Testing Equivalence vs. Runtime Monitoring*

Luca Aceto[1,2][0000−0002−2197−3018], Antonis Achilleos[2][0000−0002−1314−333X], Adrian Francalanza[3][(0000−0003−3829−7391], Anna Ingólfsdóttir[2][0000−0001−8362−3075], and Karoliina Lehtinen[4][0000−0003−1171−8790]

[1] Gran Sasso Science Institute, L'Aquila, Italy
luca.aceto@gssi.it
[2] School of Computer Science, Reykjavik University, Reykjavik, Iceland
{luca,antonios,annai}@ru.is
[3] Department of Computer Science, ICT, University of Malta, Msida, Malta
adrian.francalanza@um.edu.mt
[4] Department of Computer Science, University of Liverpool, United Kingdom
karoliina.lehtinen@liverpool.ac.uk

Abstract. Rocco De Nicola's most cited paper, which was coauthored with his PhD supervisor Matthew Hennessy, introduced three seminal testing equivalences over processes represented as states in labelled transition systems. This article relates those classic process semantics with the framework for runtime monitoring developed by the authors in the context of the project 'TheoFoMon: Theoretical Foundations for Monitorability'. It shows that may-testing semantics is closely related to the basic monitoring set-up within that framework, whereas, over strongly-convergent processes, must-testing semantics is induced by a collection of monitors that can detect when processes are unable to perform certain actions.

Keywords: Testing equivalence · Runtime monitoring · Trace equivalence · Failure equivalence · Hennessy-Milner Logic with recursion.

1 Introduction

Rocco De Nicola is probably best known for the introduction of the notions of testing equivalence over concurrent processes, in joint work with his PhD supervisor Matthew Hennessy that was reported in the conference paper [14] and the subsequent journal paper [15]. These testing equivalences embody in a natural and mathematically elegant way the intuitive idea that two processes should be equated unless they behave differently when subjected to some 'experiment'

* This research was partially supported by the projects 'TheoFoMon: Theoretical Foundations for Monitorability' (grant number: 163406-051; http://icetcs.ru.is/theofomon/) and 'Epistemic Logic for Distributed Runtime Monitoring' (grant number: 184940-051) of the Icelandic Research Fund, by the BMBF project 'Aramis II' (project number: 01IS160253) and the EPSRC project 'Solving parity games in theory and practice' (project number: EP/P020909/1).

or 'test'. The origin of this notion of equivalence can be traced back to Gottfried Wilhelm Leibniz (1646–1716), whose Identity of Indiscernibles principle states that two (mathematical) objects are equal if there is no property that distinguishes them [24, 'Discourse on Metaphysics', Section 9]. In the semantics of programming languages, its earliest precursor is, to the best of our knowledge, the notion of contextual equivalence proposed by Morris in his doctoral dissertation [26].

In general, given a set of processes, a set of tests and a relation between processes and tests that describes when a process passes a test, one can apply Leibniz's motto and declare two processes to be equivalent if they pass exactly the same set of tests. In the work of De Nicola and Hennessy, processes are states in some labelled transition system [22]. A test is itself a process, which interacts with a concurrent system under observation by hand-shake synchronisation and uses a distinguished action to report success in its observation. Since both processes and tests may be nondeterministic, the interaction between a process and a test may lead to different outcomes depending on how the two systems resolve their nondeterministic choices in the course of a computation. This led De Nicola and Hennessy to define three notions of testing semantics, which are naturally expressed in terms of preorders over processes. In the so-called *may semantics*, a process q is at least as good as some process p if the set of tests that p may pass is included in the set of tests that q may pass. In may semantics, possible failure under a test is immaterial and therefore nondeterminism is *angelic*. On the other hand, one may take the view that failure in the testing effort is *catastrophic*, in the sense that a process that may fail some test is just as bad as one that always fails it. The notion of testing semantics that captures this viewpoint is the so-called *must semantics*, according to which a process q is at least as good as some process p if the set of tests that p must pass is included in the set of tests that q must pass. Finally, a third testing preorder over processes is obtained as the intersection of the may and must preorders described above. According to this more refined view of process behaviour, a process that always fails a test is worse than one that may pass that test, which in turn is worse than one that always passes it.

De Nicola and Hennessy explored the rich theory of the testing semantics in [15] (see [19] for a book-length treatment), where each of these semantics is given operational, denotational and axiomatic accounts that are in agreement one with the other. Their ideas and the accompanying technical results have had an enormous impact on further research, as witnessed, among other things, by the over $1,650$ citations to [15][5].

Our goal in this article is to provide some evidence supporting our view that De Nicola and Hennessy's work may also be seen as providing the theoretical foundations for runtime verification [9], a line of research that is becoming increasingly important in the field of computer-aided verification. Runtime verification is a lightweight verification technique that checks whether the system

[5] Source: https://scholar.google.com/citations?user=Meb6JFkAAAAJ&hl=en, last accessed on the 24th of March 2019.

under scrutiny satisfies a correctness property by analysing its current execution. In this approach, a computational entity called a *monitor*, which is synthesised from a given correctness property, is used to observe the current system execution and to report whether the observed computation satisfies the given property.

The high-level description of runtime verification given above hints at conceptual similarities between that approach to computer-aided verification and testing equivalences à la De Nicola and Hennessy. Indeed, the monitors used in runtime verification seem to play a role akin to that of the tests in the work of De Nicola and Hennessy. In this paper, we will see that the connection between runtime verification and testing semantics can be made precise within the operational framework for runtime monitoring developed in [1,3,16,17]. More precisely, we will show that may-testing semantics is closely related to the basic monitoring set-up presented in [16,17] (Section 3), whereas must-testing semantics over strongly-convergent, finitely-branching processes is induced by a collection of monitors that can detect refusals and that stem from the framework for parameterised monitorability developed in [1] (Section 4). Together with the results presented in [7,12], we feel that Theorems 2 and 7 in this study substantiate our tenet that runtime verification owes much to the work of De Nicola and Hennessy on testing equivalences for processes.

2 Preliminaries

We begin by briefly reviewing the model of labelled transition systems used in this study (Section 2.1) and by presenting an informal account of De Nicola-Hennessy testing equivalences (Section 2.2).

