Brzozowski's and Up-To Algorithms for Must Testing

Filippo Bonchi[1], Georgiana Caltais[2], Damien Pous[1], and Alexandra Silva[3,⋆]

[1] ENS Lyon, U. de Lyon, CNRS, INRIA, UCBL
[2] Reykjavik University
[3] Radboud University Nijmegen

Abstract. Checking language equivalence (or inclusion) of finite automata is a classical problem in Computer Science, which has recently received a renewed interest and found novel and more effective solutions, such as approaches based on antichains or bisimulations up-to. Several notions of equivalence (or preorder) have been proposed for the analysis of concurrent systems. Usually, the problem of checking these equivalences is reduced to checking bisimilarity. In this paper, we take a different approach and propose to adapt algorithms for language equivalence to check one prime equivalence in concurrency theory, must testing semantics. To achieve this transfer of technology from language to must semantics, we take a coalgebraic outlook at the problem.

1 Introduction

Determining whether two systems exhibit the same behavior under a given notion of equivalence is a recurring problem in different areas from Computer Science, from compiler analysis, to program verification, to concurrency theory. A widely accepted notion of equivalence is that two systems are equivalent if they behave the same when placed in the same context.

We will focus on the equivalence problem in the context of concurrency theory and process calculi. Systems are processes and contexts will be given by sets of tests a process should obey. This leads us to consider standard behavioural equivalences and preorders for process calculi, in particular *must testing* [14]: two systems are equivalent if they pass exactly the same tests, in all their executions.

The problem of automatically checking such testing equivalences is usually reduced to the problem of checking bisimilarity, as proposed in [12] and implemented in several tools [13,10]. In a nutshell, equivalence is checked as follows. Two processes are considered, given by their labeled transition systems (LTS's). Then, the given LTS's are first transformed into "acceptance graphs", using a construction which is reminiscent of the *determinization* of non-deterministic automata (NDA). Finally, bisimilarity is checked via the *partition refinement*

⋆ Also affiliated to Centrum Wiskunde & Informatica (Amsterdam, The Netherlands) and HASLab / INESC TEC, Universidade do Minho (Braga, Portugal).

C.-c. Shan (Ed.): APLAS 2013, LNCS 8301, pp. 1–16, 2013.

algorithm [17,22]. And one can answer the question of testing equivalence because gladly bisimilarity in acceptance graphs coincides with testing equivalence in the original LTS's.

The partition refinement algorithm, which is the best-known for minimizing LTS's w.r.t. bisimilarity, is analogous to Hopcroft's algorithm [16] for minimizing deterministic automatata (DA) w.r.t. language equivalence. In both cases, a partition of the state space is iteratively refined until a fixpoint is reached. Thus, the above procedure for checking testing semantics [12] is in essence the same as the classical procedure for checking language equivalence of NDA: first determinize and then compute a (largest) fixpoint.

In this work, we propose to transfer other algorithms for language equivalence, which are not available for bisimilarity, to the world of testing semantics. In order to achieve this, we take a coalgebraic perspective at the problem in hand, which allows us to study the constructions and the semantics in a uniform fashion. The abstract framework of *coalgebras* makes it possible to study different kinds of state based systems in a uniform way [26]. In particular, both the determinization of NDA's and the construction of acceptance graphs in [12] are instances of the generalized powerset construction [28,20,11]. This is the key observation of this work, which enables us to devise the presented algorithms.

First, we consider *Brzozowski's algorithm* [9] which transforms an NDA into the minimal deterministic automaton accepting the same language in a rather magical way: the input automaton is reversed (by swapping final and initial states and reversing its transitions), determinized, reversed and determinized once more. This somewhat intriguing algorithm can be explained in terms of duality and coalgebras [4,2]. The coalgebraic outlook in [4] has several generalization of Brzozowski's algorithm to other types of transition systems, including Moore machines. This paves the way to adapt Brzozowski's algorithm for checking must semantics, which we will do in this paper.

Next, we consider several more efficient algorithms that have been recently introduced in a series of papers [32,1,7]. These algorithms rely on different kinds of *(bi)simulations up-to*, which are proof techniques originally proposed for process calculi [21,27]. From these algorithms, we choose the one in [7] (`HKC`) which has been introduced by a subset of the authors and which, as we will show, can be adapted to must testing using a coalgebraic characterization of must equivalence, which we will also introduce.

Comparing these three families of algorithms (partition refinement [12], Brzozowski and bisimulations up-to) is not a simple task: both the problems of checking language and must equivalence are PSPACE-complete [17] but, in both cases, the theoretical complexity appears not to be problematic in practice, so that an empirical evaluation is more desirable. In [31,29], experiments have shown that Brzozowski's algorithm performs better than Hopcroft for "high-density" NDA's, while Hopcroft is more efficient for generic NDA's. Both algorithms appear to be rather inefficient compared to those of the new generation [32,1,7]. It is out of the scope of this paper to present an experimental comparison of these algorithms and we confine our work to showing concrete examples where

HKC and Brzozowski's algorithm are exponentially more efficient than the other approaches.

Contributions. The main contributions of this work are:

- The coalgebraic treatment of must semantics (preorder and equivalence).
- The adaptation of HKC and Brzozowski's algorithm for must semantics. For the latter, this includes an optimization which avoids an expensive determinization step.
- The evidence that the coalgebraic analysis of systems yields not only a good mathematical theory of their semantics but also a rich playground to devise algorithms.
- An interactive applet allowing one to experiment with these algorithms [6].

The full version of this paper [5] contains further optimizations for the algorithms, their proofs of correctness, the formal connections with the work in [12] and the results of experiments checking the equivalence of an ideal and a distributed multiway synchronisation protocol [23].

Related Work. Another coalgebraic outlook on must is presented in [8] which introduces a fully abstract semantics for CSP. The main difference with our work consists in the fact that [8] builds a coalgebra from the syntactic terms of CSP, while here we build a coalgebra starting from LTS's via the generalized powerset construction [28]. Our approach puts in evidence the underlying semilattice structure which is needed for defining bisimulations up-to and HKC. As a further coalgebraic approach to testing, it is worth mentioning test-suites [18], which however do not tackle must testing. A coalgebraic characterization of other semantics of the linear time/branching time spectrum is given in [3].

