even further.

Conceptually central to the approach is a contract that is assumed to be explicitly offered by the manufacturer. The contract itself is defined by very few parameters making it easy to form an opinion about a concrete contract. And even if a manufacturer is not willing to provide such contractual guarantees, instead a contract with very generous parameters can provide convincing evidence of doping if a test uncovers the contract violation. We showed this in a real automotive example demonstrating how a legally binding reference behaviour and a contract altogether induce a finite state LTS enabling to harvest input-output conformance testing for doping tests. We developed an algorithm that can be attached directly to a system under test or in a three-step process, first generating a valid test case, which is then used to guide a human interacting with the system, thereby possibly adding distortions, followed by an a-posteriori validation of the recorded trajectory. For more effective test case selection [13, 19] we are exploring different guiding techniques [1, 2, 15] for cyber-physical systems.

Acknowledgements

We gratefully acknowledge Thomas Heinze, Michael Fries, and Peter Birtel (Automotive Powertrain Institute of HTW Saar) for sharing their automotive engineering expertise with us, and for providing the automotive test infrastructure. This work is partly supported by the ERC Grant 695614 (POWVER), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) grant 389792660 as part of TRR 248, see https://perspicuous-computing.science, by the Saarbrücken Graduate School of Computer Science, by the Sino-German CDZ project 1023 (CAP), by the Key-Area Research and Development Program Grant 2018B010107004 of Guangdong Province, by ANPCyT PICT-2017-3894 (RAFTSys), and by SeCyT-UNC 33620180100354CB (ARES).

REFERENCES

[1] Arvind S. Adimoolam, Thao Dang, Alexandre Donzé, James Kapinski, and Xiaoqing Jin. 2017. Classification and Coverage-Based Falsification for Embedded Control Systems. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (LNCS, Vol. 10426)*. Springer, 483–503. https://doi.org/10.1007/978-3-319-63387-9_24

[2] Yashwanth Annpureddy, Che Liu, Georgios E. Fainekos, and Sriram Sankaranarayanan. 2011. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In *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 (LNCS, Vol. 6605)*. Springer, 254–257. https://doi.org/10.1007/978-3-642-19835-9_21

[3] Gilles Barthe, Pedro R. D'Argenio, Bernd Finkbeiner, and Holger Hermanns. 2016. Facets of Software Doping, See [26], 601–608. http://dx.doi.org/10.1007/978-3-319-47169-3_46

[4] Kevin Baum. 2016. What the Hack Is Wrong with Software Doping?, See [26], 633–647. https://doi.org/10.1007/978-3-319-47169-3_49

[5] BBC. 2018. Audi chief Rupert Stadler arrested in diesel emissions probe. BBC, https://www.bbc.com/news/business-44517753. https://www.bbc.com/news/business-44517753 Online; accessed: 2019-01-28.

[6] Sebastian Biewer, Pedro D'Argenio, and Holger Hermanns. 2019. Doping Tests for Cyber-Physical Systems. In *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11785)*, David Parker and Verena Wolf (Eds.). Springer, 313–331. https://doi.org/10.1007/978-3-030-30281-8_18

[7] Sebastian Biewer, Pedro R. D'Argenio, and Holger Hermanns. 2021. *Doping Tests for Cyber-Physical Systems – Tool.* https://doi.org/10.5281/zenodo.4709389

[8] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity analysis of programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM, 57–70. http://doi.acm.org/10.1145/1706299.1706308

[9] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *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 (LNCS, Vol. 8414)*. Springer, 265–284. https://doi.org/10.1007/978-3-642-54792-8_15

[10] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *CSF'08*. 51–65. http://dx.doi.org/10.1109/CSF.2008.7

[11] Moritz Contag, Guo Li, Andre Pawlowski, Felix Domke, Kirill Levchenko, Thorsten Holz, and Stefan Savage. 2017. 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. IEEE Computer Society, 231–250. https://doi.org/10.1109/SP.2017.66

[12] Pedro R. D'Argenio, Gilles Barthe, Sebastian Biewer, Bernd Finkbeiner, and Holger Hermanns. 2017. Is Your Software on Dope? - Formal Analysis of Surreptitiously "enhanced" Programs. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Proceedings (LNCS, Vol. 10201)*. Springer, 83–110. https://doi.org/10.1007/978-3-662-54434-1_4

[13] R.G. de Vries. 2001. Towards Formal Test Purposes. In *Formal Approaches to Testing of Software 2001 (FATES'01) (BRICS Notes Series, NS-01-4)*. BRICS, University of Aarhus, 61–76.