2.1 Labelled Transition Systems

We assume a finite set of *external* actions Act and, following Milner [25], a distinguished *silent* action τ. We let α, a, b, c range over Act and μ over $\text{Act} \cup \{\tau\}$. A *labelled transition system* (LTS) over Act is a triple

$$L = \langle P, \text{Act}, \rightarrow_L \rangle,$$

where P is a nonempty set of system states referred to as *processes* ($p, q, \ldots \in P$), and $\rightarrow_L \subseteq P \times (\text{Act} \cup \{\tau\}) \times P$ is a transition relation. We write $p \xrightarrow{\mu}_L q$ instead of $(p, \mu, q) \in \rightarrow_L$. We use $p \stackrel{\alpha}{\Rightarrow}_L q$ to mean that, in L, p can reach q using a single α action and any number of silent actions, *i.e.,* $p(\xrightarrow{\tau}_L)^* \xrightarrow{\alpha}_L (\xrightarrow{\tau}_L)^* q$. By $p \xrightarrow{\mu}_L$ (respectively, $p \stackrel{\alpha}{\Rightarrow}_L$) we mean that there is some q such that $p \xrightarrow{\mu}_L q$ (respectively, $p \stackrel{\alpha}{\Rightarrow}_L q$) and $p \not\xrightarrow{\mu}_L$ (respectively, $p \stackrel{\alpha}{\not\Rightarrow}_L$) means that no such q exists. For a trace $s = \alpha_1 \alpha_2 \ldots \alpha_\ell \in \text{Act}^*$, $p \stackrel{s}{\Rightarrow}_L q$ means $p \stackrel{\alpha_1}{\Longrightarrow}_L \stackrel{\alpha_2}{\Longrightarrow}_L \ldots \stackrel{\alpha_\ell}{\Longrightarrow}_L q$ when $\ell \geq 1$ and $p(\xrightarrow{\tau})^* q$ when $s = \varepsilon$ is the empty trace. We say that s is a trace of p when $p \stackrel{s}{\Rightarrow}_L q$ for some q, and write $\text{traces}(p)$ for the set of all the traces of p. From now on we will omit the subscript L as the LTS will be always clear from the context.

In the rest of the paper, processes will be specified using expressions in the fragment of Milner's CCS [25] containing the operators for describing finite synchronisation trees over $\text{Act} \cup \{\tau\}$ [29].

2.2 Testing Equivalences à la De Nicola and Hennessy

We will now informally recall the testing semantics from [15,19]. We will not present the full details of the formal definitions of the testing semantics, since our technical results will rely on the alternative, test-free characterisations of the may- and must-testing preorders, which we will state in Sections 3.3 and 4.2 where they are used.

The testing equivalences over processes introduced in [15] embody in a natural and mathematically elegant way the intuitive idea that two programs should be equated unless they behave differently when subjected to some 'experiment'. In the setting of the above-mentioned paper, an experiment is itself a process, called *test*, that interacts with the observed system by communicating with it and that uses a distinguished action ω to report a successful outcome resulting from its observations.

We say that

- process p *may pass* a test t if there is *some* maximal computation resulting from the interaction between p and t in which t reports success;
- process p *must pass* a test t if t reports success in *every* maximal computation resulting from the interaction between p and t.

The classification of the possible outcomes resulting from process-test interactions leads to three different notions of semantic equivalence over processes: one in which nondeterminism is angelic (the may-testing preorder), another in which the possibility of failure is catastrophic (the must-testing preorder) and a third in which a process that may both fail and pass a test is distinguished from one that always fails it or always passes it (the intersection of the may- and must-testing preorders). Each of these semantics is given operational, denotational and axiomatic accounts that are in agreement one with the other in [15,19].

Definition 1 (Testing preorders) *For all $p, q \in P$,*

- $p \sqsubseteq_{may} q$ *iff, for each test t, p may pass t implies q may pass t;*
- $p \sqsubseteq_{must} q$ *iff, for each test t, p must pass t implies q must pass t;*
- $p \sqsubseteq_{T} q$ *iff* $p \sqsubseteq_{may} q$ *and* $p \sqsubseteq_{must} q$.

Example 1. It is well known that $\mathsf{nil} \sqsubseteq_{\text{may}} a.\mathsf{nil}$ and that $a.(b.\mathsf{nil} + c.\mathsf{nil}) \sqsubseteq_{\text{may}} a.b.\mathsf{nil} + a.c.\mathsf{nil}$. On the other hand, $\mathsf{nil} \not\sqsubseteq_{\text{must}} a.\mathsf{nil}$ and $a.(b.\mathsf{nil} + c.\mathsf{nil}) \not\sqsubseteq_{\text{must}} a.b.\mathsf{nil} + a.c.\mathsf{nil}$. Indeed, unlike nil, the process $a.\mathsf{nil}$ may fail the test $a.\mathsf{nil} + \tau.\omega.\mathsf{nil}$ (read 'ask the process under observation to do a and terminate unsuccessfully, or internally decide to succeed') and, unlike $a.(b.\mathsf{nil} + c.\mathsf{nil})$, the process $a.b.\mathsf{nil} + a.c.\mathsf{nil}$ may fail the test $a.b.\omega.\mathsf{nil}$ (read 'ask the process under observation to do a followed by b and then succeed').

3 Monitoring May Testing

We now characterise the may-testing preorder in terms of the basic framework for runtime monitoring presented in [16,17]. We first recall the needed definitions and results from those references in Sections 3.1–3.2 and then we use them to give a monitor-based version of the may-testing preorder in Section 3.3.

3.1 A Framework for Runtime Monitoring

We now review the operational framework proposed in [16,17] for runtime monitoring of properties expressed in Hennessy-Milner Logic with recursion [8,23]. In this framework, a monitor is a computational entity that observes the current system execution and uses the information so acquired to try to ascertain whether the system satisfies a given property.

Monitors We first define the notion of a monitor given in [16,17]. Monitors are states of an LTS, much like processes and tests. Syntactically, monitors are specified using expressions in a variation on the regular fragment of CCS, where the nil process is replaced by verdicts. A verdict can be one of yes, no and end, which represent acceptance, rejection and inconclusive termination, respectively.

Definition 1. *The set* Mon *of monitors is defined by the following grammar:*

$$\begin{array}{rcl} m, n \in \textsc{Mon} ::= v & \mid \alpha.m \mid m + n \mid \mathtt{rec}\ x.m \mid x \\ v ::= \mathsf{end} & \mid \mathsf{no} \mid \mathsf{yes} \end{array}$$

where x ranges over a countably infinite set of monitor variables.

An acceptance monitor *is one without occurrences of the verdict* no *and a* rejection monitor *is one that does not contain occurrences of the verdict* yes*.*

The behaviour of a monitor is defined by the derivation rules of Table 1, so monitors are states of an LTS whose transitions are those that are provable using those rules. Intuitively, a transition $m \xrightarrow{\alpha} m'$ indicates that a monitor in state m can analyse action α and become the monitor described by m' in doing so. We highlight the transition rule for verdicts in Table 1, describing the fact that, from a verdict state, any action can be analysed by transitioning to the same state; verdicts are thus irrevocable.