Notation. We denote sets by capital letters $X, Y, S, T \ldots$ and functions by lower case letters $f, g, \ldots$ Given sets X and Y, $X \times Y$ is the Cartesian product of X and Y, $X + Y$ is the disjoint union and X^Y is the set of functions $f\colon Y \to X$. The collection of *finite* subsets of X is denoted by $\mathcal{P}(X)$ (or just $\mathcal{P}X$). These operations, defined on sets, can analogously be defined on functions [26], yielding (bi-)functors on **Set**, the category of sets and functions. For a set of symbols A, A^* denotes the set of all finite words over A; ε the empty word; and $w_1 \cdot w_2$ (or $w_1 w_2$) the concatenation of words $w_1, w_2 \in A^*$. We use 2 to denote the set $\{0, 1\}$ and 2^{A^*} to denote the set of all formal languages over A. A *semilattice with bottom* $(X, \sqcup, 0)$ consists of a set X and a binary operation $\sqcup\colon X \times X \to X$ that is associative, commutative, idempotent (ACI) and has $0 \in X$ (the bottom) as identity. A *homomorphism* (of semilattices with bottom) is a function preserving $\sqcup$ and 0. Every semilattice induces a *partial order* defined as $x \sqsubseteq y$ iff $x \sqcup y = y$. The set 2 is a semilattice when taking $\sqcup$ to be the ordinary Boolean disjunction. Also the set of all languages 2^{A^*} carries a semilattice structure where $\sqcup$ is the union of languages and 0 is the empty language. More generally, for any set S, $\mathcal{P}(S)$ is a semilattice where $\sqcup$ is the union of sets and 0 is the empty set. In the rest of the paper we will indiscriminately use 0 to denote the element $0 \in 2$, the

empty language in 2^{A^*} and the empty set in $\mathcal{P}(S)$. Analogously, $\sqcup$ will denote the "Boolean or" in 2, the union of languages in 2^{A^*} and the union of sets in $\mathcal{P}(S)$.

Acknowledgments. This work was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program Investissements dAvenir (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR). In addition, Filippo Bonchi was partially supported by the projects PEPS-CNRS CoGIP, ANR-09-BLAN-0169-01, and ANR 12IS02001 PACE. Georgiana Caltais has been partially supported by the project 'Meta-theory of Algebraic Process Theories' (nr. 100014021) of the Icelandic Research Fund. Damien Pous was partially supported by the PiCoq project, ANR-10-BLAN-0305. Alexandra Silva was partially supported by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. `FCOMP-01-0124-FEDER-020537` and `SFRH/BPD/71956/2010`.

2 Background

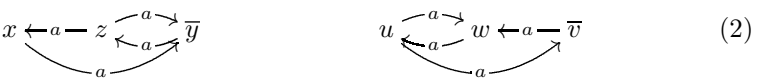
The core of this paper is about the problem of checking whether two states in a transition system are testing equivalent by reducing it to the classical problem of checking language equivalence. We will consider different types of transition systems, deterministic and non-deterministic, which we will formally describe next, together with their language semantics.

A *deterministic automaton* (DA) over the alphabet A is a pair $(S, \langle o, t\rangle)$, where S is a set of states and $\langle o, t\rangle \colon S \to 2 \times S^A$ is a function with two components: o, the output function, determines whether a state x is final ($o(x) = 1$) or not ($o(x) = 0$); and t, the transition function, returns for each state and each input letter, the next state. From any DA, there exists a function $[\![-]\!] \colon S \to 2^{A^*}$ mapping states to languages, defined for all $x \in S$ as follows:

$$[\![x]\!](\varepsilon) = o(x) \qquad\qquad [\![x]\!](a \cdot w) = [\![t(x)(a)]\!](w) \tag{1}$$

The language $[\![x]\!]$ is called the language accepted by x. Given an automaton $(S, \langle o, t\rangle)$, the states $x, y \in S$ are said to be *language equivalent* iff they accept they same language.

A *non-deterministic automaton* (NDA) is similar to a DA but the transition function returns a set of next-states instead of a single state. Thus, an NDA over the input alphabet A is a pair $(S, \langle o, t\rangle)$, where S is a set of states and $\langle o, t\rangle \colon S \to 2 \times (\mathcal{P}(S))^A$. An example is depicted below (final states are overlined, labeled edges represent transitions).



(2)

Classically, in order to recover language semantics of NDA, one uses the *subset (or powerset) construction*, transforming every NDA $(S, \langle o, t\rangle)$ into the DA

$(\mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$ where $o^\sharp \colon \mathcal{P}(S) \to 2$ and $t^\sharp \colon \mathcal{P}(S) \to \mathcal{P}(S)^A$ are defined for all $X \in \mathcal{P}(S)$ as

$$o^\sharp(X) = \bigsqcup_{x \in X} o(x) \qquad\qquad t^\sharp(X)(a) = \bigsqcup_{x \in X} t(x)(a) \ .$$

For instance with the NDA from (2), $o^\sharp(\{x, y\}) = 0 \sqcup 1 = 1$ (*i.e.*, the state $\{x, y\}$ is final) and $t^\sharp(\{x, y\})(a) = \{y\} \sqcup \{z\} = \{y, z\}$ (*i.e.*, $\{x, y\} \xrightarrow{a} \{y, z\}$).

Since $(\mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$ is a deterministic automaton, we can now apply (1), yielding a function $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ mapping *sets* of states to languages. Given two states x and y, we say that they are language equivalent iff $[\![\{x\}]\!] = [\![\{y\}]\!]$. More generally, for two sets of states $X, Y \subseteq S$, we say that X and Y are language equivalent iff $[\![X]\!] = [\![Y]\!]$.

In order to introduce the algorithms in full generality, it is important to remark here that the sets 2, $\mathcal{P}(S)$, $\mathcal{P}(S)^A$, $2 \times \mathcal{P}(S)^A$ and 2^{A^*} carry semilattices with bottom and that the functions $\langle o^\sharp, t^\sharp \rangle \colon \mathcal{P}(S) \to 2 \times \mathcal{P}(S)^A$ and $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ are homomorphisms.

2.1 Checking Language Equivalence via Bisimulation Up-To

We recall the algorithm `HKC` from [7]. We first define a notion of bisimulation on sets of states. We make explicit the underlying notion of progression.

Definition 1 (Progression, Bisimulation). *Let $(S, \langle o, t \rangle)$ be an NDA. Given two relations $R, R' \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$, R* progresses to *R', denoted $R \rightarrowtail R'$, if whenever $X \ R \ Y$ then*

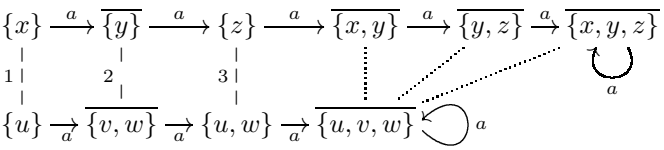
$$\textit{1. } o^\sharp(X) = o^\sharp(Y) \quad \textit{ and } \quad \textit{2. for all } a \in A,\ t^\sharp(X)(a) \ R' \ t^\sharp(Y)(a).$$

A bisimulation *is a relation R such that $R \rightarrowtail R$.*

This definition considers the states, the transitions and the outputs of the *determinized* NDA. For this reason, the bisimulation proof technique is sound and complete for language equivalence rather than for the standard notion of bisimilarity by Milner and Park [21].

