Is Your Software on Dope?
Formal Analysis of Surreptitiously "enhanced" Programs

Pedro R. D'Argenio[1,2](✉), Gilles Barthe[3], Sebastian Biewer[2], Bernd Finkbeiner[2], and Holger Hermanns[2]

[1] FaMAF, Universidad Nacional de Córdoba – CONICET, Córdoba, Argentina
dargenio@famaf.unc.edu.ar
[2] Computer Science, Saarland Informatics Campus, Saarland University, Saarbrücken, Germany
[3] IMDEA Software, Madrid, Spain

Abstract. Usually, it is the software manufacturer who employs verification or testing to ensure that the software embedded in a device meets its main objectives. However, these days we are confronted with the situation that economical or technological reasons might make a manufacturer become interested in the software slightly deviating from its main objective for dubious reasons. Examples include lock-in strategies and the NO$_x$ emission scandals in automotive industry. This phenomenon is what we call *software doping*. It is turning more widespread as software is embedded in ever more devices of daily use.

The primary contribution of this article is to provide a hierarchy of simple but solid formal definitions that enable to distinguish whether a program is *clean* or *doped*. Moreover, we show that these characterisations provide an immediate framework for analysis by using already existing verification techniques. We exemplify this by applying self-composition on sequential programs and model checking of HyperLTL formulas on reactive models.

1 Introduction

The Volkswagen exhaust emissions scandal [43] has put *software doping* in the spotlight: Proprietary embedded control software does not always exploit functionality offered by a device in the best interest of the device owner. Instead the software may be tweaked in various manners, driven by interests different from those of the owner or of society. This is indeed a common characteristics for the manner how different manufacturers circumvented [12,25] the diesel emission regulations around the world. The exhaust software was manufactured in such a way that it heavily polluted the environment, unless the software detected the

This work is partly supported by the ERC Grants 683300 (OSARES) and 695614 (POWVER), by the Saarbrücken Graduate School of Computer Science, by the Sino-German CDZ project 1023 (CAP), by ANPCyT PICT-2012-1823, by SeCyT-UNC 05/BP12 and 05/B497, and by the Madrid Region project S2013/ICE-2731 N-GREENS Software-CM.

H. Yang (Ed.): ESOP 2017, LNCS 10201, pp. 83–110, 2017.
DOI: 10.1007/978-3-662-54434-1_4

car to be (likely) fixed on a particular test setup used to determine the NO_x footprint data officially published. Phenomena resembling the emission scandal have also been reported in the context of smart phone designs [2], where software was tailored to perform better when detecting it was running a certain benchmark, and otherwise running in lower clock speed. Another smart phone case, disabling the phone [11] via a software update after "non-authorised" repair, has later been undone [36].

Usually, it is the software manufacturer who employs verification or testing to ensure that the software embedded in a device meets its main objectives. However, these days we are confronted with the situation that economical or technological reasons might make a manufacturer become interested in the software slightly deviating from its main objective for dubious reasons. This phenomenon is what we call *software doping*. It is turning more widespread as software is embedded in ever more devices of daily use.

The simplest and likely most common example of software doping (effectuating a customer lock-in strategy [3]) is that of ink printers [42] refusing to work when supplied with a toner or ink cartridge of a third party manufacturer [41], albeit being technically compatible. Similarly, cases are known where laptops refuse to charge [40] the battery if connected to a third-party charger. More subtle variations of this kind of doping just issue a warning message about the risk of using a "foreign" cartridge [20]. In the same vein, it is known that printers emit "low toner" warnings [33] earlier than needed, so as to drive or force the customer into replacing cartridges prematurely. Moreover, there are allegations that software doping has occurred in the context of electronic-voting so as to manipulate the outcome [1]. Tampering with voting machines has been proved a relatively easy task [21]. Common to all these examples is that the software user has little or no control over its execution, and that the functionality in question is against the interests of user or of society.

Despite the apparently pervasive presence of software doping, a systematic investigation or formalisation from the software engineering perspective is not existing. Fragmentary attention has been payed in the security domain with respect to cryptographic protections being sabotaged by insiders [37]. Typical examples are the many known backdoors, including the prominent dual EC deterministic random bit generator standardised by NIST [14]. Software doping however goes far beyond inclusion of backdoors.

Despite the many examples, it is not at all easy to provide a crisp characterisation of what constitutes software doping. This paper explores this issue, and proposes a hierarchy of formal characterisations of software doping. We aim at formulating and enforcing rigid requirements on embedded software driven by public interest, so as to effectively ban software doping. In order to sharpen our intuition, we offer the following initial characterisation attempt [5].

> A software system is doped if the manufacturer has included a hidden functionality in such a way that the resulting behaviour intentionally favors a designated party, against the interest of society or of the software licensee. (1)

So, a doped software induces behaviour that can not be justified by the interest of the licensee or of society, but instead serves another usually hidden interest. It thereby favors a certain brand, vendor, manufacturer, or other market participant. This happens intentionally, and not by accident. However, the question whether a certain behaviour is intentional or not is very difficult to decide. To illustrate this, we recall that the above mentioned smart phone case, to be specific the iPhone-6, where "non-authorised" repair rendered the phone unusable [11] after an iOS update, seemed to be intentional when it surfaced, but was actually tracked down to a software glitch of the update and fixed later. Notably, if the iOS designers would have had the particular intention to mistreat licensees who went elsewhere for repair, the same behaviour could well have qualified as software doping in the above sense (1). As a result, we will look at software doping according to the above characterisation, keeping in mind the possibility of intentionality but not aiming to capture it in a precise manner.

In our work, we use concise examples that are directly inspired by the real cases reviewed above. They motivate our hierarchy of formal characterisations of *clean* or *doping-free* software.

A core observation will be that software doping can be characterised by considering the program if started from two different but compatible initial states. If the obtained outputs are not compatible, then this implies that the software is *doped*. Thinking in terms of the printer, one would expect that printing with different but compatible cartridges would yield the same printout without any alteration in the observed alerts. As a consequence, the essence of the property of being clean can be cast as a *hyperproperty* [16,17].

We first explore characterisations on sequential software (Sect. 2). We introduce a characterisation that ensures the proper functioning of the system whenever it is confined to standard parameters and inputs. Afterwards, we give two other characterisations that limit the behaviour of the system whenever it goes beyond such standard framework. We then revise these characterisations so as to apply to reactive non-deterministic systems (Sect. 3).

Traditionally hyperproperties require to be analysed in an *ad-hoc* manner depending on the particular property. However, a general framework is provided by techniques based on, e.g., self-composition techniques [6] or specific logic such as HyperLTL [15]. Indeed, we show (Sect. 4) how these properties can be analysed using self-composition on deterministic programs, particularly using weakest precondition reasoning [18], and we do the same (Sect. 5) for reactive systems using HyperLTL. In both settings we demonstrate principal feasibility by presenting verification studies of simple but representative examples.

2 Software Doping on Sequential Programs

Think of a program as a function that accepts some initial parameters and, given some inputs, produces some outputs, maybe in a non-deterministic manner. Thus, a parameterised sequential non-deterministic program is a function $S : \mathsf{Param} \to \mathsf{In} \to 2^{\mathsf{Out}}$, where Param is a set of parameters, each one of them fixing a particular instance of the program S, and In and Out being respectively the sets of inputs accepted by S and outputs produced by S. Notice that for a fixed parameter p and input $\mathsf{i} \in \mathsf{In}$, the run of program $S(\mathsf{p})(\mathsf{i})$ may give a set of possible outputs.

```
procedure PRINTER($cartridge\_info$)
    if TYPE($cartridge\_info$) $\in$ Compatible
    then
        READ($document$)
        PRINT($stdout, document$)
    else
        TURNON($alert\_led$)
    end if
end procedure
```

Fig. 1. A simple printer.

```
procedure PRINTER($cartridge\_info$)
    if BRAND($cartridge\_info$) $=$ $my\text{-}brand$
    then
        READ($document$)
        PRINT($stdout, document$)
    else
        TURNON($alert\_led$)
    end if
end procedure
```

Fig. 2. A doped printer.

To understand a first possible definition, consider the program embedded in a printer (a simple abstraction is given in Fig. 1). This program may check compatibility of the ink or toner cartridge and print whenever the cartridge is compatible. In this case, we can think of the program PRINTER as a function parameterised with the information on the cartridge, that receives a document as input and produces a sequence of pages as outputs whenever the cartridge is compatible, otherwise it turns on an alert led. In this setting, we expect that the printer shows the same input-output behaviour for any compatible cartridge.

A printer manufacturer may manipulate this program in order to favour its own cartridge brand. An obvious way is displayed in Fig. 2. This is a sort of discrimination based on parameter values. Therefore, a first approach to characterising a program as *clean* (or *doping-free*) is that it should behave in a similar way for all parameters of interest. By "similar behaviour" we mean that the visible output should be the same for any given input in two different instances of the same (parameterised) program. Also, by "all parameters of interest", we refer to all parameter values we are interested in. In the case of the printer, we expect that it works with any *compatible* cartridge, but not with every cartridge. Such a compatibility domain defines a first scope within which a software is evaluated to be clean or doped.

Formally, if $\mathsf{PIntrs} \subseteq \mathsf{Param}$, we could say that a parameterised program S is clean (or doping-free) if for all pairs of parameters of interest $\mathsf{p}, \mathsf{p}' \in \mathsf{PIntrs}$ and input $\mathsf{i} \in \mathsf{In}$, $S(\mathsf{p})(\mathsf{i}) = S(\mathsf{p}')(\mathsf{i})$. Thus, the program of Fig. 1 satisfies this constraint whenever $\mathsf{Compatible}$ is the set of parameters of interest (i.e. $\mathsf{Compatible} = \mathsf{PIntrs}$). Instead, the program of Fig. 2 would be rejected as *doped* by the previous definition.

We could imagine, nonetheless, that the printer manufacturer may like to provide extra functionalities for its own product which is outside of the standard for compatibility. For instance (and for the sake of this discussion) suppose the printer manufacturer develops a new file format that is more efficient or versatile at the time of printing, but this requires some new technology on the cartridge (we could compare this to the introduction of the postscript language when standard printing was based on dots or ASCII code). The manufacturer still wants to provide the usual functionality for standard file formats that work with standard compatible cartridges and comes up with the program of Fig. 3. Notice that this program does not conform to the specification of a clean program as given above since it behaves differently when a document of the new (non-standard) type is given. This is clearly not in the spirit of the program in Fig. 3 which is actually conforming to the expected requirements.

Thus, our first definition states that a program is *clean* if, for any possible instance from the set of parameters of interest, it exhibits the same visible outputs when supplied with the same input, provided this input complies with a given standard. Formally, we assume a set $\mathsf{PIntrs} \subseteq \mathsf{Param}$ of parameters of interest and a set $\mathsf{StdIn} \subseteq \mathsf{In}$ of standard inputs and propose the following definition.

```
procedure PRINTER($cartridge\_info$)
    if TYPE($cartridge\_info$) $\in$ Compatible then
        READ($document$)
        if ($\neg$NEWTYPE($document$)
            $\vee$ SUPPORTSNEWTYPE($cartridge\_info$))
        then
            PRINT($stdout$, $document$)
        else
            TURNON($alert\_led$)
        end if
    else
        TURNON($alert\_signal$)
    end if
end procedure
```

Fig. 3. A clean printer.