Monitored system Monitors are intended to run in conjunction with the system (process) they are analysing. While monitoring a process $p \in P$, a monitor $m \in \textsc{Mon}$ tries to mirror every visible action p performs. If m cannot match an action performed by p and it cannot perform an internal action, then p performs that action and continues executing, while m becomes the inconclusive end verdict. We are only looking at the visible actions and so we allow m and p to perform silent τ actions independently of each other.

$$\textsc{mAct}\ \frac{}{\alpha.m \xrightarrow{\alpha} m} \qquad \textsc{mRec}\ \frac{}{\mathtt{rec}\, x.m \xrightarrow{\tau} m[\mathtt{rec}\, x.m/x]}$$

$$\textsc{mSelL}\ \frac{m \xrightarrow{\mu} m'}{m + n \xrightarrow{\mu} m'} \qquad \textsc{mSelR}\ \frac{n \xrightarrow{\mu} n'}{m + n \xrightarrow{\mu} n'}$$

$$\textsc{mVerd}\ \frac{}{v \xrightarrow{\alpha} v}$$

where $\alpha \in \textsc{Act}$ and $\mu \in \textsc{Act} \cup \{\tau\}$.

Table 1. Monitor Dynamics

$$\textsc{iMon}\ \frac{p \xrightarrow{\alpha} p' \quad m \xrightarrow{\alpha} m'}{m \triangleleft p \xrightarrow{\alpha} m' \triangleleft p'} \qquad \textsc{iTer}\ \frac{p \xrightarrow{\alpha} p' \quad m \overset{o}{\not\longrightarrow} \quad m \overset{\tau}{\not\longrightarrow}}{m \triangleleft p \xrightarrow{\alpha} \mathsf{end} \triangleleft p'}$$

$$\textsc{iAsyP}\ \frac{p \xrightarrow{\tau} p'}{m \triangleleft p \xrightarrow{\tau} m \triangleleft p'} \qquad \textsc{iAsyM}\ \frac{m \xrightarrow{\tau} m'}{m \triangleleft p \xrightarrow{\tau} m' \triangleleft p}$$

Table 2. Monitored Systems

Definition 2. *A* monitored system *consists of a monitor $m \in \textsc{Mon}$ and a process $p \in P$ that run side-by-side, denoted $m \triangleleft p$. The behaviour of a monitored system is defined by the derivation rules in Table 2.*

The following lemmata describe how the monitor and system LTSs can be composed and decomposed according to instrumentation [17].

Lemma 1 (General Unzipping). *$m \triangleleft p \overset{s}{\Rightarrow} n \triangleleft q$ implies*

- *$p \overset{s}{\Rightarrow} q$ and*
- *$m \overset{s}{\Rightarrow} n$ or ($\exists s_1, s_2, \alpha\, \exists m'.\ s = s_1 \alpha s_2$, $m \overset{s_1}{\Rightarrow} m' \overset{\tau}{\not\longrightarrow}$, $m' \overset{o}{\not\longrightarrow}$ and $n = \mathsf{end}$).*

Lemma 2 (Zipping). *($p \overset{s}{\Rightarrow} q$ and $m \overset{s}{\Rightarrow} n$) implies $m \triangleleft p \overset{s}{\Rightarrow} n \triangleleft q$.*

If a monitored system $m \triangleleft p$ can reach a configuration where the monitor component is the yes verdict, we say that m accepts p, and similarly m rejects p if the monitored system can reach a configuration where the monitor component is no.

Definition 3 (Acceptance/Rejection). *We define*

$$\mathbf{acc}(m,p) \overset{def}{=} \exists s, p'.\ m \triangleleft p \overset{s}{\Rightarrow} \mathsf{yes} \triangleleft p' \quad \text{and}$$
$$\mathbf{rej}(m,p) \overset{def}{=} \exists s, p'.\ m \triangleleft p \overset{s}{\Rightarrow} \mathsf{no} \triangleleft p'.$$

The Logic We use μHML, the Hennessy-Milner logic with recursion, to describe properties of processes.

Definition 4. *The formulae of* μHML *are constructed using the following grammar:*

$$\begin{array}{rlll} \varphi, \psi \in \mu\text{HML} ::= & \mathit{tt} & \mid & \mathit{ff} \\ \mid & \varphi \wedge \psi & \mid & \varphi \vee \psi \\ \mid & \langle \alpha \rangle \varphi & \mid & [\alpha]\varphi \\ \mid & \min X.\varphi & \mid & \max X.\varphi \\ \mid & X & & \end{array}$$

where X *ranges over a countably infinite set of logical variables* LVar.

Formulae are evaluated in the context of a labelled transition system and an environment, $\rho : \text{LVar} \to 2^P$, which gives values to the logical variables in the formula. For an environment ρ, variable X, and set $S \subseteq P$, we write $\rho[X \mapsto S]$ for the environment which maps X to S and all $Y \neq X$ to $\rho(Y)$. The semantics for μHML formulae is given through a function $[\![\cdot]\!]$, which, given an environment ρ, maps each formula to a set of processes — namely the processes that satisfy the formula under the assumption that each $X \in \text{LVar}$ is satisfied by the processes in $\rho(X)$. The function $[\![\cdot]\!]$ is defined as follows:

$$\begin{aligned} [\![\mathtt{tt}, \rho]\!] &\stackrel{def}{=} P \quad \text{and} \quad [\![\mathtt{ff}, \rho]\!] \stackrel{def}{=} \emptyset \\ [\![\varphi_1 \wedge \varphi_2, \rho]\!] &\stackrel{def}{=} [\![\varphi_1, \rho]\!] \cap [\![\varphi_2, \rho]\!] \\ [\![\varphi_1 \vee \varphi_2, \rho]\!] &\stackrel{def}{=} [\![\varphi_1, \rho]\!] \cup [\![\varphi_2, \rho]\!] \\ [\![[\alpha]\varphi, \rho]\!] &\stackrel{def}{=} \left\{ p \mid \forall q.\ p \stackrel{\alpha}{\Rightarrow} q \text{ implies } q \in [\![\varphi, \rho]\!] \right\} \\ [\![\langle \alpha \rangle \varphi, \rho]\!] &\stackrel{def}{=} \left\{ p \mid \exists q.\ p \stackrel{\alpha}{\Rightarrow} q \text{ and } q \in [\![\varphi, \rho]\!] \right\} \\ [\![\max X.\varphi, \rho]\!] &\stackrel{def}{=} \bigcup \{ S \mid S \subseteq [\![\varphi, \rho[X \mapsto S]]\!] \} \\ [\![\min X.\varphi, \rho]\!] &\stackrel{def}{=} \bigcap \{ S \mid S \supseteq [\![\varphi, \rho[X \mapsto S]]\!] \} \\ [\![X, \rho]\!] &\stackrel{def}{=} \rho(X). \end{aligned}$$

A formula is closed when every occurrence of a variable X is in the scope of recursive operator $\max X$ or $\min X$. Note that the environment ρ has no effect on the semantics of a closed formula. Thus, for a closed formula φ, we often drop the environment from the notation for $[\![\cdot]\!]$ and write $[\![\varphi]\!]$ instead of $[\![\varphi, \rho]\!]$.