Proposition 1 (Coinduction [7]). *For all $X, Y \in \mathcal{P}(S)$, $[\![X]\!] = [\![Y]\!]$ iff there exists a bisimulation that relates X and Y.*

For an example, we want to check the equivalence of $\{x\}$ and $\{u\}$ of the NDA in (2). The part of the determinized NDA that is reachable from $\{x\}$ and $\{u\}$ is depicted below. The relation consisting of dashed and dotted lines is a bisimulation which proves that $[\![\{x\}]\!] = [\![\{u\}]\!]$.



(3)

The dashed lines (numbered by 1, 2, 3) form a smaller relation which is not a bisimulation, but a *bisimulation up-to congruence*: the equivalence of $\{x, y\}$ and $\{u, v, w\}$ can be immediately deduced from the fact that $\{x\}$ is related to $\{u\}$ and $\{y\}$ to $\{v, w\}$. In order to formally introduce bisimulations up-to congruence, we need to define first the *congruence closure* $c(R)$ of a relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$. This is done inductively, by the following rules:

$$\frac{X\ R\ Y}{X\ c(R)\ Y} \qquad \frac{}{X\ c(R)\ X} \qquad \frac{X\ c(R)\ Y}{Y\ c(R)\ X} \tag{4}$$

$$\frac{X\ c(R)\ Y\ \ Y\ c(R)\ Z}{X\ c(R)\ Z} \qquad \frac{X_1\ c(R)\ Y_1\ \ X_2\ c(R)\ Y_2}{X_1 \sqcup X_2\ c(R)\ Y_1 \sqcup Y_2}$$

Note that the term "congruence" here is intended w.r.t. the semilattice structure carried by the state space $\mathcal{P}(S)$ of the determinized automaton. Intuitively, $c(R)$ is the smallest equivalence relation containing R and which is closed w.r.t $\sqcup$.

Definition 2 (Bisimulation up-to congruence). *A relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ is a* bisimulation up-to c *if $R \rightarrowtail c(R)$,* i.e., *whenever $X\ R\ Y$ then*

1. $o^\sharp(X) = o^\sharp(Y)$ and 2. for all $a \in A$, $t^\sharp(X)(a)\ c(R)\ t^\sharp(Y)(a)$.

Theorem 1 ([7]). *Any bisimulation up-to c is contained in a bisimulation.*

The corresponding algorithm (`HKC`) is given in Figure 1 (top). Starting from an NDA $(S, \langle o, t\rangle)$ and considering the determinized automaton $(\mathcal{P}(S), \langle o^\sharp, t^\sharp\rangle)$, it can be used to check language equivalence of two sets of states X and Y. Starting from the pair (X, Y), the algorithm builds a relation R that, in case of success, is a bisimulation up-to congruence. In order to do that, it employs the set *todo* which, intuitively, at any step of the execution, contains the pairs (X', Y') that must be checked: if (X', Y') already belongs to $c(R \cup todo)$, then it does not need to be checked. Otherwise, the algorithm checks if X' and Y' have the same outputs. If $o^\sharp(X') \neq o^\sharp(Y')$ then X and Y are different, otherwise the algorithm inserts (X', Y') in R and, for all $a \in A$, the pairs $(t^\sharp(X')(a), t^\sharp(Y')(a))$ in *todo*. The check $(X', Y') \in c(R \cup todo)$ at step `2.2` is done with the rewriting algorithm of [7, Section 3.4].

Proposition 2. *For all $X, Y \in \mathcal{P}(S)$, $[\![X]\!] = [\![Y]\!]$ iff `HKC`(X, Y).*

The iterations corresponding to the execution of `HKC`$(\{x\}, \{u\})$ on the NDA in (2) are concisely described by the numbered dashed lines in (3). Observe that only a small portion of the determinized automaton is explored; this fact usually makes `HKC` more efficient than the algorithms based on minimization, that need to build the whole reachable part of the determinized automaton.

2.2 Checking Language Equivalence via Brzozowski's Algorithm

The problem of checking language equivalence of two sets of states X and Y of a non-deterministic finite automaton can be reduced to that of building the

minimal DA for $[\![X]\!]$ and $[\![Y]\!]$ and checking whether they are the same (up to isomorphism). The most well-known procedure consists in first determinizing the NDA and then minimizing it with the Hopcroft algorithm [16]. Another interesting solution is Brzozowski's algorithm [9].

To explain the latter, it is convenient to consider a set of *initial states* I. Given an NDA $(S, \langle o, t\rangle)$ and a set of states I, Brzozowski's algorithm computes the minimal automaton for the language $[\![I]\!]$ by performing the 4 steps in Figure 1 (bottom).

The operation `reverse and determinize` takes as input an NDA $(S, \langle o, t\rangle)$ and returns a DA $(\mathcal{P}(S), \langle \overline{o}_R, \overline{t}_R\rangle)$ where the functions $\overline{o}_R \colon \mathcal{P}(S) \to 2$ and $\overline{t}_R \colon \mathcal{P}(S) \to \mathcal{P}(S)^A$ are defined for all $X \in \mathcal{P}(S)$ as $\overline{o}_R(X) = 1$ iff $X \cap I \neq 0$ and $\overline{t}_R(X)(a) = \{x \in S \mid t(x)(a) \cap X \neq 0\}$. The new initial state is the set of accepting states of the original NDA: $\overline{I}_R = \{x \mid o(x) = 1\}$. The second step consists in taking the part of $(\mathcal{P}(S), \langle \overline{o}_R, \overline{t}_R\rangle)$ which is reachable from $\overline{I}_R$. The third and the fourth steps perform this procedure once more.

As an example, consider the NDA in (2) with the set of initial states $I = \{x\}$. Brzozowski's algorithm builds the minimal DA accepting $[\![\{x\}]\!]$ as follows. After the first two steps, it returns the following DA where the initial state is $\{y\}$.

$$\{y\} \xrightarrow{a} \overline{\{x,z\}} \xrightarrow{a} \{z,y\} \xrightarrow{a} \overline{\{x,y,z\}} \overset{a}{\curvearrowleft}$$

After steps `3` and `4`, it returns the DA below with initial state $\{\{x,z\}\{x,y,z\}\}$.

$$\{\{x,z\}\{x,y,z\}\} \xrightarrow{a} \overline{\{\{y\}\{z,y\}\{x,y,z\}\}} \xrightarrow{a} \{\{x,z\}\{z,y\}\{x,y,z\}\} \downarrow a \quad \overline{\{\{y\}\{x,z\}\{z,y\}\{x,y,z\}\}} \overset{a}{\curvearrowleft}$$

Computing the minimal NDA in (2) with the set of initial states $I = \{u\}$ results in an isomorphic automaton, showing the equivalence of x and u.