[14] René G. de Vries and Jan Tretmans. 2000. On-the-fly Conformance Testing using SPIN. *STTT* 2, 4 (2000), 382–393. https://doi.org/10.1007/s100090050044

[15] Jyotirmoy V. Deshmukh, Xiaoqing Jin, James Kapinski, and Oded Maler. 2015. Stochastic Local Search for Falsification of Hybrid Systems. In *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings (LNCS, Vol. 9364)*. Springer, 500–517. https://doi.org/10.1007/978-3-319-24953-7_35

[16] Laurent Doyen, Thomas A. Henzinger, Axel Legay, and Dejan Nickovic. 2010. Robustness of Sequential Circuits. In *10th International Conference on Application of Concurrency to System Design, ACSD 2010, Braga, Portugal, 21-25 June 2010.* IEEE Computer Society, 77–84. https://doi.org/10.1109/ACSD.2010.26

[17] Jack Ewing. 2018. Ex-Volkswagen C.E.O. Charged With Fraud Over Diesel Emissions. New York Times, https://www.nytimes.com/2018/05/03/business/volkswagen-ceo-diesel-fraud.html. https://www.nytimes.com/2018/05/03/business/volkswagen-ceo-diesel-fraud.html Online; accessed: 2019-01-28.

[18] Georgios E. Fainekos and George J. Pappas. 2009. Robustness of temporal logic specifications for continuous-time signals. *Theor. Comput. Sci.* 410, 42 (2009), 4262–4291. https://doi.org/10.1016/j.tcs.2009.06.021

[19] Loe M. G. Feijs, Nicolae Goga, Sjouke Mauw, and Jan Tretmans. 2002. Test Selection, Trace Distance and Heuristics. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002, Berlin, Germany, March 19-22, 2002 (IFIP Conference Proceedings, Vol. 210)*. Kluwer, 267–282.

[20] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. 2015. Algorithms for Model Checking HyperLTL and HyperCTL*. In *CAV 2015 (LNCS, Vol. 9206)*. Springer, 30–48. http://dx.doi.org/10.1007/978-3-319-21690-4_3

[21] Alexander Graf-Brill and Holger Hermanns. 2017. Model-Based Testing for Asynchronous Systems. In *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 (Lecture Notes in Computer Science, Vol. 10471)*, Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti (Eds.). Springer, 66–82. https://doi.org/10.1007/978-3-319-67113-0_5

[22] Dick Hamlet. 2002. Continuity in sofware systems. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002.* ACM, 196–200. https://doi.org/10.1145/566172.566203

[23] Claude Jard and Thierry Jéron. 2005. TGV: theory, principles and algorithms. *STTT* 7, 4 (2005), 297–315. https://doi.org/10.1007/s10009-004-0153-x

[24] Maximilian A. Köhl, Holger Hermanns, and Sebastian Biewer. 2018. Efficient Monitoring of Real Driving Emissions. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11237)*, Christian Colombo and Martin Leucker (Eds.). Springer, 299–315. https://doi.org/10.1007/978-3-030-03769-7_17

[25] Rupak Majumdar and Indranil Saha. 2009. Symbolic Robustness Analysis. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009.* IEEE Computer Society, 355–363. https://doi.org/10.1109/RTSS.2009.17

[26] Tiziana Margaria and Bernhard Steffen (Eds.). 2016. *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Part II.* LNCS, Vol. 9953. http://dx.doi.org/10.1007/978-3-319-47169-3

[27] S. Pettersson and B. Lennartson. 1996. Stability and robustness for hybrid systems. In *Proceedings of 35th IEEE Conference on Decision and Control*, Vol. 2. 1202–1207 vol.2.

[28] Charles Riley. 2018. Volkswagen's diesel scandal costs hit $30 billion. CNN Business, https://money.cnn.com/2017/09/29/investing/volkswagen-diesel-cost-30-billion/index.html. https://money.cnn.com/2017/09/29/investing/volkswagen-diesel-cost-30-billion/index.html Online; accessed: 2019-01-28.

[29] Paulo Tabuada, Ayca Balkan, Sina Y. Caliskan, Yasser Shoukry, and Rupak Majumdar. 2012. Input-output robustness for discrete systems. In *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.* ACM, 217–226. http://doi.acm.org/10.1145/2380356.2380396

[30] The European Parliament and the Council of the European Union. 1998. Directive 98/69/EC of the European Parliament and of the Council. *Official Journal of the European Communities* (1998). http://eur-lex.europa.eu/LexUriServ/

LexUriServ.do?uri=CELEX:31998L0069:EN:HTML

[31] Jan Tretmans. 1992. *A formal approach to conformance testing*. Ph.D. Dissertation. University of Twente, Enschede, Netherlands. http://purl.utwente.nl/publications/58114

[32] Jan Tretmans. 1996. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems* 29, 1 (1996), 49–79. https://doi.org/10.1016/S0169-7552(96)00017-7

[33] Jan Tretmans. 2008. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers (LNCS, Vol. 4949)*. Springer, 1–38. https://doi.org/10.1007/978-3-540-78917-8_1

[34] United Nations. 2013. UN Vehicle Regulations - 1958 Agreement, Revision 2, Addendum 100, Regulation No. 101, Revision 3 — E/ECE/324/Rev.2/Add.100/Rev.3. http://www.unece.org/trans/main/wp29/wp29regs101-120.html

[35] Rob van Glabbeek. 2001. The Linear Time-Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes. In *Handbook of Process Algebra*. Elsevier, 3–99.