The safety fragment of μHML, denoted by sHML, and its dual co-safety fragment, cHML, are defined by the grammar:

$$\begin{array}{llllllll} \varphi, \psi \in \text{sHML} ::= & \mathtt{tt} & \mid \mathtt{ff} & \mid [\alpha]\varphi & \mid \varphi \wedge \psi & \mid \max X.\varphi & \mid X \\ \varphi, \psi \in \text{cHML} ::= & \mathtt{tt} & \mid \mathtt{ff} & \mid \langle \alpha \rangle \varphi & \mid \varphi \vee \psi & \mid \min X.\varphi & \mid X. \end{array}$$

Definition 5 (Monitorable Formulae). *We say that a rejection monitor* m *monitors a formula* $\varphi \in \mu$HML *for violation when, for each process* p, $\mathbf{rej}(m, p)$

if and only if $p \notin [\![\varphi]\!]$. *Similarly, an acceptance monitor* m *monitors a formula* $\varphi \in \mu\text{HML}$ *for satisfaction when, for each process* p, $\mathbf{acc}(m, p)$ *if and only if* $p \in [\![\varphi]\!]$. *A formula* $\varphi \in \mu\text{HML}$ *is* monitorable *if there is a monitor that monitors it for satisfaction or violation.*

3.2 Previous Results

The main result from [16,17] is to define a monitorable subset of μHML and show that it is maximally expressive. This subset is called MHML and consists of the safety and co-safety syntactic fragments of μHML: $\text{MHML} \stackrel{def}{=} \text{SHML} \cup \text{CHML}$. From now on, we focus on SHML, but the case of CHML is dual. The interested reader can see [16,17] for more details.

In order to prove that SHML is monitorable, in [16,17] Francalanza, Aceto, and Ingólfsdóttir define a monitor synthesis function, $(\!|\cdot|\!)$, which maps formulae to monitors, and show that for each $\varphi \in \text{SHML}$, $(\!|\varphi|\!)$ monitors φ for violation, in that $\mathbf{rej}((\!|\varphi|\!), p)$ holds exactly for those processes p for which $p \notin [\![\varphi]\!]$.

Definition 6 (Monitor Synthesis).

$$(\!|\mathit{tt}|\!) \stackrel{def}{=} \mathit{yes} \qquad (\!|\mathit{ff}|\!) \stackrel{def}{=} \mathit{no} \qquad (\!|X|\!) \stackrel{def}{=} x$$

$$(\!|[\alpha]\psi|\!) \stackrel{def}{=} \begin{cases} \alpha.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathit{yes} \\ \mathit{yes} & \text{otherwise} \end{cases}$$

$$(\!|\psi_1 \wedge \psi_2|\!) \stackrel{def}{=} \begin{cases} (\!|\psi_1|\!) & \text{if } (\!|\psi_2|\!) = \mathit{yes} \\ (\!|\psi_2|\!) & \text{if } (\!|\psi_1|\!) = \mathit{yes} \\ (\!|\psi_1|\!) + (\!|\psi_2|\!) & \text{otherwise} \end{cases}$$

$$(\!|\max X.\psi|\!) \stackrel{def}{=} \begin{cases} \mathtt{rec}\, x.(\!|\psi|\!) & \text{if } (\!|\psi|\!) \neq \mathit{yes} \\ \mathit{yes} & \text{otherwise} \end{cases}$$

Lemma 3. *For every formula* $\varphi \in \text{SHML}$, $(\!|\varphi|\!)$ *monitors* φ *for violation.*

Definition 7 (Formula Synthesis). *We define a formula synthesis function* $\|\cdot\|$ *from rejection monitors to* SHML.

$$\|\mathit{end}\| = \mathit{tt} \qquad \|\mathit{no}\| = \mathit{ff} \qquad \|x\| = X$$
$$\|\alpha.m\| = [\alpha]\|m\| \qquad \|m + n\| = \|m\| \wedge \|n\| \qquad \|\mathit{rec}\ x.m\| = \max X.\|m\|.$$

Lemma 4. *Every monitor* m *monitors* $\|m\|$ *for violation.*

As previously mentioned, dual results hold for CHML, whose formulae can be monitored for satisfaction using acceptance monitors.

3.3 May Testing via Monitors

The goal of this section is to show how the monitoring framework we just reviewed can be used to give an alternative characterisation of classic may-testing semantics à la De Nicola and Hennessy. As a first step, we define three natural preorders over states of LTSs that are induced by monitors. We will then show that these three preorders coincide with the may-testing preorder. In what follows, we assume a fixed LTS $L = \langle P, \textsc{Act}, \rightarrow \rangle$. All the results we present in this section hold for arbitrary LTSs.

Definition 2 (Monitoring preorders) *For all $p, q \in P$,*

- $p \sqsubseteq^A_M q$ *iff, for each acceptance monitor m,* $\mathbf{acc}(m, p)$ *implies* $\mathbf{acc}(m, q)$*;*
- $p \sqsubseteq^R_M q$ *iff, for each rejection monitor m,* $\mathbf{rej}(m, p)$ *implies* $\mathbf{rej}(m, q)$*;*
- $p \sqsubseteq q$ *iff* $p \sqsubseteq^A_M q$ *and* $p \sqsubseteq^R_M q$*.*

The following alternative characterization of the may testing preorder is well known—see [15,19].

Theorem 1 *For all $p, q \in P$, $p \sqsubseteq_{may} q$ iff traces$(p) \subseteq$ traces(q).*

One of the consequences of the above result is that tests of the form

$$a_1 . \ldots . a_n . \omega . \mathsf{nil},$$

with $n \geq 0$ and $a_1, \ldots, a_n \in \textsc{Act}$, suffice to characterize the may-testing preorder. Another one is that deciding the may-testing preorder and its induced equivalence over states in finite LTSs is PSPACE-complete [28].