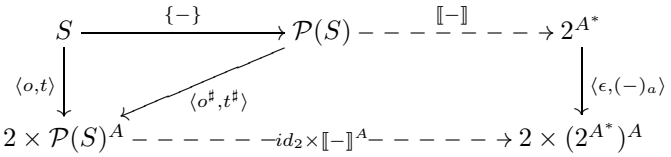
2.3 Generalized Powerset Construction

The notions introduced above can be easily described using *coalgebras*. Given a functor $F \colon \mathbf{Set} \to \mathbf{Set}$, an *$F$-coalgebra* is a pair (S, f) where S is a set of states and $f \colon S \to F(S)$ is its *transition structure*. F intuitively determines the "type" of the transitions. An *F-homomorphism* from an F-coalgebra (S, f) to an F-coalgebra (T, g) is a function $h \colon S \to T$ preserving the transition structure, *i.e.*, $g \circ h = F(h) \circ f$. An F-coalgebra (Ω, ω) is said to be *final* if for any F-coalgebra (S, f) there exists a unique F-homomorphism $[\![-]\!] \colon S \to \Omega$. Intuitively, Ω represents the universe of "F-behaviours" and $[\![-]\!]$ represents the semantic map associating states to their behaviours. Two states $x, y \in X$ are said *F-behaviourally equivalent* iff $[\![x]\!] = [\![y]\!]$. Such equivalence can be proved using *F-bisimulations* [26]. For lack of space, we refer the reader to [25] for their categorical definitions. Given a behaviour $b \in \Omega$, the *minimal coalgebra* realizing b is the part of (Ω, ω) that is reachable from b.

Let us exemplify for DA's how these abstract notions yield the expected concrete notions. DA's are coalgebras for the functor $F(S) = 2 \times S^A$. The final coalgebra of this functor is the set 2^{A^*} of formal languages over A, or more precisely, the pair $(2^{A^*}, \langle \epsilon, (-)_a \rangle)$ where $\langle \epsilon, (-)_a \rangle$, given a language L, determines whether or not the empty word is in the language ($\epsilon(L) = 1$ or $\epsilon(L) = 0$, resp.) and, for each input letter a, returns the *a-derivative* of L: $L_a = \{w \in A^* \mid aw \in L\}$. The unique map $[\![-]\!]$ into the final coalgebra 2^{A^*} is precisely the map which assigns to each state the language that it recognizes. For any language $L \in 2^{A^*}$, the minimal automaton for L is the part of $(2^{A^*}, \langle \epsilon, (-)_a \rangle)$ that is reachable from L.

In Section 3, we will use *Moore machines* which are coalgebras for the functor $F(S) = B \times S^A$. These are like DA's, but with outputs in a fixed set B. The unique F-homomorphism to the final coalgebra $[\![-]\!] \colon S \to B^{A^*}$ is defined exactly as for DA's by the equations in (1). Note that the behaviours of Moore machines are functions $\varphi \colon A^* \to B$, rather than subsets of A^*. For each behaviour $\varphi \in B^{A^*}$, there exists a minimal Moore machine realizing it.

Recall that an NDA is a pair $(S, \langle o, t \rangle)$, where $\langle o, t \rangle \colon S \to 2 \times (\mathcal{P}(S))^A$. As explained above, to recover language semantics one needs to use the subset construction, which transforms an NDA into a DA. More abstractly, this can be captured by observing that the type functor of NDA's – $2 \times \mathcal{P}(-)^A$ – is a composition of the functor $F(S) = 2 \times S^A$ (that is the functor for DA's) and the monad $T(S) = \mathcal{P}(S)$. $\mathcal{P}$-algebras are exactly semilattices with bottom and $\mathcal{P}$-algebra morphisms are the ones of semilattices with bottom. Now note that (a) the F-coalgebra $(\mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$ resulting of the powerset construction is a morphism of semilattices, (b) 2^{A^*} carries a semilattice structure and (c) $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ is a morphism of semilattices. This is summarized by the following commuting diagram:



In the diagram above, one can replace $2 \times -^A$ and $\mathcal{P}$ by arbitrary F and T as long as $FT(S)$ *carries a T-algebra structure*. In fact, given an *FT-coalgebra*, that is $(S, f \colon S \to FT(S))$, if $FT(S)$ carries a T-algebra structure h, then (a) one can define an F-coalgebra $(T(S), f^\sharp = h \circ Tf)$ where $f^\sharp \colon T(S) \to FT(S)$ is a T-algebra morphism (b) the final F-coalgebra (Ω, ω) carries a T-algebra and (c) the F-homomorphism $[\![-]\!] \colon T(S) \to \Omega$ is a T-algebra morphism.

The F-coalgebra $(T(S), f^\sharp)$ is (together with the multiplication $\mu \colon TT(S) \to T(S)$) a *bialgebra* for some distributive law $\lambda \colon FT \Rightarrow TF$ (we refer the reader to [19] for a nice introduction on this topic). The behavioural equivalence of bialgebras can be proved either via bisimulation, or, like in Section 2.1, via *bisimulation up-to congruence* [20,25]: the result that justifies `HKC` (Theorem 1) generalises to this setting – the congruence being taken w.r.t. the algebraic structure μ. This is what allows us to move to must semantics.

HKC(X, Y):

```
(1) R is empty; todo is {(X', Y')};
(2) while todo is not empty, do
 (2.1) extract (X', Y') from todo;
 (2.2) if (X', Y') ∈ c(R ∪ todo) then continue;
 (2.3) if o♯(X') ≠ o♯(Y') then return false;
 (2.4) for all a ∈ A,
         insert (t♯(X')(a), t♯(Y')(a)) in todo;
 (2.5) insert (X', Y') in R;
(3) return true;
```

Brzozowski:

```
(1) reverse and determinize;
(2) take the reachable part;
(3) reverse and determinize;
(4) take the reachable part.
```

Fig. 1. Top: Generic HKC algorithm, parametric on $o^\sharp$, $t^\sharp$ and c. Bottom: Generic Brzozowski's algorithm, parametric on `reverse and determinize`. Instantiations to language and must equivalence in Sections 2 and 3.