Definition 1. *A parameterised program S is* clean *(or* doping-free*) if for all pairs of parameters of interest* $\mathsf{p}, \mathsf{p}' \in \mathsf{PIntrs}$ *and input* $\mathsf{i} \in \mathsf{In}$*, if* $\mathsf{i} \in \mathsf{StdIn}$ *then* $S(\mathsf{p})(\mathsf{i}) = S(\mathsf{p}')(\mathsf{i})$*. If the program is not clean we will say that it is* doped*.*

The characterisation given above is based on a comparison of the behaviour of two instances of a program, each of them responding to different parameter values within PIntrs. A second, different characterisation may instead require to compare a reference specification capturing the essence of clean behaviour against any possible instance of the program. The first approach seems more general than the second one in the sense that the specification could be considered as one of the possible instances of the (parameterised) program. However, we can consider a distinguished parameter $\hat{\mathsf{p}}$ so that the instance $S(\hat{\mathsf{p}})$ is actually the specification of the program, in which case, both definitions turn out to be equivalent. In any case, it is important to observe that the specification may not be available since it is also made by the software manufacturer, and only the expected requirements may be known.

We remark that Definition 1 entails the existence of a contract which defines the set of parameters of interest and the set of standard inputs. In fact, Definition 1 only asserts doping-freedom if the program is well-behaved within such a contract, namely, as long as the parameters are within PIntrs and inputs are within StdIn. A behaviour outside this realm is deemed immediately correct since it is of no interest. This view results too mild in some cases where the change of behaviour of a program between a standard input and a non-standard but yet not-so-different input is extreme.

Consider the electronic control unit (ECU) of a diesel vehicle, in particular its exhaust emission control module. For diesel engines, the controller injects a certain amount of a specific fluid (an aqueous urea solution) into the exhaust pipeline in order to lower mono-nitrogen oxides (NO_x) emissions. We simplify this control problem to a minimal toy example. In Fig. 4 we display a function that reads the *throttle* position and calculates which is the dose of diesel exhaust fluid (DEF) (stored in *def_dose*) that should be injected to reduce the NO_x emission. The last line of the program precisely models the NO_x emission by storing it in the output variable *NOx* after a (made up) calculation directly depending on the *throttle* value and inversely depending on the *def_dose*.

```
procedure EmissionControl()
    read(throttle)
    def_dose := SCRModel(throttle)
    NOx := throttle^3 / (k · def_dose)
end procedure
```

Fig. 4. A simple emission control.

The Volkswagen emission scandal arose precisely because their software was instrumented so that it works as expected *only if* operating in or very close to the lab testing conditions [19]. For our simplified example, this behaviour is exemplified by the algorithm of Fig. 5. Of course, the real case was less simplistic. Precisely, in this setting, the lab conditions define the set of standard inputs, i.e., the set StdIn is actually ThrottleTestValues and, as a consequence, a software like this one trivially meets the characterisation of *clean* given in Definition 1. However, this unit is intentionally programmed to defy the regulations when being unobserved and hence it falls directly within our intuition of what a doped software is (see (1)).

```
procedure EmissionControl()
    read(throttle)
    if throttle ∈ ThrottleTestValues then
        def_dose := SCRModel(throttle)
    else
        def_dose := altSCRModel(throttle)
    end if
    NOx := throttle^3 / (k · def_dose)
end procedure
```

Fig. 5. A doped emission control.

The spirit of the emission tests is to verify that the amount of NO_x in the car exhaust gas does not exceed a given threshold *in general*. Thus, one would expect that if the input values of the EmissionControl function deviates 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". A similar rationale could be applied for regulation of other systems such as speed limit controllers in scooters and electric bikes.

Therefore, we need to introduce two notions of distance $d_{\mathsf{In}} : (\mathsf{In} \times \mathsf{In}) \to \mathbb{R}_{\geq 0}$ and $d_{\mathsf{Out}} : (\mathsf{Out} \times \mathsf{Out}) \to \mathbb{R}_{\geq 0}$ on inputs and outputs respectively. In principle, we do not require them to be metrics, but they need to be commutative and satisfy that $d_{\mathsf{In}}(\mathsf{i}, \mathsf{i}) = d_{\mathsf{Out}}(\mathsf{o}, \mathsf{o}) = 0$ for all $\mathsf{i} \in \mathsf{In}$ and $\mathsf{o} \in \mathsf{Out}$. Since programs are non-deterministic, we need to lift the output distance to sets of outputs and for that we will use the Hausdorff lifting which, as we will see, is exactly what we need. Given a distance d, the Hausdorff lifting $\mathcal{H}(d)$ is defined by

$$\mathcal{H}(d)(A,B) = \max\left\{\sup_{a\in A}\inf_{b\in B} d(a,b), \sup_{b\in B}\inf_{a\in A} d(a,b)\right\} \qquad (2)$$

Based on this, we provide a new definition 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.

Definition 2. *A parameterised program S is* robustly clean *if for all pairs of parameters of interest* $\mathsf{p},\mathsf{p}' \in \mathsf{PIntrs}$ *and inputs* $\mathsf{i},\mathsf{i}' \in \mathsf{In}$*, if* $\mathsf{i} \in \mathsf{StdIn}$ *is a standard input and* $d_{\mathsf{In}}(\mathsf{i},\mathsf{i}') \leq \kappa_{\mathsf{i}}$ *then* $\mathcal{H}(d_{\mathsf{Out}})(S(\mathsf{p})(\mathsf{i}), S(\mathsf{p}')(\mathsf{i}')) \leq \kappa_{\mathsf{o}}$.

Requiring that $\mathcal{H}(d_{\mathsf{Out}})(S(\mathsf{p})(\mathsf{i}), S(\mathsf{p}')(\mathsf{i}')) \leq \kappa_{\mathsf{o}}$ is equivalent to demand that

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

Notice that this is what we actually need for the non-deterministic case: each output of one of the program instances should be matched within "reasonable distance" by some output of the other program instance.

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 $S(\mathsf{p}')(\mathsf{i}')$ will be requested to be within κ_{o} distance of some output generated by the respective execution induced by a standard input. In addition, notice that if the program S is deterministic and terminating we could simply write that $d_{\mathsf{Out}}(S(\mathsf{p})(\mathsf{i}), S(\mathsf{p}')(\mathsf{i}')) \leq \kappa_{\mathsf{o}}$.

The concept of robustly clean programs generalises that of clean programs. Indeed, by taking $d_{\mathsf{In}}(\mathsf{i},\mathsf{i}) = 0$ and $d_{\mathsf{In}}(\mathsf{i},\mathsf{i}') > \kappa_{\mathsf{i}}$ for all $\mathsf{i} \neq \mathsf{i}'$, and $d_{\mathsf{Out}}(\mathsf{o},\mathsf{o}) = 0$ and $d_{\mathsf{Out}}(\mathsf{o},\mathsf{o}') > \kappa_{\mathsf{o}}$ for all $\mathsf{o} \neq \mathsf{o}'$, we see that Definition 1 is subsumed by Definition 2. Also, notice that the tolerance parameters κ_{i} and κ_{o} are values that should be provided as well as the notions of distance d_{In} and d_{Out}, and, together with the set PIntrs of parameters of interest and the set StdIn of standard inputs, are part of the contract that ensures that the software is robustly clean. Moreover, the limitation to these tolerance values has to do with the fact that, beyond it, particular requirements (e.g. safety) may arise. For instance, a smart battery may stop accepting charge if the current emitted by a standardised but foreign charger is higher than "reasonable" (i.e. than the tolerance values); however, it may still proceed in case it is dealing with a charger of the same brand for which it may know that it can resort to a customised protocol allowing ultra-fast charging in a safe manner.

Example 3. We remark that Definition 2 will actually detect as doped the program of Fig. 5 for appropriate distances d_{In} and d_{Out} and tolerance parameters κ_{i} and κ_{o}. Indeed, suppose that $\text{SCRModel}(x) = x^2$, $\text{altSCRModel}(x) = x$, and $\mathsf{k} = 2$. To check if the programs are robustly clean, take $\mathsf{In} = (0, 2]$ (these are the values that variable *throttle* takes), $\mathsf{StdIn} = (0, 1]$, let the distances d_{In} and d_{Out} be the absolute values of the differences of the values that take *throttle* and NOx, respectively, and let $\kappa_{\mathsf{i}} = 2$ and $\kappa_{\mathsf{o}} = 1$. With this setting, the program of Fig. 4 is robustly clean while the program of Fig. 5 is not.

Definition 2 can be further generalised by adjusting to a precise desired granularity given by a function $f : \mathbb{R} \to \mathbb{R} \cup \{\infty\}$ that relates the distances of the input with the distances of the outputs as follows.

Definition 4. *A parameterised program S is f-*clean *if for all pairs of parameters of interest* $\mathsf{p}, \mathsf{p}' \in \mathsf{PIntrs}$ *and inputs* $\mathsf{i}, \mathsf{i}' \in \mathsf{In}$*, if $i \in \mathsf{StdIn}$ is a standard input then $\mathcal{H}(d_{\mathsf{Out}})(S(\mathsf{p})(\mathsf{i}), S(\mathsf{p}')(\mathsf{i}')) \leq f(d_{\mathsf{In}}(\mathsf{i}, \mathsf{i}'))$.*

Like for Definition 2, the definition of f-clean does not require i' to satisfy StdIn. Moreover, notice that it is important that f can map into ∞, in which case it means that input i' becomes irrelevant to the property. Also here the Hausdorff distance is elegantly encoding the requirement that

1. for all $\mathsf{o} \in S(\mathsf{p})(\mathsf{i})$ there exists $\mathsf{o}' \in S(\mathsf{p}')(\mathsf{i}')$ s.t. $d_{\mathsf{Out}}(\mathsf{o}, \mathsf{o}') \leq f(d_{\mathsf{In}}(\mathsf{i}, \mathsf{i}'))$, and
2. for all $\mathsf{o}' \in S(\mathsf{p}')(\mathsf{i}')$ there exists $\mathsf{o} \in S(\mathsf{p})(\mathsf{i})$ s.t. $d_{\mathsf{Out}}(\mathsf{o}, \mathsf{o}') \leq f(d_{\mathsf{In}}(\mathsf{i}, \mathsf{i}'))$.

This definition is strictly more general than Definition 2, which can be seen by taking f defined by $f(x) = \kappa_{\mathsf{o}}$ whenever $x \leq \kappa_{\mathsf{i}}$ and $f(x) = \infty$ otherwise. (Notice here the use of ∞.) Also, if the program S is deterministic, we could simply require that $d_{\mathsf{Out}}(S(\mathsf{p})(\mathsf{i}), S(\mathsf{p}')(\mathsf{i}')) \leq f(d_{\mathsf{In}}(\mathsf{i}, \mathsf{i}'))$.

In this new definition, the bounding function f, together with the distances d_{In} and d_{Out}, the set PIntrs of parameters of interest and the set StdIn of standard inputs, are part of the contract that ensures that the software is f-clean.

Example 5. For the example of the emission control take the setting as in Example 3 and let $f(x) = x/2$. Then the program of Fig. 4 is f-clean while the program of Fig. 5 is not.

We remark that the notion of f-clean strictly relates the distance of the input values with the distance of the output values. Thus, e.g., the accepted distance on the outputs may grow according the distance of the input grows. Compare it to the notion of robustly clean in which the accepted distance on the outputs is only bounded by a constant (κ_{o}), regardless of the proximity of the inputs (which is only observed w.r.t. to constant κ_{i}).

3 Software Doping on Reactive Programs

Though we use the Volkswagen ECU case study as motivation for introducing Definitions 2 and 4, this program is inherently reactive: the DEF dosage depends not only of the current inputs but also on the current state (which in turn is set according to previous inputs). Therefore, in this section, we revise the definitions given in the previous section within the framework of reactive programs.

We consider a parameterised reactive program as a function $S : \mathsf{Param} \to \mathsf{In}^{\omega} \to 2^{(\mathsf{Out}^{\omega})}$ so that any instance of the program reacts to the k-th input in the input sequence producing the k-th output in each respective output sequence. Thus each instance of the program can be seen, for instance, as a (non-deterministic) Mealy or Moore machine. In this setting, we require that $\mathsf{StdIn} \subseteq \mathsf{In}^{\omega}$. Thus, the definition of a clean reactive program strongly resembles Definition 1.