Theorem 2 *For all $p, q \in P$, the following are equivalent:*

1. $p \sqsubseteq_{may} q$,
2. $p \sqsubseteq^A_M q$,
3. $p \sqsubseteq^R_M q$ *and*
4. $p \sqsubseteq q$.

To show the above result, we first prove that the preorder over processes induced by trace inclusion, which coincides with the may-testing preorder by Theorem 1, is included in both $\sqsubseteq^A_M$ and $\sqsubseteq^R_M$.

Lemma 3 *For all $p, q \in P$, if traces$(p) \subseteq$ traces(q) then $p \sqsubseteq^A_M q$ and $p \sqsubseteq^R_M q$.*

Proof. Assume that traces$(p) \subseteq$ traces(q). We first show that $p \sqsubseteq^A_M q$ holds.

To this end, let m be an acceptance monitor such that $\mathbf{acc}(m, p)$. By definition, this means that $m \triangleleft p \stackrel{s}{\Rightarrow} \mathsf{yes} \triangleleft p'$ for some $s \in \textsc{Act}^*$ and process p'. Using the 'unzipping lemma' (Lemma 1), this yields that $m \stackrel{s}{\Rightarrow} \mathsf{yes}$ and $p \stackrel{s}{\Rightarrow} p'$. So s is a trace of p and, by the proviso of the lemma, also of q. Thus, $q \stackrel{s}{\Rightarrow} q'$ for some q'. Using the 'zipping lemma' (Lemma 2), we obtain that $m \triangleleft q \stackrel{s}{\Rightarrow} \mathsf{yes} \triangleleft q'$,

which means that $\mathbf{acc}(m, q)$. Since m was an arbitrary acceptance monitor, we conclude that $p \sqsubseteq_M^A q$, and we are done.

The argument proving $p \sqsubseteq_M^R q$ is similar. Simply replace acceptance monitors with rejection monitors, **acc** with **rej** and yes with no in the above proof. □

Next, we establish that the converse inclusions also hold.

Lemma 4 *For all $p, q \in P$, if $p \sqsubseteq_M^A q$ or $p \sqsubseteq_M^R q$ then traces$(p) \subseteq$ traces(q).*

Proof. We limit ourselves to proving that if $p \sqsubseteq_M^A q$ then $\mathrm{traces}(p) \subseteq \mathrm{traces}(q)$, as the proof of the other implication is similar. To this end, assume that $p \sqsubseteq_M^A q$ and that $p \stackrel{s}{\Rightarrow} p'$ for some p'. We will show that $s \in \mathrm{traces}(q)$.

First of all, observe that, for each $t \in \text{Act}^*$, we can construct an acceptance monitor $m(t)$ thus:

$$\begin{aligned} m(\varepsilon) &= \mathsf{yes} \\ m(at') &= a.m(t') \ . \end{aligned}$$

Note that, for each $t \in \text{Act}^*$, by construction,

$$m(t) \stackrel{t'}{\Rightarrow} \mathsf{yes} \text{ iff } t = t' \ .$$

Since $p \stackrel{s}{\Rightarrow} p'$, the 'zipping lemma' (Lemma 2) yields that $m(s) \triangleleft p \stackrel{s}{\Rightarrow} \mathsf{yes} \triangleleft p'$. Thus $\mathbf{acc}(m(s), p)$ and, from the assumption that $p \sqsubseteq_M^A q$, we may infer that $\mathbf{acc}(m(s), q)$. By definition and the observation above, this means that $m(s) \triangleleft q \stackrel{s}{\Rightarrow} \mathsf{yes} \triangleleft q'$ for some q'. The 'unzipping lemma' (Lemma 1) now yields that $q \stackrel{s}{\Rightarrow} q'$, which was to be shown. □

Theorem 2 and the monitorability results presented in [1,17] can now be combined to obtain logical characterization results for the may-testing preorder. Even though these results are folklore, we believe that recasting them in terms of monitorability builds a pleasing connection between a classic testing preorder and runtime monitoring for μHML.

In the statement of the following result, for each process p, we define

$$\begin{aligned} \text{cHML}(p) &= \{\varphi \mid \varphi \in \text{cHML and } p \models \varphi\} \text{ and} \\ \text{sHML}(p) &= \{\varphi \mid \varphi \in \text{sHML and } p \models \varphi\}. \end{aligned}$$

Theorem 5 *For all $p, q \in P$, the following statements hold:*

1. $p \sqsubseteq_{may} q$ *iff* $\text{cHML}(p) \subseteq \text{cHML}(q)$.
2. $p \sqsubseteq_{may} q$ *iff* $\text{sHML}(q) \subseteq \text{sHML}(p)$.

Proof. We limit ourselves to presenting the proof of the second statement. The proof of the first statement is similar.

In order to establish the 'only if' implication, assume that $p \sqsubseteq_{\text{may}} q$ and $p \not\models \varphi$, for some $\varphi \in \text{sHML}$. We claim that $q \not\models \varphi$. To this end, observe that, as $p \not\models \varphi$ by assumption, Lemma 3 yields that $\mathbf{rej}(\llparenthesis \varphi \rrparenthesis, p)$. By Theorem 2 and

$p \sqsubseteq_{\text{may}} q$, we have that $p \sqsubseteq_M^R q$. Hence, $\mathbf{rej}(\langle\!|\varphi|\!\rangle, q)$ and, using Lemma 3 again, we may conclude that $q \not\models \varphi$, as claimed.

To prove the 'if' implication, we assume that $\text{sHML}(q) \subseteq \text{sHML}(p)$ and show that $p \sqsubseteq_{\text{may}} q$. By Theorem 2, this suffices to establish that claim. Suppose that $\mathbf{rej}(m, p)$ for some rejection monitor m. By Lemma 4, we have that $p \not\models \|m\| \in \text{sHML}$. By assumption, this means that $q \not\models \|m\|$ either. Hence, again using Lemma 4, we conclude that $\mathbf{rej}(m, q)$, and we are done. □

4 Monitoring Must Testing

As Theorem 2 indicates, the monitoring framework presented in [16,17] is not expressive enough to characterise the must-testing preorder, as monitor acceptance and rejection are only determined by the traces processes can perform. This means that monitors from the basic framework reviewed in Section 3.1 cannot distinguish, for instance, the processes described by the CCS expressions $a.(b.\mathsf{nil} + c.\mathsf{nil})$ and $a.b.\mathsf{nil} + a.c.\mathsf{nil}$, which are not must-testing equivalent because $a.(b.\mathsf{nil} + c.\mathsf{nil}) \not\sqsubseteq_{\text{must}} a.b.\mathsf{nil} + a.c.\mathsf{nil}$.