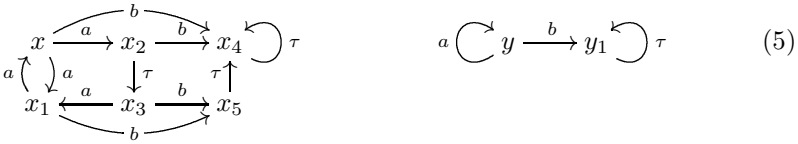
3 Must Semantics

The operational semantics of concurrent systems is usually given by *labelled transition systems* (LTS's), labelled by actions that are either visible to an external observer or internal actions (usually denoted by a special symbol τ). Different kinds of semantics can be defined on these structures (*e.g.*, linear or branching time, strong or weak semantics). In this paper we consider *must semantics* [14] which, intuitively, equates those systems that pass exactly the same tests, in all their executions.

Before formally introducing *must semantics* as in [12], we fix some notations: $\stackrel{\varepsilon}{\Rightarrow}$ denotes $\stackrel{\tau}{\rightarrow}^*$ the reflexive and transitive closure of $\stackrel{\tau}{\rightarrow}$ and, for $a \in A$, $\stackrel{a}{\Rightarrow}$ denotes $\stackrel{\tau}{\rightarrow}^* \stackrel{a}{\rightarrow} \stackrel{\tau}{\rightarrow}^*$. For $w \in A^*$, $\stackrel{w}{\Rightarrow}$ is defined inductively, in the obvious way. The *acceptance set of* x *after* w is $A(x, w) = \{\{a \in A \mid x' \stackrel{a}{\rightarrow}\} \mid x \stackrel{w}{\Rightarrow} x' \wedge x' \stackrel{\tau}{\not\rightarrow}\}$. Intuitively, it represents the set of actions that can be fired after "maximal" executions of w from x, those that cannot be extended by some τ-labelled transitions. The possibility of executing τ-actions forever is referred to as *divergence*. We write $x \not\downarrow$ whenever x diverges. Dually, the convergence relation $x \downarrow w$ for a state x and a word $w \in A^*$ is inductively defined as follows: $x \downarrow \varepsilon$ iff x does not diverge and $x \downarrow aw'$ iff (a) $x \downarrow \varepsilon$ and (b) if $x \stackrel{a}{\Rightarrow} x'$, then $x' \downarrow w'$. Given two sets $B, C \in \mathcal{PP}(A)$, we write $B \subset\subset C$ iff for all $B_i \in B$, there exists $C_i \in C$ such that $C_i \subseteq B_i$. With these ingredients, it is possible to introduce must preorder and equivalence.

Definition 3 (Must semantics [12]). *Let x and y be two states of an LTS. We write $x \sqsubseteq_{mst} y$ iff for all words $w \in A^*$, if $x \downarrow w$ then $y \downarrow w$ and $A(y,w) \subset\subset A(x,w)$. We say that x and y are must-equivalent ($x \sim_{mst} y$) iff $x \sqsubseteq_{mst} y$ and $y \sqsubseteq_{mst} x$.*

As an example, consider the LTS depicted below. States x_4, x_5 and y_1 are divergent. All the other states diverge for words containing the letter b and converge for words on a^*. For these words and states x, x_1, x_2, x_3 and y, the corresponding acceptance sets are $\{\{a,b\}\}$. In particular, note that $A(x_2,\varepsilon)$ is $\{\{a,b\}\}$ and not $\{\{b\},\{a,b\}\}$. It is therefore easy to conclude that x, x_1, x_2, x_3 and y are all must equivalent.



(5)

3.1 A Coalgebraic Characterization of Must Semantics

In what follows we show how $\sqsubseteq_{mst}$ can be captured in terms of coalgebras. This will further allow adapting the algorithms introduced in Section 2 for checking $\sim_{mst}$ and $\sqsubseteq_{mst}$.

First, we model LTS's in terms of coalgebras $(S, t\colon S \to (1+\mathcal{P}(S))^A)$, where $1 = \{\top\}$ is the singleton set, and for $x \in S$,

$$t(x)(a) = \top, \text{ if } x \not\downarrow a \qquad t(x)(a) = \{y \mid x \stackrel{a}{\Rightarrow} y\}, \text{ otherwise.}$$

Intuitively, a state $x \in S$ that displays divergent behaviour with respect to an action $a \in A$ is mapped to $\top$. Otherwise t computes the set of states that can be reached from x through a (by possibly performing a finite number of τ-transitions). At this point we need some additional definitions: for a function $\varphi\colon A \to \mathcal{P}(S)$, $I(\varphi)$ denotes the set of all labels "enabled" by φ, given by $I(\varphi) = \{a \in A \mid \varphi(a) \neq 0\}$, while $\mathit{Fail}(\varphi)$ denotes the set $\{Z \subseteq A \mid Z \cap I(\varphi) = 0\}$. With these definitions, we decorate the states of an LTS by means of an output function $o\colon S \to 1 + \mathcal{P}(\mathcal{P}(A))$ defined as follows:

$$o(x) = \top, \text{ if } x \not\downarrow \qquad o(x) = \bigcup_{x \xrightarrow{\tau} x'} o(x') \text{ if } x \xrightarrow{\tau}, \quad o(x) = \mathit{Fail}(t(x)), \text{ otherwise.}$$

Note that $(S, \langle o, t\rangle)$ is an FT-coalgebra for the functor $F(S) = (1+\mathcal{P}\mathcal{P}A) \times S^A$ and the monad $T(S) = 1 + \mathcal{P}(S)$. Algebras for such monad T are semilattices with bottom and an extra element $\top$ acting as *top* (*i.e.*, such that $x \sqcup \top = \top$ for all x). For any set U, $1+\mathcal{P}(U)$ carries a semilattice with bottom and top: bottom is the empty set; top is the element $\top \in 1$; $X \sqcup Y$ is defined as the union for arbitrary subsets $X, Y \in \mathcal{P}(U)$ and as $\top$ otherwise. Consequently, $1 + \mathcal{P}(\mathcal{P}A)$, $1 + \mathcal{P}(S)$, $(1 + \mathcal{P}(S))^A$ and $FT(S)$ carry a T-algebra structure as well. This enables the application of the generalized powerset construction (Section 2.3)

associating to each FT-coalgebra $(S, \langle o, t\rangle)$ the F-coalgebra $(1+\mathcal{P}(S), \langle o^\sharp, t^\sharp\rangle)$ defined for all $X \in 1+\mathcal{P}(S)$ as expected:

$$o^\sharp(X) = \begin{cases} \top & \text{if } X = \top \\ \bigsqcup_{x\in X} o(x) & \text{if } X \in \mathcal{P}(S) \end{cases} \qquad t^\sharp(X)(a) = \begin{cases} \top & \text{if } X = \top \\ \bigsqcup_{x\in X} t(x)(a) & \text{if } X \in \mathcal{P}(S) \end{cases}$$