Definition 6. *A parameterised reactive program S is* clean *if for all pairs of parameters of interest* $\mathsf{p}, \mathsf{p}' \in \mathsf{PIntrs}$ *and input* $\mathsf{i} \in \mathsf{In}^\omega$*, if* $\mathsf{i} \in \mathsf{StdIn}$ *then* $S(\mathsf{p})(\mathsf{i}) = S(\mathsf{p}')(\mathsf{i})$*.*

Naively, we may think that the definition of robustly clean may be also reused as given in Definition 2 by considering metrics on ω-traces. Unfortunately this definition does not work as expected: suppose two input sequences in In^ω that only differ by a single input in some late k-th position but originates a distance larger than κ_i. Now the program under study may become clean even if the respective outputs differ enormously at an early k'-th position ($k' < k$). Notice that there is no justification for such early difference on the output, since the input sequences are the same up to position k'.

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

Therefore, we consider distances on finite traces: $d_\mathsf{In} : (\mathsf{In}^* \times \mathsf{In}^*) \to \mathbb{R}_{\geq 0}$ and $d_\mathsf{Out} : (\mathsf{Out}^* \times \mathsf{Out}^*) \to \mathbb{R}_{\geq 0}$. Now, we provide a definition of robustly clean on reactive programs that ensures that, as long as all j-th prefix of a given input sequence, with $j \leq k$, are within κ_i distance, the k-th prefix of the output sequence are within κ_o distance, for any $k \geq 0$. In the following definition, we denote with $\mathsf{i}[..k]$ the k-th prefix of the input sequence i (and similarly for output sequences).

Definition 7. *A parameterised reactive program S is* robustly clean *if for all pairs of parameters of interest* $\mathsf{p}, \mathsf{p}' \in \mathsf{PIntrs}$ *and input sequences* $\mathsf{i}, \mathsf{i}' \in \mathsf{In}^\omega$*, if* $\mathsf{i} \in \mathsf{StdIn}$ *then, for all $k \geq 0$ the following must hold*

$$(\forall j \leq k : d_\mathsf{In}(\mathsf{i}[..j], \mathsf{i}'[..j]) \leq \kappa_\mathsf{i}) \to \mathcal{H}(d_\mathsf{Out})(S(\mathsf{p})(\mathsf{i})[..k], S(\mathsf{p}')(\mathsf{i}')[..k]) \leq \kappa_\mathsf{o},$$

where $S(\mathsf{p})(\mathsf{i})[..k] = \{\mathsf{o}[..k] \mid \mathsf{o} \in S(\mathsf{p})(\mathsf{i})\}$ *and similarly for* $S(\mathsf{p}')(\mathsf{i}')[..k]$*.*

By having as precondition that $d_\mathsf{In}(\mathsf{i}[..j], \mathsf{i}'[..j]) \leq \kappa_\mathsf{i}$ *for all* $j \leq k$, this definition considers the fact that once one instance of the program deviates too much from the normal behaviour (i.e. beyond κ_i distance at the input), this instance is not obliged any longer to meet (within κ_o distance) the output, even if later inputs get closer again. This enables robustly clean programs to stop if an input outside the standard domain may result harmful for the system. Also, notice that, by considering the conditions through all k-th prefixes the definition encompasses the safety nature of the robustly cleanness property.

Example 8. A slightly more realistic version of the emission control system on the ECU is given in Fig. 6. It is a closed loop where the calculation of the DEF dosage also depends on the previous reading of NO$_x$. Moreover, the DEF dosage does not affect deterministically in the NO$_x$ emission. Instead, there is a margin of error on the NO$_x$ emission which is represented by the factor λ and the non-deterministic assignment of variable *NOx* in the penultimate line within the loop.

This non-deterministic assignment is an (admittedly unrealistic) abstraction of the chemical reaction between the exhaust gases and the DEF dosage. Figure 7 gives the version of the emission control system instrumenting the cheating hack. We define the selective catalytic reduction (SCR) models as follows:

```
procedure EmissionControl()
    $NOx := 0$
    loop
        read($throttle$)
        $def\_dose := \text{SCRModel}(throttle, NOx)$
        $NOx :\in \left[(1-\lambda)\frac{throttle^3}{\mathsf{k}\cdot def\_dose}, (1+\lambda)\frac{throttle^3}{\mathsf{k}\cdot def\_dose}\right]$
        output($NOx$)
    end loop
end procedure
```

Fig. 6. An emission control (reactive).

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

where $\lambda = 0.1$ and $\mathsf{k} = 2$, and $\text{altSCRModel}(x, n) = x$ (i.e., it ignores the feedback of the NO_x emission resulting in the same altSCRModel as in Example 3). We also take $\mathsf{In} = (0, 2]$ (recall that these are the values that variable *throttle* takes). The idea of the feedback in SCRModel is that if the previous emission was higher than expected with the planned current dosage, then the actual current dosage is an extra λ portion above the planned dosage.

For the contract required by robustly cleanness, we let $\mathsf{StdIn} = (0, 1]^\omega$ and define $d_{\mathsf{In}}(\mathsf{i}, \mathsf{i}') = |\text{last}(\mathsf{i}) - \text{last}(\mathsf{i}')|$ and similarly $d_{\mathsf{Out}}(\mathsf{o}, \mathsf{o}') = |\text{last}(\mathsf{o}) - \text{last}(\mathsf{o}')|$, where $\text{last}(t)$ is the last element of the finite trace t. We take $\kappa_{\mathsf{i}} = 2$ and $\kappa_{\mathsf{o}} = 1.1$. ($\kappa_{\mathsf{o}}$ needs to be a little larger than in Example 3 due to the non-deterministic assignment to NOx.)

```
procedure EmissionControl()
    $NOx := 0$
    loop
        read($throttle$)
        if $throttle \in \mathsf{ThrottleTestValues}$ then
            $def\_dose := \text{SCRModel}(throttle, NOx)$
        else
            $def\_dose := \text{altSCRModel}(throttle, NOx)$
        end if
        $NOx :\in \left[(1-\lambda)\frac{throttle^3}{\mathsf{k}\cdot def\_dose}, (1+\lambda)\frac{throttle^3}{\mathsf{k}\cdot def\_dose}\right]$
        output($NOx$)
    end loop
end procedure
```

Fig. 7. A doped emission control (reactive).

In Sect. 6 we will use a model checking tool to prove that the algorithm in Fig. 6 is robustly clean, while the algorithm of Fig. 7 is not.

As before, Definition 7 can be further generalised by adjusting to a precise desired granularity given by a function $f : \mathbb{R} \to \mathbb{R} \cup \{\infty\}$ that relates the distances of the input with the distances of the outputs as follows.

Definition 9. *A parameterised reactive program S is* f-clean *if for all pairs of parameters of interest* $\mathsf{p}, \mathsf{p}' \in \mathsf{PIntrs}$ *and input sequences* $\mathsf{i}, \mathsf{i}' \in \mathsf{In}^\omega$, *if* $\mathsf{i} \in \mathsf{StdIn}$ *then for all* $k \geq 0$, $\mathcal{H}(d_{\mathsf{Out}})(S(\mathsf{p})(\mathsf{i})[..k], S(\mathsf{p}')(\mathsf{i}')[..k]) \leq f(d_{\mathsf{In}}(\mathsf{i}[..k], \mathsf{i}'[..k]))$.

Like for Definition 7, the definition of f-cleanness also considers distance on prefixes to ensure that major differences in late inputs do not impact on differences of early outputs, capturing also the safety nature of the property.

We observe that Definition 9 is more general than Definition 7. As before, define f by $f(x) = \kappa_{\mathsf{o}}$ whenever $x \leq 1$ and $f(x) = \infty$ otherwise, but also redefine the metric on the input domain as follows:

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

for all $\mathsf{i}, \mathsf{i}' \in \mathsf{In}$ and $k \geq 0$.

Example 10. For the example of the emission control take the setting as in Example 8 and let $f(x) = x/2 + 0.3$. The variation of f w.r.t. Example 5 is necessary to cope with the non-determinism introduced in these models. With this setting, in Sect. 6 we will check that the program of Fig. 6 is f-clean while the program of Fig. 7 is not.

4 Analysis Through Self-composition

In this section we will focus on sequential deterministic programs and we will see them in the usual way: as state transformers. Thus, if $\mu, \mu' : \mathsf{Var} \to \mathsf{Val}$ are states mapping the variables of a program into values within their domain, we denote with $(S, \mu) \Downarrow \mu'$ that a program S, initially taking values according to μ, executes and terminates in state μ'. We indicate with $(S, \mu) \Downarrow \perp$ that the program S starting at state μ does not terminate. As usual, we denote by $\mu \models \phi$ that a predicate ϕ holds on a state μ.

In this new setting, and restricting to deterministic programs, Definition 1 could be alternatively formulated as in Proposition 11. For this, we will assume that S contains sets of variables $\vec{x}_{\mathsf{p}}$, $\vec{x}_{\mathsf{i}}$, and $\vec{x}_{\mathsf{o}}$ which are respectively parameter variables, input variables and output variables. Moreover, let PIntrs and StdIn be predicates on states containing only program variables in $\vec{x}_{\mathsf{p}}$ and $\vec{x}_{\mathsf{i}}$, respectively. They characterise the set of parameters of interest and the set of standard inputs. Now, we can state,

Proposition 11. *A sequential and deterministic program S is clean if and only if for all states μ_1, μ_2 and μ_1' such that $\mu_1 \models \mathsf{PIntrs} \wedge \mathsf{StdIn}$, $\mu_2 \models \mathsf{PIntrs} \wedge \mathsf{StdIn}$, $\mu_1(\vec{x}_{\mathsf{i}}) = \mu_2(\vec{x}_{\mathsf{i}})$ and $(S, \mu_1) \Downarrow \mu_1'$, it holds that $(S, \mu_2) \Downarrow \mu_2'$ and $\mu_1'(\vec{x}_{\mathsf{o}}) = \mu_2'(\vec{x}_{\mathsf{o}})$ for some μ_2'.*

The proof of the proposition is straightforward since it is basically a notation change, hence we omit it. Also, notice that we omit any explicit reference to non-terminating programs. This is not necessary due to the symmetric nature of the predicates.

In the nomenclature of [7] relations

$$\begin{aligned}\mathcal{I} &= \{(\mu_1, \mu_2) \mid \mu_1 \models \mathsf{PIntrs} \wedge \mathsf{StdIn}, \\ &\qquad\qquad \mu_2 \models \mathsf{PIntrs} \wedge \mathsf{StdIn}, \text{ and } \mu_1(\vec{x}_\mathsf{i}) = \mu_2(\vec{x}_\mathsf{i})\} \\ \mathcal{I}' &= \{(\mu_1, \mu_2) \mid \mu_1(\vec{x}_\mathsf{o}) = \mu_2(\vec{x}_\mathsf{o})\}\end{aligned}$$

are called *indistinguishable criteria*[1], and if $(\mu_1, \mu_2) \in \mathcal{I}$ then we say that μ_1 and μ_2 are *$\mathcal{I}$-indistinguishable*[2]. Similarly, for $\mathcal{I}'$. Thus, Proposition 11 characterises what in [7] is called *termination-sensitive* $(\mathcal{I}, \mathcal{I}')$*-security* and, by [7, Proposition 3], the property of cleanness can be analysed using the weakest (conservative) precondition (wp) [18] through self-composition.

Proposition 12. *Let $[\vec{x}/\vec{x}']$ indicate the substitution of each variable x by variable x'. Then a deterministic program S is clean if and only if*

$$\left(\begin{array}{l}(\mathsf{PIntrs} \wedge \mathsf{StdIn}) \wedge (\mathsf{PIntrs} \wedge \mathsf{StdIn})[\vec{x}/\vec{x}'] \\ \wedge\, \vec{x}_\mathsf{i} = \vec{x}'_\mathsf{i} \wedge \mathrm{wp}(S, \mathit{true})\end{array}\right) \;\Rightarrow\; \mathrm{wp}(S; S[\vec{x}/\vec{x}'], \vec{x}_\mathsf{o} = \vec{x}'_\mathsf{o}).$$

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

With the same setting as before, and taking d_In, d_Out, κ_i and κ_o as for Definition 2, we obtain an alternative definition of robustly cleanness for deterministic programs.