The first four authors presented a framework for parameterised monitorability in [1] and studied several of its instantiations. In what follows, we will first present one such instantiation (Section 4.1) and then show how a natural restriction of that specific monitoring framework offers a characterisation of must-testing semantics in terms of monitors (Section 4.2).

4.1 A Framework for Runtime Monitoring with Refusals

The instance of the monitoring framework from [1] we consider here is the one obtained by extending the syntax for rejection monitors given in Definition 1 with 'conditions' of the form $\text{ref}(a)$, where $a \in \text{Act}$. (In the terminology of [1], 'conditions' are predicates over processes.)

Formally, following [1, Sections 4.1 and 5.2], we extend the formation rules for monitors given in Definition 1 with those of the form $\text{ref}(a).m$, for each $a \in \text{Act}$. In the rest of this paper, we use the term *refusal monitors* for the monitors generated by that augmented grammar. In the behaviour of monitors, $\text{ref}(a)$ is treated as an ordinary action prefixing operator and thus the rules in Table 1 are extended with the following ones:

$$\frac{}{\text{ref}(a).m \xrightarrow{\text{ref}(a)} m}, \ a \in \text{Act}.$$

Intuitively, in the spirit of Phillips' refusal testing [27], a monitor of the form $\text{ref}(a).m$ checks whether the system it observes can *refuse* action a and, if so, continues monitoring as m. This is expressed by the following instrumentation rules for such conditions, which are added to the rules in Table 2:

$$\frac{m \xrightarrow{\text{ref}(a)} m' \quad p \overset{\tau}{\not\longrightarrow} \quad p \overset{a}{\not\longrightarrow}}{m \triangleleft p \xrightarrow{\tau} m' \triangleleft p} \ a \in \text{Act}. \tag{1}$$

In what follows, we say that p *refuses* a when $p \stackrel{\tau}{\not\rightarrow}$ and $p \stackrel{a}{\not\rightarrow}$.

The syntax for refusal monitors allows one to write monitors such as

$$a.\mathrm{ref}(b).c.\mathrm{ref}(d).\mathsf{no} \ .$$

Since our goal is to define a monitor-based characterisation of must-testing semantics, monitors that alternate the observation of action occurrences with that of refusals arbitrarily are too powerful. Indeed, they would characterise *failure-trace semantics*, which coincides with Phillips' refusal testing over image-finite processes [18]. Therefore, in what follows, we only consider the sub-language $\textsc{Mon}_F$ of refusal monitors that consists of the monitors m that are generated by the following grammar:

$$\begin{array}{rcl} m, n \in \textsc{Mon}_F ::= v & \mid \alpha.m \mid \mathrm{ref}(a).r \mid m + n \mid \mathtt{rec}\, x.m \mid x \\ r ::= \mathsf{no} & \mid \mathrm{ref}(a).r \\ v ::= \mathsf{end} & \mid \mathtt{no}, \end{array}$$

where x comes from a countably infinite set of monitor variables. We refer to those monitors as *failure monitors* and use them to define a preorder over processes as follows.

Definition 3 (Failure monitoring preorder) *For all $p, q \in P$,*

$$p \sqsubseteq_M^{Ref} q \text{ iff, for each failure monitor } m \in \textsc{Mon}_F, \ \mathbf{rej}(m, q) \text{ implies } \mathbf{rej}(m, p).$$

Intuitively, as in must-testing semantics, $p \sqsubseteq_M^{\mathrm{Ref}} q$ means that q is 'at least as well behaved as' p when its executions are observed by a failure monitor, in the sense that each failure monitor that rejects q will also reject p, and being rejected by a monitor is considered harmful. However, there might be some monitor that rejects p, but not q. For example, it is not too hard to see that $a.b.\mathsf{nil} + a.c.\mathsf{nil} \sqsubseteq_M^{\mathrm{Ref}} a.(b.\mathsf{nil} + c.\mathsf{nil})$, as each failure monitor that rejects $a.(b.\mathsf{nil} + c.\mathsf{nil})$ will also reject $a.b.\mathsf{nil} + a.c.\mathsf{nil}$. On the other hand, the monitor $a.\mathrm{ref}(b).\mathsf{no}$ rejects $a.b.\mathsf{nil} + a.c.\mathsf{nil}$, but not $a.(b.\mathsf{nil} + c.\mathsf{nil})$.

The following lemma describes how failure-monitor and system LTSs can be composed and decomposed according to instrumentation (cf. Lemmas 1 and 2).

Lemma 5 (Unzipping and zipping for failure monitors). *Let m be a failure monitor and let $p \in P$.*

1. *Assume that $m \triangleleft p \stackrel{s}{\Rightarrow} \mathsf{no} \triangleleft q$. Then*
   - *$p \stackrel{s}{\Rightarrow} q$ and*
   - *$m \stackrel{sref(a_1)\cdots ref(a_\ell)}{\Longrightarrow} \mathsf{no}$ for some $\ell \geq 0$ and $a_1 \ldots a_\ell \in \textsc{Act}^*$ such that q refuses a_i for each $i \in \{1, \ldots, \ell\}$.*
2. *Assume that $p \stackrel{s}{\Rightarrow} q$ and $m \stackrel{sref(a_1)\cdots ref(a_\ell)}{\Longrightarrow} \mathsf{no}$, for some $\ell \geq 0$ and $a_1 \ldots a_\ell \in \textsc{Act}^*$ such that q refuses a_i for each $i \in \{1, \ldots, \ell\}$. Then $m \triangleleft p \stackrel{s}{\Rightarrow} \mathsf{no} \triangleleft q$.*

4.2 Must Testing via Monitors

The goal of this section is to show how the monitoring framework we just reviewed can be used to give an alternative characterisation of classic must-testing semantics à la De Nicola and Hennessy over strongly-convergent, finitely-branching processes, which we now proceed to define.