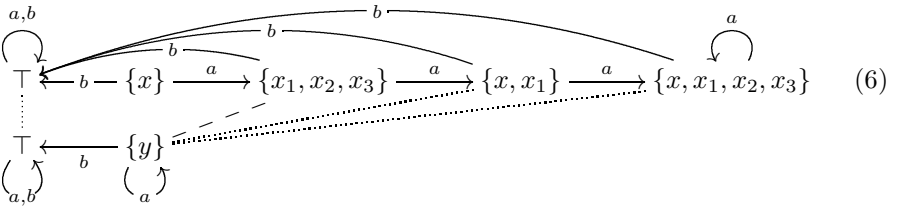
Note that in the above definitions, $\sqcup$ is not simply the union of subsets, but it is the join operation in $1+\mathcal{P}\mathcal{P}A$ and $1+\mathcal{P}(S)$. Moreover, $(1+\mathcal{P}S, \langle o^\sharp, t^\sharp\rangle)$ is a Moore machine with output in $1+\mathcal{P}\mathcal{P}A$ and, therefore, the equations in (1) induce a function $[\![-]\!]\colon (1+\mathcal{P}(S)) \to (1+\mathcal{P}\mathcal{P}A)^{A^*}$. The semilattice structure of $1+\mathcal{P}\mathcal{P}A$ can be easily lifted to $(1+\mathcal{P}\mathcal{P}A)^{A^*}$: bottom, top and $\sqcup$ are defined pointwise on A^*. If $\sqsubseteq_{\mathcal{M}}$ represents the preorder on $(1+\mathcal{P}\mathcal{P}A)^{A^*}$ induced by this semilattice, then the following theorem holds.

Theorem 2. *$x \sqsubseteq_{mst} y$ iff $[\![\{y\}]\!] \sqsubseteq_{\mathcal{M}} [\![\{x\}]\!]$ and $x \sim_{mst} y$ iff $[\![\{x\}]\!] = [\![\{y\}]\!]$.*

Note that according to the definition of $\sqsubseteq_{\mathcal{M}}$, $[\![\{y\}]\!] \sqsubseteq_{\mathcal{M}} [\![\{x\}]\!]$ iff $[\![\{y\}]\!] \sqcup [\![\{x\}]\!] = [\![\{x\}]\!]$, and since $[\![-]\!]$ is a T-homomorphism (namely it preserves bottom, top and $\sqcup$), the latter equality holds iff $[\![\{y, x\}]\!] = [\![\{x\}]\!]$. Summarizing,

$$x \sqsubseteq_{mst} y \text{ iff } [\![\{x, y\}]\!] = [\![\{x\}]\!].$$

Consider, once more, the LTS in (5). The part of the Moore machine $(1+\mathcal{P}(S), \langle o^\sharp, t^\sharp\rangle)$ which is reachable from $\{x\}$ and $\{y\}$ is depicted below (the output function $o^\sharp$ maps $\top$ to $\top$ and the other states to $\{0\}$). The relation consisting of dashed and dotted lines is a bisimulation proving that $[\![\{x\}]\!] = [\![\{y\}]\!]$, *i.e.*, that $x \sim_{mst} y$.



(6)

Our construction is closely related to the one in [12], that transforms LTS's into (deterministic) acceptance graphs. We refer the interested reader to a detailed comparison provided in the full version of this paper [5]. There we also show an optimization for representing outputs by means of $I(t(x))$ rather $Fail(t(x))$.

3.2 HKC for Must Semantics

The coalgebraic characterization discussed in the previous section guarantees soundness and completeness of bisimulation up-to congruence for must equivalence. Bisimulations are now relations $R \subseteq (1+\mathcal{P}(S)) \times (1+\mathcal{P}(S))$ on the state space $1+\mathcal{P}(S)$ where $o^\sharp$ and $t^\sharp$ are defined as in Section 3.1. Now, the congruence closure $c(R)$ of a relation $R \subseteq (1+\mathcal{P}(S)) \times (1+\mathcal{P}(S))$ is defined by the rules in (4) where $\sqcup$ is the join in $(1+\mathcal{P}(S))$ (rather than the union in

$\mathcal{P}(S)$). By simply redefining $o^\sharp$, $t^\sharp$ and $c(R)$, the algorithm in Figure 1 can be used to check must equivalence and preorder (the detailed proof can be found in the full version of the paper [5]). In particular, note that the check at step `2.1` can be done with the same algorithm as in [7, Section 3.4].

Suppose, for example, that we want to check whether the states x and y of the LTS in (5) are must equivalent. The relation $R = \{(\{x\}, \{y\}), (\{x_1, x_2, x_3\}, \{y\})\}$ depicted by the dashed lines in (6) is not a bisimulation, but a bisimulation up-to congruence, since both $(\top, \top) \in c(R)$ and $(\{x, x_1\}, \{y\}) \in c(R)$. For the latter, observe that

$$\{x, x_1\}\ c(R)\ \{y, x_1\}\ c(R)\ \{x_1, x_2, x_3\}\ c(R)\ \{y\}.$$

It is important to remark here that `HKC` computes this relation without the need of exploring all the reachable part of the Moore machine $(1 + \mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$. So, amongst all the states in (6), `HKC` only explores $\{x\}$, $\{y\}$ and $\{x_1, x_2, x_3\}$.

3.3 Brzozowski's Algorithm for Must Semantics

A variation of the Brzozowski algorithm for Moore machines is given in [4]. We could apply such algorithm to the Moore machine $(1 + \mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$ which is induced by the coalgebra $(S, \langle o, t \rangle)$ introduced in Section 3.1. Here, we propose a more efficient variation that skips the first determinization from $(S, \langle o, t \rangle)$ to $(1 + \mathcal{P}(S), \langle o^\sharp, t^\sharp \rangle)$.

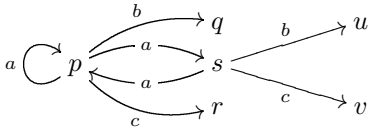
The novel algorithm consists of the four steps described in Section 2.2, where the procedure `reverse and determinize` is modified as follows: $(S, \langle o, t \rangle)$ with the set of initial states I is transformed into $((1 + \mathcal{P}\mathcal{P}(A))^S, \langle \overline{o}_R, \overline{t}_R \rangle)$ where $\overline{o}_R \colon (1 + \mathcal{P}\mathcal{P}A)^S \to 1 + \mathcal{P}\mathcal{P}A$ and $\overline{t}_R \colon (1 + \mathcal{P}\mathcal{P}A)^S \to ((1 + \mathcal{P}\mathcal{P}A)^S)^A$ are defined for all functions $\psi \in (1 + \mathcal{P}\mathcal{P}A)^S$ as

$$\overline{o}_R(\psi) = \bigsqcup_{x \in I} \psi(x) \qquad \overline{t}_R(\psi)(a)(x) = \begin{cases} \top & \text{if } t(x)(a) = \top \\ \bigsqcup_{y \in t(x)(a)} \psi(y) & \text{otherwise} \end{cases}$$

and the new initial state is $\overline{I}_R = o$.