Proposition 13. *A sequential and deterministic program S is robustly clean if and only if for all states μ_1, μ_2, and μ' such that $\mu_1 \models \mathsf{PIntrs} \wedge \mathsf{StdIn}$, $\mu_2 \models \mathsf{PIntrs}$, and $d_\mathsf{In}(\mu_1(\vec{x}_\mathsf{i}), \mu_2(\vec{x}_\mathsf{i})) \leq \kappa_\mathsf{i}$, the following two conditions hold:*

1. *if $(S, \mu_1) \Downarrow \mu'$, then $(S, \mu_2) \Downarrow \mu'_2$ and $d_\mathsf{Out}(\mu'(\vec{x}_\mathsf{o}), \mu'_2(\vec{x}_\mathsf{o})) \leq \kappa_\mathsf{o}$ for some μ'_2; and*
2. *if $(S, \mu_2) \Downarrow \mu'$, then $(S, \mu_1) \Downarrow \mu'_1$ and $d_\mathsf{Out}(\mu'_1(\vec{x}_\mathsf{o}), \mu'(\vec{x}_\mathsf{o})) \leq \kappa_\mathsf{o}$ for some μ'_1.*

In this case, the indistinguishability criteria are

$$\begin{aligned}\mathcal{I} &= \{(\mu_1, \mu_2) \mid \mu_1 \models \mathsf{PIntrs} \wedge \mathsf{StdIn}, \mu_2 \models \mathsf{PIntrs}, \text{ and } d_\mathsf{In}(\mu_1(\vec{x}_\mathsf{i}), \mu_2(\vec{x}_\mathsf{i})) \leq \kappa_\mathsf{i}\} \\ \mathcal{I}' &= \{(\mu_1, \mu_2) \mid d_\mathsf{Out}(\mu_1(\vec{x}_\mathsf{o}), \mu_2(\vec{x}_\mathsf{o})) \leq \kappa_\mathsf{o}\}\end{aligned}$$

Notice that $\mathcal{I}$ is not symmetric. Then the first item of Proposition 13 characterises termination-sensitive $(\mathcal{I}, \mathcal{I}')$-security while the second item characterises termination-sensitive $(\mathcal{I}^{-1}, \mathcal{I}')$-security. Using again [7, Proposition 3], the property of robustly cleanness can be analysed using wp through self-composition.

[1] In this definition, states should actually be considered as tuples of values rather than state mappings in order to exactly match the definitions of [7, Sect. 3].

[2] Also, to strictly follow notation in [7, Sect. 3] we should have written $\mu_1 \sim^{\mathcal{I}}_{id} \mu_2$ instead of $(\mu_1, \mu_2) \in \mathcal{I}$.

Proposition 14. *A deterministic program S is robustly clean if and only if*

$$\begin{aligned}&\mathsf{PIntrs} \wedge \mathsf{StdIn} \wedge \mathsf{PIntrs}[\vec{x}/\vec{x}'] \wedge d_{\mathsf{In}}(\vec{x}_{\mathsf{i}}, \vec{x}'_{\mathsf{i}}) \leq \kappa_{\mathsf{i}} \\ &\quad\Rightarrow \left(\begin{array}{l} \quad\ \mathrm{wp}(S, \textit{true}) \Rightarrow \mathrm{wp}(S; S[\vec{x}/\vec{x}'], d_{\mathsf{Out}}(\vec{x}_{\mathsf{o}}, \vec{x}'_{\mathsf{o}}) \leq \kappa_{\mathsf{o}}) \\ \wedge\ \ \mathrm{wp}(S[\vec{x}/\vec{x}'], \textit{true}) \Rightarrow \mathrm{wp}(S[\vec{x}/\vec{x}']; S, d_{\mathsf{Out}}(\vec{x}_{\mathsf{o}}, \vec{x}'_{\mathsf{o}}) \leq \kappa_{\mathsf{o}}) \end{array}\right)\end{aligned}$$

Proceeding in a similar manner, we can also obtain an alternative definition of f-cleanness for deterministic programs.

Proposition 15. *A sequential and deterministic program S is f-clean if and only if for all states μ_1, μ_2, and μ' such that $\mu_1 \models \mathsf{PIntrs} \wedge \mathsf{StdIn}$, and $\mu_2 \models \mathsf{PIntrs}$, the following two conditions hold:*

1. *if $(S, \mu_1) \Downarrow \mu'$, then $(S, \mu_2) \Downarrow \mu'_2$ and $d_{\mathsf{Out}}(\mu'(\vec{x}_{\mathsf{o}}), \mu'_2(\vec{x}_{\mathsf{o}})) \leq f(d_{\mathsf{In}}(\mu_1(\vec{x}_{\mathsf{i}}), \mu_2(\vec{x}_{\mathsf{i}}))$ for some μ'_2; and*
2. *if $(S, \mu_2) \Downarrow \mu'$, then $(S, \mu_1) \Downarrow \mu'_1$ and $d_{\mathsf{Out}}(\mu'_1(\vec{x}_{\mathsf{o}}), \mu'(\vec{x}_{\mathsf{o}})) \leq f(d_{\mathsf{In}}(\mu_1(\vec{x}_{\mathsf{i}}), \mu_2(\vec{x}_{\mathsf{i}}))$ for some μ'_1.*

Notice that the term $f(d_{\mathsf{In}}(\mu_1(\vec{x}_{\mathsf{i}}), \mu_2(\vec{x}_{\mathsf{i}}))$ appears in the conclusion of the implications of both items. This may look unexpected since it seems to be related to the input requirements rather than the output requirements, in particular because it refers to the input states. This makes this case a little less obvious than the previous one. To overcome this situation, we introduce a constant $Y \in \mathbb{R}_{\geq 0}$ which we assume universally quantified. Using this, we define the following indistinguishability criteria

$$\begin{aligned}\mathcal{I}_Y &= \{(\mu_1, \mu_2) \mid \mu_1 \models \mathsf{PIntrs} \wedge \mathsf{StdIn}, \\ &\qquad\qquad \mu_2 \models \mathsf{PIntrs},\ \text{and } f(d_{\mathsf{In}}(\mu_1(\vec{x}_{\mathsf{i}}), \mu_2(\vec{x}_{\mathsf{i}}))) = Y\} \\ \mathcal{I}'_Y &= \{(\mu_1, \mu_2) \mid d_{\mathsf{Out}}(\mu_1(\vec{x}_{\mathsf{o}}), \mu_2(\vec{x}_{\mathsf{o}})) \leq Y\}\end{aligned}$$

By using this, by Proposition 15, we have that S is f-clean if and only if for every $Y \in \mathbb{R}_{\geq 0}$, and for all states μ_1, μ_2, and μ' such that $(\mu_1, \mu_2) \in \mathcal{I}_Y$

1. if $(S, \mu_1) \Downarrow \mu'$, then $(S, \mu_2) \Downarrow \mu'_2$ and $(\mu', \mu'_2) \in \mathcal{I}'_Y$ for some μ'_2; and
2. if $(S, \mu_2) \Downarrow \mu'$, then $(S, \mu_1) \Downarrow \mu'_1$ and $(\mu'_1, \mu') \in \mathcal{I}'_Y$ for some μ'_1.

With this new definition, and taking into account again the asymmetry of $\mathcal{I}_Y$, the first item characterises termination-sensitive $(\mathcal{I}_Y, \mathcal{I}'_Y)$-security while the second one characterises termination-sensitive $(\mathcal{I}_Y^{-1}, \mathcal{I}'_Y)$-security. From this and [7, Prop. 3], the property of f-cleanness can be analysed using wp and self-composition.

Proposition 16. *A deterministic program S is f-clean if and only if for all $Y \in \mathbb{R}_{\geq 0}$*

$$\begin{aligned}&\mathsf{PIntrs} \wedge \mathsf{StdIn} \wedge \mathsf{PIntrs}[\vec{x}/\vec{x}'] \wedge f(d_{\mathsf{i}}(\vec{x}_{\mathsf{i}}, \vec{x}'_{\mathsf{i}})) = Y \\ &\quad\Rightarrow \left(\begin{array}{l} \quad\ \mathrm{wp}(S, \textit{true}) \Rightarrow \mathrm{wp}(S; S[\vec{x}/\vec{x}'], d_{\mathsf{Out}}(\vec{x}_{\mathsf{o}}, \vec{x}'_{\mathsf{o}}) \leq Y \\ \wedge\ \ \mathrm{wp}(S[\vec{x}/\vec{x}'], \textit{true}) \Rightarrow \mathrm{wp}(S[\vec{x}/\vec{x}']; S, d_{\mathsf{Out}}(\vec{x}_{\mathsf{o}}, \vec{x}'_{\mathsf{o}}) \leq Y \end{array}\right)\end{aligned}$$

$$\begin{aligned}
\mathrm{wp}(x := e, Q) &= Q[e/x] \\
\mathrm{wp}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end if}, Q) &= b \Rightarrow \mathrm{wp}(S_1, Q) \wedge \neg b \Rightarrow \mathrm{wp}(S_2, Q) \\
\mathrm{wp}(S_1; S_2, Q) &= \mathrm{wp}(S_1, \mathrm{wp}(S_2, Q)) \\
\mathrm{wp}(\textbf{while } b \textbf{ do } S \textbf{ end do}, Q) &= \exists k : k \geq 0 : H_k(Q)
\end{aligned}$$

where $H_0(Q) = \neg b \wedge Q$ and $H_{k+1}(Q) = (b \wedge \mathrm{wp}(S, H_k(Q))) \vee H_0(Q)$

Fig. 8. Equations for the wp calculus

Example 17. In this example, we use Proposition 16 to prove correct our statements in Example 3. First, we recall the definition of wp in Fig. 8, and rewrite the programs in Figs. 4 and 5 with all functions and values properly instantiated in the way we need it here (see Figs. 9 and 10).

```
$def\_dose := thrtl^2$
$NOx := thrtl^3 \;/\; (2 \cdot def\_dose)$
```

Fig. 9. Program EC.

```
if $thrtl \in \mathsf{ThrottleTestValues}$
then
    $def\_dose := thrtl^2$
else
    $def\_dose := thrtl$
end if
$NOx := thrtl^3 \;/\; (2 \cdot def\_dose)$
```

Fig. 10. Program AEC.

On the one hand, none of the programs have parameters, then $\mathsf{PIntrs} = \mathsf{true}$. On the other hand, $\mathsf{StdIn} = (thrtl \in (0, 1])$. Since $\mathrm{wp}(\textsc{ec}, \mathsf{true}) = \mathsf{true}$ we have to prove that

$$thrtl \in (0,1] \wedge \left(\tfrac{|thrtl - thrtl'|}{2} = Y\right) \Rightarrow \left(\begin{array}{c} \mathrm{wp}(\textsc{ec}; \textsc{ec}', |NOx - NOx'| \leq Y) \\ \wedge\ \mathrm{wp}(\textsc{ec}'; \textsc{ec}, |NOx - NOx'| \leq Y) \end{array}\right) \tag{3}$$

where $\textsc{ec}'$ is another instance of $\textsc{ec}$ with every program variable x renamed by x'. Moreover, function f and distances d_{In} and d_{Out} are already instantiated. It is not difficult to verify that $\mathrm{wp}(\textsc{ec}; \textsc{ec}', |NOx - NOx'| \leq Y) \equiv \left(\frac{|thrtl - thrtl'|}{2} \leq Y\right)$ and $\mathrm{wp}(\textsc{ec}'; \textsc{ec}, |NOx - NOx'| \leq Y) \equiv \left(\frac{|thrtl' - thrtl|}{2} \leq Y\right)$ from which the implication follows and hence $\textsc{ec}$ is f-clean.