Definition 4 (Strongly convergent and stable processes) *A process $p \in P$ is* convergent *iff it cannot perform an infinite sequence of τ transitions, that is, there is no infinite sequence $p_0, p_1, p_2, \ldots$ of processes in P such that $p_0 = p$ and $p_i \xrightarrow{\tau} p_{i+1}$ for each $i \geq 0$. We say that $p \in P$ is* strongly convergent *iff each of the processes that can be reached from it via a sequence of transitions is convergent.*

A process $p \in P$ is stable *iff it cannot perform a τ transition, that is, $p \stackrel{\tau}{\not\rightarrow}$.*

Definition 5 (Finitely branching processes) *A process $p \in P$ is* finitely branching *iff each of the processes that can be reached from it via a sequence of transitions has only finitely many outgoing transitions, that is, the set*

$$\{(\mu, q') \mid q \xrightarrow{\mu} q'\}$$

is finite for each q such that $p \stackrel{s}{\Rightarrow} q$ for some $s \in \text{Act}^*$.

The alternative characterisation of the must-testing preorder in terms of failures, which we will present in Theorem 6 to follow, is by now folklore in concurrency theory. To the best of our knowledge, it was first proved by Rocco De Nicola in [13] and offers a connection between must-testing semantics and failures semantics [11] that, at the time, was considered rather unexpected. As a corollary of that result and a classic one by Kanellakis and Smolka [21, Theorem 5.1], deciding the must-testing preorder and equivalence is PSPACE-complete.

Definition 6 (Initials and failures of a process) *Let $p \in P$.*

- *The set $I(p)$ of* initials *of p is $\{a \mid p \stackrel{a}{\Rightarrow}\}$.*
- *A pair (s, A) is a* failure *of a process $p \in P$ iff $s \in \text{Act}^*$, $A \subseteq \text{Act}$ and $I(p') \cap A = \emptyset$ for some stable p' such that $p \stackrel{s}{\Rightarrow} p'$. We write failures($p$) for the set of failures of process p.*

Theorem 6 (De Nicola [13]) *For all strongly convergent, finitely branching $p, q \in P$, $p \sqsubseteq_{must} q$ iff failures(q) $\subseteq$ failures(p).*

Remark 1. In the classic treatment of must-testing semantics over CCS and other process description languages, strongly convergent processes are guaranteed to be finitely branching. In this paper, for the sake of clarity, we have chosen to make the requirement that processes be finitely branching explicit.

Using the above theorem, we will now show the following result, to the effect that the must testing preorder coincides with the failure monitoring preorder from Definition 3.

Theorem 7 *For all strongly convergent, finitely branching $p, q \in P$, $p \sqsubseteq_{must} q$ iff $p \sqsubseteq_M^{Ref} q$.*

Proof. Let $p, q \in P$ be strongly convergent and finitely branching. By Theorem 6, it suffices only to prove that

$$\text{failures}(q) \subseteq \text{failures}(p) \text{ iff } p \sqsubseteq_M^{\text{Ref}} q.$$

We show the two implication separately.

To prove the 'only if' implication, assume that $\text{failures}(q) \subseteq \text{failures}(p)$ and that $\mathbf{rej}(m, q)$ for some failure monitor m. We claim that $\mathbf{rej}(m, p)$ also holds. To see this, observe that, since $\mathbf{rej}(m, q)$, there are some $s \in \text{Act}^*$ and some $q' \in P$ such that $m \triangleleft q \stackrel{s}{\Rightarrow} \mathsf{no} \triangleleft q'$. By the unzipping lemma for failure monitors (Lemma 5(1), we have that

- $q \stackrel{s}{\Rightarrow} q'$ and
- $m \xrightarrow{\text{sref}(a_1)\cdots\text{ref}(a_\ell)} \mathsf{no}$ for some $\ell \geq 0$ and $a_1 \ldots a_\ell \in \text{Act}^*$ such that q' refuses a_i for each $i \in \{1, \ldots, \ell\}$.

It follows that $(s, \{a_1, \ldots, a_\ell\})$ is a failure of q and, by our assumption, also of p. This means that $p \stackrel{s}{\Rightarrow} p'$ for some p' that refuses a_i for each $i \in \{1, \ldots, \ell\}$. Using the zipping lemma for failure monitors (Lemma 5(2), we conclude that $m \triangleleft p \stackrel{s}{\Rightarrow} \mathsf{no} \triangleleft p'$ and thus $\mathbf{rej}(m, p)$, as claimed.

To prove the 'if' implication, assume that $p \sqsubseteq_M^{\text{Ref}} q$. We claim that $\text{failures}(q)$ is included in $\text{failures}(p)$. This follows from the observation that rejection monitors can be used to encode the failures of a process. More precisely, consider a failure pair $(s, \{a_1, \ldots, a_\ell\})$. We can associate with it a rejection monitor $m(s, \{a_1, \ldots, a_\ell\})$ by induction on s thus:

$$\begin{aligned} m(\varepsilon, \{a_1, \ldots, a_\ell\}) &= \text{ref}(a_1).\ldots.\text{ref}(a_\ell).\mathsf{no} \quad \text{and} \\ m(as', \{a_1, \ldots, a_\ell\}) &= a.m(s', \{a_1, \ldots, a_\ell\}). \end{aligned}$$

By induction on s, it is easy to prove that $(s, \{a_1, \ldots, a_\ell\})$ is a failure of some process p iff $\mathbf{rej}(m(s, \{a_1, \ldots, a_\ell\}), p)$. We can now complete the proof of the claim thus:

$$\begin{aligned} (s, \{a_1, \ldots, a_\ell\}) \in \text{failures}(q) &\Leftrightarrow \mathbf{rej}(m(s, \{a_1, \ldots, a_\ell\}), q) \\ &\Rightarrow \mathbf{rej}(m(s, \{a_1, \ldots, a_\ell\}), p) \quad (\text{as } p \sqsubseteq_M^{\text{Ref}} q) \\ &\Leftrightarrow (s, \{a_1, \ldots, a_\ell\}) \in \text{failures}(p), \end{aligned}$$

and we are done. □

5 Conclusions

In this celebratory article, we have provided a formal connection between the theory of testing equivalence, developed by De Nicola and Hennessy during De

Nicola's PhD studies in Edinburgh, and the increasingly important field of runtime verification. The results in this study are not deep, but we hope that they highlight the pervasive nature of the ideas that underlie the definition of the testing equivalences from [15] and will convince our readers that the field of runtime monitoring owes much to the seminal work by De Nicola and Hennessy. Some of us were influenced by that work at the start of their careers [5,6,20] and are still working on testing-based approaches to the analysis of concurrent processes after about thirty years.

An interesting avenue for future research is to investigate whether the must-testing-like preorders over clients studied by Bernardi and Francalanza in [10] capture some interesting properties of monitors. So far, our work on monitorability has used the trace-based notions of *verdict equivalence* and *ω-verdict equivalence* over monitors—see, for instance, the papers [2,3,4].