Note that the result of this procedure is a Moore machine. Brzozowski's algorithm in Section 2.2 transforms an NDA $(S, \langle o, t \rangle)$ with initial state I into the minimal DA for $[\![I]\!]$. Analogously, our algorithm transforms an LTS into the minimal Moore machine for $[\![I]\!]$.

Let us illustrate the minimization procedure by means of an example. Take the alphabet $A = \{a, b, c\}$ and the LTS depicted below on the left.



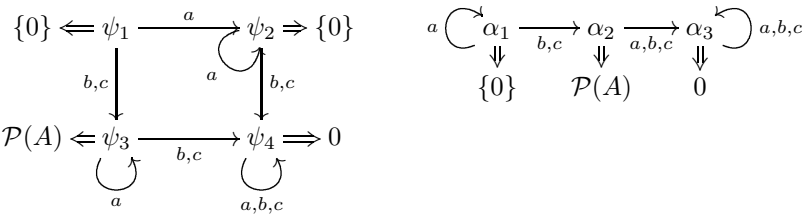
$$o(p) = \{0\} \quad o(s) = \{0\}$$
$$o(q) = \mathcal{P}(A) \quad o(u) = \mathcal{P}(A)$$
$$o(r) = \mathcal{P}(A) \quad o(v) = \mathcal{P}(A)$$

Since there are no τ transitions, the function $t \colon S \to (1 + \mathcal{P}(S))^A$ is defined as on the left, and the function $o \colon S \to (1 + \mathcal{P}\mathcal{P}A)$ (given on the right) assigns to

each state x the set $\mathit{Fail}(t(x))$. Suppose we want to build the minimal Moore machine for the behaviour $[\![\{p\}]\!]\colon A^* \to 1 + \mathcal{P}\mathcal{P}A$, which is the function

$$[\![\{p\}]\!]\colon a^* \mapsto \{0\},\ a^*b \mapsto \mathcal{P}(A),\ a^*c \mapsto \mathcal{P}(A),\ \_ \mapsto 0$$

where $\_$ denotes all the words different from a^*, a^*b and a^*c. By applying our algorithm to the coalgebra $(S, \langle o, t\rangle)$, we first obtain the intermediate Moore machine on the left below, where a double arrow $\psi \Rightarrow Z$ means that the output of ψ is the set Z. The initial state is $\psi_1\colon S \to 1 + \mathcal{P}\mathcal{P}A$ which, by definition, is the output function o above. The explicit definitions of the other functions ψ_i can be computed according to the definition of $\overline{t}_R$.



Observe that $[\![\psi_1]\!]$ is the "reversed" of $[\![\{p\}]\!]$. For instance, triggering ba^* from ψ_1 leads to ψ_3 with output $\mathcal{P}(A)$; this is the same output we get by executing a^*b from p, according to $[\![\{p\}]\!]$. Executing `reverse and determinize` once more (step `3`) and taking the reachable part (step `4`), we obtain the minimal Moore machine on the right, with initial state α_1.

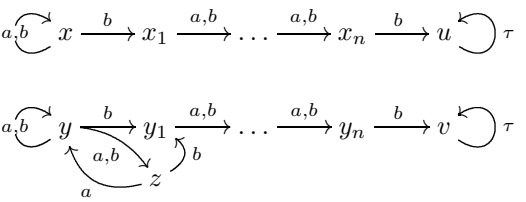
We have proved the correctness of this algorithm in the full version of this paper [5]; it builds on the coalgebraic perspective on Brzozowski's algorithm given in [4].

4 A Family of Examples

As discussed in the introduction, the problem of checking must equivalence is PSPACE-complete [17]. Hence, a theoretical comparison of `HKC`, Brzozowski (`BRZ`) and the partition refinement (`PR`) of [12] will be less informative than a thorough experimental analysis. Designing adequate experiments is out of the scope of this paper. We will instead just show the reader some concrete examples. It is possible to show some concrete cases where (a) `HKC` takes polynomial time while `BRZ` and `PR` exponential time and (b) (`BRZ`) polynomial time while `HKC` and `PR` exponential time. There are also examples where (c) `PR` is polynomial and `BRZ` is exponential, but it is impossible to have `PR` polynomial and `HKC` exponential. Indeed, cycle `2` of `HKC` is repeated at most $1 + |A|\cdot|R|$ times where $|A|$ is the size of the alphabet and $|R|$ is the size of the produced relation R. Such relation always contains at most n pairs of states, for n being the size of the reachable part of the determinised system. Therefore, if `HKC` takes exponential time, then also `PR` takes exponential time since it always needs to build the reachable part of the determinised LTS.

In this section we show an example for (a). Examples for (b) and (c) can be found in the full version of this paper [5].

Consider the following LTS, where n is an arbitrary natural number. After the determinization, $\{x\}$ can reach all the states of the shape $\{x\} \cup X_N$, where $X_N = \{x_i \mid i \in N\}$ for any $N \subseteq \{1, \ldots, n\}$. For instance for $n = 2$, $\{x\} \stackrel{aa}{\to} \{x\}$, $\{x\} \stackrel{ab}{\to} \{x, x_1\}$, $\{x\} \stackrel{ba}{\to} \{x, x_2\}$ and $\{x\} \stackrel{bb}{\to} \{x, x_1, x_2\}$. All those states are distinguished by must and, therefore, the minimal Moore machine for $[\![\{x\}]\!]$ has at least 2^n states.



One can prove that x and y are must equivalent by showing that relation

$$R = \{(\{x\}, \{y\}), (\{x\}, \{y, z\}), (\top, \top)\} \\ \cup \{(\{x\} \cup X_N, \{y, z\} \cup Y_N) \mid N \subseteq \{1, \ldots, n\}\}$$

is a bisimulation (here $Y_N = \{y_i \mid i \in N\}$). Note that R contains $2^n + 2$ pairs.

In order to check $[\![\{x\}]\!]$=$[\![\{y\}]\!]$, HKC builds the following relation,

$$R' = \{(\{x\}, \{y\}), (\{x\}, \{y, z\})\} \cup \{(\{x, x_i\}, \{y, z, y_i\}) \mid i \in \{1, \ldots, n\}\}$$

which is a bisimulation up-to and which contains only $n + 2$ pairs. It is worth to observe that R' is like a "basis" of R: all the pairs $(X, Y) \in R$ can be generated by those in R' by iteratively applying the rules in (4). Therefore, HKC proves $[\![\{x\}]\!]$=$[\![\{y\}]\!]$ in polynomial time, while minimization-based algorithms (such as [12] or Brzozowski's algorithm) require exponential time.