For $\textsc{aec}$ we also have that $\mathrm{wp}(\textsc{aec}, \mathsf{true}) = \mathsf{true}$ and hence we have to prove a formula similar to 3. In this case, $\mathrm{wp}(\textsc{aec}; \textsc{aec}', |NOx - NOx'| \leq Y)$ is

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

The predicate is the same for $\mathrm{wp}(\textsc{aec}'; \textsc{aec}, |NOx - NOx'| \leq Y)$, since $|a - b| = |b - a|$. Then, the predicate

$$\left(thrtl \in (0,1] \wedge \tfrac{|thrtl - thrtl'|}{2} = Y\right) \Rightarrow \left(\begin{array}{c} \mathrm{wp}(\textsc{aec}; \textsc{aec}', |NOx - NOx'| \leq Y) \\ \wedge\ \mathrm{wp}(\textsc{aec}'; \textsc{aec}, |NOx - NOx'| \leq Y) \end{array}\right)$$

is equivalent to

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

which can be proved false if, e.g., $thrtl = 1$ and $thrtl' = 1.5$.

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

We could profit from [7] for the use of other verification techniques, including separation logic and model checking where the properties can be expressed in terms of LTL and CTL. Particularly, CTL permits the encoding of the full non-deterministic properties given in Sect. 2. We will not dwell on this since in the next section we explore the encoding of the reactive properties through a more general setting.

5 Analysis of Reactive Programs with HyperLTL

HyperLTL [15] is a temporal logic for the specification of hyperproperties of reactive systems. HyperLTL extends linear-time temporal logic (LTL) with trace quantifiers and trace variables, which allow the logic to refer to multiple traces at the same time. The problem of model checking a HyperLTL formula over a finite-state model is decidable [24]. In this section, we focus on reactive non-deterministic programs and use HyperLTL to encode the different definitions of clean reactive programs given in Sect. 3. In the following, we interpret a program as a set $S \subseteq (2^{\mathsf{AP}})^\omega$ of infinite traces over a set AP of atomic propositions.

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

$$\begin{array}{rcl} \psi & ::= & \exists \pi.\, \psi \mid \forall \pi.\, \psi \mid \phi \\ \phi & ::= & a_\pi \mid \neg \phi \mid \phi \vee \phi \mid \mathsf{X}\,\phi \mid \phi\, \mathsf{U}\, \phi \end{array} \tag{4}$$

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

A *trace assignment* is a partial function $\Pi : \mathcal{V} \to (2^{\mathsf{AP}})^\omega$ that assigns traces to variables. Let $\Pi[\pi \mapsto t]$ denote the same function as Π except that π is mapped to the trace t. For $k \in \mathbb{N}$, let $t[k]$, $t[k..]$, and $t[..k]$ denote respectively the k-th element of t, the k-th suffix of t, and the k-th prefix of t. The trace assignment suffix $\Pi[k..]$ is defined by $\Pi[k..](\pi) = \Pi(\pi)[k..]$. By $\Pi \models_S \psi$ we mean that formula ϕ is satisfied by the program S under the trace assignment Π. Satisfaction is recursively defined as follows.

$$\begin{array}{lll}
\Pi \models_S \exists\pi.\,\psi & \text{iff} & \Pi[\pi \mapsto t] \models_S \psi \text{ for some } t \in S \\
\Pi \models_S \forall\pi.\,\psi & \text{iff} & \Pi[\pi \mapsto t] \models_S \psi \text{ for every } t \in S \\
\Pi \models_S a_\pi & \text{iff} & a \in \Pi(\pi)[0] \\
\Pi \models_S \neg\phi & \text{iff} & \Pi \not\models_S \phi \\
\Pi \models_S \phi_1 \vee \phi_2 & \text{iff} & \Pi \models_S \phi_1 \text{ or } \Pi \models_S \phi_2 \\
\Pi \models_S \mathsf{X}\,\phi & \text{iff} & \Pi[1..] \models_S \phi \\
\Pi \models_S \phi_1\,\mathsf{U}\,\phi_2 & \text{iff} & \text{there exists } k \geq 0 \text{ s.t. } \Pi[k..] \models_S \phi_2 \text{ and} \\
& & \qquad\qquad\qquad \text{for all } 0 \leq j < k, \Pi[j..] \models_S \phi_1
\end{array}$$

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

In the following, we give the different characterisations of cleanness for reactive programs in terms of HyperLTL. For this, let $\mathsf{AP} = \mathsf{AP_p} \cup \mathsf{AP_i} \cup \mathsf{AP_o}$ where $\mathsf{AP_p}$, $\mathsf{AP_i}$, and $\mathsf{AP_o}$ are the atomic propositions that define the parameter values, the input values, and the output values respectively. Thus, we take $\mathsf{Param} = 2^{\mathsf{AP_p}}$, $\mathsf{In} = 2^{\mathsf{AP_i}}$ and $\mathsf{Out} = 2^{\mathsf{AP_o}}$. Therefore, a program $S \subseteq (2^{\mathsf{AP}})^\omega$ can be seen as a function $\hat{S} : \mathsf{Param} \to \mathsf{In}^\omega \to 2^{(\mathsf{Out}^\omega)}$ where

$$t \in S \quad \text{if and only if} \quad (t \downarrow \mathsf{AP_o}) \;\in\; \hat{S}(t[0] \cap \mathsf{AP_p})(t \downarrow \mathsf{AP_i}), \tag{5}$$

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

For the propositions appearing in the rest of this sections, we will assume that distances between traces are defined only according to its last element. That is, for the distance $d_{\mathsf{In}} : (\mathsf{In}^* \times \mathsf{In}^*) \to \mathbb{R}_{\geq 0}$ there exists a distance $\hat{d}_{\mathsf{In}} : (\mathsf{In} \times \mathsf{In}) \to \mathbb{R}_{\geq 0}$ such that $d_{\mathsf{In}}(\mathsf{i}, \mathsf{i}') = \hat{d}_{\mathsf{In}}(\mathrm{last}(\mathsf{i}), \mathrm{last}(\mathsf{i}'))$ for every $\mathsf{i}, \mathsf{i}' \in \mathsf{In}^*$, and similarly for $d_{\mathsf{Out}} : (\mathsf{Out}^* \times \mathsf{Out}^*) \to \mathbb{R}_{\geq 0}$. Let us call these type of distances *past-forgetful*. Moreover, we will need the abbreviations given in Table 1 for a clear presentation of the formulas.

The set of parameters of interest $\mathsf{PIntrs} \subseteq \mathsf{Param}$ defines a Boolean formula which we ambiguously call PIntrs. Also, we let StdIn be an LTL formula with atomic propositions in $\mathsf{AP_i}$, that is, a formula obtained with the grammar in the second line of (4) where atomic propositions have the form $a \in \mathsf{AP_i}$ (instead of a_π). Thus StdIn characterises the set of all input sequences through an LTL formula. With StdIn_π we represent the HyperLTL formula that is exactly like StdIn but where each occurrence of $a \in \mathsf{AP_i}$ has been replaced by a_π. Likewise, we let PIntrs_π represent the Boolean formula that is exactly like PIntrs with each occurrence of $a \in \mathsf{AP_p}$ replaced by a_π. We are now in conditions to state the characterisation of a clean program in terms of HyperLTL.

Table 1. Syntactic sugar for comparisons between traces

$\mathsf{p}_\pi = \mathsf{p}_{\pi'}$	iff	$\bigwedge_{a \in \mathsf{AP}_\mathsf{p}} a_\pi \leftrightarrow a_{\pi'}$
$\mathsf{i}_\pi = \mathsf{i}_{\pi'}$	iff	$\bigwedge_{a \in \mathsf{AP}_\mathsf{i}} a_\pi \leftrightarrow a_{\pi'}$
$\mathsf{o}_\pi = \mathsf{o}_{\pi'}$	iff	$\bigwedge_{a \in \mathsf{AP}_\mathsf{o}} a_\pi \leftrightarrow a_{\pi'}$
$\hat{d}_{\mathsf{In}}(\mathsf{i}_\pi, \mathsf{i}_{\pi'}) \leq \kappa_\mathsf{i}$	iff	$\bigvee_{\substack{\mathsf{i},\mathsf{i}' \in \mathsf{In} \\ \hat{d}(\mathsf{i},\mathsf{i}') \leq \kappa_\mathsf{i}}} \bigwedge_{a \in \mathsf{i}} a_\pi \wedge \bigwedge_{a \in \mathsf{i}'} a_{\pi'}$
$\hat{d}_{\mathsf{Out}}(\mathsf{o}_\pi, \mathsf{o}_{\pi'}) \leq \kappa_\mathsf{o}$	iff	$\bigvee_{\substack{\mathsf{o},\mathsf{o}' \in \mathsf{Out} \\ \hat{d}(\mathsf{o},\mathsf{o}') \leq \kappa_\mathsf{o}}} \bigwedge_{a \in \mathsf{o}} a_\pi \wedge \bigwedge_{a \in \mathsf{o}'} a_{\pi'}$
$\hat{d}_{\mathsf{Out}}(\mathsf{o}_\pi, \mathsf{o}_{\pi'}) \leq f(\hat{d}_{\mathsf{In}}(\mathsf{i}_\pi, \mathsf{i}_{\pi'}))$	iff	$\bigvee_{\substack{\mathsf{o},\mathsf{o}' \in \mathsf{Out}, \mathsf{i},\mathsf{i}' \in \mathsf{In} \\ \hat{d}(\mathsf{o},\mathsf{o}') \leq f(\hat{d}(\mathsf{i},\mathsf{i}'))}} \bigwedge_{a \in \mathsf{i}} a_\pi \wedge \bigwedge_{a \in \mathsf{i}'} a_{\pi'} \wedge \bigwedge_{a \in \mathsf{o}} a_\pi \wedge \bigwedge_{a \in \mathsf{o}'} a_{\pi'}$

Proposition 18. *A reactive program S is clean if and only if it satisfies the HyperLTL formula*

$$\begin{aligned} &\forall\pi_1.\,\forall\pi_2.\,\exists\pi_2'.(\mathsf{PIntrs}_{\pi_1} \wedge \mathsf{PIntrs}_{\pi_2} \wedge \mathsf{StdIn}_{\pi_1}) \\ &\qquad\qquad \rightarrow \big(\mathsf{p}_{\pi_2} = \mathsf{p}_{\pi_2'} \wedge \mathsf{G}(\mathsf{i}_{\pi_1} = \mathsf{i}_{\pi_2'} \wedge \mathsf{o}_{\pi_1} = \mathsf{o}_{\pi_2'})\big) \end{aligned} \tag{6}$$

As it is given, the formula actually states that

$$\forall \mathsf{p}_1 : \forall \mathsf{p}_2 : \forall \mathsf{i} : \mathsf{p}_1, \mathsf{p}_2 \in \mathsf{PIntrs} \wedge \mathsf{i} \in \mathsf{StdIn} : \hat{S}(\mathsf{p}_1)(\mathsf{i}) \subseteq \hat{S}(\mathsf{p}_2)(\mathsf{i})$$

Because of the symmetry of this definition (namely, interchanging p_1 and p_2), this is indeed equivalent to Definition 6. Notice that in (6), π_2 quantifies universally the parameter of the second instance, while π_2' represents the existence of the output sequence in such instance. The proofs of Propositions 18 to 20 follow the same structures. So we only provide the proof of Proposition 19 which is the most involved.

In fact, Proposition 19 below states the characterisation of a robustly clean program in terms of two HyperLTL formulas (or as a single HyperLTL formula by taking the conjunction).