Acknowledgments We are grateful to the anonymous reviewers for their suggestions, which helped us to improve the paper. Luca Aceto thanks Ugo Montanari, who asked him a question that led to the work presented in this article during a talk he gave at IMT Lucca in July 2018. Luca Aceto and Anna Ingólfsdóttir have been lucky to count Rocco De Nicola as one of their friends and mentors for many years. Luca Aceto's 'tesi di laurea' was jointly supervised by Rocco De Nicola and Alessandro Fantechi, and he was one of the first two students to graduate under Rocco De Nicola's supervision in 1986.

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A.: A framework for parameterized monitorability. In: Baier, C., Dal Lago, U. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018. Lecture Notes in Computer Science, vol. 10803, pp. 203–220. Springer (2018), https://doi.org/10.1007/978-3-319-89366-2_11
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö.: On the complexity of determinizing monitors. In: Carayol, A., Nicaud, C. (eds.) Implementation and Application of Automata - 22nd International Conference, CIAA 2017. Lecture Notes in Computer Science, vol. 10329, pp. 1–13. Springer (2017), https://doi.org/10.1007/978-3-319-60134-2_1
3. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in monitorability: From branching to linear time and back again. Proceedings of the ACM on Programming Languages **3**(POPL), 52:1–52:29 (2019), https://dl.acm.org/citation.cfm?id=3290365
4. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: The cost of monitoring alone. CoRR **abs/1902.05152** (2019), http://arxiv.org/abs/1902.05152
5. Aceto, L., De Nicola, R., Fantechi, A.: Testing equivalences for event structures. In: Venturini Zilli, M. (ed.) Mathematical Models for the Semantics of Parallelism, Advanced School. Lecture Notes in Computer Science, vol. 280, pp. 1–20. Springer (1986), https://doi.org/10.1007/3-540-18419-8_9

6. Aceto, L., Ingólfsdóttir, A.: A theory of testing for ACP. In: Baeten, J.C.M., Groote, J.F. (eds.) CONCUR '91, 2nd International Conference on Concurrency Theory. Lecture Notes in Computer Science, vol. 527, pp. 78–95. Springer (1991), https://doi.org/10.1007/3-540-54430-5_82
7. Aceto, L., Ingólfsdóttir, A.: Testing Hennessy-Milner logic with recursion. In: Thomas, W. (ed.) Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99. Lecture Notes in Computer Science, vol. 1578, pp. 41–55. Springer (1999), https://doi.org/10.1007/3-540-49019-1_4
8. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, New York, NY, USA (2007). https://doi.org/10.1017/cbo9780511814105
9. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457. Springer (2018), https://doi.org/10.1007/978-3-319-75632-5
10. Bernardi, G.T., Francalanza, A.: Full-abstraction for client testing preorders. Science of Computer Programming **168**, 94–117 (2018), https://doi.org/10.1016/j.scico.2018.08.004
11. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3), 560–599 (1984), https://doi.org/10.1145/828.833
12. Cerone, A., Hennessy, M.: Process behaviour: Formulae vs. tests (extended abstract). In: Fröschle, S.B., Valencia, F.D. (eds.) Proceedings 17th International Workshop on Expressiveness in Concurrency, EXPRESS'10. Electronic Proceedings in Theoretical Computer Science, vol. 41, pp. 31–45 (2010), https://doi.org/10.4204/EPTCS.41.3
13. De Nicola, R.: Extensional equivalences for transition systems. Acta Informatica **24**(2), 211–237 (1987), https://doi.org/10.1007/BF00264365
14. De Nicola, R., Hennessy, M.: Testing equivalence for processes. In: Díaz, J. (ed.) Automata, Languages and Programming, 10th Colloquium. Lecture Notes in Computer Science, vol. 154, pp. 548–560. Springer (1983), https://doi.org/10.1007/BFb0036936
15. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science **34**, 83–133 (1984), https://doi.org/10.1016/0304-3975(84)90113-0
16. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: On verifying Hennessy-Milner logic with recursion at runtime. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification - 6th International Conference, RV 2015. Lecture Notes in Computer Science, vol. 9333, pp. 71–86. Springer (2015), https://doi.org/10.1007/978-3-319-23820-3_5
17. Francalanza, A., Aceto, L., Ingolfsdottir, A.: Monitorability for the Hennessy–Milner Logic with recursion. Formal Methods in System Design **51**(1), 87–116 (2017), http://dx.doi.org/10.1007/s10703-017-0273-z
18. van Glabbeek, R.J.: The linear time – branching time spectrum I: The semantics of concrete, sequential processes. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, chap. 1, pp. 3–99. Elsevier (2001)
19. Hennessy, M.: Algebraic Theory of Processes. Foundations of Computing, MIT Press (May 1988)
20. Hennessy, M., Ingólfsdóttir, A.: A theory of communicating processes with value passing. Information and Computation **107**(2), 202–236 (1993), https://doi.org/10.1006/inco.1993.1067
21. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation **86**(1), 43–68 (1990), https://doi.org/10.1016/0890-5401(90)90025-D

22. Keller, R.M.: Formal verification of parallel programs. Communications of the ACM **19**(7), 371–384 (1976), https://doi.org/10.1145/360248.360251
23. Larsen, K.G.: Proof systems for satisfiability in Hennessy-Milner logic with recursion. Theoretical Computer Science **72**(2), 265–288 (1990). https://doi.org/10.1016/0304-3975(90)90038-J, http://www.sciencedirect.com/science/article/pii/030439759090038J
24. Loemker, L.E. (ed.): G. W. Leibniz: Philosophical Papers and Letters. Dordrecht: D. Reidel, 2nd edn. (1969)
25. Milner, R.: A Calculus of Communicating Systems. Springer, Berlin, Heidelberg (1980)
26. Morris, J.H.: Lambda-Calculus Models of Programming Languages. Ph.D. thesis, Massachusetts Institute of Technology (1968)
27. Phillips, I.: Refusal testing. Theoretical Computer Science **50**(3), 241–284 (1987). https://doi.org/10.1016/0304-3975(87)90117-4, http://www.sciencedirect.com/science/article/pii/0304397587901174
28. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time: Preliminary report. In: Aho, A.V., Borodin, A., Constable, R.L., Floyd, R.W., Harrison, M.A., Karp, R.M., Strong, H.R. (eds.) Proceedings of the 5th Annual ACM Symposium on Theory of Computing. pp. 1–9. ACM (1973), https://doi.org/10.1145/800125.804029
29. Winskel, G.: Synchronization trees. Theoretical Computer Science **34**, 33–82 (1984), https://doi.org/10.1016/0304-3975(84)90112-9