5 Conclusions and Future Work

We have introduced a coalgebraic characterization of must testing semantics by means of the *generalized powerset construction* [28]. This allowed us to adapt proof techniques and algorithms that have been developed for language equivalence to must semantics. In particular, we showed that *bisimulations up-to congruence* (that was recently introduced in [7] for NDA's) are sound also for must semantics. This fact guarantees the correctness of a generalization of HKC [7] for checking must equivalence and preorder and suggests that the *antichains*-based algorithms [32,1] can be adapted in a similar way. We have also proposed a variation of Brzozowski's algorithm [9] to check must semantics, by exploiting the abstract theory in [4]. Our contribution is not a simple instantiation of [4], but developing our algorithm has required some ingenuity to avoid the preliminary

determinization that would be needed to directly apply [4]. We implemented these algorithms together with an interactive applet available online [6].

We focused on must testing semantics because it is challenging to compute, but our considerations hold also for may testing and for several decorated trace semantics of the *linear time/branching time spectrum* [30] (namely, those that have been studied in [3]). Adapting these algorithms to check *fair testing* [24] seems to be more complicated: while it is possible to coalgebraically capture failure trees, we do not know how to model fair testing equivalence. We believe that this is a challenging topic to investigate in the future. Moreover, since coalgebras can easily model probabilistic systems, it is worth to investigate whether our approach can be extended to the testing semantics of probabilistic and nondeterministic processes (e.g. [15]).

References

1. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
2. Bezhanishvili, N., Kupke, C., Panangaden, P.: Minimization via duality. In: Ong, L., de Queiroz, R. (eds.) WoLLIC 2012. LNCS, vol. 7456, pp. 191–205. Springer, Heidelberg (2012)
3. Bonchi, F., Bonsangue, M., Caltais, G., Rutten, J., Silva, A.: Final semantics for decorated traces. Elect. Not. in Theor. Comput. Sci. 286, 73–86 (2012)
4. Bonchi, F., Bonsangue, M.M., Rutten, J.J.M.M., Silva, A.: Brzozowski's algorithm (Co)Algebraically. In: Constable, R.L., Silva, A. (eds.) Kozen Festschrift. LNCS, vol. 7230, pp. 12–23. Springer, Heidelberg (2012)
5. Bonchi, F., Caltais, G., Pous, D., Silva, A.: Brzozowski's and up-to algorithms for must testing (full version), `http://www.alexandrasilva.org/files/brz-hkc-must-full.pdf`
6. Bonchi, F., Caltais, G., Pous, D., Silva, A.: Web appendix of this paper, with implementation of the algorithms (July 2013), `http://perso.ens-lyon.fr/damien.pous/brz`
7. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: POPL, pp. 457–468. ACM (2013)
8. Boreale, M., Gadducci, F.: Processes as formal power series: a coinductive approach to denotational semantics. TCS 360(1), 440–458 (2006)
9. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. Mathematical Theory of Automata 12(6), 529–561 (1962)
10. Calzolai, F., De Nicola, R., Loreti, M., Tiezzi, F.: TAPAs: A tool for the analysis of process algebras. In: Jensen, K., van der Aalst, W.M.P., Billington, J. (eds.) ToPNoC I. LNCS, vol. 5100, pp. 54–70. Springer, Heidelberg (2008)
11. Cancila, D., Honsell, F., Lenisa, M.: Generalized coiteration schemata. Elect. Not. in Theor. Comput. Sci. 82(1) (2003)
12. Cleaveland, R., Hennessy, M.: Testing equivalence as a bisimulation equivalence. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 11–23. Springer, Heidelberg (1990)
13. Cleaveland, R., Parrow, J., Steffen, B.: The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. TOPLAS 15(1), 36–72 (1993)

14. De Nicola, R., Hennessy, M.: Testing equivalences for processes. TCS 34, 83–133 (1984)
15. Deng, Y., van Glabbeek, R.J., Hennessy, M., Morgan, C.: Real-reward testing for probabilistic processes. In: QAPL. EPTCS, vol. 57, pp. 61–73 (2011)
16. Hopcroft, J.E.: An n log n algorithm for minimizing in a finite automaton. In: Proc. Int. Symp. of Theory of Machines and Computations, pp. 189–196. Academic Press (1971)
17. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. In: PODC 1983, pp. 228–240. ACM, New York (1983)
18. Klin, B.: A coalgebraic approach to process equivalence and a coinduction principle for traces. Elect. Not. in Theor. Comput. Sci. 106, 201–218 (2004)
19. Klin, B.: Bialgebras for structural operational semantics: An introduction. TCS 412(38), 5043–5069 (2011)
20. Lenisa, M.: From set-theoretic coinduction to coalgebraic coinduction: some results, some problems. Elect. Not. in Theor. Comput. Sci. 19, 2–22 (1999)
21. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
22. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
23. Parrow, J., Sjödin, P.: Designing a multiway synchronization protocol. Computer Communications 19(14), 1151–1160 (1996)
24. Rensink, A., Vogler, W.: Fair testing. Inf. Comput. 205(2), 125–198 (2007)
25. Rot, J., Bonsangue, M., Rutten, J.: Coalgebraic bisimulation-up-to. In: van Emde Boas, P., Groen, F.C.A., Italiano, G.F., Nawrocki, J., Sack, H. (eds.) SOFSEM 2013. LNCS, vol. 7741, pp. 369–381. Springer, Heidelberg (2013)
26. Rutten, J.: Universal coalgebra: a theory of systems. TCS 249(1), 3–80 (2000)
27. Sangiorgi, D.: On the bisimulation proof method. Math. Struc. in CS 8, 447–479 (1998)
28. Silva, A., Bonchi, F., Bonsangue, M., Rutten, J.: Generalizing the powerset construction, coalgebraically. In: Proc. FSTTCS. LIPIcs, vol. 8, pp. 272–283 (2010)
29. Tabakov, D., Vardi, M.: Experimental evaluation of classical automata constructions. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 396–411. Springer, Heidelberg (2005)
30. van Glabbeek, R.: The linear time - branching time spectrum I. The semantics of concrete, sequential processes. In: Handbook of Process Algebra, pp. 3–99. Elsevier (2001)
31. Watson, B.W.: Taxonomies and Toolkits of Regular Language Algorithms. PhD thesis, Eindhoven University of Technology, the Netherlands (1995)
32. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)