Proposition 19. *A reactive program S is robustly clean under past-forgetful distances d_{In} and d_{Out} if and only if S satisfies the following two HyperLTL formulas*

$$\begin{aligned} &\forall\pi_1.\,\forall\pi_2.\,\exists\pi_2'. \\ &\quad (\mathsf{PIntrs}_{\pi_1} \wedge \mathsf{PIntrs}_{\pi_2} \wedge \mathsf{StdIn}_{\pi_1}) \\ &\quad \rightarrow \Big(\mathsf{p}_{\pi_2} = \mathsf{p}_{\pi_2'} \wedge \mathsf{G}(\mathsf{i}_{\pi_2} = \mathsf{i}_{\pi_2'}) \wedge \big((\hat{d}_{\mathsf{Out}}(\mathsf{o}_{\pi_1}, \mathsf{o}_{\pi_2'}) \leq \kappa_\mathsf{o})\ \mathsf{W}\ (\hat{d}_{\mathsf{In}}(\mathsf{i}_{\pi_1}, \mathsf{i}_{\pi_2'}) > \kappa_\mathsf{i})\big)\Big) \\ &\forall\pi_1.\,\forall\pi_2.\,\exists\pi_1'. \\ &\quad (\mathsf{PIntrs}_{\pi_1} \wedge \mathsf{PIntrs}_{\pi_2} \wedge \mathsf{StdIn}_{\pi_1}) \\ &\quad \rightarrow \Big(\mathsf{p}_{\pi_1} = \mathsf{p}_{\pi_1'} \wedge \mathsf{G}(\mathsf{i}_{\pi_1} = \mathsf{i}_{\pi_1'}) \wedge \big((\hat{d}_{\mathsf{Out}}(\mathsf{o}_{\pi_1'}, \mathsf{o}_{\pi_2}) \leq \kappa_\mathsf{o})\ \mathsf{W}\ (\hat{d}_{\mathsf{In}}(\mathsf{i}_{\pi_1'}, \mathsf{i}_{\pi_2}) > \kappa_\mathsf{i})\big)\Big) \end{aligned} \tag{7}$$

The difference between the first and second formula is subtle, but reflects the fact that, while the first formula has the universal quantification on the outputs of the program that takes standard input and the existential quantification on the program that may deviate, the second one works in the other way around. Thus each of the formulas capture each of the sup-inf terms in the definition of Hausdorff distance (see (2)). To notice this, follow the existentially quantified variable (π'_2 for the first formula, and π'_1 for the second one). Also, the weak until operator W has exactly the behaviour that we need to represent the interaction between the distances of inputs and the distances of outputs. The semantics of $\phi \,\mathsf{W}\, \psi$ is defined by

$$t \models \phi \,\mathsf{W}\, \psi \text{ iff } \quad \forall k \geq 0 : (\forall j \leq k : t[j..] \models \neg\psi) \rightarrow t[k..] \models \phi \tag{8}$$

Next, we prove Proposition 19.

Proof. We only prove that the first formula captures the bound on the left sup-inf term of the definition of Hausdorff distance (see eq. (2)) in Definition 7. The other condition is proved in the same way and corresponds to the other sup-inf term of the Hausdorff distance. Taking into account the semantics of the weak until operator given in Eq. 8, the semantics of HyperLTL in general and using abbreviations in Table 1, formula 7 is equivalent to the following statement

$$\begin{aligned}
&\forall t_1 \in S : \forall t_2 \in S : \exists t'_2 \in S :\\
&\quad (t_1 \models \mathsf{PIntrs} \wedge t_2 \models \mathsf{PIntrs} \wedge t_1 \models \mathsf{StdIn})\\
&\quad \rightarrow \Big((t_2[0] \cap \mathsf{AP_p}) = (t'_2[0] \cap \mathsf{AP_p}) \wedge (\forall j \geq 0 : t_2[j] \cap \mathsf{AP_i} = t'_2[j] \cap \mathsf{AP_i})\\
&\qquad\quad \wedge \forall k \geq 0 : (\forall j \leq k : \hat{d}_{\mathsf{In}}(t_1[j] \cap \mathsf{AP_i}, t'_2[j] \cap \mathsf{AP_i}) \leq \kappa_{\mathsf{i}})\\
&\qquad\qquad\qquad\qquad \rightarrow \hat{d}_{\mathsf{Out}}(t_1[k] \cap \mathsf{AP_o}, t'_2[k] \cap \mathsf{AP_o}) \leq \kappa_{\mathsf{o}}\Big)
\end{aligned}$$

By applying some definitions and notation changes, this is equivalent to

$$\begin{aligned}
&\forall t_1 \in S : \forall t_2 \in S : \exists t'_2 \in S :\\
&\quad ((t_1[0] \cap \mathsf{AP_p}) \in \mathsf{PIntrs} \wedge (t_2[0] \cap \mathsf{AP_p}) \in \mathsf{PIntrs} \wedge (t_1 \downarrow \mathsf{AP_i}) \in \mathsf{StdIn})\\
&\quad \rightarrow \Big((t_2[0] \cap \mathsf{AP_p}) = (t'_2[0] \cap \mathsf{AP_p}) \wedge (t_2 \downarrow \mathsf{AP_i}) = (t'_2 \downarrow \mathsf{AP_i})\\
&\qquad\quad \wedge \forall k \geq 0 : (\forall j \leq k : \hat{d}_{\mathsf{In}}(t_1[j] \cap \mathsf{AP_i}, t'_2[j] \cap \mathsf{AP_i}) \leq \kappa_{\mathsf{i}})\\
&\qquad\qquad\qquad\qquad \rightarrow \hat{d}_{\mathsf{Out}}(t_1[k] \cap \mathsf{AP_o}, t'_2[k] \cap \mathsf{AP_o}) \leq \kappa_{\mathsf{o}}\Big)
\end{aligned}$$

which, by logic manipulation, is equivalent to

$$\begin{aligned}
&\forall \mathsf{p}_1 : \forall \mathsf{p}_2 : \forall \mathsf{i}_1 : \forall \mathsf{i}_2 : \forall \mathsf{o}_1 :\\
&\Big(\exists t_1 \in S : \exists t_2 \in S :\\
&\qquad (\mathsf{p}_1 = (t_1[0] \cap \mathsf{AP_p}) \in \mathsf{PIntrs}) \wedge (\mathsf{p}_2 = (t_2[0] \cap \mathsf{AP_p}) \in \mathsf{PIntrs})\\
&\qquad \wedge \mathsf{i}_1 = (t_1 \downarrow \mathsf{AP_i}) \wedge \mathsf{i}_2 = (t_2 \downarrow \mathsf{AP_i}) \wedge \mathsf{o}_1 = (t_1 \downarrow \mathsf{AP_o}) \wedge \mathsf{i}_1 \in \mathsf{StdIn}\Big)
\end{aligned}$$

$$\begin{aligned}&\to \exists \mathsf{o}_2 : \exists t'_2 \in S : \\ &\qquad \Big(\mathsf{p}_2 = (t'_2[0] \cap \mathsf{AP_p}) \wedge \mathsf{i}_2 = (t'_2 \downarrow \mathsf{AP_i}) \wedge \mathsf{o}_2 = (t'_2 \downarrow \mathsf{AP_o}) \\ &\qquad \wedge \forall k \geq 0 : (\forall j \leq k : \hat{d}_{\mathsf{In}}(\mathsf{i}_1[j], \mathsf{i}_2[j]) \leq \kappa_\mathsf{i}) \to \hat{d}_{\mathsf{Out}}(\mathsf{o}_1[k], \mathsf{o}_2[k]) \leq \kappa_\mathsf{o}\Big)\end{aligned}$$

By (5) and the fact that distances are past-forgetful, the previous equation is equivalent to

$$\begin{aligned}&\forall \mathsf{p}_1 : \forall \mathsf{p}_2 : \forall \mathsf{i}_1 : \forall \mathsf{i}_2 : \forall \mathsf{o}_1 : \\ &\Big(\mathsf{p}_1, \mathsf{p}_2 \in \mathsf{PIntrs} \wedge \mathsf{i}_1 \in \mathsf{StdIn} \wedge \forall k \geq 0 : (\forall j \leq k : d_{\mathsf{In}}(\mathsf{i}_1[..j], \mathsf{i}_2[..j]) \leq \kappa_\mathsf{i}) \\ &\qquad \wedge \mathsf{o}_1 \in \hat{S}(\mathsf{p}_1)(\mathsf{i}_1)\Big) \to \big(\exists \mathsf{o}_2 \in \hat{S}(\mathsf{p}_2)(\mathsf{i}_2) : d_{\mathsf{Out}}(\mathsf{o}_1[..k], \mathsf{o}_2[..k]) \leq \kappa_\mathsf{o}\big)\end{aligned}$$

which in turn corresponds to bounding the left sup-inf term of the Hausdorff distance (see (2)) in Definition 7,

$$\begin{aligned}&\forall \mathsf{p}_1 : \forall \mathsf{p}_2 : \forall \mathsf{i}_1 : \forall \mathsf{i}_2 : \\ &\big(\mathsf{p}_1, \mathsf{p}_2 \in \mathsf{PIntrs} \wedge \mathsf{i}_1 \in \mathsf{StdIn} \wedge \forall k \geq 0 : (\forall j \leq k : d_{\mathsf{In}}(\mathsf{i}_1[..j], \mathsf{i}_2[..j]) \leq \kappa_\mathsf{i})\big) \\ &\qquad \to \big(\sup_{\mathsf{o}_1 \in \hat{S}(\mathsf{p}_1)(\mathsf{i}_1)} \inf_{\mathsf{o}_2 \in \hat{S}(\mathsf{p}_2)(\mathsf{i}_2)} d_{\mathsf{Out}}(\mathsf{o}_1[..k], \mathsf{o}_2[..k])\big) \leq \kappa_\mathsf{o}\end{aligned}$$

thus proving this part of the proposition. □

Finally, we also give the characterisation of an f-clean program in terms of HyperLTL.

Proposition 20. *A reactive program S is f-clean under past-forgetful distances d_{In} and d_{Out} if and only if S satisfies the following two HyperLTL formulas*

$$\begin{aligned}&\forall \pi_1 . \forall \pi_2 . \exists \pi'_2 . \\ &\quad (\mathsf{PIntrs}_{\pi_1} \wedge \mathsf{PIntrs}_{\pi_2} \wedge \mathsf{StdIn}_{\pi_1}) \\ &\quad \to \Big(\mathsf{p}_{\pi_2} = \mathsf{p}_{\pi'_2} \wedge \mathsf{G}(\mathsf{i}_{\pi_2} = \mathsf{i}_{\pi'_2}) \wedge \mathsf{G}\Big(\hat{d}_{\mathsf{Out}}(\mathsf{o}_{\pi_1}, \mathsf{o}_{\pi'_2}) \leq f(\hat{d}_{\mathsf{In}}(\mathsf{i}_{\pi_1}, \mathsf{i}_{\pi'_2}))\Big)\Big) \\ &\forall \pi_1 . \forall \pi_2 . \exists \pi'_1 . \\ &\quad (\mathsf{PIntrs}_{\pi_1} \wedge \mathsf{PIntrs}_{\pi_2} \wedge \mathsf{StdIn}_{\pi_1}) \\ &\quad \to \Big(\mathsf{p}_{\pi_1} = \mathsf{p}_{\pi'_1} \wedge \mathsf{G}(\mathsf{i}_{\pi_1} = \mathsf{i}_{\pi'_1}) \wedge \mathsf{G}\Big(\hat{d}_{\mathsf{Out}}(\mathsf{o}_{\pi'_1}, \mathsf{o}_{\pi_2}) \leq f(\hat{d}_{\mathsf{In}}(\mathsf{i}_{\pi'_1}, \mathsf{i}_{\pi_2}))\Big)\Big) \qquad (9)\end{aligned}$$

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

We remark that the HyperLTL characterisations presented in Propositions 19 and 20 can be extended to any distance of bounded memory, that is, distances such that $d(t, t') = d(t[k..], t'[k..])$ for every finite traces t and t' and a fixed bound $k \in \mathbb{N}$. The solution proceeds by basically using the same formulas on an expanded and annotated model (with the expected exponential blow up w.r.t. to the original one).

Example 21. In our running example of the emission control system (see Examples 8 and 10), the property of robustly cleanness reduces to checking formula

$$\begin{aligned}&\forall \pi_1 .\, \forall \pi_2 .\, \exists \pi'_2 . \\ &\qquad \mathsf{StdIn}_{\pi_1} \rightarrow \Big(\mathsf{G}(t_{\pi_2} = t_{\pi'_2}) \wedge \big((\hat{d}_{\mathsf{Out}}(n_{\pi_1}, n_{\pi'_2}) \leq \kappa_{\mathsf{o}})\, \mathsf{W}\, (\hat{d}_{\mathsf{In}}(t_{\pi_1}, t_{\pi'_2}) > \kappa_{\mathsf{i}})\big)\Big)\end{aligned} \tag{10}$$

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

$$\begin{aligned}&\forall \pi_1 .\, \forall \pi_2 .\, \exists \pi'_2 . \\ &\qquad \mathsf{StdIn}_{\pi_1} \rightarrow \Big(\mathsf{G}(t_{\pi_2} = t_{\pi'_2}) \wedge\, \mathsf{G}\, \big(\hat{d}_{\mathsf{Out}}(n_{\pi_1}, n_{\pi'_2}) \leq f(\hat{d}_{\mathsf{In}}(t_{\pi_1}, t_{\pi'_2}))\big)\Big)\end{aligned} \tag{11}$$

and the symmetric formula.

6 Experimental Results

We verified the cleanness of the emission control system using the HyperLTL model checker MCHyper [24]. The input to the model checker is a description of the system as an Aiger circuit and a hyperproperty specified as an alternation-free HyperLTL formula. Since the HyperLTL formulas from the previous section are of the form $\forall \pi_1 \forall \pi_2 \exists \pi'_2 \ldots$, and are, hence, not alternation-free, MCHyper cannot check these formulas directly. However, it is possible to prove or disprove such formulas by strengthening the formulas and their negations manually into alternation-free formulas that are accepted by MCHyper.

In order to prove that program EC in Fig. 9 is robustly clean, we strengthen formula (10) by substituting π_2 for the existentially quantified variable π'_2. The resulting formula is alternation-free:

$$\forall \pi_1 .\, \forall \pi_2 .\; \mathsf{StdIn}_{\pi_1} \rightarrow \big((\hat{d}_{\mathsf{Out}}(n_{\pi_1}, n_{\pi_2}) \leq \kappa_{\mathsf{o}})\, \mathsf{W}\, (\hat{d}_{\mathsf{In}}(t_{\pi_1}, t_{\pi_2}) > \kappa_{\mathsf{i}})\big) \tag{12}$$

MCHyper confirms that program EC satisfies (12). The program thus also satisfies (10). Notice that we had obtained the same formula if we would have started from the formula symmetric to (10).

To prove that program AEC in Fig. 10 is doped with respect to (10), we negate (10) and obtain

$$\begin{aligned}&\exists \pi_1 .\, \exists \pi_2 .\, \forall \pi'_2 . \\ &\quad \neg\Big(\mathsf{StdIn}_{\pi_1} \rightarrow \Big(\mathsf{G}(t_{\pi_2} = t_{\pi'_2}) \wedge \big((\hat{d}_{\mathsf{Out}}(n_{\pi_1}, n_{\pi'_2}) \leq \kappa_{\mathsf{o}})\, \mathsf{W}\, (\hat{d}_{\mathsf{In}}(t_{\pi_1}, t_{\pi'_2}) > \kappa_{\mathsf{i}})\big)\Big)\Big)\end{aligned}$$

This formula is of the form $\exists\pi_1.\,\exists\pi_2.\,\forall\pi'_2.\ldots$ and, hence, again not alternation-free. We replace the two existential quantifiers with universal quantifiers and restrict the quantification to two specific throttle values, a for π_1 and b for π_2:

$$\begin{aligned}&\forall\pi_1.\,\forall\pi_2.\,\forall\pi'_2.\\ &\quad \mathsf{G}(t_{\pi_1} = a \wedge t_{\pi_2} = b) \;\rightarrow\\ &\quad \neg\Big(\mathsf{StdIn}_{\pi_1} \;\rightarrow \Big(\mathsf{G}(t_{\pi_2} = t_{\pi'_2}) \wedge \big((\hat{d}_{\mathsf{Out}}(n_{\pi_1}, n_{\pi'_2}) \leq \kappa_{\mathsf{o}})\;\mathsf{W}\;(\hat{d}_{\mathsf{In}}(t_{\pi_1}, t_{\pi'_2}) > \kappa_{\mathsf{i}})\big)\Big)\Big)\end{aligned} \tag{13a}$$

This transformation is sound as long as there actually exist traces with throttle values a and b. We establish this by checking, separately, that the following existential formula is satisfied:

$$\exists\pi_1.\,\exists\pi_2.\,\mathsf{G}(t_{\pi_1} = a \wedge t_{\pi_2} = b) \tag{14}$$

MCHyper confirms the satisfaction of both formulas, which proves that (10) is violated by program AEC. Precisely, the counterexample that shows the violation of (10) is any pair of traces π_1 and π_2 that makes $\mathsf{G}(t_{\pi_1} = a \wedge t_{\pi_2} = b)$ true in (14). We proceed similarly for the formula symmetric to (10) obtaining two formulas just as before which are also satisfied by AEC and hence the original formula is not. Also, we follow a similar process to prove that EC is f-clean but AEC is not.

Table 2. Experimental results from the verification of robust cleanness of EC and AEC

Program	NO_x Step	Model size	Circuit size		Property	Time
		#transitions	#latches	#gates		(sec.)
EC	0.05	1436	17	9749	(12)	0.92
	0.00625	60648	23	505123	(12)	22.19
AEC	0.05	3756	19	27574	(13a) $a = 0.1$	1.62
					(13b) $a = 0.1$	1.6
					(13a) $a = 1$	1.68
					(13b) $a = 1$	1.56
	0.00625	175944	25	1623679	(13a) $a = 0.1$	102.07
					(13b) $a = 0.1$	96.3
					(13a) $a = 1$	97.67
					(13b) $a = 1$	92.8

Table 2 shows experimental results obtained with MCHyper[3] version 0.91 for the verification of robustly cleanness. The Aiger models were constructed by discretizing the values of the throttle and the NO_x. We show results from two different models, where the values of the throttle was discretised in steps of 0.1

[3] https://www.react.uni-saarland.de/tools/mchyper/.

units in both models and the values of the NO_x in steps of 0.05 and 0.00625. All experiments were run under OS X "El Capitan" (10.11.6) on a MacBook Air with a 1.7GHz Intel Core i5 and 4GB 1333MHz DDR3. In Table 2, the model size is given in terms of the number of transitions, while the size of the Aiger circuit encoding the model prepared for the property is given in terms of the number of latches and gates. The specification checked by MCHyper is the formula indicated in the property column. Formula (13b) is the formula symmetric to (13a). For the throttle values a and b in formulas (13a) and (13b), we chose $b = 2$ and let a vary as specified in the property column. Table 3 shows similar experimental results for the verification of f-cleanness. With (12′), (13a′), and (13b′) we indicate the similar variations to (12), (13a), and (13b) required to verify (11). Model checking takes less than two seconds for the coarse discretisation and about two minutes for the fine discretisation.

Table 3. Experimental results from the verification of f-cleanness of EC and AEC

Program	NO_x	Model size	Circuit size		Property	Time
	step	#transitions	#latches	#gates		(sec.)
EC	0.05	1436	5	9869	(12′)	1.08
	0.00625	60648	8	505285	(12′)	21.74
AEC	0.05	3756	6	27708	(13a′) $a = 0.1$	1.71
					(13b′) $a = 0.1$	1.72
					(13a′) $a = 1$	1.72
					(13b′) $a = 1$	1.77
	0.00625	175944	9	1623855	(13a′) $a = 0.1$	95.29
					(13b′) $a = 0.1$	97.48
					(13a′) $a = 1$	95.57
					(13b′) $a = 1$	95.5

7 A Comprehensive Characterisation

If we concretely focus on the contract between the society or the licensee, and the software manufacturer, we can think in a more general but precise definition. It emerges by noticing that there is a partition on the set of inputs in three sets, each one of them fulfilling a different role within the contract:

1. The set StdIn of *standard inputs*. For these inputs, the program is expected to work exactly as regulated. It is the case, e.g., of the inputs defining the tests for the NO_x emission. Thus, it is expected that the program complies to Definition 1 when provided only with inputs in StdIn.

2. The set Comm of *committed inputs* such that $\mathsf{Comm} \cap \mathsf{StdIn} = \varnothing$. These inputs are expected to be close according to a distance to StdIn and are not strictly regulated. However, it is expected that the manufacturer commits to respect certain bounds on the outputs. This would correspond to the inputs that do not behave exactly like the tests for the NO_x emission, but yet define "reasonable behaviour" of the car on the road. The behaviour of the program under this set of inputs can be characterised either by Definition 2 or Definition 4.
3. All other inputs are supposed to be anomalous and expected to be significantly distant from the standard inputs. In our emission control example, this can occur, e.g., if the car is climbing a steep mountain or speeding up in a highway. In this realm the only expectation is that the behaviour of the output is continuous with respect to the input.

Bearing this partition in mind, we propose the following general definition.

Definition 22. *A parameterised program S is* clean *(or* doping-free*) if for all pairs of parameters of interest $\mathsf{p}, \mathsf{p}' \in \mathsf{PIntrs}$ and inputs $\mathsf{i}, \mathsf{i}' \in \mathsf{In}$,*

1. *if $\mathsf{i} \in \mathsf{StdIn}$ then $S(\mathsf{p})(\mathsf{i}) = S(\mathsf{p}')(\mathsf{i})$;*
2. *if $\mathsf{i} \in \mathsf{StdIn}$ and $\mathsf{i}' \in \mathsf{Comm}$ then $\mathcal{H}(d_{\mathsf{Out}})(S(\mathsf{p})(\mathsf{i}), S(\mathsf{p}')(\mathsf{i}')) \leq f(d_{\mathsf{In}}(\mathsf{i}, \mathsf{i}'))$.*
3. *for every $\epsilon > 0$ there exists $\delta > 0$ such that for all $\mathsf{i}' \notin \mathsf{StdIn} \cup \mathsf{Comm}$ and $\mathsf{i} \in \mathsf{In}$, $d_{\mathsf{In}}(\mathsf{i}, \mathsf{i}') < \delta$ implies $\mathcal{H}(d_{\mathsf{Out}})(S(\mathsf{p})(\mathsf{i}), S(\mathsf{p}')(\mathsf{i}')) < \epsilon$.*

Notice that, while PIntrs, StdIn, Comm, d_{In}, d_{Out}, and f are part of the contract entailed by the definition, ϵ and δ in item 3 are not since they are quantified (universally and existentially, resp.) in the definition. In this case, we choose for item 3 to require that the program S is uniformly continuous in $\mathsf{In} \setminus (\mathsf{StdIn} \cup \mathsf{Comm})$. However, we could have opted for stronger requirements such as Lipschitz continuity. The chosen type of continuity would also be part of the contract. Notice that this is the only case in which we require continuity. Instead, discontinuities are allowed in cases 1 and 2 as long as the conditions are respected since they may be part of the specification. In particular, notice that f could be *any* function. Obviously, a similar definition can be obtained for reactive systems.

We remark that cases 1 and 2 can be verified, as we showed in the paper. We have not yet explored the verification of case 3.

8 Related Work

The term "software doping" has being coined by the press about a year ago and, after the Volkswagen exhaust emissions scandal, the elephant in the room became unavoidable: software developers introduce code intended to deceive [28]. Recently, a special session at ISOLA 2016 was devoted to this topic [34]. In [9], Baum attacks the problem from a philosophical point of view and elaborates on the ethics of it. In [5], we provided a first discussion of the problem and some informal characterisations hinting at the formal proposal of this paper. Though all these works point out the need for a technical attack on the problem, none of them provide a formal proposal.

Similar to software doping, backdoored software is a class of software that does not act in the best interest of users; see for instance the recent analysis in [37]. The primary emphasis of backdoored software is on leaking confidential information while guaranteeing functionality.

Dope-freedom in sequential programs is strongly related to abstract non-intereference [6,26] as already disussed in Sect. 4. More generally, our notions of dope-freedom are hyperproperties [16], a general class that encompasses notions across different domains, in particular non-interference in security [39], robustness (a.k.a. stability) in cyber-physical systems [13], and truthfulness in algorithmic game theory [8]. There exist several methods for verifying hyperproperties, including relational and Cartesian Hoare logics [10,38,44], self-composition and product programs constructions [4,7], temporal logics [15,23,24], or games [35]. These techniques greatly vary in their completeness, efficiency, and scalability.

Another worthwhile direction to study is the use of program equivalence analysis [22,27] for the analysis of cleanness.

9 Concluding Remarks

This article has focused on a serious and yet long overlooked problem, arising if software developers intentionally and silently deviate from the intended objective of the developed software. A notorious reason behind such deviations are simple and blunt lock-in strategies, so as to bind the software licensee to a certain product or product family. However, the motivations can be more diverse and obscure. As the software manufacturer has full control over the development process, the deviation can be subtle and surreptitiously introduced in a way that the fact that the program does not quite conform to the expected requirements may go well unnoticed.

We have pioneered the formalisation of this problem domain by offering several formal characterisations of software doping. These can serve as a framework for establishing a contract between the interested parties, namely the society or the licensee, and the software manufacturer, so as to avoid and eventually ban the development of doped programs.

We have also reported on the use of existing theories and tools at hand to demonstrate that the formal characterisation can indeed be analysed in various ways. In particular, the application of the self-composition technique opens many research directions for further analysis of software doping as it has been widely studied in the area of security [29,31], semantical differences [32] and cross or relative verification [30].

As we have demonstrated, the use of HyperLTL enables the automatic analysis of reactive models with respect to software doping. However, the complexity of this technique imposes some serious limits on its applicability. Thus, further studies in this direction are needed in order to enable analysis of reactive models of relatively large size, or alternatively to analyse the program code directly.

We believe our characterisations provide a first solid step to understand software doping and that our result opens a large umbrella of new possibilities, both in the direction of more dedicated characterisations as well as specifically tailored analysis techniques. For instance, the idea of dealing with distances and thresholds already rises the question of whether such distances could be quantified by probabilities. Also, the NO_x emission example would immediately suggest that the technique should also be addressed with testing. Moreover, the fact that the characterisations are hyperproperties also invites us to investigate for static analysis of source code based on type systems, abstraction techniques, etc.

Acknowledgement. We would like to thank the Dependable Systems and Software Group (Saarland University) for a fruitful discussion during an early presentation of this work, and Nicolás Wolovick for drawing our attention to electronic voting.

References

1. Agorist, M.: WATCH: computer programmer testifies he helped rig voting machines. MintPress News (2016) http://www.mintpressnews.com/214505-2/214505/. Accessed 13 Jan 2017
2. AppleInsider: Galaxy S4 on steroids: Samsung caught doping in benchmarks (2013). http://forums.appleinsider.com/discussion/158782/galaxy-s-4-on-steroids-samsung-caught-doping-in-benchmarks. Accessed 13 Jan 2017
3. Arthur, W.B.: Competing technologies, increasing returns, and lock-in by historical events. Econ. J. **99**(394), 116–131 (1989). http://www.jstor.org/stable/2234208
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21437-0_17
5. Barthe, G., D'Argenio, P.R., Finkbeiner, B., Hermanns, H.: Facets of software doping. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 601–608. Springer, Heidelberg (2016). doi:10.1007/978-3-319-47169-3_46
6. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSFW-17, pp. 100–114. IEEE Computer Society (2004). http://doi.ieeecomputersociety.org/10.1109/CSFW.2004.17
7. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6), 1207–1252 (2011). http://dx.doi.org/10.1017/S0960129511000193
8. Barthe, G., Gaboardi, M., Arias, E.J.G., Hsu, J., Roth, A., Strub, P.: Higher-order approximate relational refinement types for mechanism design and differential privacy. In: Rajamani, S.K., Walker, D. (eds.) POPL 2015, pp. 55–68. ACM (2015). http://doi.acm.org/10.1145/2676726.2677000
9. Baum, K.: What the hack is wrong with software doping? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 633–647. Springer, Heidelberg (2016). doi:10.1007/978-3-319-47169-3_49
10. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) POPL 2004, pp. 14–25. ACM Press (2004). http://doi.acm.org/10.1145/964001.964003

11. Brignall, M.: 'Error 53' fury mounts as Apple software update threatens to kill your iPhone 6. The Guardian (2010). https://www.theguardian.com/money/2016/feb/05/error-53-apple-iphone-software-update-handset-worthless-third-party-repair. Accessed 13 Jan 2017
12. Carrel, P., Bryan, V., Croft, A.: Germany asks Opel for more information in Zafira emissions probe. Reuters (2016). http://www.reuters.com/article/us-volkswagen-emissions-germany-opel-idUSKCN0Y92GI. Accessed 13 Jan 2017
13. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity analysis of programs. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL 2010, pp. 57–70 (2010). http://doi.acm.org/10.1145/1706299.1706308
14. Checkoway, S., Niederhagen, R., Everspaugh, A., Green, M., Lange, T., Ristenpart, T., Bernstein, D.J., Maskiewicz, J., Shacham, H., Fredrikson, M.: On the practical exploitability of dual EC in TLS implementations. In: Fu, K., Jung, J. (eds.) 23rd USENIX Security Symposium. pp. 319–335. USENIX Association (2014). https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/checkoway
15. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54792-8_15
16. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: CSF 2008, pp. 51–65 (2008). http://dx.doi.org/10.1109/CSF.2008.7
17. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). http://dx.doi.org/10.3233/JCS-2009-0393
18. Dijkstra, E.: A Discipline of Programming. Prentice Hall PTR, Upper Saddle River (1997)
19. Domke, F., Lange, D.: The exhaust emissions scandal ("Dieselgate"). In: 30th Chaos Communication Congress (2015). https://events.ccc.de/congress/2015/Fahrplan/events/7331.html. Accessed 13 Jan 2017
20. Dvorak, J.C.: The secret printer companies are keeping from you. PC Mag UK (2012). http://uk.pcmag.com/printers/60628/opinion/the-secret-printer-companies-are-keeping-from-you. Accessed 13 Jan 2017
21. Feldman, A.J., Halderman, J.A., Felten, E.W.: Security analysis of the Diebold AccuVote-ts voting machine. In: Martinez, R., Wagner, D. (eds.) 2007 USENIX/ACCURATE Electronic Voting Technology Workshop, EVT 2007. USENIX Association (2007). https://www.usenix.org/conference/evt-07/security-analysis-diebold-accuvote-ts-voting-machine
22. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Crnkovic, I., Chechik, M., Grünbacher, P. (eds.) ASE 2014, pp. 349–360. ACM (2014). http://doi.acm.org/10.1145/2642937.2642987
23. Finkbeiner, B., Hahn, C.: Deciding Hyperproperties. In: Desharnais, J., Jagadeesan, R. (eds.) CONCUR 2016. LIPIcs, vol. 59, pp. 13:1–13:14. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016). http://drops.dagstuhl.de/opus/volltexte/2016/6170
24. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Heidelberg (2015). doi:10.1007/978-3-319-21690-4_3
25. Flak, A., Taylor, E., Wacket, M., Eckert, V., Stonestreet, J.: Test of fiat diesel model shows irregular emissions: Bild am Sonntag. Reuters (2016). http://www.reuters.com/article/us-fiat-emissions-germany-idUSKCN0XL0MT. Accessed 13 Jan 2017

26. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: parameterizing non-interference by abstract interpretation. In: Jones, N.D., Leroy, X. (eds.) POPL 2004, pp. 186–197. ACM (2004). http://doi.acm.org/10.1145/964001.964017
27. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. Softw. Test. Verif. Reliab. **23**(3), 241–258 (2013)
28. Hatton, L., van Genuchten, M.: When software crosses a line. IEEE Softw. **33**(1), 29–31 (2016). http://dx.doi.org/10.1109/MS.2016.6
29. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: Miller, E.L., Hand, S. (eds.) SOSP 2015, pp. 1–17. ACM (2015). http://doi.acm.org/10.1145/2815400.2815428
30. Hawblitzel, C., Lahiri, S.K., Pawar, K., Hashmi, H., Gokbulut, S., Fernando, L., Detlefs, D., Wadsworth, S.: Will you still compile me tomorrow? Static cross-version compiler validation. In: Meyer, B., Baresi, L., Mezini, M. (eds.) ESEC/FSE 2013, pp. 191–201 (2013). http://doi.acm.org/10.1145/2491411.2491442
31. Kovács, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties. In: Sadeghi, A., Gligor, V.D., Yung, M. (eds.) CCS 2013, pp. 211–222. ACM (2013). http://doi.acm.org/10.1145/2508859.2516721
32. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: a language-agnostic semantic diff tool for imperative programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 712–717. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31424-7_54
33. Manjoo, F.: Take that, stupid printer! Slate (2008). http://www.slate.com/articles/technology/technology/2008/08/take_that_stupid_printer.html. Accessed 13 Jan 2017
34. Margaria, T., Steffen, B. (eds.): ISoLA 2016. LNCS, vol. 9953. Springer, Heidelberg (2016)
35. Milushev, D., Clarke, D.: Incremental hyperproperty model checking via games. In: Riis Nielson, H., Gollmann, D. (eds.) NordSec 2013. LNCS, vol. 8208, pp. 247–262. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41488-6_17
36. Panzarino, M.: Apple apologizes and updates iOS to restore iPhones disabled by error 53. TechCrunch (2016). https://techcrunch.com/2016/02/18/apple-apologizes-and-updates-ios-to-restore-iphones-disabled-by-error-53/. Accessed 13 Jan 2017
37. Schneier, B., Fredrikson, M., Kohno, T., Ristenpart, T.: Surreptitiously weakening cryptographic systems. IACR Cryptology ePrint Archive 2015, 97 (2015). http://eprint.iacr.org/2015/097
38. Sousa, M., Dillig, I.: Cartesian Hoare logic for verifying k-safety properties. In: Krintz, C., Berger, E. (eds.) PLDI 2016, pp. 57–69. ACM (2016). http://doi.acm.org/10.1145/2908080.2908092
39. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005). doi:10.1007/11547662_24
40. Tritech Computer Solutions: Dell laptops reject third-party batteries and AC adapters/chargers. Hardware vendor lock-in? https://nctritech.wordpress.com/2010/01/26/dell-laptops-reject-third-party-batteries-and-ac-adapterschargers-hardware-vendor-lock-in/ (2010). Accessed 13 Jan 2017

41. Waller, K.: Has a printer update rendered your cartridges redundant? Which? (2016). https://conversation.which.co.uk/technology/printer-software-update-third-party-printer-ink/. Accessed 13 Jan 2017
42. Waste Ink: Epson firmware update = no to compatibles. http://www.wasteink.co.uk/epson-firmware-update-compatible-problem/ (2012). Accessed 13 Jan 2017
43. Wikipedia: Volkswagen emissions scandal. Wikipedia, The Free Encyclopedia (2016). https://en.wikipedia.org/wiki/Volkswagen_emissions_scandal. Accessed 13 Jan 2017
44. Yang, H.: Relational separation logic. Theor. Comput. Sci. **375**(1–3), 308–334 (2007). http://dx.doi.org/10.1016/j.tcs.2006.12.036